APLICAÇÃO DE WAVE FUNCTION COLLAPSE E RAY CASTING NA CRIAÇÃO DE UM JOGO ISOMÉTRICO

Thomas Ricardo Reinke, Dalton Solano dos Reis – Orientador

Curso de Bacharel em Ciência da Computação
Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil

treinke@furb.br, dalton@furb.br

Resumo: *Este artigo apresenta o desenvolvimento de uma aplicação para demonstração do funcionamento do algoritmo Wave Function Collapse juntamente com o algoritmo de Ray Casting, com o objetivo de construir um ambiente com cenário isométrico, fornecendo uma perspectiva tridimensional em uma projeção bidimensional. Os resultados alcançados foram avaliados a partir do tempo de execução de cada algoritmo abordado, assim como também, utilizando de algumas métricas específicas para cada um deles, como tamanho do mundo gerado ou quantidade de rays disparados e ângulo do campo de visão. Tratando-se de sua performance, os algoritmos se mostram eficientes, sendo possível alcançar um resultado satisfatório com a junção dos dois algoritmos. Em relação a naturalidade do terreno gerado, dependendo da quantidade de diferentes tiles, é necessário estudo mais aprofundados nas regras se fazem necessários.*

Palavras-chave*: Wave Function Collapse. Procedural Content Generation. Ray casting. Isometria.*

1 INTRODUÇÃO

Desde o surgimento dos primeiros jogos eletrônicos, com o desenvolvimento dos computadores e aumento do poder de processamento, os jogos vêm se tornando cada vez maiores e mais complexos, atingindo novos horizontes. Durante a evolução dos gráficos, várias vertentes apareceram conforme as décadas passaram, partindo de gráficos 2D, para gráficos isométricos, também conhecidos como 2,5D e alcançando gráficos completamente 3D.

Nos primeiros passos do desenvolvimento gráfico nos jogos, alguns algoritmos foram criados. Um deles, o algoritmo de Ray Casting, como Peddie (2016) descreve, é responsável por calcular onde estão os objetos mais próximos do usuário, traçando raios do ponto de vista do espectador até os objetos no cenário. Esse algoritmo foi utilizado principalmente para duas funções diferentes, a primeira sendo a visibilidade do jogador em um jogo 2D ou isométrico, e a segunda, como Peddie (2016) define, para transformar uma forma limitada de dados (uma matriz simplificada) em uma projeção 3D. Esse algoritmo, atualmente, é bastante utilizado em sensores para comparar as leituras do sensor com a distância *ground truth* entre os obstáculos de um mapa (WALSH; KARAMAN, 2018).

Devido ao segmento gráfico e geração de conteúdo para esses jogos estar constantemente se tornando mais complexo, e ter aumentado exponencialmente em tamanho, custo e tempo de desenvolvimento, o uso de algoritmos para geração de diversos tipos de conteúdo dentro do jogo está progressivamente maior. O Procedural Content Generation (PCG), como é chamado, permite que partes do jogo (mapas, texturas, itens, missões etc.) sejam gerados algoritmicamente ao invés de diretamente por um design humano (RISI *et al*., 2014).

Ao se entrar na definição de PCG, há dois tipos de jogos distintos, os *non-PCG-based*, que usufruem do PCG, porém, seriam capazes de serem jogos sem aplicar esse tema, apesar de talvez não conseguir impressionar o usuário final da mesma forma. Em contrapartida, existem os jogos *PCG-based*, que possuem toda experiência jogável estruturada pelo PCG, sendo possível assim, expandir ou até mesmo criar um gênero (SMITH *et al*., 2019). Os algoritmos PCG podem ser divididos nas seguintes quatro categorias: métodos construtivos, métodos baseados em busca, métodos baseados em restrições e métodos machine-learning (SUMMERVILLE *et al*., 2018). De acordo com Smith *et al*. (2019), os jogos que se baseiam completamente em PCG podem ser distinguidos dos demais de acordo com três diferentes itens: rejogabilidade, adaptabilidade e o controle do jogador sobre o conteúdo.

Com o passar das décadas, o PCG ficou cada vez mais comum no desenvolvimento de jogos e, com ele, o surgimento de diferentes algoritmos para a realização das necessidades propostas. Dentre eles está o algoritmo Wave Function Collapse (WFC), um algoritmo PCG proposto recentemente, baseado em resolver restrições, que possui um grande valor potencial no campo do desenvolvimento de jogos (CHENG; HAN; FEI, 2020). Esse termo surgiu da área de mecânica quântica e tem sido introduzido recentemente na área tecnológica para possibilitar, principalmente, a criação de principalmente de novas imagens que possuam uma sequência, que faça sentido, baseada em uma imagem de entrada. Por conta de sua aplicação na área da tecnologia ser recente, jogos utilizando esse algoritmo em sua base são bem escassos, possuindo os jogos desenvolvidos baseados em sua maioria em planos 2D (Caves of Qud) e 3D (Bad North: Jotunn Edition e Townscaper).

Diante do apresentado, este trabalho desenvolveu uma aplicação para demonstrar como o algoritmo de WFC e Ray Casting podem ser utilizados em conjunto para criar um ambiente isométrico. Os objetivos específicos do trabalho são: demonstrar a utilização do algoritmo de WFC com a adição de restrições em um ambiente isométrico; demonstrar a utilização do algoritmo de Ray Casting em um ambiente isométrico; comparar as performances dos diferentes algoritmos de Ray Casting. A análise de performance dos algoritmos irá se basear em métricas padrões, como tempo de execução utilizando diferentes parâmetros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta a fundamentação dos assuntos abordados no trabalho, estando dividido em quatro subseções. A subseção 2.1 descreve o que é a geração de conteúdo procedural e demonstra o funcionamento do algoritmo de WFC. A subseção 2.2 descreve o conceito e definição do algoritmo de Ray Casting e demonstra alguns diferentes algoritmos que podem ser utilizados em diferentes casos. A subseção 2.3 aborda a projeção em isometria e algumas fórmulas que podem ser utilizadas para recriar um ambiente do tipo *diamond*. Por fim, a subseção 2.4 discute três trabalhos correlatos que utilizam os temas base deste artigo, onde demonstram e explicam detalhadamente o funcionamento de cada tópico.

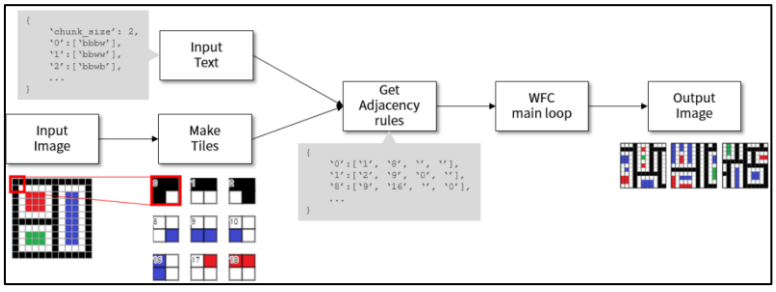
2.1 WAVE FUNCTION COLLAPSE E GERAÇÃO DE CONTEÚDO PROCEDURAL

Togelius (2011) define o Procedural Content Generation (PCG) em jogos como a criação de conteúdo automaticamente fazendo o uso de algoritmos. Alguns exemplos famosos são a geração de *dungeons* em Rogue (AI Design 1980), a geração de mapa em Civilization (MicroPose 1991), a geração aleatória de armas em Borderlands (Gearbox 2009) e a vegetação em SpeedTree (Interactive Data Visualization 2003). Sandhu, Chen e McCoy (2019) complementam que é possível reduzir a quantidade de recursos que são gastos ao se desenvolver novos *assets*, como por exemplo, em tarefas como geração de níveis, o PCG pode acelerar a criação dos *assets* e reduzir tempo que seria gasto na produção dos mesmos, podendo até mesmo contribuir para a redução do custo geral do projeto. Como diz Shaker *et al.* (2016), é possível que o PCG remova a necessidade de um designer humano ou artista que seria necessário para gerar conteúdo, humanos são lentos e de alto custo, e parece que precisamos deles cada vez mais e mais.

Apesar de serem muitos os pontos positivos na aplicação destes algoritmos em jogos, atualmente, pouca pesquisa envolvendo o PCG aborda profundamente como criar conteúdo interativo, ou jogos completamente novos. Há pesquisas relacionadas a criação de PCG que podem ser controlados por designers humanos, mas pouco conteúdo relacionado à sua utilização por jogadores (SMITH, G. 2021). Dentro da categoria de algoritmos PCG, há o algoritmo Wave Function Collapse (WFC), que surgiu recentemente e se mostra bastante promissor.

Møller, Billeskov e Palamas (2020) definem o WFC como um algoritmo de satisfação de restrição inspirado no conceito do colapso da função de onda presente na física quântica, onde um estado não observado possibilita que todos os estados sejam possíveis, enquanto observações restringem as possibilidades. O WFC ficou famoso rapidamente pois permite que sejam introduzidas restrições de design no processo generativo. Até então, o algoritmo só teria sido usado na geração de imagens. Contudo, eventualmente, sua introdução na geração de jogos e mapas foram aparecendo. Como afirma Kim *et al.* (2019), o momento em que o WFC se desprende da síntese de texturas é um aspecto chave para permitir novas aplicações surpreendentes no design de jogos. É possível ver na Figura 1 o fluxo que engloba o processo do algoritmo WFC.

Figura 1 – processo do algoritmo WFC



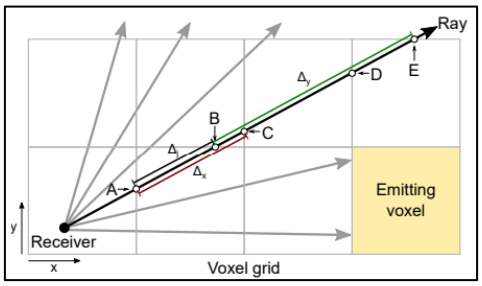
Fonte: Kim *et al.* (2019).

2.2 O ALGORITMO DE RAY CASTING

O Ray Casting foi usado primeiramente como uma técnica de renderização 3D partindo de uma matriz 2D. Quando os computadores não possuíam poder suficiente para rodar motores 3D em tempo real, o algoritmo de Ray Casting foi a primeira solução (VANDEVENNE, 2004). Este algoritmo consiste basicamente em uma quantidade definida de raios saindo de um ponto em comum, e percorrem a matriz até encontrarem algum obstáculo. Atualmente, este algoritmo é usado para leitura de imagens e, de acordo com Walsh e Karaman (2018), para comparar as leituras do sensor sobre os obstáculos de um mapa.

O algoritmo de Ray Casting pode ser usado juntamente com mais algoritmos, um exemplo deles, é o algoritmo Digital Difference Analyzer (DDA). De acordo com Museth (2014), é um conceito da computação gráfica que torna mais eficiente a rasterização de linhas, sendo um algoritmo que é bastante relacionado ao algoritmo de Bresenham. Na Figura 2 é possível verificar o funcionamento do algoritmo de Ray Casting juntamente com o algoritmo DDA.

Figura 2 – Representação do algoritmo DDA



Fonte: Hilton *et al.* (2018).

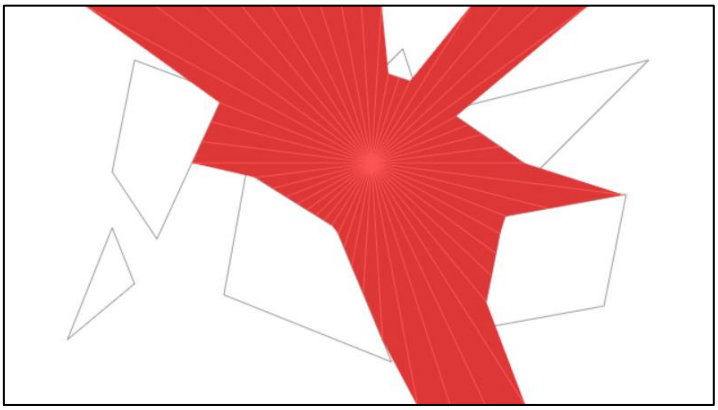
O *ray* lançado a partir do algoritmo de Ray Casting irá verificar a colisão somente quando encontrar o *grid* de algum dos eixos do plano cartesiano, seja ele `x` ou `y`. Assim, possibilitando que seja possível realizar cálculos apenas quando necessário, economizando tempo de processamento e memória. De acordo com Vandevenne (2004) quando derivada a distância `x` e `y` utilizando a fórmula de Pitágoras, é possível obter a equação `(1)` para os valores do eixo das abscissas e a equação `(2)` para os valores do eixo das ordenadas.

$$Sx \leftarrow \sqrt{1 + \left(\frac{dy}{dx}\right)^2} \quad (1) \qquad Sy \leftarrow \sqrt{1 + \left(\frac{dx}{dy}\right)^2} \quad (2)$$

Segundo Vandevenne (2004), o algoritmo DDA sempre irá pular exatamente um quadrado do plano cartesiano, podendo ser um quadrado na direção `x` ou na direção `y`. A direção de lançamento dos *rays* devem ser controladas por uma variável separada, tanto para `x` quanto para `y`, podendo possuir os valores de +1 ou -1. Assim, se a direção do *ray* tiver um `componente-x` negativo, os valores serão calculados entre a distância da posição do lançamento do *ray* até a primeira coluna do plano cartesiano à esquerda do ponto, em caso de possuir um componente-x positivo, à direita. O mesmo cálculo é realizado para o `componente-y`, em caso positivo, será calculado a distância da posição do lançamento do *ray* até a primeira linha acima, caso contrário, abaixo. Em cada repetição destas verificações é adicionado um quadrado, até que uma parede seja encontrada.

Uma outra solução seria utilizar o algoritmo de Ray Casting a partir da equação para verificar se houve intersecção entre o *ray* lançado do observador e a linha formada entre dois vértices. Case (2014), partindo de uma solução sem a lista de vértices dos obstáculos presentes nas cenas, criou um ambiente onde foram lançadas 50 *rays* espaçadas igualmente em 360°, e é possível ver na Figura 3 que a solução não atende completamente o proposto. Case (2014) afirma que mesmo com 360 *rays* representando os 360°, a projeção do polígono gerado ficaria irregular e com muitas falhas.

Figura 3 – Representação do algoritmo de Ray Casting com 50 *rays*



Fonte: Case (2014).

A solução tomada para este problema, foi de definir uma lista com todos os vértices, e por fim, lançar um *ray* até cada um destes pontos em adição com mais dois *rays* com um *offset* de +/- 0.0001 radianos. Os *rays* adicionais seriam responsáveis por atingir as paredes atrás do objeto, dando a impressão de visão (CASE, 2014).

2.3 PROJEÇÃO EM ISOMETRIA

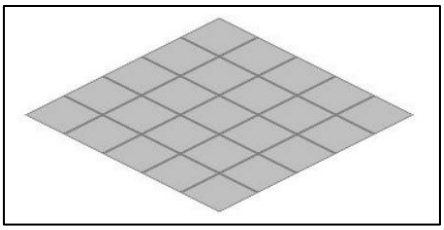
De acordo com Sampaio e Ramalho (2003), para entender o que são projeções isométricas, primeiro é necessário entender o que são projeções axonométricas. As projeções axonométricas são projeções do espaço 3D para o 2D que possuem as seguintes características:

a) a projeção no espaço 2D não possui "ponto de fuga";
b) linhas paralelas no espaço 3D continuam paralelas no espaço 2D;
c) objetos que estão distantes possuem o mesmo tamanho de objetos que estão pertos.

Já as projeções isométricas seguem estas mesmas características, porém, as medidas usadas nos eixos x, y e z são as mesmas, ou seja, uma unidade no eixo x é, em comprimento, igual a uma unidade nos eixos y e z.

Há vários tipos de mapas isométricos, e de acordo com Sampaio e Ramalho (2003), um dos principais tipos de mapas isométricos, são os do tipo *diamond*, geralmente utilizados em jogos de estratégia em tempo real. Na Figura 4, é possível ver um mapa do tipo *diamond*, que possui este nome por conta do seu formato.

Figura 4 – Representação de um mapa do tipo *diamond*



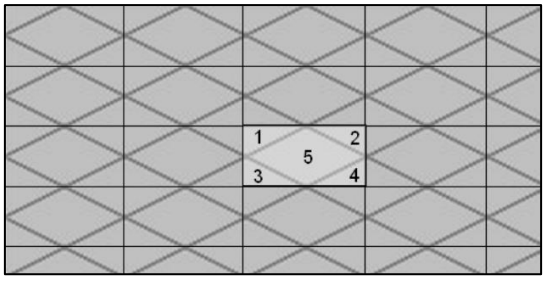
Fonte: Sampaio e Ramalho (2003).

Conforme Sampaio e Ramalho (2001), há alguns problemas relacionados com a plotagem isométrica, sendo um deles, a ordem com que os *tiles* são plotados. Quando os *tiles* possuem objetos que vão além da fronteira de sua *sprite*, é necessário seguir uma ordem de renderização para que a tela não fique inconsistente. As seguintes regras devem ser seguidas:

a) *tiles* precisam ser renderizados do fundo para frente, de forma com que nenhum *tile* seja plotado após outro que está em sua frente;
b) se uma parte da tela for atualizada, todos os *tiles* modificados e adjacentes devem ser redesenhados, seguindo a regra anterior.

Outro problema é descobrir em qual *tile* exatamente está um ponto na tela. Por conta de cada *tile* não ser um retângulo, é necessário cálculos matemáticos para efetuar a conversão de um espaço para outro. Na Figura 5 é possível verificar o que acontece ao dividir o mapa isométrico em retângulos para que se possa definir em qual dos índices da matriz o ponto está localizado. Uma das soluções possíveis seria a utilização da fórmula de verificar se um ponto está contido dentro de um triângulo retângulo, que seria formado em cada um dos vértices deste novo retângulo.

Figura 5 – Existência de 5 regiões ao dividir o mapa isométrico em retângulos



Fonte: Sampaio e Ramalho (2003).

2.4 TRABALHOS CORRELATOS

A seguir serão apresentados três trabalhos que possuem características semelhantes aos temas centrais do trabalho proposto e que utilizam algumas das técnicas apresentadas anteriormente. O Quadro 1 apresenta o trabalho de Sandhu, Chen e McCoy (2019) que demonstra a aplicação do algoritmo WFC para geração de terrenos com a utilização de restrições de design. Já o Quadro 2 apresenta o desenvolvimento de um aplicativo por Kasapakis, Gavalas e Galatis (2016), que utiliza o algoritmo de Ray Casting para o posicionamento de pontos de interesse em locais com base em realidade aumentada. E por fim, o Quadro 3 apresenta um *framework* para a criação de jogos isométricos desenvolvido por Sampaio e Ramalho (2003).

Quadro 1 – Trabalho Correlato 1

Referência	Sandhu, Chen e McCoy (2019)
Objetivos	Contribuir com o estudo do algoritmo WFC, demonstrando a possibilidade de implementar restrições de design ao cálculo da WFC, como por exemplo, recalculação de peso e alterações nos ciclos de observação e propagação do algoritmo.
Principais funcionalidades	São criadas a restrição *non-local constraints* que modifica diretamente o ciclo de observação/propagação do algoritmo, introduzindo variáveis externas e utilizando temas de dependência e frequência em que os itens adicionados devem aparecer; a restrição *weight recalculation* que recalcula o peso do *tile* informado para descobrir se mesmo após mais ciclos de preenchimento, o *tile* escolhido anteriormente ainda é o mais adequado; e a restrição de *area propagation* que irá modificar o passo de propagação do algoritmo, fazendo com que seja possível trabalhar com áreas maiores ao invés de *tiles* sozinhos.
Ferramentas de desenvolvimento	É criado um aplicativo desktop utilizando Electron.
Resultados e conclusões	Os testes de performance comparando o algoritmo WFC puro e alterado por restrições apontam que as restrições apresentadas possuem uma eficácia suficiente para serem utilizadas em tempo de execução.

Fonte: elaborado pelo autor.

Quadro 2 – Trabalho Correlato 2

Referência	Kasapakis, Gavalas e Galatis (2016)
Objetivos	Aplicar o algoritmo de Ray Casting juntamente com a realidade aumentada, para assim, produzir um novo aplicativo de celular que permite a visualização apenas de pontos de interesse (Point of Interest - POIs) que estão no campo de visão do usuário.
Principais funcionalidades	É aplicado o algoritmo de Ray Casting juntamente com a realidade aumentada e geolocalização para possibilitar que o usuário veja na tela apenas os POIs que estão visíveis, ou seja, não estão atrás de estruturas.
Ferramentas de desenvolvimento	Não especificado. Utiliza realidade aumentada e GPS.
Resultados e conclusões	Os testes de performance são realizados utilizando diferentes ângulos e comprimentos de *rays*, concluindo com os resultados que há possibilidade de utilizar o algoritmo em tempo real. Também é feito um estudo de caso com o aplicativo no sítio arqueológico de Cnossos na ilha de Creta, na Grécia, mostrando que a disponibilização apenas de POIs visíveis interessou mais os alunos por conta da diferente tecnologia e estilos de interação.

Fonte: elaborado pelo autor.

Quadro 3 – Trabalho Correlato 3

Referência	Sampaio e Ramalho (2003)
Objetivos	Reaproveitar o *framework* antecessor (Forge V8) e reconstruí-lo para o desenvolvimento de jogos isométricos.
Principais funcionalidades	Desenvolvido um *framework* composto por módulos de gráfico, entrada, saída, som, log, IA, modelagem e editor de cenários, para o desenvolvimento de jogos isométricos, facilitando a criação dos jogos.
Ferramentas de desenvolvimento	Desenvolvido em C# e utilizando Assembly em alguns pontos onde era necessária mais performance. Foi utilizada a biblioteca multimídia DirectX da Microsoft.
Resultados e conclusões	Os resultados foram medidos com base na validação do *framework* na disciplina de Projeto e Implementação de Jogos do curso de Graduação em Ciência da Computação do Centro de Informática da UFPE, por intermédio do desenvolvimento de três jogos. A avaliação do *framework* pelas equipes que o utilizaram para o desenvolvimento dos jogos foi bastante positiva, considerando-o como fácil de utilizar, robusto e bastante útil.

Fonte: elaborado pelo autor.

3 DESCRIÇÃO DO PROTÓTIPO

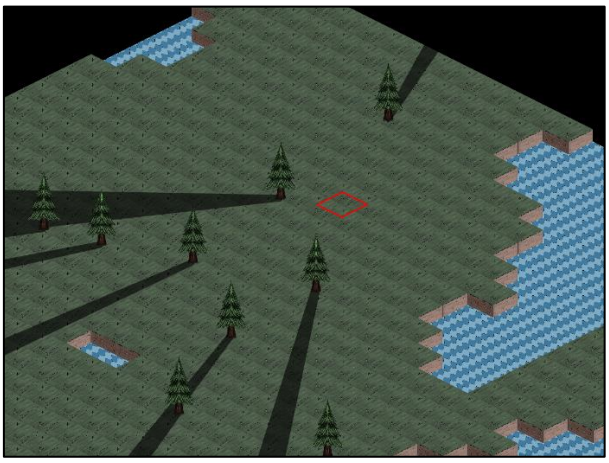
Esta seção apresenta a especificação do protótipo com suas principais funcionalidades e características. A subseção 3.1 mostra os principais aspectos do protótipo, as funcionalidades disponíveis, o fluxo de execução do protótipo e a interface gráfica final. A subseção 3.2 aborda a implementação de três diferentes algoritmos de Ray Casting, o algoritmo de WFC e alguns cálculos necessários para possibilitar a projeção isométrica do plano. O protótipo, código fonte e documentação complementar se encontram disponíveis no GitHub, no repositório de Reinke (2023).

3.1 ESPECIFICAÇÃO

No desenvolvimento deste protótipo foi utilizado a biblioteca Pygame, um grupo de módulos desenhados especificamente para criar jogos na linguagem Python. O protótipo desenvolvido é apenas uma demonstração de como o algoritmo WFC pode ser utilizado para gerar um mapa aleatoriamente e integrar diretamente com o Ray Casting para delimitar o que o jogador pode ou não ver, dependendo de sua posição e obstáculos presentes em tela. Assim, o desenvolvimento e execução do protótipo é realizado diretamente em um computador, desde que possua o Python versão 3.11.3 e o Pygame 2.3.0.

Como é possível ver na Figura 6, a execução do protótipo retorna um mundo gerado aleatoriamente pelo algoritmo WFC e pode constar com alguns obstáculos responsáveis por bloquear o campo de visão do usuário através do algoritmo de Ray Casting.

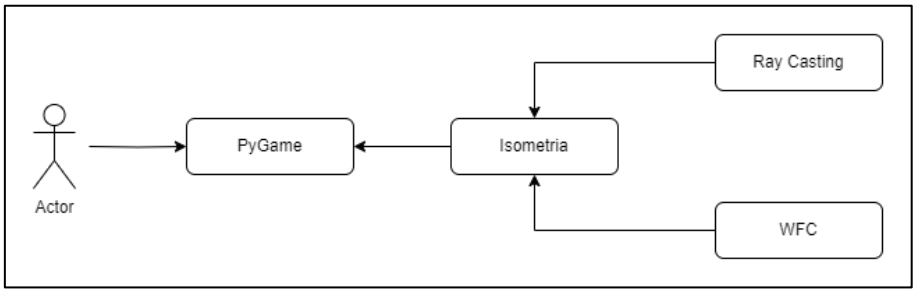
Figura 6 – Mapa gerado pelo algoritmo WFC em projeção isométrica



Fonte: elaborado pelo autor.

O protótipo se baseia na utilização de três partes principais: o *loop* principal, que roda o jogo e instancia a classe `IsometricGame`, esta responsável por gerenciar toda a seção de apresentação dos *sprites*, em conjunto com os cálculos necessários para definir o *tile* selecionado atualmente, *chunk* em que o usuário está e *chunks* visíveis, calculando por meio de uma constante quais *chunks* devem ser carregados em tela; a classe `WaveFunctionCollapse` que fica responsável por toda a lógica de gerar o ambiente, obtendo como parâmetro a quantidade de linhas e colunas que devem ser geradas; e responsável pela parte do cálculo do algoritmo de Ray Casting, foi desenvolvido a classe `RayCaster`, destinada a realizar o cálculo do polígono visível pelo usuário. O fluxo com que os algoritmos são executados pode ser observado na Figura 7, contando com as principais operações realizadas.

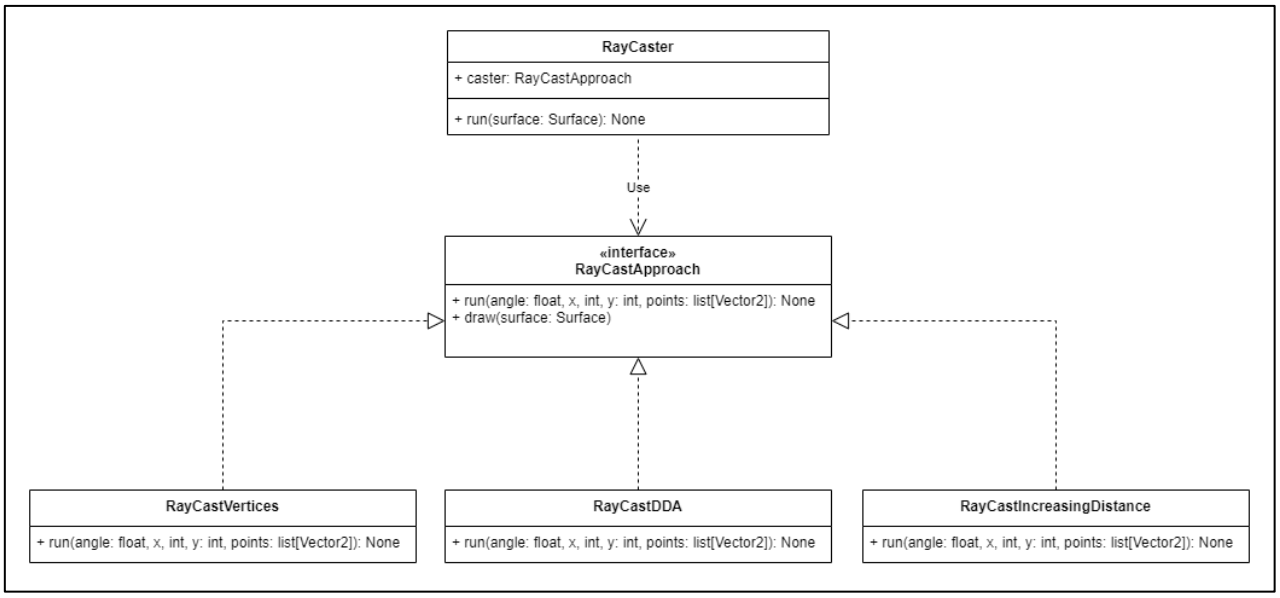
Figura 7 – Fluxo de funcionamento dos algoritmos



Fonte: elaborado pelo autor.

Sendo existente a possibilidade de aplicação de algoritmos distintos de Ray Casting, este trabalho conta com o desenvolvimento de uma interface responsável por obrigar os algoritmos a seguirem um padrão, para que sejam implementados sem dificuldades. Na Figura 8, a classe `RayCaster`, que recebe como parâmetro alguma classe que implementa a interface `RayCastApproach`, e é responsável por calcular o campo visível do jogador, recebendo os pontos dos obstáculos como parâmetro.

Figura 8 – Diagrama de classes dos algoritmos de Ray Casting



Fonte: elaborado pelo autor.

3.2 IMPLEMENTAÇÃO

Esta seção apresenta a implementação dos principais algoritmos utilizados na construção do protótipo, sendo eles o algoritmo WFC, que será abordado em detalhes na subseção 3.2.1, os algoritmos de Ray Casting, que serão abordados na subseção 3.2.2 e os cálculos envolvendo a projeção isométrica, que serão abordados na subseção 3.2.3. Toda implementação foi realizada na linguagem de programação Python utilizando a biblioteca Pygame. Sendo assim, esta seção abordará os temas mencionados acima, demonstrando na prática os principais pontos dos algoritmos necessários para o funcionamento deste protótipo. Grande parte da implementação utiliza um componente da biblioteca Pygame chamado `Vector2` que representa um vetor com valores `x` e `y`.

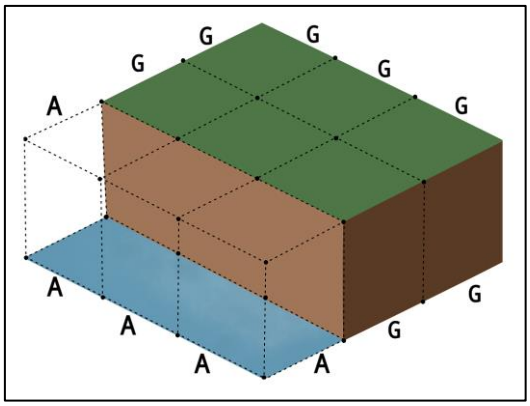
3.2.1 Wave Function Collapse

Para iniciar o algoritmo WFC, inicialmente é necessário definir a estrutura do código que será escrito. Neste caso, a estrutura se fez em três etapas principais, que serão abordadas a seguir. A subseção 3.2.1.1 deste artigo explicará como exatamente devem ser escritas as regras responsáveis pelo rearranjo dos *tiles* conforme o algoritmo progride; a subseção 3.2.1.2 demonstrará como aplicar o conceito de entropia e buscar sempre o caso que apresente menor entropia, a fim de buscar garantir que haja uma resposta definitiva para as restrições do algoritmo. Por último, a subseção 3.2.1.3 abordará o tema do *weighted choice* para possibilitar que sejam definidos pesos para cada uma das regras, para que caso haja mais de uma possível resposta, ao invés de escolher aleatoriamente um *tile* para que seja a solução daquela pergunta, um peso será incluído para favorecer algum deles, de modo que sejam gerados terrenos com maiores números de blocos "terra" do que de "água" por exemplo, ou vice-versa. O código completo pode ser encontrado no repositório de Reinke (2023) para consulta.

3.2.1.1 Regras

Para entender como as regras do algoritmo WFC funcionam temos como base a Figura 9. É necessário dividir todos os lados da *sprite* em três partes iguais (esse valor pode variar dependendo da necessidade do desenvolvedor). Partindo da primeira divisão feita na parte de cima da *sprite*, é feita a notação seguindo em sentido horário e resultando na sequência: "cima", "direita", "baixo" e "direita". É preciso lembrar que lados iguais devem conter a mesma notação, neste artigo iremos utilizar `G` para "grama", e `A` para "água".

Figura 9 – Divisão do *sprite* para definição das regras do WFC



Fonte: elaborado pelo autor.

Portanto, neste caso, ficaria a seguinte notação, na parte superior `GGG`, e seguindo no sentido horário, `GGA`, `AAA` e `AGG`. É necessário que sejam escritas regras para cada um dos *tiles* criados, sendo elas essenciais para que seja possível relacionar *tiles* entre si, desde que lados adjacentes tenham regras simétricas. Neste trabalho, foram usadas duas *sprites* diferentes, de diferentes alturas para dar um relevo maior ao mapa gerado. Ao criar uma regra, é importante ter em mente que todas as possibilidades devem ser concebidas, para que assim, seja gerado um terreno mais natural, sem repetições de padrões indesejados. No Quadro 4, podemos ver todas as regras que foram definidas neste trabalho, como mencionado anteriormente, são necessárias regras para todas as possibilidades possíveis de interação entre os *sprites*.

Quadro 4 – Regras desenvolvidas para o algoritmo WFC

```
self.file_lookup_table = {
    "water_bottom": ("GGG", "GGA", "AAA", "AGG"),
    "water_right": ("GGA", "AAA", "AGG", "GGG"),
    "water_left": ("AGG", "GGG", "GGA", "AAA"),
    "water_top": ("AAA", "AGG", "GGG", "GGA"),
    "water_top_right": ("AAA", "AAA", "AGG", "GGA"),
    "water_top_left": ("AAA", "AGG", "GGA", "AAA"),
    "water_bottom_right": ("GGA", "AAA", "AAA", "AGG"),
    "water_bottom_left": ("AGG", "GGA", "AAA", "AAA"),
    "grass_bottom_left": ("GGA", "AGG", "GGG", "GGG"),
    "grass_bottom_right": ("AGG", "GGG", "GGG", "GGA"),
    "grass_top_left": ("GGG", "GGA", "AGG", "GGG"),
    "grass_top_right": ("GGG", "GGG", "GGA", "AGG"),
    "grass_stone_bottom": ("GGG", "GGG", "PPP", "GGG"),
    "grass_stone_left": ("GGG", "GGG", "GGG", "PPP"),
    "grass_stone_right": ("GGG", "PPP", "GGG", "GGG"),
    "grass_stone_up": ("PPP", "GGG", "GGG", "GGG"),
    "grass": ("GGG", "GGG", "GGG", "GGG"),
    "water": ("AAA", "AAA", "AAA", "AAA"),
    "stone": ("PPP", "PPP", "PPP", "PPP"),
}
```

Fonte: elaborado pelo autor.

3.2.1.2 Entropia

Para Sandhu, Chen e McCoy (2019), a entropia é a probabilidade de um elemento ser escolhido, sendo um conceito similar ao conceito presente na física. A entropia é um fator importante que dirige as escolhas que o WFC faz enquanto itera sobre o espaço generativo. Sandhu, Chen e McCoy (2019) complementam que o desenvolvedor pode obter mais controle do espaço generativo adicionando restrições que manipulem a entropia. A entropia é definida pelo número de possibilidades presentes na escolha de um *tile* em específico. No início do algoritmo, todos os *tiles* possuem a mesma entropia, por conta de não haver nenhuma restrição até o momento, porém, após a primeira escolha, que geralmente é feita aleatoriamente, a entropia dos *tiles* em volta diminuem. Como mencionado anteriormente, os *tiles* são ordenados da menor entropia para a maior, sendo que a menor deve sempre ser escolhida, pois aumenta a chance de o algoritmo realizar todo o processo sem ter a necessidade de executar um *backtracking* para correção.

Neste trabalho a entropia é calculada passando como parâmetro uma posição `(x, y)` da matriz 2D, e dentro da função, é calculado para cada *tile* adjacente, as restrições que cada um apresenta e que influencia a escolha do *tile* `(x, y)`. Após aplicar todas as restrições, será gerada uma lista de possíveis *tiles* que podem ser utilizados naquele espaço,

sendo a entropia, a quantidade de possibilidades que foram calculadas. No Quadro 5, temos o exemplo de cálculo do lado direito.

Quadro 5 – Cálculo de entropia do lado direito de um *tile*

```
if (x + 1, y) in self.filled_set:
    ref = self.board[y][x + 1]
    if ref is not None:
        ref = ref.left[::-1]
        localset = {tile for tile in self.tiles if self.tiles[tile].right == ref}
else:
    localset = set(self.tiles)

outset &= localset
```

Fonte: elaborado pelo autor.

Como mencionado, este código somente verifica o *tile* adjacente à direita, ou seja, x + 1. Se já estiver populado, é tomada a regra daquele *tile* como referência, e por fim, verificado qual o lado esquerdo da regra desta referência (lado virado para o *tile* que está sendo calculado). Após essa verificação, todos os *tiles* que possuem esta mesma regra do lado direito são definidos como uma possível resposta e são adicionados ao resultado final. E em caso do *tile* à direita estar vazio, todas as opções podem ser válidas. Depois de realizar essa verificação em todos os lados, as respostas são adicionadas à uma lista utilizando o operador `AND` para que somente sejam armazenadas respostas que atendam os cálculos de todas as adjacências. Por fim, após todos os cálculos, é retornado o tamanho da lista gerada para aquelas coordenadas `(x, y)`, sendo assim, sua entropia.

3.2.1.3 Weighted Choice

Após ser calculada a entropia e selecionada a menor dentre elas, iremos ter uma lista de possíveis *tiles* que podem ocupar o *tile* que está sendo calculado. Aplicando um algoritmo simples de seleção aleatória em que seja levado em conta o peso, para isso, cada uma das regras criadas deve possuir um peso, sendo o valor, um número inteiro. Quando o algoritmo for escolher de fato um *tile* para ser adicionado à matriz principal, a seleção por *weighted choice* possibilita que seja gerado um mapa mais aleatório, sem padrões se repetindo por conta de dois principais itens:

a) a escolha não é completamente definida, não há obrigatoriedade de sempre escolher o *tile* com mais peso, por mais que um *tile* possa possuir muito mais influência, a escolha ainda permanece aleatória, porém com mais chances para um do que para outro.
b) a escolha não é completamente aleatória, utilizando uma escolha 100% aleatória, ou seja, sem pesos envolvidos, o mundo gerado se comporta de forma esquisita, não mantendo uma razão de mais grama do que água, por exemplo.

3.2.2 Ray Casting

O algoritmo de Ray Casting pode ser construído de várias formas. A forma a ser definida irá depender do resultado que o desenvolvedor espera e do ambiente em que ele será aplicado. Nesta seção, serão demonstrados três diferentes algoritmos que possuem a mesma finalidade, porém, com lógicas diferentes e para diferentes fins, dentre eles: Ray Casting por pixel, Ray Casting utilizando Digital Diference Analyzer (DDA) e Ray Casting utilizando os vértices dos obstáculos. Dos algoritmos mencionados, os dois primeiros somente funcionam em mapas com duas dimensões (2D), e no terceiro caso, é possível adaptá-lo para utilizar em qualquer ambiente, desde que seja possível obter a posição dos vértices dos obstáculos.

3.2.2.1 Ray Casting por Pixel

Este algoritmo leva como base a posição do jogador, o ângulo para qual ele está se direcionando e uma lista com todos os obstáculos (geralmente sendo valores inteiros em uma matriz 2D). Neste tipo de algoritmo, cada *ray* disparado do observador aumenta de pixel em pixel, e verifica se o ponto atual está colidindo com um bloco ou não. Neste trabalho, a função responsável pelo cálculo recebe os parâmetros de ângulo do campo de visão (Field Of View – FOV) do jogador e as coordenadas `x` e `y` do jogador. No Quadro 6 podemos ver o principal cálculo realizado para verificar se o *ray* encontrou ou não uma parede.

Quadro 6 – Cálculo da posição do ponto alvo no algoritmo de Ray Casting

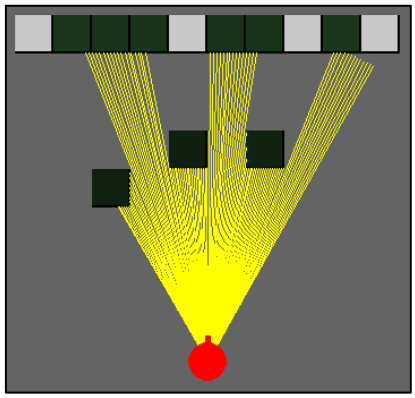
```
start_angle = player_angle - (self.fov / 2)

for ray in range(self.rays_to_cast):
    target_sin = math.sin(start_angle)
    target_cos = math.cos(start_angle)

    for depth in range(self.max_depth):
        target = Vector2(player_x - target_sin * depth, player_y + target_cos * depth)
```

Fonte: elaborado pelo autor.

Inicialmente é realizada uma repetição dependendo da quantidade de *rays* que serão geradas. Após isso, o seno e cosseno da direção que a *ray* irá tomar é calculado utilizando a biblioteca própria do Python. Posteriormente, teremos outra repetição, que irá de 0 até o tamanho máximo que as *rays* podem chegar, e em cada repetição, um ponto é definido levando em conta a direção da *ray* e o tamanho que a *ray* terá. Isso se repete até que seja encontrado algum bloco, ou até que a *ray* chegue em seu comprimento máximo, onde o *loop* é quebrado.

Para verificação de colisão com paredes, é possível traduzir as coordenadas em tela para coordenadas dentro da matriz, e assim, verificando que valor se encontra naquela posição. Neste trabalho, ao encontrar uma parede, o bloco é colorido com a cor verde para melhor visualização das colisões. Na Figura 10 é possível verificar o resultado deste algoritmo utilizando 120 *rays* em um FOV de $\pi/3$.

Figura 10 – Demonstração do algoritmo de Ray Casting percorrendo pixel por pixel



Fonte: elaborado pelo autor.

A performance deste algoritmo é baseada na quantidade de *rays* lançados e no comprimento de cada um deles, sendo possível diminuir a quantidade de *rays* para melhorar drasticamente a performance.

3.2.2.2 Ray Casting utilizando DDA

Este algoritmo tem como base os mesmos parâmetros do algoritmo anterior, porém os cálculos são realizados de uma forma diferente, utilizando o algoritmo Digital Difference Analyzer (DDA). Este algoritmo realiza os cálculos de colisão a cada determinada distância, chamado de *stepsize*, ao invés de realizar em todos os pixels, incrementando assim, a performance dos cálculos. No Quadro 7 são apresentados os principais cálculos que serão realizados nessa operação.

Quadro 7 – Cálculo da normalização e do *stepsize*

```
player_in_map = Vector2(player_x // self.map.tile_size, player_y // self.map.tile_size)

normalized_direction = (target - player).normalize()
normalized_direction.x -= 0.0000001
normalized_direction.y -= 0.0000001

ray_stepsize = Vector2(
    math.sqrt(1 + math.pow((normalized_direction.y / normalized_direction.x), 2)),
    math.sqrt(1 + math.pow((normalized_direction.x / normalized_direction.y), 2))
)
```

Fonte: elaborado pelo autor.

Inicialmente é calculado a direção que o *ray* estará apontado, da mesma forma que o algoritmo anterior, é possível realizar isso calculando para `x` $player_x - \sin angle * depth$ e para `y` $player_y - \cos angle * depth$. Após o cálculo da direção do ponto objetivo, será necessário normalizar a direção, que neste trabalho, foi feito utilizando a função `normalize` do Pygame, que coloca os valores de `x` e `y` do vetor, em valores de 0 até 1. É possível que este resultado seja

igual a 0, então é necessário tratar esse possível erro, por `Exception` ou diminuindo/acrescentando um valor que não fará diferença no cálculo.

Após o cálculo da direção normalizada, iremos calcular os valores referentes ao tamanho de cada *step* que deverá ser tomado, tanto no eixo `x` quanto no eixo `y`. Serão retomadas as equações `(1)` e `(2)` que foram mencionadas anteriormente, substituindo `d` por `normalized_direction`, resultando em um vetor com um valor de *step* para `x` e para `y`. Será necessário verificar se os valores `x` e `y` são positivos ou negativos, para definir em qual direção os cálculos irão se dirigir. No Quadro 8, são realizadas as validações de direção, verificando se marcham em direções positivas ou negativas no plano cartesiano, e o *loop* para avançar para as próximas validações.

Quadro 8 – Cálculo da posição do ponto alvo no algoritmo de Ray Casting

```
if normalized_direction.x < 0:
    step.x = -1
    ray_length.x = (ray_start.x - map_position.x) * ray_stepsize.x
else:
    step.x = 1
    ray_length.x = ((map_position.x + 1) - ray_start.x) * ray_stepsize.x

if normalized_direction.y < 0:
    step.y = -1
    ray_length.y = (ray_start.y - map_position.y) * ray_stepsize.y
else:
    step.y = 1
    ray_length.y = ((map_position.y + 1) - ray_start.y) * ray_stepsize.y

found_block = False
distance = 0

# Algoritmo DDA
while (distance * self.map.tile_size) < self.max_depth - 50:
    if ray_length.x < ray_length.y:
        map_position.x += step.x
        distance = ray_length.x
        ray_length.x += ray_stepsize.x
    else:
        map_position.y += step.y
        distance = ray_length.y
        ray_length.y += ray_stepsize.y

    if map_position.x >= 0 and map_position.x < number_rows and
       map_position.y >= 0 and map_position.y < number_cols):
        if self.map.environment[int(map_position.y)][int(map_position.x)] == 1:
            intersection_point = ray_start + normalized_direction * distance
            found_block = True
            break
```

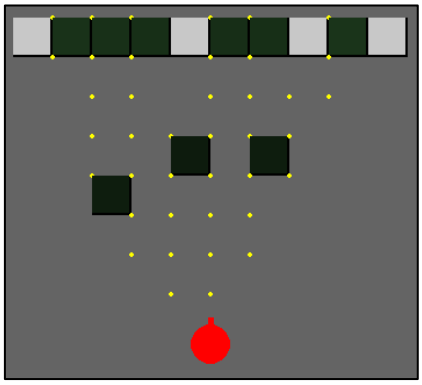
Fonte: elaborado pelo autor.

Ao validar se a direção do vetor normalizado é positiva ou negativa, caso o vetor tenha um componente `x` negativo, o *step* será igual a -1, e em caso positivo, +1, o mesmo serve para o componente `y`. Caso o componente `x` seja negativo, a distância percorrida no eixo `x` será igual a distância da posição inicial, até a primeira coluna do *grid* à esquerda, caso possua o componente `x` positivo, a primeira coluna do *grid* à direita. Da mesma forma para o componente `y`, sendo calculado para cima ou para baixo. A posição do mapa que está sendo validada é utilizada e a posição de início do *ray* será subtraído, e há casos em que o valor 1 é adicionado por conta da direção ser negativa tanto no componente `x` quanto no y. Após isso, será obtido a distância perpendicular ao lado do *grid*, e então, realizada a multiplicação por `ray_stepsize.x` ou `ray_stepsize.y` para obter a distância Euclidiana inicial definitiva.

Agora o algoritmo DDA começa definitivamente, em suma é um *loop* que irá incrementar o *ray* com uma unidade de `stepsize` em cada repetição, até encontrar uma parede, ou no caso deste trabalho, também chegar à distância máxima definida para cada *ray*. Em cada repetição, o algoritmo irá pular uma unidade de `stepsize` no eixo `x` ou uma unidade de `stepsize` no eixo `y`. O menor valor entre os componentes `x` e `y` da variável `ray_length`, que armazena o valor da distância entre o ponto que está sendo validado até a próxima linha do *grid* cartesiano, tanto no eixo `x` quanto no `y`, é escolhida, e após isso, a posição validada irá ser incrementada. Por fim, a variável `ray_length` recebe o valor da próxima distância a ser percorrida naquele eixo. Caso a validação daquela posição na matriz seja verdadeira, um bloco foi atingido.

Na Figura 11, é demonstrado o resultado do algoritmo mencionado acima, lançando 120 *rays* em um FOV de $\pi/3$, onde a cada *loop* do DDA é desenhado uma bolinha de cor amarela, representando o ponto que está sendo calculado, e em caso de colisão com uma parede, a parede é colorida com a cor verde.

Figura 11 – Demonstração do algoritmo de Ray Casting utilizando DDA



Fonte: elaborado pelo autor.

3.2.2.3 Ray Casting utilizando Vértices

Este algoritmo, por sua vez, funciona de uma forma diferente dos algoritmos apresentados anteriormente. Seu funcionamento é baseado em uma lista contendo os vértices de cada obstáculo presente no ambiente, onde seu objetivo, além de permitir que sejam identificados os pontos de colisão, é também criar um polígono representando a visão do usuário. O algoritmo se baseia em definir todos os vértices em uma lista, e a partir desta, lançar três *rays* se baseando no ângulo calculado utilizando o arco tangente de 2 argumentos (`atan2`). Sendo uma diretamente ao ponto, outra em um ângulo 0,0001 à esquerda e outra em um ângulo 0,0001 à direita, para que o *ray* possa continuar para um possível obstáculo atrás deste que foi testado (Quadro 9).

Quadro 9 – Definição dos ângulos que serão calculados posteriormente

```
unique_angles: list[float] = []
for point in unique_points:
    angle = math.atan2(point[0].y - player_y, point[0].x - player_x)
    unique_angles.append(angle - 0.00001)
    unique_angles.append(angle)
    unique_angles.append(angle + 0.00001)
```

Fonte: elaborado pelo autor.

Após a separação de todos os ângulos que serão calculados é necessário buscar se a *ray* lançada em determinado ângulo, partindo da posição do usuário, possuirá intersecção com algum segmento de reta criado pelos vértices presentes no ambiente. A função em si recebe como parâmetros, um *ray*, sendo ele composto por sua origem, que será a posição do usuário, e por sua direção, calculada utilizando a função de seno e cosseno no ângulo calculado anteriormente, recebendo também como parâmetro, um segmento de reta, composto por dois pontos de algum polígono.

Para buscar a intersecção mais próxima entre a *ray* e os segmentos, é necessário transformar ambos em suas formas paramétricas, sendo a fórmula $Point + Direction \cdot T$. Nos resultando em quatro equações, representando o componente `x` e `y`, de ambos *ray* e segmento. Antes de continuar os cálculos, é necessário que seja validado que o *ray* e o segmento não são paralelos, pois, em caso afirmativo, não haverá intersecção em nenhum ponto. O cálculo necessário para verificação de paralelismo é mostrado no Quadro 10.

Quadro 10 – Verificação de paralelismo entre duas retas

```
r_mag = math.sqrt(r_dx * r_dx + r_dy * r_dy)
s_mag = math.sqrt(s_dx * s_dx + s_dy * s_dy)
if r_dx / r_mag == s_dx / s_mag and r_dy / r_mag == s_dy / s_mag:
    return None
```

Fonte: elaborado pelo autor.

Caso as retas não sejam paralelas, é possível que haja uma intersecção, e caso se intersectem, os componentes `x` e `y` serão iguais, nos deixando as paramétricas do *ray* e do segmento de reta, de acordo com Case (2014), resultando fórmulas `(3)` e `(4)`

$$r\_px + r\_dx * T1 = s\_px + s\_dx * T2 \quad (3) \qquad r\_py + r\_dy * T1 = s\_py + s\_dy * T2 \quad (4)$$

Com isso, é necessário isolar primeiramente uma variável (`T1`), e após obter o resultado da equação, calcular o valor da outra variável (`T2`) substituindo a variável restante com o valor obtido no cálculo anterior. É necessário realizar

uma tratativa que impeça que o valor do divisor seja igual a zero, por conta de ser impossível dividir qualquer valor por zero. Neste trabalho, esta tratativa foi criada utilizando a `Exception ZeroDivisionError`, padrão da linguagem Python. Por último, é necessário validar se $T1 > 0$ e que $0 < T2 < 1$, pois caso contrário, a intersecção não está dentro dos limites definidos do segmento, logo, não deverá haver uma intersecção. O cálculo desta etapa pode ser observado no Quadro 11, que demonstra todos os cálculos necessários para verificação de intersecção entre o *ray* e o segmento de reta.

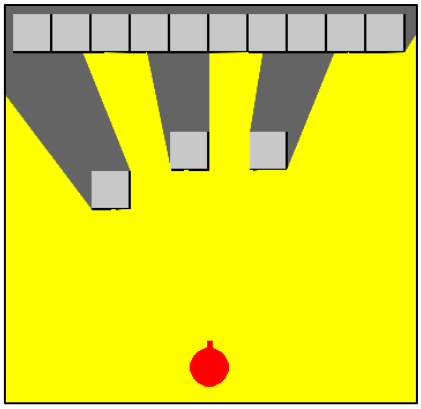
Quadro 11 – Cálculo referente à existência ou não de uma intersecção

```
try:
    T2 = (r_dx * (s_py - r_py) + r_dy * (r_px - s_px)) / (s_dx * r_dy - s_dy * r_dx)
except ZeroDivisionError:
    T2 = (r_dx * (s_py - r_py) + r_dy * (r_px - s_px)) / (s_dx * r_dy - s_dy * r_dx -
0.00001)

try:
    T1 = (s_px + s_dx * T2 - r_px) / r_dx
except ZeroDivisionError:
    T1 = (s_px + s_dx * T2 - r_px) / (r_dx - 0.00001)

if T1 < 0: return None
if T2 < 0 or T2 > 1: return None

return {
    "x": r_px + r_dx * T1,
    "y": r_py + r_dy * T1,
    "param": T1
}
```

Fonte: elaborado pelo autor.

Esse cálculo deverá ser realizado para cada ângulo definido anteriormente, e para cada ângulo, deverá ser checada a intersecção com todos os segmentos presentes em tela. Por fim, é possível juntar todos os pontos de intersecção, e renderizar um polígono passando por todos estes pontos, o resultado é demonstrado na Figura 12.

Figura 12 – Demonstração do algoritmo de Ray Casting utilizando Vértices



Fonte: elaborado pelo autor.

3.2.3 Projeção Isométrica

Ao projetar isometricamente, são necessários alguns cuidados. Este trabalho usou como base o algoritmo WFC para gerar uma matriz 2D e posteriormente plotar esses dados como *sprites* em uma tela utilizando PyGame. Esta seção será subdividida em três subseções que serão abordadas a seguir. A subseção 3.2.3.1 deste trabalho abordará as técnicas e equações responsáveis por plotar os *tiles*. A subseção 3.2.3.2 dissertará sobre como é possível definir em qual *tile* um ponto na tela se encontra, evitando o problema de ser impossível selecionar alguns *tiles* pela tela. E por último, a subseção 3.2.3.3 que demonstrará formas mais rápidas de renderizar todos os *sprites* necessários.

3.2.3.1 Renderização de *tiles*

Conforme mencionado anteriormente, há algumas regras que devem ser seguidas ao renderizar os *tiles* na tela, por exemplo, ordenar todos os objetos pela posição `y` antes de renderizá-los na tela, para que nenhum sobreponha o outro. De forma sucinta, serão utilizadas duas principais equações, para que as coordenadas da matriz 2D geradas pelo algoritmo WFC estejam em harmonia ao serem organizadas em tela, de acordo com Barr (2019) é necessário que sejam aplicadas as funções `(5)` e `(6)` para a posição `x` e `y` respectivamente.

$$Sx \leftarrow (x - y) \cdot \frac{Tw}{2} \quad (5) \qquad Sy \leftarrow (x + y) \cdot \frac{Th}{2} \quad (6)$$

Desse modo, é necessário levar em consideração o tamanho do *tile*, o *offset* em que a tela se encontra e o vetor de origem. No Quadro 12 demonstra as funções responsáveis por transformar as coordenadas da tela para matriz e vice-versa.

Quadro 12 – Funções responsáveis por transformar coordenadas

```
def to_screen_coordinates(x: int, y: int, offset: Vector2):
    return Vector2(
         (VECTOR_ORIGIN.x*VECTOR_TILESIZE.x)+(x-y)*(VECTOR_TILESIZE.x/2)+offset.x,
         (VECTOR_ORIGIN.y*VECTOR_TILESIZE.y)+(x+y)*(VECTOR_TILESIZE.y/2)+offset.y,
    )

def to_list_coordinates(x: int, y: int):
    return Vector2(
        int((y - VECTOR_ORIGIN.y) + (x - VECTOR_ORIGIN.x)),
        int((y - VECTOR_ORIGIN.y) - (x - VECTOR_ORIGIN.x))
    )
```

Fonte: elaborado pelo autor.

A princípio é definida uma constante `VECTOR_ORIGIN` que será responsável apenas por mover o *tile* inicial do nosso terreno, para que o mesmo não seja gerado muito à esquerda da tela (definido como $x = 3$ e $y = 1$). A constante `VECTOR_TILESIZE` irá definir o tamanho geral de cada *tile* que será plotado (definido como $x = 30$ e $y = 15$), como as *sprites* são todas baseadas em uma largura sendo o dobro da altura, esta razão deve ser mantida na hora de definir esta variável. A variável `offset` armazena o valor do *offset*, para que seja possível "andar" pelo terreno, basicamente movendo o lugar de renderização para o lado oposto que o usuário se mover, dando a impressão de movimento.

3.2.3.2 Seleção de *tiles*

Como mencionado anteriormente, há um problema que ocorre por conta de a projeção isométrica não ser retangular, consequentemente ocasionando o funcionamento incorreto da seleção de *tiles*. Ao pensar que todos os *tiles* devem ser transformados em retângulos para realizar este cálculo, conforme a Figura 5, a abordagem utilizada neste trabalho foi realizar cálculos para verificar se o ponto em que o mouse se encontra, está dentro ou fora de algum dos quatro triângulos retângulos gerados a partir dessa retangularização dos *tiles*. O Quadro 13 demonstra a implementação do algoritmo responsável por verificar se um determinado ponto `point` está dentro de um triângulo formado pelos vértices `vertice_a`, `vertice_b` e `vertice_c`.

Quadro 13 – Funções responsáveis por transformar coordenadas

```
def sign(p1: Vector2, p2: Vector2, p3: Vector2):
    return (p1.x - p3.x) * (p2.y - p3.y) - (p2.x - p3.x) * (p1.y - p3.y)

def is_point_inside(point: Vector2, vertice_a: Vector2, vertice_b: Vector2, vertice_c:
Vector2):

    d1 = sign(point, vertice_a, vertice_b)
    d2 = sign(point, vertice_b, vertice_c)
    d3 = sign(point, vertice_c, vertice_a)

    has_neg = (d1 < 0) or (d2 < 0) or (d3 < 0)
    has_pos = (d1 > 0) or (d2 > 0) or (d3 > 0)

    return not (has_neg and has_pos)
```

Fonte: elaborado pelo autor.

Ao utilizar a função acima, é possível verificar se o mouse está dentro de algum dos quatro triângulos retângulos, e em caso positivo, realizar a operação necessária para movimentar os índices da matriz corretamente para o espaço selecionado. As coordenadas de cada triângulo irão se basear inteiramente em obter a mediana de cada aresta do retângulo selecionado, e juntá-las com os seus respectivos vértices.

Após cada uma das verificações, em caso positivo, é necessário realizar uma operação de subtração ou adição à variável que armazena o valor do *tile* selecionado. Caso esteja dentro do triângulo superior esquerdo, a posição `x` deve ser diminuída em 1; caso esteja dentro do triângulo inferior esquerdo, a posição `y` deve ser acrescentada em 1; caso esteja dentro do triângulo superior direito, a posição `y` deve ser diminuída em 1; caso esteja dentro do triângulo inferior direito, a posição `x` deve ser acrescentada em 1; e caso não esteja dentro de nenhum triângulo, o *tile* correto já está selecionado.

3.2.3.3 Renderização em *chunks*

Para tornar a renderização mais eficiente, é possível renderizar uma porção do terreno gerado ao invés de renderizá-lo inteiramente. Normalmente ao construir uma matriz 2D para armazenar os dados de cada *tile*, é criada uma gigante lista de listas, representando linhas e colunas, contendo todos os dados. Neste trabalho, entretanto, foi adotada uma perspectiva de renderização por *chunk*, que seriam porções do terreno geral. A matriz 2D se tornou um dicionário chave valor, sendo a chave, as coordenadas do *chunk* divididas por vírgula, e o valor, uma matriz 2D de tamanho igual para todos os *chunks*, contendo os dados somente daquela porção de terreno.

Ao criar os dados a partir do algoritmo WFC, uma constante `CHUNK_SIZE` é definida e representa a quantidade de linhas e colunas que serão armazenadas em cada *chunk*. Após isso, a matriz 2D retornada do algoritmo WFC é recortada em vários blocos de `CHUNK_SIZE x CHUNK_SIZE` e são armazenados no novo dicionário que representa aquele mundo, utilizando as posições dos *chunks* como chave, para que possa ser acessado posteriormente. O Quadro 14 representa a porção de código responsável por ler a saída do algoritmo WFC e transformá-lo em um dicionário de *chunks*

Quadro 14 – Código principal responsável pela criação dos *chunks*

```
map_array = wave_function.run(size_x, size_y)

self.col_nums = len(map_array[0])
self.row_nums = len(map_array)

for row in range(self.row_nums):
    for col in range(self.col_nums):
        vector_world = to_screen_coordinates(col, row, self.offset)

        chunk_x = col // self.CHUNK_SIZE
        chunk_y = row // self.CHUNK_SIZE
        target_chunk = f'{chunk_x},{chunk_y}'

        if target_chunk not in self.map_dict:
            self.map_dict[target_chunk] = self.create_empty_chunk()

        self.map_dict[target_chunk][row % self.CHUNK_SIZE].append((image, vector_world))

def create_empty_chunk(self):
    return [[] * self.CHUNK_SIZE for i in range(self.CHUNK_SIZE)]
```

Fonte: elaborado pelo autor.

Ao adicionar todos os *tiles* gerados em um dicionário, neste trabalho foi criada uma constante `CHUNK_RENDER_DISTANCE` que fica responsável por armazenar o valor de *chunks* que irão ser carregados em cada uma das direções do *chunk* em que o usuário está. A função responsável pela seleção de quais *chunks* devem ser exibidos, recebe como parâmetros o valor `x` e `y` do *chunk* em que o usuário se encontra e a distância de renderização, para assim, ter uma base de qual é o central e quão longe será necessário buscar os demais. Como as chaves salvas no dicionário representam valores positivos e crescentes, sendo eles a posição de cada um dos *chunks*, é realizado um simples *loop* para obter todos os adjacentes em um determinado raio, como é possível ver no Quadro 15.

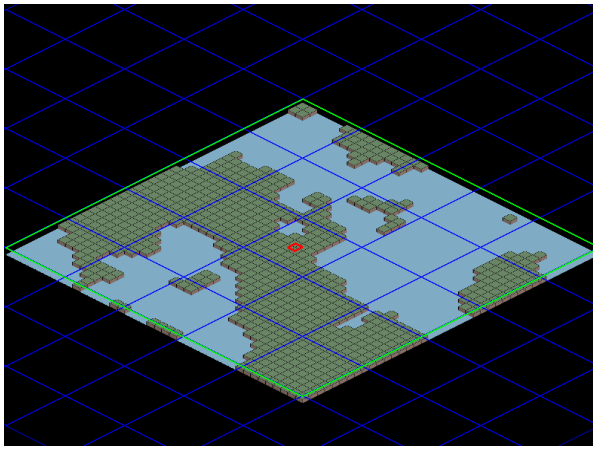
Quadro 15 – Código responsável por obter todos os *chunks* que devem ser renderizados

```
def get_rendered_chunks(x_chunk: int, y_chunk: int, distance: int):
      response = []

      for y in range(int(y_chunk) - distance, int(y_chunk) + distance + 1):
            for x in range(int(x_chunk) - distance, int(x_chunk) + distance + 1):
                  response.append(f'{x},{y}')

      return response
```

Fonte: elaborado pelo autor.

Após a definição dos *chunks* que precisam ser renderizados, somente é necessário procurar as chaves retornadas pela função `get_rendered_chunks` no dicionário do mapa, e renderizar normalmente a lista com linhas e colunas contidas como valor naquela posição. A realização desta etapa aumenta consideravelmente a performance da plotagem, permitindo que o código se preocupe com outras operações, não sendo necessário renderizar *tiles* que não se encontram na tela. Conforme a Figura 13, é possível notar todos os *chunks* tracejados em azul e os *chunks* visíveis tracejados com verde. A renderização só é feita conforme o valor definido para `CHUNK_RENDER_DISTANCE`, possibilitando assim, o aumento de performance.

Figura 13 – Demonstração da renderização por *chunks*



Fonte: elaborado pelo autor.

4 RESULTADOS

Esta seção apresenta os resultados que foram alcançados a partir do trabalho desenvolvido, focando nos algoritmos WFC e Ray Casting, e para facilitar a leitura dos resultados, as tabelas e gráficos presentes neste capítulo irão considerar apenas a média geral dos dados analisados. Para a análise do algoritmo de WFC foram gerados mapas de diferentes tamanhos, a fim de comparar a velocidade com que são gerados e por fim verificar até onde o algoritmo se mostra eficiente. Para a análise dos algoritmos de Ray Casting, foram utilizados a quantidade de *rays* e o comprimentos dos *rays* como parâmetro, sendo estes alterados em cada etapa dos testes. Na subseção 4.1 é analisado o algoritmo de WFC, apresentando os resultados obtidos, e em seguida, na subseção 4.2, são analisados os três algoritmos de Ray Casting mencionados na seção 3.2.2 e apresentados os resultados obtidos.

4.1 WAVE FUNCTION COLLAPSE

Para a análise de performance do algoritmo WFC, foram alterados apenas os parâmetros de entrada do algoritmo, sendo eles, o número de colunas e linhas que serão geradas a partir do algoritmo. Para possibilitar que uma análise seja feita partindo de diferentes tamanhos de mapa, o algoritmo foi aplicado à 14 regras para os *tiles* e 14 pesos para cada uma das regras e dimensões de 5x5, 10x10, 25x25, 50x50, 75x75 e 100x100. Todos os testes foram realizados em um computador Windows 10 Home 64 bits com processador i5-1135G7 e 8 GB de RAM.

A Tabela 1 apresenta a média de tempo em milissegundos (ms) das 50 execuções do algoritmo WFC em cada uma das dimensões mencionadas anteriormente. Também é exibida a quantidade de *tiles* gerados por interação, sendo assim, é possível perceber que o tempo de execução aumenta drasticamente em relação à quantidade de *tiles* gerados, sendo o principal problema enfrentado no desenvolvimento deste trabalho. Com base no algoritmo implementado, o maior empecilho ocorre na função responsável por retornar o *set* com os *tiles* vizinhos válidos de cada um dos *tiles* para poder calcular a entropia posteriormente, um estudo posterior seria necessário para verificar se é possível remover esse problema e manter a proporção de quantidade de *tiles* gerados e tempo de execução estável.

Tabela 1 – Médias em milissegundos de 50 execuções do algoritmo WFC

Dimensões	Qtd. *Tiles*	ms
5x5	25	0,8
10x10	100	7,2
25x25	625	223,4
50x50	2.500	3.443,2
75x75	5.625	15.111,0
100x100	10.000	119.395,9

Fonte: elaborado pelo autor.

A partir destes resultados observa-se que o tempo de execução aumenta exponencialmente em relação à quantidade de *tiles* gerados pelo algoritmo. Por conta do algoritmo necessitar que o resultado seja armazenado em uma matriz e seja reprocessado diversas vezes, o tempo de execução piora conforme a quantidade de *tiles* que necessita ser armazenada. Sendo assim, é possível a utilização do algoritmo para geração de terrenos, porém são necessárias algumas tratativas, como por exemplo, gerar o terreno em pequenas partições e juntar o resultado das diferentes execuções, para assim, dividir a construção de um terreno maior em várias etapas, realizando os cálculos em menos tempo.

4.2 RAY CASTING

Para a análise de performance dos algoritmos de Ray Casting, foram alterados os parâmetros envolvendo os *rays*, como comprimento, quantidade e FOV. Por conta do algoritmo envolvendo o cálculo baseado em vértices não possuir parâmetros como os mencionados anteriormente, os algoritmos serão divididos em duas categorias, e terão testes diferentes para verificar as performances de cada um. Os algoritmos envolvendo DDA e pixel serão testados variando os parâmetros envolvendo as alterações nas *rays* para saber até que ponto é mais vantajoso usar algum deles.

A Tabela 2 apresenta a média de tempo em milissegundos (ms) das 50 execuções do algoritmo de Ray Casting por pixel e por DDA, os parâmetros utilizados para realizar os testes foram, a quantidade de *rays* 150, 100, 50 e a distância máxima que cada *ray* pode chegar, alternando entre 450 e 300. Tanto os obstáculos, como os movimentos realizados, foram realizados igualmente nos dois casos, para evitar que os resultados dos testes possuam discrepâncias que não envolvam os parâmetros.

Tabela 2 – Médias em milissegundos de 50 execuções de dois algoritmos de Ray Casting

Algoritmo	Qtd *rays*	Distância máxima	ms
Pixel	150	300	31,8
	150	450	51,3
	100	300	28,5
	100	450	38,8
	50	300	13,8
	50	450	17,5
DDA	150	300	2,4
	150	450	5,1
	100	300	2,2
	100	450	3,5
	50	300	1,3
	50	450	1,6

Fonte: elaborado pelo autor.

Ao analisar os resultados, é possível afirmar que a abordagem utilizando o algoritmo DDA tem uma performance muito superior à abordagem que utiliza sua marcha baseada em pixels. Isso acontece por conta da quantidade de validações que ambos os algoritmos fazem, sendo o DDA, responsável por calcular apenas os pontos que intersectam com as colunas e linhas do plano cartesiano, enquanto a abordagem por pixels realiza o mesmo cálculo para cada pixel presente em sua direção, até que atinja algum ponto. Portanto, caso seja necessário a utilização de um algoritmo Ray Casting que precise calcular pontos específicos em uma matriz 2D, que utiliza como base o plano cartesiano, a utilização de DDA é imprescindível para um aumento na performance da execução do Ray Casting.

Em relação ao algoritmo envolvendo vértices, os testes realizados foram feitos alternando a quantidade de vértices presentes no cenário, para descobrir então, com até quantos vértices é vantajoso utilizar esse formato de algoritmo na execução de um algoritmo de Ray Casting. A Tabela 3 apresenta a média de tempo em milissegundos (ms) de 50 execuções do algoritmo de Ray Casting por vértices, sendo o único parâmetro validado, a quantidade de vértices calculados, sendo eles os valores de 20, 40, 80, 120 e 160.

Tabela 3 – Médias em milissegundos de 50 execuções do algoritmo de Ray Casting por vértices

Qtd vértices	ms
20	2,5
40	6,6
80	21,0
120	45,2
160	86,0

Fonte: elaborado pelo autor.

A utilização da abordagem envolvendo vértices se baseia na quantidade de vértices que estão presentes no ambiente, sendo assim, caso o ambiente possua apenas pontos que coincidem com o plano cartesiano, a abordagem envolvendo o algoritmo DDA continua sendo a mais proveitosa. Porém, caso os vértices estejam em pontos aleatórios (sem estarem necessariamente em cima de alguma linha ou coluna do plano cartesiano), este algoritmo se mostra vantajoso ao invés de utilizar a abordagem por pixel. Ainda assim, é necessário avaliar a quantidade de vértices que serão calculados no terreno, portanto, não sendo uma solução que possa ser utilizada em qualquer ambiente. Por conta de o tempo de execução aumentar conforme a quantidade de pontos a serem calculados, é necessária uma tratativa mais aprofundada ao utilizar esse algoritmo, sendo ela limitando a quantidade de vértices presentes, ou aplicando algoritmos externos para obter apenas os pontos extremamente necessários para o cálculo do Ray Casting utilizando essa abordagem.

5 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um protótipo para demonstração da geração de mundos isométricos com o algoritmo de Wave Function Collapse (WFC), juntamente com o algoritmo de Ray Casting para possibilitar a definição do campo de visão do usuário (FOV). As principais implementações do algoritmo de Ray Casting foram a tratativa por pixel, utilizando *digital differential analyzer* e por vértices. O algoritmo WFC construído, utilizou uma ordenação por entropia, e após isso, um *weighted choice* para definir a melhor escolha para o *tile* a ser definido. Todavia, a abordagem de Ray Casting utilizada foi por vértices, por conta de ser a mais apropriada para construir um campo de visão em si e não ter a necessidade de calcular quais paredes estão sendo atingidas ou não. Na projeção isométrica, foi utilizado o estilo *diamond* por ser um dos modelos mais comuns e ser simples de utilizar seu sistema de coordenadas. A utilização da linguagem de programação Python utilizando Pygame, se deu por conta da facilidade no desenvolvimento de aspectos gerais do protótipo, não sendo necessário um estudo muito aprofundado em uma nova linguagem ou biblioteca.

Em relação aos aspectos de performance do protótipo, os algoritmos alcançaram os objetivos desejados, sendo alguns mais eficientes que outros em determinados ambientes. Considerando os algoritmos de Ray Casting, o mais eficiente para percorrer um *grid* 2D definitivamente é a abordagem que utiliza o algoritmo de DDA, sendo em média 13x mais performático do que a abordagem calculando pixel por pixel. O algoritmo que utiliza a abordagem fundamentada em vértices, por sua vez, possui sua performance embasada na quantidade de objetos que terão que ser calculados, não possuindo uma comparação direta. Caso o objetivo do trabalho seja utilizar o algoritmo de Ray Casting para transformar o 2D em uma projeção 3D, de acordo com Vandevenne (2004), o algoritmo utilizando DDA é muito mais performático, por conta das poucas validações necessárias de serem feitas.

O algoritmo WFC por outro lado, possui resultados promissores para a geração apenas de mapas menores como demonstrado na Tabela 1. Pois, como o tempo necessário para concluir o algoritmo aumenta exponencialmente conforme a quantidade de *tiles* gerados, a geração de mapas muitos grandes, em uma primeira análise, não traz resultados muitos animadores. Isso acontece por conta da matriz responsável por guardar a resposta do algoritmo ser atualizada diversas vezes, trazendo um déficit de performance da operação.

Referente a naturalidade com que os terrenos foram gerados, pode-se concluir que a naturalidade em si, dependerá diretamente das regras aplicadas dentro da lógica do algoritmo WFC. Conforme mais regras são aplicadas, a aleatoriedade do algoritmo diminui, aumentando as chances de que o ambiente gerado possua uma maior uniformidade entre as transições de diferentes blocos. A inclusão de um *weighted choice* dentro do algoritmo WFC para que sejam selecionados apenas os melhores *tiles* para o caso sendo analisado trouxe melhorias significativas aos resultados, resultando em terrenos bem menos aleatórios e mais homogêneos em relação à escolha aleatória. Mesmo com a adição de uma escolha baseada em pesos, a alteração não ocasionou em uma perda significativa de performance.

Por mais que existam aplicações do algoritmo WFC e Ray Casting em diferentes áreas e estilos de jogos, suas aplicações em jogos isométricos tendem a não ser uma abordagem recorrente, sendo que de todos os autores mencionados neste trabalho, nenhum deles chegou a apresentar a junção destes algoritmos para gerar uma projeção isométrica. Nestes casos, os algoritmos são usados individualmente para diversas aplicações, não apresentando dados precisos como tempo de execução por quantidade de *rays* para o algoritmo de Ray Casting e tempo de execução por quantidade de *tiles* para o algoritmo de WFC. Neste aspecto, mesmo que o trabalho desenvolvido não tenha atingido resultados promissores em todos os casos, acredita-se que a análise comparativa entre as diferentes abordagens dos algoritmos possa servir como base e incentivo para novas pesquisas nestas áreas, como por exemplo, a aplicação dos algoritmos em projeções 2D ou até mesmo 3D, melhoria na aplicação dos algoritmos já utilizados, ou até mesmo a comprovação ou não de suas viabilidades neste contexto.

A partir dos temas acima, levantam-se algumas extensões para expandir este trabalho, sendo estas:

a) utilizar diferentes abordagens de cálculo para o algoritmo de Ray Casting;
b) criar um algoritmo para reduzir a quantidade de pontos calculados pela abordagem do Ray Casting que utiliza vértices, realizando uma espécie de `convex-hull` nos pontos para obter apenas os pontos exteriores de cada figura;
c) incluir diferentes camadas de solo à geração do algoritmo de WFC, possibilitando que sejam gerados terrenos isométricos com diferentes alturas;
d) utilizar os algoritmos de Ray Casting e WFC para a geração de ambientes com projeções diferentes da isométrica, sendo elas 2D ou até mesmo 3D, como descrito por Vandevenne (2004);
e) implementar o *backtracking* ao algoritmo WFC conforme proposto por Gumin (2019) e por Sandhu *et al.* (2019), para realizar a reconstrução em caso de impossibilidade de continuação;
f) gerar pequenos novos *chunks* de forma procedural, conforme o usuário se aproxima dos limites do terreno já gerado, para possibilitar uma execução mais rápida do algoritmo;

REFERÊNCIAS

BARR, David. **PixelGameEngine**. 2019. Disponível em https://github.com/OneLoneCoder/Javidx9/blob/master/PixelGameEngine/SmallerProjects/OneLoneCoder_PGE_IsometricTiles.cpp. Acesso em: 15 abr. 2023.

CASE, Nicky. **SIGHT & LIGHT: how to create 2D visibility/shadow effects for your game**. 2014. Disponível em: https://ncase.me/sight-and-light/. Acesso em: 15 abr. 2023.

CHENG, Darui; HAN, Honglei; FEI, Guangzheng. Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm. In: International Conference on Electrical Contacts, 30., 2020, Suiça. **Proceedings of the International Federation for Information Processing (IFIP)**, 2020, p. 37-50. Disponível em: https://www.researchgate.net/publication/348204502_Automatic_Generation_of_Game_Levels_Based_on_Controllable_Wave_Function_Collapse_Algorithm. Acesso em: 05 dez. 2022.

GUMIN, Maxim. **mxgmn/WaveFunctionCollapse**. 2019. Disponível em: https://github.com/mxgmn/WaveFunctionCollapse. Acesso em: 05 dez. 2022.

HILTON, James E.; LEONARD, Justin; BLANCHI, Raphaele; NEWNHAM, Glenn; OPIE, Kimberley; RUCINSKI, Chris; SWEDOSH, William. Dynamic modelling of radiant heat from wildfires. **22nd International Congress on Modelling and Simulation (MODSIM2017)**. dez. 2017. Disponível em: https://www.researchgate.net/publication/329503247_Dynamic_modelling_of_radiant_heat_from_wildfires. Acesso em: 15 abr. 2023.

KASAPAKIS, Vlasios; GAVALAS, Damianos; GALATIS, Panagiotis. Augmented reality in cultural heritage: Field of view awareness in an archaeological site mobile guide. **Journal of Ambient Intelligence and Smart Environments**, v. 8, n. 5, p. 501 – 514, 31 out. 2016. Disponível em: https://sci-hub.se/10.3233/ais-160394. Acesso em: 05 dez. 2022.

KIM, Hwanhee; LEE, Seongtaek; LEE, Hyundong; HAHN, Teasung; KANG, Shinjin. Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm. **IEEE Conference on Games**. p. 1 – 4, ago. 2019. Disponível em: https://ieeexplore.ieee.org/document/8848019. Acesso em: 15 abr. 2023.

MØLLER, Tobias N.; BILLESKOV, Jonas; PALAMAS, George. Expanding Wave Function Collapse with Growing Grids for Procedural Map Generation. In: International Conference on the Foundations of Digital Games, 15., 2020, Nova Iorque. **Proceedings of the 15th International Conference on the Foundations of Digital Games**. Nova Iorque: Association for Computing Machinery, 2020. p. 1 – 4. Disponível em: https://dl.acm.org/doi/10.1145/3402942.3402987. Acesso em 15 abr 2023.

MORRIS, Quentin Edward. **Modifying Wave Function Collapse for more Complex Use in Game Generation and Design**. Orientador: Mark Lewis. 2021. 62 f. Tese de Honra (Ciência da Computação) – Curso de Ciência da Computação, Universidade de Trinity, Texas, 2021. Disponível em: https://digitalcommons.trinity.edu/cgi/viewcontent.cgi?article=1058&context=compsci_honors. Acesso em: 05 dez 2022.

MUSETH, Ken. Hierarchical Digital Differential Analyzer for Efficient Ray-Marching in OpenVDB. In: Special Interest Group on Computer Graphics and Ineractive Techniques Conference, 14., 2014, Vancouver. **Proceedings SIGGRAPH '14: ACM SIGGRAPH 2014 Talks**. Nova Iorque: Association for Computing Machinery, 2014. p. 1. Disponível em: https://dl.acm.org/doi/10.1145/2614106.2614136. Acesso em: 15 abr. 2023.

PEDDIE, Jon. **What's the Difference Between Ray Tracing, Ray Casting, and Ray Charles?**. 2016. Disponível em: https://www.electronicdesign.com/technologies/displays/article/21801219/whats-the-difference-between-ray-tracing-ray-casting-and-ray-charles. Acesso em 17 set. 2022.

REINKE, Thomas R. **Aplicação de Wave Function Collapse e Ray Casting na criação de um jogo isométrico**. Pomerode, 2023. Disponível em: https://github.com/thrnkk/tcc. Acesso em: 25 jun. 2023.

RISI, Sebastian; LEHMAN, Joel; D'AMBROSIO, David B.; STANLEY, Kenneth O., Automatically Categorizing Procedurally Generated Content for Collecting Games. In: International Conference on the Foundations of Digital Games, 9., 2014, Nova Iorque. **Proceedings of the Workshop on Procedural Content Generation in Games**. Nova Iorque: Association for Computing Machinery, 2014. p. 1 - 7. Disponível em: http://www.fdg2014.org/workshops/pcg2014_paper_02.pdf. Acesso em: 05 dez. 2022.

SAMPAIO, Eduardo José Torres; RAMALHO, Geber Lisboa. **Forge 16V**: um framework para o desenvolvimento de jogos isométricos. 2003. Dissertação (Mestrado). Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Pernambuco, Recife. Disponível em: https://repositorio.ufpe.br/bitstream/123456789/2531/1/arquivo4819_1.pdf. Acesso em 05 dez. 2022.

SANDHU, Arunpreet; CHEN, Zeyuan; MCCOY, Joshua. Enhancing wave function collapse with design-level constraints. In: International Conference on the Foundations of Digital Games, 14., 2019, Nova Iorque. **Proceedings of the Workshop on Procedural Content Generation in Game.** Nova Iorque: Association for Computing Machinery, 2019. p. 1 - 9. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3337722.3337752. Acesso em: 05 dez. 2022.

SHAKER, Noor; TOGELIUS, Julian; NELSON, Mark J. **Procedural Content Generation in Games**. Nova Iorque: Springer Publishing Company, 2016. Disponível em: https://link.springer.com/book/10.1007/978-3-319-42716-4. Acesso em: 15 abr. 2023.

SMITH, Gillian; GAN, Elaine; OTHENIN-GIRARD, Alexei; WHITEHEAD, Jim. PCG-Based Game Design: Enabling New Play Experiences through Procedural Content Generation. In: International Workshop on Procedural Content Generation in Games, 2., 2011, Nova Iorque. **Proceedings of the 2nd International Workshop on Procedural Content Generation in Games**. Nova Iorque: Association for Computing Machinery, 2011. p 1 - 4. Disponível em: https://dl.acm.org/doi/pdf/10.1145/2000919.2000926. Acesso em: 05 dez. 2022.

SMITH, Gillian. The Future of Procedural Content Generation in Games. **AIIDE Workshop Technical Report**, v. 10, n. 3, p. 53 – 57, jun. 2021. Disponível em: https://ojs.aaai.org/index.php/AIIDE/article/view/12748/12596.

SUMMERVILLE, Adam; SNODGRASS, Sam; GUZDIAL, Matthew; HOLMGÅRD, Christoffer; HOOVER, Amy K.; ISAKSEN, Aaron; NEALEN, Andy; TOGELIUS, Julian. Procedural Content Generation via Machine Learning (PCGML). **IEEE Transactions on Games**, v. 10, n. 3, p. 257 – 270, set. 2018. Disponível em: https://arxiv.org/pdf/1702.00539.pdf. Acesso em: 05 dez. 2022.

TOGELIUS, Julian; KASTBJERG, Emil; SCHEDL, David; YANNAKAKIS, Georgios N. What is Procedural Content Generation? Mario on the borderline. In: International Workshop on Procedural Content Generation in Games, 2., 2011, Nova Iorque. **Proceedings of the 2nd International Workshop on Procedural Content Generation in Games**. Nova Iorque: Association for Computing Machinery, 2011. p. 1 – 6. Disponível em: https://dl.acm.org/doi/10.1145/2000919.2000922. Acesso em: 15 abr. 2023.

VANDEVENNE, Lode. **Raycasting**. 2004. Disponível em https://lodev.org/cgtutor/raycasting.html. Acesso em: 15 abr. 2023.

WALSH, Corey H.; KARAMAN, Sertac. **CDDT: Fast Approximate 2D Ray Casting for Accelerated Localization**. In: International Conference on Robotics and Automation (ICRA), Australia. 2018, p. 3677 - 3684. Disponível em: https://arxiv.org/pdf/1705.01167.pdf. Acesso em: 05 dez. 2022.