UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME CORRÊA MILAK

COMPARAÇÃO DOS ALGORITMOS DE BUSCA DE CAMINHO, A* E DIJKSTRA, APLICADO EM CENÁRIOS DE JOGOS

CRICIÚMA

2018

GUILHERME CORRÊA MILAK

COMPARAÇÃO DOS ALGORITMOS DE BUSCA DE CAMINHO, A* E DIJKSTRA, APLICADO EM CENÁRIOS DE JOGOS

Trabalho de Conclusão de Curso, apresentado para a obtenção do grau de Bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: Prof. MSc. Luciano Antunes

Coorientador: Prof. MSc. Matheus Leandro Ferreira

CRICIÚMA

2018

GUILHERME CORRÊA MILAK

COMPARAÇÃO DOS ALGORITMOS DE BUSCA DE CAMINHO, A* E DIJKSTRA, APLICADO EM CENÁRIOS DE JOGOS

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Teoria dos Grafos.

Criciúma, 26 de junho de 2018.

BANCA EXAMINADORA

Prof. Luciano Antunes – Me. – (UNESC) - Orientador

Prof. Matheus Leandro Ferreira – Esp. – (UNESC) – Coorientador

Prof. Marlon de Matos de Oliveira - Dr. – (UFSC – Araranguá)

Prof. Kristian Madeira - Dr. – (UNESC)

AGRADECIMENTOS

Agradeço aos meus familiares, principalmente aos meus pais, por tornarem possível a conclusão de mais esta etapa da minha vida.

Aos meus colegas e amigos que me deram apoio para não desistir e continuar com a realização deste trabalho.

E ao meu orientador Luciano e coorientador Matheus que me deram todo o suporte necessário e me guiaram da melhor forma possível em direção aos objetivos.

“O rio atinge seus objetivos porque aprendeu a contornar obstáculos.”

Lao Tsé

RESUMO

O mercado de Jogos Digitais cresce de forma constante e acelerada. A indústria de jogos lança todo ano diversos títulos que contam com, cada vez mais, recursos gráficos de alto poder tecnológico. O uso destes recursos depende muitas vezes do hardware de processamento gráfico, mas também está diretamente ligado à forma como um jogo digital é idealizado e implementado. Acompanhando o crescimento dos jogos digitais, surgem também as *engines,* termo utilizado para definir os motores de jogos, muitos disponibilizados gratuitamente e responsáveis por facilitar o desenvolvimento de jogos. Por meio do uso de um motor de jogo, é possível gerar cenários tridimensionais e implementar vários outros recursos presentes em jogos digitais de forma mais rápida. Em diversos gêneros de jogos, o comportamento dos personagens e de outras entidades relacionadas ao jogo também impactam na performance do mesmo, e consequentemente, na experiência do jogador. Uma das técnicas responsáveis por prover inteligência à uma entidade de um jogo é conhecida como busca de caminho, esta técnica consiste na determinação e projeção de uma rota para que um objeto do jogo possa navegar entre dois ou mais pontos do cenário, conhecendo o ambiente a qual está percorrendo e, consequentemente, demonstrando mais inteligência e fluidez na navegação. Para o uso do recurso de busca de caminho, se faz necessário a implementação de um algoritmo que será responsável por interpretar os pontos navegáveis do cenário a qual está inserido e então, gerar uma rota entre estes pontos. Existem vários tipos de algoritmos que realizam esta tarefa, dentre eles estão os algoritmos de Dijkstra e A*. Nesta pesquisa, utilizando-se do motor de jogo *Unity3D* para a modelagem de cenários tridimensionais, foram implementados os algoritmos de busca de caminho Dijkstra e A* que foram utilizados para a geração de caminhos dentro dos cenários modelados, viabilizando uma comparação entre ambos. Dentre os cenários testados, os resultados obtidos demonstraram que, conforme a complexidade dos cenários aumenta, os algoritmos tendem a impactar de forma crescente o fluxo do jogo.

Palavras-chave: Jogos Digitais. Unity3D. Busca de caminho. Algoritmo de Dijkstra. Algoritmo A*.

ABSTRACT

The game design market has been expanding in a spiking fashion. Each year the game industry launches a variety of titles relying more and more on technologically advanced graphic resources. The increasing use of these resources oftentimes depend on the graphics processing hardware unit. However, it is directly linked to how the game is conceived. Parallel to the game market growth, the game engine hype can be noticed as well, since a multitude of these can be used free of charge to ease the struggle of game development. By using these engines, it is possible to generate three dimensional scenarios and implement other features that are pillars in gaming in the least amount of time. In a variety of genres, the behaviors of the characters and other entities can also further impact the overall performance and, consequently, the player experience. One of the methods responsible for providing intelligence to a game entity is known as pathfinding. It consists of determining and projecting a path so the object can navigate between two or more spots in a scenario, identifying the environment that is being traversed and displaying wit and flow while navigating. In order to approach the pathfinding concept, an algorithm has to be developed to interpret the reachable spots in a given scenario and plot a path between these two or more spots. There is a wide range of algorithms that can accomplish such a task, among them the Dijkstra algorithm and A*. In this project, the Unity 3D game engine was in charge of three-dimensional terrain modelling, in which the pathfinding algorithms were implemented and the paths were generated, further allowing them to be compared to each other. Among the scenarios that were subject to testing, the gathered results demonstrate that as the obstacle complexity rises, the impact of the algorithms in the game's flow increases as well.

Keywords: Digital gaming. Unity3D. Pathfinding. Dijkstra algorithm. A* algorithm.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2D	2-Dimensional
3D	3-Dimensional
BNL	Brookhaven National Laboratory
E3	Electronic Entertainment Expo
GDD	Game Design Document
IA	Inteligência Artificial
IDSA	Interactive Digital Software Association
MIT	Massachusetts Institute of Technology
NPC	Non-Player Character
RPG	Role Playing Game

SUMÁRIO

1 INTRODUÇÃO

Um jogo pode ser definido como uma atividade lúdica muito mais ampla que um fenômeno físico ou reflexo psicológico, sendo ainda, um ato voluntário concretizado como evasão da vida real, limitado pelo tempo e espaço, criando a ordem através de uma perfeição temporária. Adicionalmente, apresenta tensão, expressa sob forma de incerteza e acaso, no sentido de que em um jogo jamais se deve conhecer desfecho. O desconhecimento do desfecho, por sua vez, é uma característica importante nos jogos, pois seu desenvolvimento depende dos mais variados fatores, internos e externos, como as estratégias adotadas e as respostas fornecidas pelo ambiente (LUCCHESE; RIBEIRO, 2009).

De acordo com Schuytema (2008) um jogo eletrônico é uma atividade lúdica formada por ações e decisões que resultam numa condição final. Tais ações e decisões são limitadas por um conjunto de regras e por um universo, que no contexto dos jogos digitais, são regidos por um programa de computador.

O jogo eletrônico é composto de três partes: enredo, motor e interface interativa. O enredo define o tema, a trama, os objetivos do jogo e a sequência com a qual os acontecimentos surgem. O motor do jogo é o mecanismo que controla a reação do ambiente às ações e decisões do jogador, efetuando as alterações de estado neste ambiente. Por fim, a interface interativa permite a comunicação entre o jogador e o motor do jogo, fornecendo um caminho de entrada para as ações do jogador e um caminho de saída para as respostas audiovisuais referentes às mudanças do estado do ambiente (LUCCHESE; RIBEIRO, 2009).

Atualmente, com o surgimento de motores de jogos gratuitos e poderosos como o Unity3D e o Unreal Engine o mercado de jogos digitais deixou de ser estimulado apenas por grandes empresas, mas também por desenvolvedores *indies*, termo este utilizado para designar desenvolvedores independentes. Tendo estes motores de jogos se tornados tão acessíveis, o uso de recursos como Inteligência Artificial (IA), que se referindo a jogos digitais, incorpora funções essenciais para o funcionamento do mesmo, foi altamente facilitado. Funcionalidades como som, cálculos e iluminação já veem implementados e previamente configurados.

Um dos desafios de um jogador em um determinado jogo é encontrar o melhor caminho que o personagem deverá percorrer até chegar ao seu ponto de destino. Os motores de jogos permitem que algoritmos de busca de melhor caminho

como o A*, Floyd-Warshall, Johnson, Dijkstra, Bellman-Ford, entre outros, sejam implementados. Porém, dentre estes algoritmos, a dúvida é, qual deles é o mais adequado para uso nessas situações. Em consequência disto, este estudo busca implementar cenários de jogos dotados de diferentes níveis de complexidade a fim de se aplicar os algoritmos de busca de caminho A* e Dijkstra, para se obter uma comparação empírica entre estes algoritmos, realizando análises de desempenho, como o tempo para atingir o ponto de destino a partir do ponto de partida, o número de iterações de cada algoritmo, o impacto dos algoritmos na média dos *frames*, termo este utilizado para caracterizar a medida do número de imagens ou quadros que um dispositivo gráfico processa e exibe em uma unidade de tempo, do jogo e o tempo consumido por *frame* para a realização das buscas.

1.1 OBJETIVO GERAL

Comparar os algoritmos de busca de caminho A* e Dijkstra com o uso do motor de jogo Unity3D, utilizando cenários distintos.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos desta pesquisa se constituem de:

a) demonstrar o funcionamento de motores de jogos;
b) demonstrar o conceito do problema do caminho mínimo;
c) interpretar o funcionamento do algoritmo de busca de caminho mínimo Dijkstra;
d) interpretar o funcionamento do algoritmo de busca de caminho mínimo A*;
e) reproduzir ambientes de jogos em um motor de jogo;
f) aplicar testes com os algoritmos estudados nos ambientes gerados a fim de obter uma comparação de ambos.

1.3 JUSTIFICATIVA

Atualmente, jogos por computador têm sido largamente explorados em termos comerciais. Dados da *Interactive Digital Software Association* (IDSA), a

associação que organiza a *Electronic Entertainment Expo* (E3) demonstram que em 1999 foram comercializados em torno de US$ 6,1 bilhões em software de entretenimento só nos Estados Unidos. Em 1996, este volume era US$ 3,7 bilhões, com venda de 109 milhões unidades de software de jogos, sendo 59 milhões de programas para PCs. Pesquisa da IDSA também mostra que, em 1996, jogos em PCs eram mais usados do que processadores de texto (BATTAIOLA, 2000).

Ainda segundo Battaiola (2000) um aspecto importante observado sobre jogos por computador é que eles, ao contrário de pacotes tradicionais, como, por exemplo, formatadores de texto, compiladores, sistemas gráficos, etc., são potencialmente mais competitivos em termos de mercado. Este fato se justifica porque cada jogo é um produto em particular e o seu sucesso não está totalmente relacionado com a sua sofisticação computacional, mas sim aos atrativos lúdicos que ele fornece aos usuários.

Mover-se de um lugar a outro, utilizando um caminho razoável, ao mesmo tempo em que se desvia de obstáculos, é um requisito fundamental para qualquer entidade que queira demostrar algum sinal de inteligência em um jogo. Um dos aspectos mais importantes relacionados à implementação de funcionalidades de IA em jogos, e de impacto visual mais obvio, é então a determinação de caminhos (KARLSSON, 2006). Em relação ao comportamento de movimentação em jogos, um dos problemas que os desenvolvedores encontram é a minimização da carga do processador versus maximizar a precisão e a inteligência do movimento (POTTINGER, 1999, tradução nossa).

O aumento expressivo na popularização dos jogos eletrônicos fez com que cada vez mais novas empresas, novos cursos de especialização, e, consequentemente, novos tipos de profissionais surgissem. Um outro tipo de programa de computador também teve um grande aumento no mercado, os motores de jogos, responsáveis por facilitar o desenvolvimento de jogos tanto para equipes pequenas ou desenvolvedores *indies*, quanto para grandes empresas, pois oferecem grandes quantidades de recursos como cálculos matemáticos, principalmente para simular física, ferramentas para facilitar a criação de cenários, personagens, animações, efeitos especiais, entre outros. Existem muitos motores de criação de jogos no mercado, alguns exemplos são: Construct 2, Unreal Engine, CryEngine, GameMaker e Unity3D, sendo este um dos motores de jogos que mais teve destaque no mercado nos últimos anos. Dentre os fatores que contribuíram para

isso, pode-se destacar o fato de o motor ser totalmente gratuito, oferecer um ambiente de desenvolvimento simplificado e de fácil aprendizagem, utilizar uma linguagem de programação de alto nível para implementação do código do jogo, e ainda, possuir uma loja de recursos com modelos tridimensionais, efeitos sonoros, *scripts*, dentre outros recursos utilizados no desenvolvimento de jogos, sendo muitos desses recursos gratuitos e livres para a utilização.

No processo de desenvolvimento de um jogo, quando o desenvolvedor ou a equipe de desenvolvimento se depara com a necessidade de utilização de recursos de busca de caminho, se faz necessário a escolha de qual algoritmo usar, uma vez que existem vários tipos de algoritmos com este propósito.

Neste ponto, há uma variável importante, o desempenho dos algoritmos, pois esta é uma variável de grande importância para o desenvolvimento de jogos, uma vez que impacta diretamente na experiência do jogador, e poucas pesquisas e trabalhos que têm como objetivo realizar testes nestes algoritmos para relatar suas características de desempenho foram encontradas.

Por estes motivos, aliado ao fato de que cada vez mais os jogos estão ganhando espaço no mercado, como citado acima, procura-se por meio deste trabalho, aplicar os algoritmos de busca de caminho A* e Dijkstra em diferentes cenários de jogos, a fim de demonstrar qual dos algoritmos é mais indicado para a solução do problema de melhor caminho.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é dividido em cinco capítulos principais. No primeiro capítulo é realizada uma breve introdução do surgimento dos jogos digitais e o papel dos problemas de caminhos neste contexto.

O segundo capítulo descreve jogos digitais, onde é apresentado sua definição, a modalidade de jogos tridimensionais, uma breve história, os principais gêneros, além de relatar também sobre o mercado de jogos. Também neste capítulo, é descrito as etapas de desenvolvimento de um jogo, motor de jogo, com destaque para o Unity 3D e por fim, técnicas de mapeamento de cenários de jogos.

No terceiro capítulo, relata-se sobre problemas de caminhos, apresentando sua definição e aplicações, seguindo com uma contextualização em cenários de jogos, e finalizando com a explanação dos algoritmos de busca de

caminho mínimo Dijkstra e A*, onde se realiza uma breve introdução seguido de uma demonstração detalhada de suas implementações.

Os trabalhos correlatos que foram usados como base para a realização deste projeto são apresentados no capítulo 4. Neste capítulo, são apresentadas pesquisas relacionadas ao conceito de busca de caminho em jogos digitais, como forma de complementar o conhecimento na área.

No quinto capítulo está o trabalho proposto, onde encontra-se a explicação do que será explorado de acordo com a metodologia. Na metodologia, por sua vez, é demonstrada toda a implementação dos algoritmos Dijkstra e A*, bem como a forma de mapear o cenário e por fim, a modelagem dos ambientes utilizadas como base para as comparações. Em seguida, é realizada a coleta dos dados referentes aos comportamentos de ambos algoritmos e realizada a comparação entre os mesmos.

2 JOGOS

Antigamente, os jogos se caracterizavam por ser uma simulação que era executada apenas em nossas mentes, onde os jogadores interagiam entre si e com o ambiente por meio de narrações de histórias em grupo e em tempo real. Era comum, nestes tipos de jogos, que muitos elementos fossem abstraídos pelas mentes dos jogadores simulando batalhas e ações que na maioria das vezes não existiam na realidade (ZAMBIASI; PINHEIRO, 2010).

2.1 JOGOS DIGITAIS

Jogos desenvolvidos digitalmente podem ser descritos como uma ramificação de jogos não digitais. Dentre os usuários de computadores, grande parte já passou horas se divertindo em algum tipo de jogo digital (BATTAIOLA, 2000).

Um jogo digital, ou eletrônico, como também pode ser chamado, é descrito por Schuytema (2008), como uma atividade lúdica elaborada por ações e decisões que, juntas, resultam em uma condição final. Neste contexto, um programa de computador limita essas ações por meio de um universo criado virtualmente, e um conjunto de regras. O universo concretiza as ações dos jogadores, enquanto as regras definem o que pode e o que não pode ser realizado, assim como geram as consequências das ações do jogador.

Um jogo digital pode ser definido como um sistema composto de três partes básicas, o enredo, um motor e uma interface de interação com o jogador. Battaiola (2000), descreve esses três elementos como sendo:

a) **enredo:** os desafios que o jogo irá oferecer aos jogadores, a história, quando existir uma, os objetivos a qual o jogador, por meio de uma sequência de etapas, terá de se esforçar para atingir;
b) **interface interativa:** esta, realiza o gerenciamento entre o jogador e o motor do jogo, reportando mudanças de ambientes, regras ou estados do jogo. O desenvolvimento desta interface, leva em consideração aspectos artísticos, cognitivos e técnicos. Artístico, na capacidade de valorizar a representação do jogo; cognitivo, na capacidade de interpretar corretamente as informações gráficas pelo usuário; e técnico

envolvendo complexidade, performance e portabilidade dos elementos gráficos;

c) **motor:** é o sistema de controle de jogo, que está por trás do usuário controlando os estados do jogo de acordo com determinada ação do jogador. Este, está envolvido em uma série de aspectos computacionais, dentre eles, escolha da linguagem de programação apropriada em função de portabilidade e facilidade, desenvolvimento de algoritmos específicos, características do tipo de interface com o usuário, entre outros. No item 2.7 desta pesquisa, o motor de jogo e suas características serão descritos de forma mais aprofundada.

A principal característica responsável pela diferenciação de jogos digitais dos não digitais é a existência de mundos fictícios, que por sua vez, são fundamentalmente abstratos. É fato que, em jogo não-digitais, o jogador acaba imaginando seu próprio mundo fictício, mas este, fica limitado à imaginação do usuário, e não é compartilhado com os demais jogadores, diferentemente dos jogos digitais, onde a representação do universo é a mesma para todos os jogadores (JUUL, 2005).

Outra diferença marcante em jogos digitais se dá na criação das regras de determinado jogo. Em um jogo não digital, sempre há espaço para manipulação das regras, isto é, modificá-las para atender a um certo requisito imposto pelos jogadores. Entretanto, quando se trata de um jogo digital, essa flexibilidade não existe, pelo fato de que as regras são pré-computadas pelo algoritmo do jogo. De fato, existem jogos em que é possível personalizar algumas regras por meio de um menu de configurações, mas esses mecanismos não são tão flexíveis quanto os meios de comunicação exercidos em jogos não-digitais (LUCCHESE; RIBEIRO, 2009).

Adicionalmente, de acordo com Lucchese e Ribeiro (2009), com relação à existência de mundos fictícios, os jogos digitais fornecem ao jogador uma experiência audiovisual muito superior aos jogos não digitais, uma vez que o mercado de jogos evoluiu drasticamente, com novas tecnologias que elevaram o poder de processamento gráfico, sonoro e de controle, e que estão em constante evolução, fornecendo cada vez mais experiências nunca antes vistas neste contexto de jogo.

2.2 JOGOS 3D

A partir da década de 90, com o início da redução dos preços de processadores de vídeo mais potentes, as conhecidas, placas aceleradoras 3D, os jogos digitais passaram a serem mais imersivos. Com novas tecnologias de desenvolvimento de universos tridimensionais, surge então o primeiro jogo 3D que se tornou um sucesso no ano de 1992, *Wolfenstein 3D.* Partindo-se de experiências anteriores em que os jogos utilizavam imagens distorcidas impressas sobre sólidos para criar uma falsa ilusão de profundidade, o jogo apresentava pela primeira vez um elemento que se joga no cenário, uma arma, o que garantia ao jogador a experiência de participar visualmente na ação (CLUA; BITTENCOURT, 2005).

Diferentemente dos jogos bidimensionais, onde só é possível movimentar o personagem em duas direções, sendo elas a horizontal (X) e vertical (Y), um jogo tridimensional oferece mais imersão ao jogador incluindo um eixo de direção a mais, o eixo de Z, que representa a profundidade do cenário, o que requer mais processamento gráfico, uma vez que os objetos precisam ser modelados tridimensionalmente para respeitar a profundeza do cenário (MARQUES, 2015).

Um jogo tridimensional (3D) é um programa de computador considerado muito especial. Este necessita explorar ao máximo o *hardware* dedicado, ou, as conhecidas placas aceleradoras 3D. Vários módulos se fazem necessários para a construção de um jogo 3D, como computação gráfica, inteligência artificial, multimídia, redes de computadores, entre outros. Além disso, todos esses módulos citados precisam trabalhar em conjunto e respeitando uma característica fundamental de um jogo: ser um programa de computador em tempo real (CLUA; BITTENCOURT, 2005).

2.3 BREVE HISTÓRIA

As primeiras pesquisas abrangendo jogos surgiram por volta de 1952. Douglas A.S. em seu doutorado desenvolveu uma versão gráfica do famoso jogo-da-velha utilizando um computador de válvulas *EDSAC.* Mais adiante, em 1958 surgia um simulador simples de um jogo de tênis, inspirado no esporte mundialmente conhecido, em que o jogador rebate uma bola com uma raquete através de uma rede, com o objetivo de atingir a quadra do oponente. Este foi desenvolvido por

cientistas no laboratório de pesquisas militares *Brookhaven National Laboratory.* A partir dessas duas invenções, houveram quase cinco décadas de desenvolvimento de outros tipos de jogos, até que, em 1961, estudantes do *Massachusetts Institute of Technology* (MIT) publicam o *Spacewar,* que contava com batalhas aeroespaciais e inspirado nos livros do autor E.E Smith (BENIN, 2007).

A partir das décadas de 70 e 80, com o avanço da tecnologia computacional desta época, começou a ser possível levar os computadores pessoais para os lares, e por consequência, o aumento no interesse da população por jogos, principalmente de crianças e jovens. Por consequência, houve um grande aumento na produção de jogos, uma vez que a demanda buscava por diferentes gêneros de jogos (BENIN, 2007).

Benin (2007) ainda conclui que, como citado anteriormente, durante quase cinquenta anos de pesquisa em desenvolvimento de jogos, diversos gêneros surgiram e foram explorados em busca de novas formas, estilos e jogabilidades. No próximo item desta pesquisa, serão explorados e descritos os principais gêneros de jogos existentes no mercado.

2.4 GÊNEROS DE JOGOS

De uma forma geral, os jogos são normalmente classificados por agrupamentos de seus tipos, que obedecem ou apresentam grandes características similares entre si. Dentre essas características e critérios, podem ser citados a forma como o jogador interage com o universo do jogo ou ambiente, o tipo de interação que o jogador possui com o seu personagem, o objetivo do jogo, principalmente, e por último o contexto na qual se insere o jogador (LUCCHESE; RIBEIRO, 2009).

O tipo de um jogo é responsável por algumas de suas características de motor e interface, e este gera implicações claras nas técnicas de sua implementação. Battaiola (2000) descreve os gêneros de jogos como sendo:

a) **estratégia:** jogos que exigem mais da capacidade intelectual do jogador, do que apenas reflexos. Nesse gênero, os objetivos do usuário geram grandes consequências no andamento e no estado do jogo. Podem ser citados como exemplos *SimCity* e *SimCity 2000* como jogos de planejamento de países e cidades, *WarCraft* como jogo de estratégia de guerra, este, onde o jogador comanda exércitos e

esquadras, e além disso, qualquer outro jogo onde a percepção e a forma com que o jogador submete suas escolhas são mais importantes do que a velocidade ou o reflexo. Adicionalmente, a lógica de operação desses jogos geralmente são complexas, pois contam com estruturas e bases de dados extensas, e não requerem interação em tempo real;

b) **simuladores:** neste gênero, incluem-se jogos que simulam carros, aviões, naves espaciais e qualquer outro simulador que age na tentativa de modelar alguma situação do mundo real, em uma perspectiva de primeira pessoa. É importante destacar ainda que, estes tipos de jogos buscam dar grande valor à física do ambiente, pelo fato de que o principal objetivo é imergir o usuário no ambiente proposto. Normalmente, a interface desses jogos são construídas tridimensionalmente, utilizando texturas e polígonos de alta qualidade. E por fim, possuem grande complexidade na lógica operacional, e estruturas de dados grandes, principalmente por conta de cálculos de física do ambiente;

c) **aventura:** jogos em que o principal objetivo do jogador é finalizar os estágios do jogo solucionando enigmas e quebra-cabeças, unindo ações que se baseiam em raciocínio e reflexo. O usuário age em terceira pessoa, geralmente atuando como um auxiliar de um personagem principal. Neste gênero, as bases de dados também podem ser consideradas extensas, pelo fato de o jogo normalmente oferecer muita liberdade ao jogador, porém, o grau de complexidade do motor diminui, pois utilizam-se muitas texturas com padrões constantes;

d) **infantil:** são geralmente jogos de quebra-cabeças, educativos ou que possuem histórias simples, a fim de divertir seu público-alvo que é formado por crianças. São jogos caracterizados por possuírem imagens muito coloridas e com visual próximo ao de desenhos animados. Neste gênero, a principal prioridade é na facilidade de interação com o jogo, ou seja, a interface. A criança atua em terceira pessoa, normalmente auxiliando um personagem tido como principal, progredindo quando consegue resolver algum pequeno desafio;

e) **passatempo:** esses são caracterizados por não possuírem história,

cujo principal objetivo é atingir uma pontuação alta em jogos simples como por exemplo rápidos quebra-cabeças. Possuem usualmente uma simples interface bidimensional, contudo, pode possuir um motor com lógica complexa;

f) **RPG:** são os *Role Playing Game* implementados em programas de computador com os mesmos objetivos de um RPG convencional. Inicialmente, a perspectiva desse gênero de jogo se passava em primeira pessoa, porém, esta tendência está mudando para terceira pessoa. Possui uma implementação complexa, pois exige uma grande base de dados;

g) **esporte:** jogos que simulam esportes populares da realidade, como futebol, vôlei, basquete, boxe, entre outros. Pelo fato de que o jogador comanda um time inteiro neste gênero, a interface é tridimensional e em termos de programação, possui os mesmo problemas dos simuladores, pois exigem que a ação ocorra em tempo real;

h) **educação/treinamento:** este gênero de jogo pode incluir características de qualquer outro gênero citado anteriormente, diferenciando-se dos outros pelo fato de apresentar critérios didáticos e pedagógicos.

O modo de classificação apresentado por Battaiola aparenta definir o gênero do jogo pela sua principal característica, ou seja, a que fica mais evidente. Por este motivo, é normal que um jogo possa estar em mais de uma categoria de gênero ao mesmo tempo. O jogo *Portal* serve como um exemplo de uma junção de aspectos de um jogo de tiro em primeira pessoa com elementos originários de jogos de passatempo, como quebra-cabeças (LUCCHESE; RIBEIRO, 2009).

2.5 MERCADO DE GAMES

Os jogos digitais, antes considerados programas de computador apenas com o intuito de divertir os usuários sem gerar lucro algum, passaram com o decorrer do tempo, a ocupar grandes setores da economia mundial.

A indústria de jogos por meio do seu potencial de geração de emprego e renda, e também por promover contribuições importantes de inovação tecnológica em diferentes setores da economia como arquitetura, construção civil, marketing,

publicidade, áreas de saúde, treinamento e capacitação, dentre outros, tornou-se uma área que deixou de ser apenas voltada à diversão para ser também uma área de grande valor no mercado (FLEURY; NAKANO; CORDEIRO, 2014).

De acordo com Fleury, Nakano e Cordeiro (2014) segundo a consultoria *PricewaterhouseCoopers* (PwC, 2014), no ano 2013, o mercado de jogos movimentou 65,7 bilhões de dólares e estima-se que em 2018, este valor aumente para 89 bilhões de dólares, projetando uma taxa de crescimento de 6,3% ao ano. A nível de comparação, a indústria de filmes e entretenimento movimentou no ano 2013, 88,2 bilhões de dólares e a projeção para 2018 é 110 bilhões de dólares, o que gera um aumento de 4,5% ao ano.

Dentre os motivos que trouxeram essa área para uma posição de tamanha importância no mercado, um deles é o fato de que os jogos pararam de serem consumidos apenas pelo público de jovens do sexo masculino, como se pensava tradicionalmente, como também por crianças, mulheres e idosos (FLEURY; NAKANO; CORDEIRO, 2014).

Um outro ponto de grande importância a se destacar é o constante avanço e sofisticação nos dispositivos utilizados para executar os jogos, como computadores, videogames, *smartphones*, entre outros. A evolução desses dispositivos estimulam cada vez mais as empresas desenvolvedoras de jogos digitais a criarem jogos que contam com mais realismo gráfico e intelectual (MELLO; ZENDRON, 2015).

Mello e Zendron (2015) complementam ainda a respeito da expansão e aumento da capacidade das redes de computadores, que beneficiaram ainda mais o mercado de jogos digitais, por meio de vendas de jogos em lojas online, não necessitando da compra do jogo físico, como no CD por exemplo. Além disso, com a melhora nos pacotes de internet, os jogos online, onde os jogadores jogam juntos conectados pelo mundo todo, foram fortemente alavancados, se expandindo e tendo elevadas taxas de crescimento no mercado.

Battaiola (2000) reforça ainda que, os jogos contribuem com a capacitação técnica pelo fato de que, desenvolver um jogo, requer múltiplos profissionais de diversas subáreas computacionais, como inteligência artificial, programação, sistemas operacionais, computação gráfica, etc., o que contribui com a geração de empregos para novos tipos de profissionais.

2.6 ETAPAS DE DESENVOLVIMENTO DE UM JOGO

Apesar de um jogo ser um programa de computador e depender das técnicas de desenvolvimento e dos programadores, o seu sucesso é quase inteiramente dependente de sua concepção (BRASILIENSE, 2006).

Neste item, será descrito de forma resumida as principais etapas de desenvolvimento de um jogo, envolvendo desde a etapa inicial, na criação do *Game Design Document (GDD)*, documento a qual possui todas as informações do jogo a ser desenvolvido, até as etapas finais de publicação de um jogo.

A elaboração de um jogo digital envolve diversos profissionais específicos e especializados em diferentes áreas do conhecimento humano. Para a concepção de um jogo digital, podem-se citar como as principais e mais críticas áreas o roteiro e a narrativa (estes presentes no *GDD*); toda a arte do jogo como conceitos de personagens, objetos do jogo, cenários, entre outros; a modelagem desses objetos; texturização e programação (MARQUES, 2015).

Coutinho et al. (2014) destaca ainda que o processo de elaboração de um jogo, dependendo de sua complexidade e seu tamanho, mesmo contando com uma grande equipe de desenvolvimento, pode demorar meses ou até mesmo anos para ter seu primeiro protótipo jogável pronto.

O processo de desenvolvimento pode ser dividido em três grandes categorias, sendo elas: pré-produção, produção e pós-produção (SCHUYTEMA, 2008).

Marques (2015) explana as três categorias justificando que a primeira, é onde todo o time responsável pelo jogo, e não somente os desenvolvedores, definem ideias, o estilo do jogo, os objetivos, dentre outros. Já na etapa de produção, é onde entra a maior parte das áreas citadas anteriormente, com a modelagem e texturização dos recursos, definição do motor de jogo a ser utilizado, construção do design dos níveis, definição da mecânica do jogo e a criação e revisão do código implementado pela equipe de programação. E por fim, a pós-produção, iniciada a partir do lançamento do jogo, e consiste na produção de conteúdo extra para *download,* como expansões ou atualizações, se for o caso.

De modo geral, as etapas de elaboração de um jogo digital consistem na definição do *Game Design Document* (GDD), a criação das artes conceituais, a modelagem dos recursos visuais, a texturização das artes e a programação do jogo.

2.6.1 O *Game Design Document (GDD)*

Este documento é elaborado na pré-produção do jogo, onde contém todas as características do jogo em questão a ser desenvolvido, como história, regras, objetivos, personagens, mecânica, perspectiva (primeira pessoa, terceira pessoa ou outra), dentre outros. Esse processo gera uma série de outros documentos que serão utilizados por todos os integrantes da equipe de arte e programação do jogo durante todo o seu tempo de desenvolvimento (BRASILIENSE, 2006).

Os responsáveis pela elaboração deste documento são os chamados *Game Designers*, e é deles a responsabilidade de ditar para a equipe de desenvolvimento o que será programado, modelado e as imagens que serão utilizadas. É deles também a palavra final para aprovar o que foi desenvolvido (BRASILIENSE, 2006).

2.6.2 Arte Conceitual

Esta etapa representa a criação de conceitos, seja de personagens, objetos, cenários e qualquer outro item visual que será utilizado no jogo. Esta representação é feita de forma parecida a um rascunho, para se ter noção do produto final. Esta etapa é de grande importância, visto que ela pode ser considerada o alicerce e ponto de partida para a construção de um jogo de qualidade (MARQUES, 2015).

> Desenvolver um conceito para um novo jogo é como fazer um esboço. Uma ideia é tirada de algum estado precoce e transformada em algo mais elaborado. Mediante esse processo, os detalhes são trabalhados e o conceito se torna mais “real”. (RABIN, 2012, p. 119)

2.6.3 Modelagem tridimensional

Na produção de um jogo, esta é a etapa responsável por gerar e popular os cenários de todo o universo do jogo com os modelos desenvolvidos por profissionais da área de modelagem 3D.

Para isso, este profissional necessita de um programa de computador que o auxilie na geração desses materiais. Marques (2015) cita alguns desses

programas disponíveis no mercado, como: Autodesk Maya, Autodesk 3D Studio Max, Cinema 4D, Blender (este sendo gratuito e de código fonte aberto), entre outros.

É importante ressaltar que o modelador precisa tomar alguns cuidados quando está criando um novo objeto. Ele precisa estar atento à forma, estilo e com a contagem de polígonos do seu modelo, principalmente, pois este diz respeito à eficiência do mesmo. Um modelo com uma alta contagem de polígonos gera mais trabalho para o *hardware* de processamento gráfico, o que, consequentemente reduz o desempenho do jogo (RABIN, 2013).

2.6.4 Texturização

Esta é a etapa que dará cor e realismo aos objetos anteriormente criados pelo profissional de modelagem 3D.

Um artista digital realiza um processo de pintura em uma imagem bidimensional utilizando algum programa de computador de edição de imagem como auxiliador, pode-se citar por exemplo, o *Photoshop*. Com isso, o mapa de textura é gerado (MARQUES, 2015).

Este mapa é então aplicado ao objeto 3D por meio de uma técnica parecida com a de embrulhar um objeto, como uma bola de futebol, por exemplo. Por esta razão, o artista digital deve preparar a textura da melhor forma possível para encaixar de forma adequada ao modelo 3D (RABIN, 2013).

Marques (2015) finaliza explanando que a texturização é o processo que dá vida ao jogo, dando cor, acabamento, noção de profundidade e vincos aos modelos tridimensionais. Sem ela, esses modelos permaneceriam na sua cor original, cinza, que possuem quando estão sendo modelados no programa de modelagem 3D.

2.6.5 Programação

Para um jogo possuir mecânicas e ser funcional, ele precisa do trabalho dos programadores para gerarem os *scripts* que são anexados aos objetos ou personagens com o objetivo de impor alguma inteligência nestes objetos.

Utilizando-se de linguagens de programação, a equipe de programadores

do jogo desenvolve os códigos responsáveis pela movimentação, interação, interface com o usuário, dentre outros, para serem anexados às entidades do jogo.

Em jogos de um único jogador, onde o usuário interage com personagens que são controlados pelo computador, esses personagens precisam ser inteligentes e interativos, do contrário, o jogador não teria desafio ao jogar o jogo (BRASILIENSE, 2006).

A linguagem de programação a ser utilizada pela equipe depende de alguns fatores, sendo um deles, a escolha do motor para o jogo. Se o jogo utilizará algum motor já pronto, como exemplos, Construct 2, Unreal Engine ou Unity3D, a equipe não terá muita escolha a não ser codificar na linguagem suportada pelo mesmo. Em casos em que a equipe desenvolverá seu próprio motor, como em grandes empresas, por exemplo a *Blizzard,* desenvolvedora de jogos como *StarCraft* e *World of Warcraft*, fica a critério da mesma a escolha da linguagem, uma vez que desenvolver um motor passa a não ser um trabalho custoso para grandes equipes e com grandes orçamentos.

2.6.6 Outras considerações

Além de envolver todas as áreas da computação citadas anteriormente, há ainda outras categorias que se fazem importante no desenvolvimento de um jogo. Por exemplo, um jogo, dependendo do seu gênero, precisa ter uma história que prende o jogador. Para definir esses elementos, se faz necessário o uso de outras áreas que saem fora do contexto da computação, como por exemplo psicologia, história e estratégia (BRASILIENSE, 2006).

Ainda, de acordo com Brasiliense (2006), outro grande ponto que é um desafio para a equipe de desenvolvimento é tornar o jogo divertido. Isso se deve pelo fato de que, mesmo que todos os itens anteriores sejam feitos corretamente, isso não garantirá que o jogo será divertido. Para contornar esse desafio, é de grande valia que a equipe do jogo utilize *feedbacks* dos jogadores de acordo com o andamento do desenvolvimento do jogo, por meio de fóruns e lançamento de versões de testes, as chamadas versões *betas.*

No próximo item será descrito o funcionamento de um motor de jogo, com um aprofundamento no motor Unity3D, que será utilizado como base para as comparações dos algoritmos Dijkstra e A*.

2.7 MOTOR DE JOGO

Ao iniciar o processo de desenvolvimento de um jogo, muitas vezes é necessário um grande esforço da equipe de desenvolvimento com preocupações que vão além da projeção do jogo em si, como por exemplo, implementar códigos para manipulação de imagens, animações, gerência de memória, física, dentre outras funcionalidades que estão envolvidas indiretamente em jogos. Isso geralmente faz com que a equipe se atrase e acabe envolvendo muito tempo em atividades que não fazem parte do desenvolvimento do jogo em si.

Esse conjunto de funcionalidades que não envolvem diretamente o desenvolvimento do jogo podem serem abstraídos por um motor de jogo. Este, é um programa de computador projetado para abstrair funcionalidades do desenvolvedor oferecendo módulos e funcionalidades prontas e otimizadas para serem usadas pelo criador do jogo. Em essência, as características de um bom motor envolvem possuir um sistema de renderização 3D, simulador de física, suporte para uma ou mais linguagens de programação para a criação dos *scripts*, editor de cena integrado e capacidade de importação de dados como imagens, sons, modelos, entre outros. (PASSOS et al., 2009).

Um motor de jogo também possui como outra característica importante a de oferecer suporte para diversas plataformas, desde computadores e consoles como *Playstation, Xbox, Nintendo* até os dispositivos móveis mais recentes como Android, iOS, *Windows Phone*. (COUTINHO et al., 2014).

Um motor de jogo se constitui de diferentes componentes que geralmente são organizados de acordo com sua importância. Coutinho et al. (2014) cita os seguintes componentes de um motor como sendo os principais:

a) **núcleo do motor:** considerado o principal elemento de um motor de jogo, responsável por tratar renderização de imagens, manipular os elementos do jogo, interpretar e obedecer aos comandos do utilizador (comandos de entrada, como mouse e teclado), entre outros;
b) **kit de desenvolvimento do motor:** este se trata do código fonte do núcleo do motor, podendo ou não ser disponibilizado pelo desenvolvedor ou pela empresa desenvolvedora do motor. Alterações neste código poderá afetar o funcionamento do mesmo;

c) **editores de nível:** embutidos junto com o ambiente de desenvolvimento do motor, facilitam a criação e manipulação de ambientes do jogo, não apenas visualmente, mas também aos comportamentos destes ambientes quando se referindo às ações do utilizador;

d) **exportadores/importadores:** responsáveis por importar dados de outras aplicações, como modelos 3D, sons, materiais, texturas, dentre outros, e também por exportar informações para serem usadas por outros programas, como por exemplo, o executável do jogo para diferentes plataformas;

e) **inteligência artificial:** representam conjuntos de funções programadas pelo desenvolvedor que facilitam na criação da interatividade e inteligência dos elementos presentes no jogo.

Adicionalmente, um motor de jogo possui ainda o que pode ser considerado de sub-motores, sendo eles: o motor de renderização e o de física. O primeiro, diz respeito à forma com que os objetos visuais e suas informações são tratados na tela, sendo responsável por características como posicionamento, texturização, efeitos de luz, reflexão, amostragem dos elementos na tela, dentre outros. Já o segundo, é responsável por tratar da interação entre os objetos presentes em uma cena de um jogo, trazendo-os mais próximos da realidade com cálculos de física simulando velocidade, massa, inércia, gravidade, dentre outros. É responsável também por calcular as consequências de colisões entre objetos (COUTINHO et al. 2014).

Atualmente, o número deste tipo de *software* tem aumentado constantemente no mercado, justificando o fato de que mais jogos estão sendo desenvolvidos por mais profissionais se interessando pela área. Como exemplos de motores disponíveis no mercado, é possível citar alguns como: Unreal Engine, CryEngine, Construct 2, Cocos2D e Unity 3D. A seguir será descrito as principais funcionalidades do motor de jogo Unity 3D, uma vez que este será utilizado como motor base para a realização das comparações nesta pesquisa.

2.7.1 Unity 3D

O motor de jogo Unity 3D tem se tornado cada vez mais famoso entre os desenvolvedores de jogos, principalmente os chamados *indies,* termo este utilizado para denominar desenvolvedores independentes. A razão desta fama advém de vários fatores proporcionados pelo motor. A ferramenta dispõe de recursos para renderização de gráficos 2D e 3D, oferece as tecnologias mais atualizadas em relação à renderização de modelos 3D, imagens, texturas, materiais, partículas e iluminação. O motor conta ainda com um criador de terrenos em tempo real, motor de física, suporte para mais de uma linguagem de programação, áudio e rede, para criação de jogos *multiplayer* (XAVIER, 2011).

A Unity3D abstrai do desenvolvedor a necessidade de ter que utilizar bibliotecas como *DirectX* ou *OpenGL* (apesar deste ser possível, se necessário), que são bibliotecas gráficas de mais baixo nível em relação às linguagens suportadas pelo motor. Há ainda a possibilidade de se produzir *shaders,* utilizando a linguagem Cg da *Nvidia*. Conta com o motor de física *PhysX* também da *Nvidia*, e para a linguagem de *scripts*, utiliza uma implementação de código aberto do *framework* .Net da Microsoft (PASSOS, 2009).

Um outro ponto que chama a atenção é o fato de o motor possuir versão gratuita. Há três tipos de licença presentes na Unity, sendo elas a versão *Personal*, *Plus* e *Pro.* A versão *Personal* é para quem está aprendendo ou começando a desenvolver sem querer ter gastos, podendo inclusive publicar seus jogos sem ter custos. Já as versões *Plus* e *Pro* são para desenvolvedores que desejam mais recursos do motor, como o chamado Unity ADS, um serviço prestado pela desenvolvedora do motor onde a mesma oferece determinados recursos para os jogadores que assistirem um vídeo de publicidade, por exemplo, em troca de determinados recursos no jogo, garantindo assim uma receita mensal aos criadores do jogo (MARQUES, 2017).

Marques (2017) ainda cita como diferenciais presentes no motor em relação a outros, a sua simplicidade de uso, a possibilidade de utilizar sua versão gratuita e liberdade para comercializar jogos mesmo utilizando a versão gratuita, como citado anteriormente. Algumas características que destacam o motor em relação à outros estão relacionadas ao uso de materiais e *shaders*, a

compatibilidade de arquivos, as plataformas suportas, as linguagens de programação dos *scripts* além do motor de física.

2.7.1.1 Materiais e *shaders*

É notável que cada vez mais os novos lançamentos em jogos digitais contam com gráficos realísticos. E grande parte dos efeitos utilizados para dar impressão de realismo em jogos, advém dos *shaders. Shaders* são pequenos programas que atuam na placa de vídeo, ao invés de atuarem no processador, manipulando os pixels ou ainda o próprio modelo 3D para gerar filtros de imagens ou distorções na malha do objeto, por exemplo.

O motor Unity 3D permite a criação desses programas de forma facilitada, por meio de uma linguagem de *shader* própria chamada *ShaderLab*. Essa linguagem possui sintaxe simples e inclui algumas capacidades poderosas, como reuso de código, criação de texturas procedurais, entre outros (PASSOS, 2009).

A conexão entre um *shader* e um objeto 3D ocorre por meio de um material. Um material funciona como um container para as várias propriedades visuais que um objeto pode ter, e o ambiente de desenvolvimento do motor oferece recursos para criação de materiais de forma rápida e facilitada (PASSOS, 2009).

2.7.1.2 Importação e compatibilidade de arquivos

No processo de desenvolvimento de um jogo, diversos tipos e formatos de arquivos são utilizados para a sua construção. Geralmente, dentro da equipe de desenvolvimento, existem pequenas subequipes cuidando de diferentes reponsabilidades, como uma equipe de programação, de modelagem, de criação de áudios, etc. No meio deste processo todo, é essencial que o motor de jogo a qual a equipe utiliza, suporte diversos formatos de arquivos (XAVIER, 2011).

O motor oferece uma forma simplificada de importar recursos, bastando arrastar o arquivo para dentro do ambiente de desenvolvimento que este estará pronto para ser usado na cena do jogo. Alguns dos formatos mais populares e que são suportados pelo motor incluem arquivos (.fbx) para modelos 3D, (.wav, .mp3, .ogg) para áudio e (.jpg, .png, .bmp ou até mesmo .psd) para texturas e imagens. Há

ainda, mais formatos suportados, e uma lista deles pode ser obtida acessando a documentação do motor (PASSOS, 2009).

2.7.1.3 Plataformas suportadas

De acordo com a documentação do motor, a busca por suporte para novas plataformas está em constante avanço, e algumas das plataformas dentre as mais de 20 que já possuem suporte incluem computadores com sistema operacional *Windows, Linux ou Mac*, consoles como *Playstation 4, Xbox One, Nintendo 3DS, Wii U,* dispositivos móveis equipados com *Android*, *IOs, FireOS* além de oferecer suporte a realidade aumentada.

2.7.1.4 Linguagens de programação de *scripts*

O motor fornece três tipos de linguagens de programação para o desenvolvimento de *scripts* relacionados ao jogo sendo elas o *C#* ou C Sharp, *Javascript* e *Boo*. Um detalhe interessante sobre as linguagens, é a possibilidade de serem utilizadas todos os três tipos em um mesmo projeto, sendo possível então um *script* criado em *javascript* se comunicar com um *script* criado em C Sharp, o que traz outro ponto positivo ao motor, pois partes diferentes do jogo podem serem implementadas por diferentes programadores (XAVIER, 2011).

Passos (2009) chama a atenção a um outro detalhe referente à arquitetura dos *scripts*. Uma vez que os *scripts* no motor são acoplados aos objetos em cena, é possível projetá-los de maneira modular, facilitando a flexibilidade do código e o seu reuso.

2.7.1.5 Motor de física

O motor de física utilizado pela Unity 3D é o popular *PhysX*, desenvolvido pela *Nvidia*. O *PhysX* é responsável pelo tratamento de colisões e pelas fórmulas relacionadas à física incluindo corpos rígidos. Sendo considerado um dos mais completos motores de física do mercado, ele foi utilizado em alguns jogos popular como *Mass Effect* e *Medal of Honor: Airbone* (Passos 2009).

Uma das funcionalidades que o ambiente de desenvolvimento da Unity 3D oferece, é a possibilidade de editar graficamente as propriedades de física em um determinado objeto que possua anexado a si algum tipo de física, acessando a aba de propriedades de objetos.

2.7.1.6 Outras vantagens

Algumas outras vantagens descritas por Xavier (2011) incluem a possibilidade de desenvolvimento em tempo real, onde programadores e testadores podem atuar ao mesmo tempo, possibilidade de edição de arquivos e *scripts* ao mesmo momento em que o jogo está sendo executado, além de ser uma ferramenta de fácil aprendizagem e contar com uma extensa documentação em sua página web.

Outra vantagem tem a ver com seu grande crescimento, sendo hoje um dos motores de jogos mais conhecidos, principalmente pelo público que quer iniciar em desenvolvimento de jogos, além de contar com constantes atualizações que acompanham as mais atualizadas novidades relacionadas a gráficos.

2.8 CENÁRIOS DE JOGOS

Dentre as variadas etapas do processo de desenvolvimento de um jogo eletrônico, a forma como o cenário é mapeado receberá atenção especial nesta pesquisa.

Um dos principais objetivos ao se modelar um cenário de um jogo, é prender a atenção do jogador à história, dando relevância e beleza por meio dos objetos, sejam esses, construções, terrenos, etc., que representam a fantasia e que criam a imersão dentro do jogo. Além disso, a modelagem do cenário auxilia nos testes de colisão e na determinação da visibilidade, que ajuda no *pipeline* de renderização (SANTOS; SANTOS, 2009).

Adicionalmente, quando um jogo necessita do uso de algoritmos de busca de caminho, um outro ponto que ganha relevância é o mapeamento do cenário. O mapeamento do cenário determina quais objetos inseridos no mesmo possuirão colisão, se tornando um obstáculo, e ainda, determina as áreas que serão livres para o agente se movimentar. Deste modo, os algoritmos de busca de caminho realizam

uma leitura do cenário para determinar os locais ou pontos dentro do terreno que podem ser transitáveis, isto é, livre de obstáculos, e os pontos por onde não é possível se movimentar, onde obstáculos interferem na movimentação do agente.

Rabin (2012) descreve que para realizar a busca de caminhos, um agente ou o sistema de busca de caminhos precisa entender o nível, não necessitando de um modelo completo e detalhado, mas uma representação que leve em consideração as informações mais importantes e relevantes.

Para ser capaz de entender o nível, o cenário do jogo precisa ser mapeado. Dentre as formas de mapeamento de cenários, serão abordadas nesta pesquisa as seguintes: Mapeamento com base em grades ou *tiles*, por gráficos de *waypoints*, e por último, mapeamento baseado em malhas de navegação.

2.8.1 Mapeamento de cenário baseado em grades

O mapeamento de um cenário utilizando grades, ou *tiles* como também são chamados, é uma forma simples e intuitiva de representar o nível. Jogos como *Age of Empires* e *Warcraft III* utilizam este tipo de representação em seus cenários (RABIN, 2012).

O mapeamento em grades ou *tiles* possibilita a criação de níveis dividindo a imagem de uma cena em uma grade regular, onde os elementos que serão repetidos ao longo da grade, são carregados apenas uma vez, e em seguida replicados nas suas devidas posições (SANTOS; SANTOS, 2009).

Quando se desenvolve um jogo baseado em grade, cada célula presente nesta grade tem um significado diferente. Uma pode representar o chão, outra, uma rocha sólida e assim em diante. Neste tipo de jogo, os objetos ou o personagem são movidos pelo jogador ou pelo computador a partir da grade de representação do cenário (PAZERA, 2001).

A técnica foi inicialmente desenvolvida para o uso em jogos bidimensionais, considerando o fato de que a grade é representada por vetor bidimensional, ou seja, cada célula da grade é representada por uma determinada estrutura contendo a sua posição em x e y no cenário. Porém, ainda de acordo com Rabin (2012), isto não significa que este tipo de mapeamento não pode ser usado em jogos tridimensionais. Jogos como *Warcraft III* em que a elevação do cenário não interfere na posição do agente, pode ter seu mapa de jogo reduzido a uma

representação bidimensional, mesmo que os objetos do cenário sejam tridimensionais, permitindo que esta técnica possa ser aplicada sem problemas.

A figura 1 demonstra um cenário construído tridimensionalmente aplicando a técnica de grade para o mapeamento dos objetos no cenário.

Figura 1 – Representação de cenário tridimensional em grade

Fonte: Cui e Shi (2011)

A técnica de grade possui algumas vantagens, uma delas é que, dada uma posição no cenário, rapidamente é possível encontrar a célula que a representa. Outra vantagem é o fato de que a partir de uma célula, é possível acessar as células vizinhas com facilidade (RABIN, 2012).

O tratamento de colisão em um sistema de grades ocorre individualmente, ou seja, cada célula possui ou não possui colisão, e a detecção ocorre quando um agente intercepta os limites de outra célula. Essa estratégia de colisão pode apresentar inconsistências com a realidade em algumas ocasiões. Um exemplo disso seria um obstáculo baixo, como uma pequena caixa, o jogador espera poder atravessá-la, mas neste caso, se a célula em que a caixa se encontra estiver marcada como obstáculo, a movimentação através do objeto não será permitida (SANTOS; SANTOS, 2009).

Assim sendo, o algoritmo de busca de caminho determina a célula em que o agente está, e a célula a qual se deseja chegar. Em seguida o algoritmo atua coletando as informações de cada célula vizinha individualmente, analisando se esta

é um obstáculo ou não, fazendo isso repetidamente, gerando um único caminho até a célula destino, considerado o melhor.

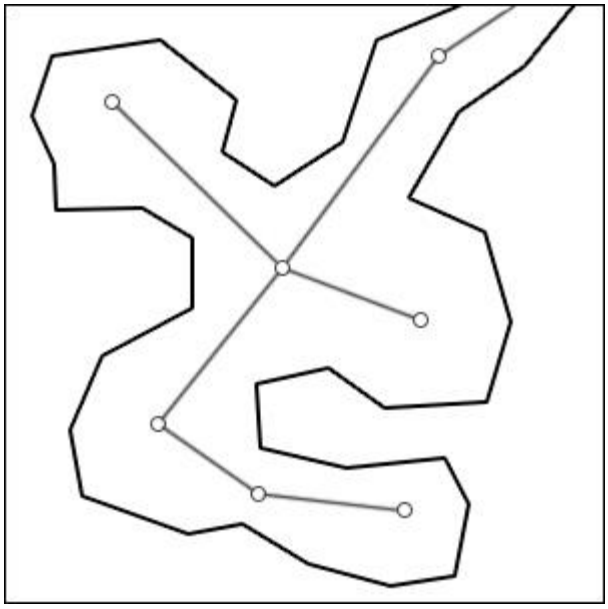
2.8.2 Mapeamento por gráficos de *waypoints*

Os gráficos de *waypoints* foram utilizados em grande escala antes do surgimento e popularização das malhas de navegação, que será descrita no item 1.9.3. Eles representam o mundo como um gráfico abstrato, e não possuem um mapeamento explícito entre os nós no gráfico e o espaço a percorrer (RABIN, 2014).

Neste esquema, a navegação é realizada mapeando-se pontos de visibilidade ou *waypoints*, inseridos nas partes mais importantes do cenário. Cada ponto necessita ter ao menos uma conexão ou *link*, como também pode ser chamado, direta com um outro ponto do cenário. Um *link* conecta exatamente dois pontos entre si, indicando que um agente pode se movimentar sem encontrar grandes obstáculos entre esses dois pontos (RABIN, 2012).

Na figura 2, é demonstrado parte de um cenário mapeado com pontos inseridos nos locais definidos como mais importantes para a navegação.

Figura 2 – Parte de cenário mapeado com *waypoints*



Fonte: Do autor

Para encontrar um caminho neste tipo de mapeamento, um algoritmo procura o ponto mais próximo da posição do agente, e o ponto mais próximo da posição a que se quer chegar. Em seguida, o algoritmo de busca de caminho

procura por um caminho entre o ponto de partida e o ponto de chegada, sempre percorrendo os outros pontos localizados no cenário, e então, move o agente.

O conjunto de *waypoints* de um cenário pode ser demarcado manualmente pelos desenvolvedores, ou por um algoritmo que insira o conjunto automaticamente. É comum em jogos de estratégia em tempo real os desenvolvedores deixarem livre a opção de o jogador ser capaz de adicionar pontos nos cenários dinamicamente, para calcular uma determinada rota em tempo de jogo. (PATEL, 2017).

Uma das vantagens dos *waypoints* em relação ao mapeamento por grades, descrito no tópico anterior, é que, enquanto uma célula em um sistema de grades pode ser tipicamente ligada à oito células vizinhas, um ponto no sistema de *waypoints* pode ser ligado a quaisquer outros pontos, tornando este sistema mais flexível que as grades (RABIN, 2012).

Mesmo com a vantagem citada acima, isso não significa que esta técnica é superior à técnica de grade, os desenvolvedores devem sempre analisar qual serão as necessidades do cenário em relação à movimentação. No próximo item, será descrito o mapeamento por malhas de navegação. Dentre os três tipos demonstrados, esta técnica é a mais atual lançada e já conquistou um espaço grandioso entre os desenvolvedores, pela simplicidade e nível de detalhamento que a mesma proporciona.

2.8.3 Mapeamento baseado em malhas de navegação

A malha de navegação é uma técnica para representar o cenário de um jogo usando polígonos. Devido à sua simplicidade e alta eficiência na representação do ambiente, a malha de navegação tornou-se uma escolha dominante para jogos tridimensionais (CUI; SHI, 2012, tradução nossa).

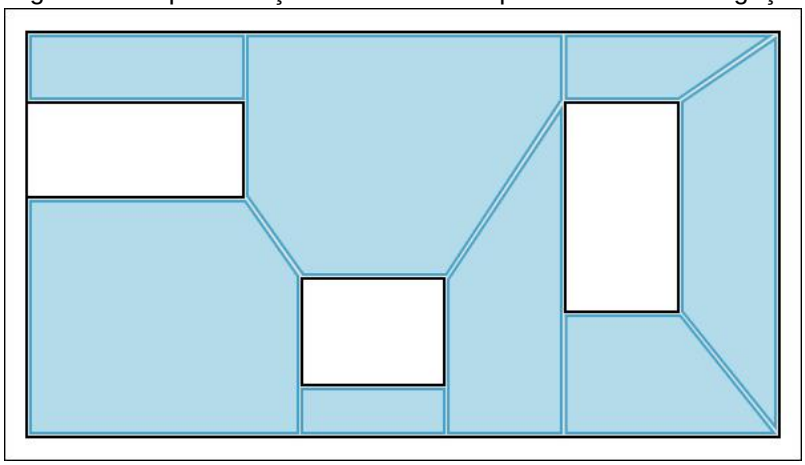
A representação cobre apenas as partes do cenário navegáveis (livre de obstáculos) no mundo do jogo, com polígonos convexos. Os polígonos convexos garantem que qualquer linha direta formada de um ponto a outro dentro de um mesmo nó (polígono), seja um caminho válido (CANSIAN, 2011, tradução nossa).

As malhas de navegação possuem várias semelhanças em relação aos gráficos de *waypoints*. O principal aspecto que diferem ambas, é que, ao contrário dos gráficos de *waypoints* que representam cada nó como um ponto no cenário, nos

mapas baseados em malhas cada nó é representado por um polígono convexo (RABIN, 2012).

A figura 3 demonstra um cenário simples representado por meio de malhas de navegação. Cada polígono, representado pela cor azul, se conecta a outros por meio de suas bordas e os retângulos em cor branca representam os obstáculos da cena.

Figura 3 – Representação de um cenário por malhas de navegação



Fonte: Do autor

Outra diferença importante a ser mencionada segundo Rabin (2012), é o fato de que, ao contrário das grades e malhas de navegação, um mapeamento baseado em *waypoints* não especifica as áreas transitáveis e intransitáveis. Assim, um gráfico de *waypoints* é considerado uma representação incompleta do cenário, ao contrário da representação em malha, que é capaz de criar uma representação completa e detalhada do terreno.

Inicialmente, é necessário a geração da malha de navegação. Ela pode ser gerada manualmente, ou por um algoritmo, e os nós da malha precisam ser formados por triângulos ou polígonos convexos.

Após a geração dos polígonos, o cenário estará preparado para ser usado pelo algoritmo de busca de caminho, que irá calcular e retornar uma determinada rota válida entre os pontos de início e destino. Cansian (2011, tradução nossa) descreve ainda que, normalmente neste processo, inicia-se descobrindo em qual polígono a posição inicial do agente e a posição destino se encontram, podendo ocorrer três situações:

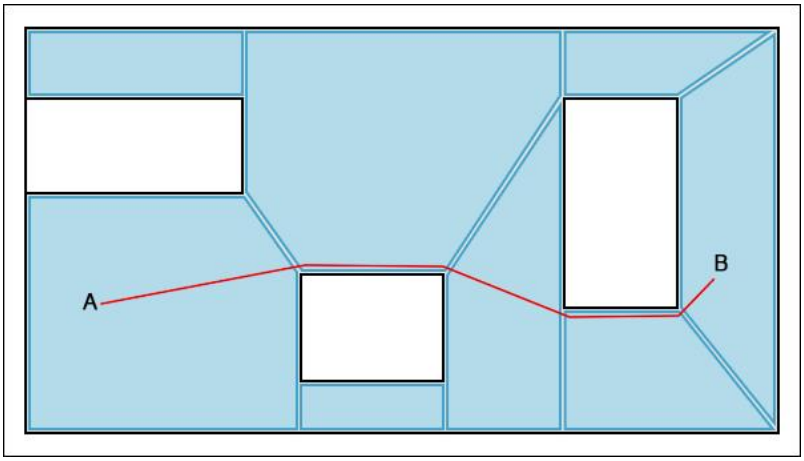
a) **ambos os pontos estão localizados no mesmo polígono:** isto significa que o caminho pode ser representado por uma linha reta direta conectando os dois pontos, e este será o melhor caminho;

b) **um ou mais pontos está(ão) localizado(s) fora de todos os polígonos:** nesta situação, é recomendado que o processo de busca de caminho seja interrompido e o algoritmo informe que não há um caminho válido;

c) **os pontos estão localizados em polígonos diferentes:** esta é a situação que normalmente ocorre, e neste caso o algoritmo deverá prosseguir com o processo de busca de caminho.

O algoritmo de busca de caminho então inicia os cálculos para a geração da rota, percorrendo e validando cada polígono, semelhante ao processo de geração de rota do mapeamento por *waypoints.* A figura 4 demonstra o resultado final deste processo, o ponto A e B representam o ponto de partida e o ponto de chegada, respectivamente, e a linha em vermelho representa a rota gerada.

Figura 4 – Rota gerada com base em uma malha de navegação



Fonte: Do autor

Há ainda, a possibilidade de suavizar a rota tornando-a mais curta e direta, utilizando-se de algoritmos de suavização de rota. Como um exemplo destes algoritmos pode-se citar o *Simple Stupid Funnel Algorithm,* desenvolvido pelo programador líder de Inteligência Artificial do jogo Crysis, mas este não será foco de estudo nesta pesquisa.

A diferença entre os mapeamentos por *waypoints* e por malhas de navegação se torna mais explícita a partir da suavização de rota. Segundo Kalani

(2008, tradução nossa) mapas de *waypoints* não contam com nenhuma informação do cenário por fora da rota, isto é, o agente só possui conhecimento do cenário enquanto permanece dentro dos pontos que estão conectados diretamente por *links*, e uma tentativa de suavização da rota poderia fazer com que o mesmo caísse em uma parte do cenário ou colidisse com um obstáculo, ficando trancado e impossibilitando-o de continuar sua movimentação.

Adicionalmente, Kalani (2008, tradução nossa) descreve que este problema não ocorre quando se utiliza malhas de navegação, considerando o fato de que nesta técnica, cada nó é um polígono, e o agente pode se movimentar para qualquer local dentro deste polígono, desde que não ultrapasse os limites de suas bordas. A diferença em questão é ilustrada na figura 5.

Figura 5 - Suavização de rota em mapeamento por *waypoints* e malha de navegação

Fonte: Kalani (2008)

Ainda na figura 5, o mapeamento por *waypoints* é demonstrado à esquerda, e a representação por malha de navegação, à direita. Na representação por *waypoints*, o agente é obrigado a permanecer na rota conhecida, diferentemente da representação por malha, onde o agente pode se movimentar livremente, desde que permaneça dentro de um polígono.

3 PROBLEMAS DE CAMINHO

No problema do caminho mínimo, o objetivo consiste em localizar o melhor caminho entre dois pontos distintos, ou nós, como também são chamados, em um determinado terreno, podendo este ser desde um mapa geográfico do mundo real, até cenários virtuais de jogos digitais.

Um caminho é uma lista de células, pontos ou nós que um agente tem de percorrer a fim de se chegar em uma posição final, partindo de uma posição inicial. Na maioria das situações, um grande número de caminhos diferentes pode ser tomado para alcançar a meta (RABIN, 2012).

3.1 PROBLEMAS DE CAMINHO MÍNIMO EM JOGOS

Em jogos digitais, é muito comum ocorrerem situações envolvendo busca de caminhos. Nos seus diversos gêneros, principalmente em jogos de estratégia em tempo real, onde o número de agentes e rotas são maiores, é preciso estabelecer trajetórias para guiar os *Non-Player Characters* (NPCs), termo este designado a uma entidade do jogo que é controlada pelo computador, e não pelo jogador, através do ambiente. Nestes tipos de jogos se faz necessária a solução do problema da busca de caminhos. Um exemplo de utilização seria o famoso jogo Pacman, em que os fantasmas poderiam utilizar-se de um algoritmo de busca de caminhos para calcular a rota repetidamente em direção à posição do player (OSÓRIO et al., 2007).

Segundo Rabin (2012) a busca de caminhos envolve uma estrutura complexa de tarefas, tais como análise do nível de jogo e seus obstáculos, cálculo de trajetórias, busca e identificação de outras unidades e do jogador, segmento de um alvo, sistemas de detecção de colisão, dentre outros.

Em muitos jogos é necessário simular milhares ou milhões de agentes interagindo com o ambiente, e dentre essas interações, a navegação é de grande importância, pois é por meio desta que os agentes podem perseguir, procurar ou interceptar outros agentes no cenário do jogo. A busca de caminho fornece aos personagens a capacidade de navegar em um ambiente de forma autônoma (PELECHANO; FUENTES, 2016, tradução nossa).

Se um agente não pode encontrar seu caminho no nível de jogo, ele vai parecer completamente incompetente. A busca de caminhos não é uma tarefa trivial.

Isso ocorre em parte porque os recursos que um sistema de busca de caminhos consome podem rapidamente sair do controle (RABIN, 2012).

Uma outra questão importante a se tratar sobre problema de caminho em um jogo, é a necessidade do algoritmo agir de forma rápida e performática. Vermette (2011), descreve que a necessidade de técnicas altamente eficientes é evidente à medida que os jogos modernos exigem altas demandas de processamento e memória, além da necessidade de tempo para renderizar gráficos, simulação de física, renderização de objetos e texturas do cenário, cálculos de iluminação, *scripts* de comportamento do próprio jogo, entre outras tarefas, tudo isso sendo realizado em tempo real. Nesta situação, qualquer atraso na execução do algoritmo, impacta diretamente a experiência do jogador, podendo causar queda nos *frames* e até curtos travamentos.

3.2 ALGORITMOS DE BUSCA DE CAMINHO MÍNIMO

Durante muitos anos, o problema do caminho mínimo tem sido um assunto que gerou pesquisas intensivas, o que levou ao surgimento de muitas técnicas e algoritmos com a proposta de solução para este problema (SUN-GI; SUNG-WOO; JU-JANG, 1995, tradução nossa).

O problema possui uma estrutura particularmente simples, o que permitiu aos pesquisadores desenvolver vários algoritmos intuitivamente atraentes para solucioná-lo (AHUJA, 1993, tradução nossa).

Dentre vários algoritmos que foram desenvolvidos ao longo do tempo, alguns ganharam mais destaque em relação a outros e ficaram mais conhecidos, como o algoritmo de Floyd-Warshall, Dijkstra, Bellman-Ford, Johnson e A*.

Um dos critérios importantes de um algoritmo de busca de caminhos é a qualidade do caminho que ele encontra. Alguns algoritmos garantem encontrar um caminho mais adequado, enquanto outros não garantem até mesmo encontrar um caminho. Os algoritmos que garantem encontrar um caminho são chamados de algoritmos completos, e algoritmos que garantem sempre encontrar o caminho mais adequado são conhecidos como algoritmos ótimos (RABIN, 2012).

A seguir, será demonstrado de forma detalhada, o funcionamento dos algoritmos Dijkstra e A*, algoritmos estes utilizados para a realização das comparações desenvolvidas neste trabalho, e que, de acordo com Cui e Shi (2011)

são dois algoritmos clássicos de busca de caminho em grafos ponderados, que podem encontrar o melhor caminho entre um nó para qualquer outro nó.

3.2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra foi introduzido pelo cientista da computação Edsger Dijkstra em 1956, e publicado em 1959, com o objetivo de calcular o menor caminho em uma rede, partindo de um nó inicial e calculando as menores distâncias para todos os outros nós presentes na rede (REDDY et al., 2015).

Tendo escolhido um ponto, ou nó, como origem, o algoritmo calcula o menor caminho para todos os outros nós presente no grafo, partindo de uma estimativa inicial para cada nó. Conforme a iteração acontece entre esses nós, o algoritmo vai ajustando essa estimativa, sempre com o objetivo de diminuir o custo de navegação entre um ponto e outro (SILVA; SANCHES, 2009).

3.2.1.1 Descrição do algoritmo de Dijkstra

De acordo com Qing, Zheng e Yue (2017, tradução nossa), o algoritmo de Dijkstra inicia a pesquisa pelo menor caminho através do nó inicial e itera entre seus adjacentes, como dito anteriormente. O nó de destino, assim como todos os outros nós presentes no grafo, também é interpretado como um nó intermediário. O algoritmo realiza a busca verificando todos os nós adjacentes ao atual nó intermediário, sempre selecionando como próximo nó atual, o que possui menor custo de trajeto, e continua realizando esta tarefa até que todos os nós do grafo sejam visitados.

A complexidade do algoritmo de Dijkstra é $O(n^2)$ operações, onde n representa o número de nós do grafo, para determinar o caminho mais curto entre dois nós em um grafo (CARDOSO, 2009).

A figura 6 representa a formalização do algoritmo de Dijkstra.

Figura 6 – Pseudocódigo algoritmo Dijkstra

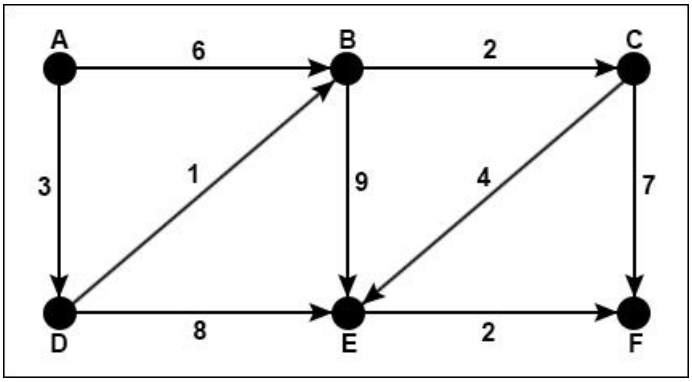
Algoritmo 1 Algoritmo Dijkstra

```
for i = 1 to n do
    L(v_i) ← ∞
end for
L(a) ← 0
S ← ∅
while z ∉ S do
    u ← um vértice não pertencente a S com L(u) mínimo
    S ← S ∪ {u}
    for todos os vértices v ∉ S do
        if L(u) + w(u, v) < L(v) then
            L(v) ← L(u) + w(u, v)
        end if
    end for
end while
```

Fonte: Cardoso (2009).

Na figura 7, está sendo ilustrado um grafo ponderado. Um grafo é dito ponderado quando todas as suas arestas possuem anexadas a si, seus devidos pesos (Cardoso, 2009). Este grafo será utilizado para a demonstração do algoritmo em sua execução. O objetivo da busca será partir do nó A, e procurar pelo menor caminho até o ponto F, definido como ponto de destino para este caso.

Figura 7 – Grafo para demonstração Dijkstra



Fonte: Do autor

Na sua primeira iteração, o algoritmo de Dijkstra analisa todos os nós adjacentes ao atual, que nesta iteração é o nó com o rótulo A. Um nó é dito adjacente a outro nó, quando estes possuem uma aresta os conectando diretamente. Na figura 7, os nós adjacentes ao nó A, são os nós B e D.

A figura 8 representa uma tabela que será preenchida conforme as iterações do algoritmo.

Figura 8 – Algoritmo Dijkstra valores iteração nó A

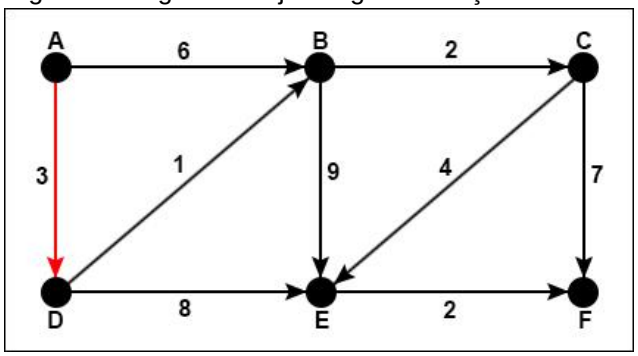
Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)					
B	(6, A)					
C	-					
D	(3, A)					
E	-					
F	-					

Fonte: Do autor

Na figura 8, os custos iniciais são preenchidos pelo algoritmo, onde os custos para ir do nó atual até ele mesmo é zero (0, A), e os custos para ir ao nó B e D são (6, A) e (3, A), respectivamente. Como os nós C, E e F não são adjacentes ao nó atual (A), eles não são preenchidos com nenhum valor. O custo do nó A está em coloração vermelha a fim de representar que o menor caminho para aquele nó já foi encontrado.

O algoritmo selecionará o próximo nó que possuir menor custo de trajeto, com exceção do nó a qual já teve seu menor caminho encontrado (A). De acordo com a figura 8, o nó D possui o menor custo (3, A). O algoritmo então o seleciona como nó atual e define que o menor caminho até o mesmo foi encontrado. A figura 9 demonstra a situação do grafo para a primeira iteração.

Figura 9 – Algoritmo Dijkstra grafo iteração nó D



Fonte: Do autor

A partir do nó atual, ele fará o mesmo processo descrito anteriormente, analisará os nós adjacentes ao nó D, sendo os nós A, B e E. Como o nó A já foi avaliado, serão avaliados os nós B e E. Partindo do nó atual (D), o caminho para chegar até o nó B, tem custo 1. Este custo será somado ao valor gasto para ir do

anterior nó atual (A), até o nó atual (D). Sendo assim, como o custo para chegar de A até D é 3, o custo de D até B será 4. Por fim, para o custo do nó D ao E, será tido como valor o custo atual de D somado ao custo da aresta que conecta os nós D e E.

A figura 10 demonstra a tabela com os valores atualizados para a iteração descrita.

Figura 10 – Algoritmo Dijkstra valores iteração nó D

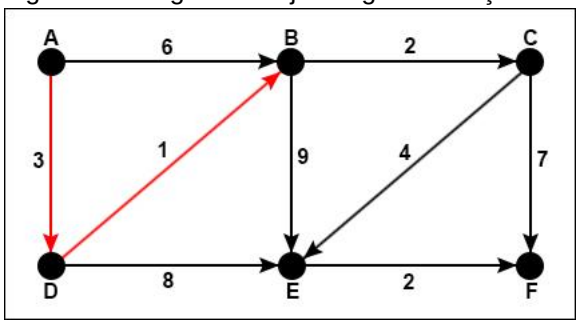
Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)	-	-	-	-	-
B	(6, A)	(4, D)				
C	-	-				
D	(3, A)	(3, A)				
E	-	(11, D)				
F	-	-				

Fonte: Do autor

Na figura 10, o valor do custo 2 para o nó B foi atualizado de (6, A) para (4, D), o que significa que o menor caminho para chegar ao nó B será através do percurso A – D – B, ao invés de ir para o nó B diretamente pelo nó A.

O próximo nó a ser escolhido como atual será o nó B, uma vez que entre os valores (4, D) e (11, D), o valor (4, D) é menor. Nesta iteração, o nó B é fechado e o menor caminho para chegar até este nó, será seguindo o percurso A – D – B, como descrito anteriormente. A figura 11 demonstra a atual situação do grafo.

Figura 11 – Algoritmo Dijkstra grafo iteração nó B



Fonte: Do autor

Tendo o nó B como atual, o algoritmo analisará todos os nós adjacentes a B, sendo eles C e E. Tomando o nó C primeiramente, o custo para atingi-lo será o custo obtido para chegar até B, somado ao valor do custo da aresta que conecta B e

C, sendo este, 2. Logo, o valor para chegar a C será 6, vindo de B (6, B). Para o nó E, o mesmo processo é feito, resultando em 13 vindo de B (13, B).

A figura 12 demonstra a tabela com os valores da iteração 3.

Figura 12 – Algoritmo Dijkstra valores iteração nó B

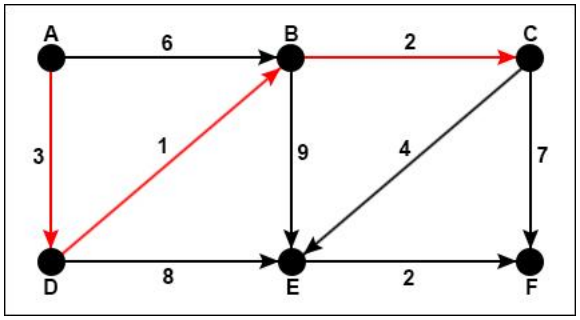
Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)	-	-	-	-	-
B	(6, A)	(4, D)	(4, D)			
C	-	-	(6, B)			
D	(3, A)	(3, A)	-			
E	-	(11, D)	(13, B)			
F	-	-	-			

Fonte: Do autor

Próxima iteração, o algoritmo selecionará novamente o nó com menor custo de trajeto, sendo o nó C (6, B). A partir do nó C, verificará os nós E e F, que são seus nós adjacentes. Para o custo do nó E, será somado o valor gasto para chegar até o nó atual (C), que equivale a 6, somado ao custo da aresta que conecta os nós C e E, possuindo valor 4. Logo, o custo para atingir o nó E a partir de C será 10. Para o nó F, o custo obtido será 13.

Na figura 13, está representado o grafo e o melhor caminho encontrado até a iteração do nó C.

Figura 13 – Algoritmo Dijkstra grafo iteração nó C



Fonte: Do autor

Em sequência está sendo demonstrado a tabela para a iteração referente ao nó C, na figura 14.

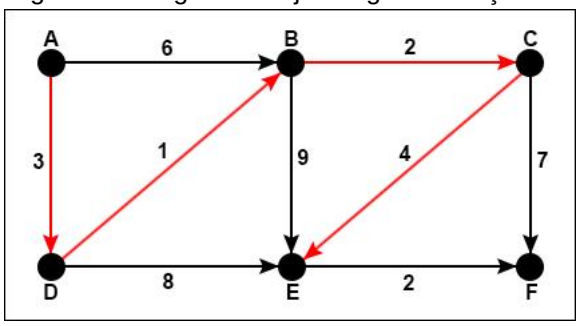
Figura 14 – Algoritmo Dijkstra valores iteração nó C

Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)	-	-	-	-	-
B	(6, A)	(4, D)	(4, D)	-		
C	-	-	(6, B)	(6, B)		
D	(3, A)	(3, A)	-	-		
E	-	(11, D)	(13, B)	(10, C)		
F	-	-	-	(13, C)		

Fonte: Do autor

Em sequência, o algoritmo seleciona o nó E, que possui menor custo entre (10, C) e (13, C). A figura 15 abaixo, representa a atual situação da busca, demonstrando o caminho já percorrido.

Figura 15 – Algoritmo Dijkstra grafo iteração nó E



Fonte: Do autor

A partir do nó E, selecionado como atual, o único nó adjacente é o nó F, o algoritmo então realiza a soma do custo já acumulado para chegar até o nó atual (10) com o valor gasto para percorrer o nó F a partir do nó atual (2), resultando em 12. Na figura 16 encontram-se os valores atualizados da iteração do nó E.

Figura 16 – Algoritmo Dijkstra valores iteração nó E

Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)	-	-	-	-	-
B	(6, A)	(4, D)	(4, D)	-	-	
C	-	-	(6, B)	(6, B)	-	
D	(3, A)	(3, A)	-	-	-	
E	-	(11, D)	(13, B)	(10, C)	(10, C)	
F	-	-	-	(13, C)	(12, E)	

Fonte: Do autor

Ao atingir o nó F, o algoritmo de Dijkstra chega em sua última iteração. Selecionando o único nó restante como atual (F), e não tendo mais nó adjacente para verificar, o algoritmo conclui sua busca tendo encontrando o menor caminho a partir do ponto A utilizando o percurso A – D – B – C – E – F.

A figura 17 demonstra o valor final da tabela de busca do algoritmo Dijkstra.

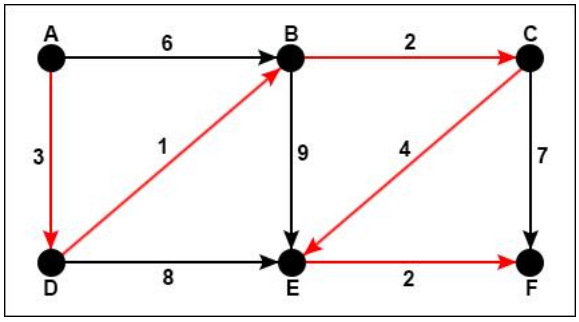
Figura 17 – Algoritmo Dijkstra grafo iteração nó F (final)

Nó	Custo 1	Custo 2	Custo 3	Custo 4	Custo 5	Custo 6
A	(0, A)	-	-	-	-	-
B	(6, A)	(4, D)	(4, D)	-	-	-
C	-	-	(6, B)	(6, B)	-	-
D	(3, A)	(3, A)	-	-	-	-
E	-	(11, D)	(13, B)	(10, C)	(10, C)	-
F	-	-	-	(13, C)	(12, E)	(12, E)

Fonte: Do autor

O percurso final realizado pelo algoritmo no grafo é ilustrado na figura 18, o caminho em questão está destacado em coloração vermelha.

Figura 18 – Algoritmo Dijkstra grafo iteração nó F (final)



Fonte: Do autor

No próximo item do trabalho, será explanado sobre o algoritmo A*, sua definição e implementação, de forma similar à explicação do algoritmo de Dijkstra.

3.2.2 Algoritmo A*

O algoritmo A*, assim como o Dijkstra, também encontra um caminho entre dois pontos distintos em um determinado mapa. Embora haja diferentes tipos

de algoritmos que realizam este mesmo trabalho, o A* encontrará o caminho mais curto entre dois pontos, se houver um caminho, e fará isso relativamente rápido (RABIN, 2002, tradução nossa).

Adicionalmente, de acordo com Rabin (2002, tradução nossa), este algoritmo é considerado direcionado, ou seja, ao decorrer da busca, utilizando-se de heurísticas, o algoritmo avalia a melhor direção para ir ao encontro do local de destino, não gastando tempo avaliando todas as direções possíveis, o que o torna um algoritmo bastante flexível.

O algoritmo A* combina partes de informações dos algoritmos Dijkstra, quando usa informações dos nós mais próximos ao nó de partida, e do algoritmo *Best-First-Search*, ao utilizar informações dos nós mais próximos ao nó de destino, estabelecendo um balanço entre essas duas informações à medida que se move entre o ponto de partida e o ponto de destino, para realizar a busca seguindo uma direção, chamada de heurística. Por esta razão, pode se dizer que o A* é uma combinação dos algoritmos Dijkstra e *Best-First-Search* (PATEL, 2017).

3.2.2.1 Descrição do algoritmo A*

> O algoritmo A* tenta minimizar o custo total da solução usando uma função de avaliação para escolher um nó vizinho do espaço de estados de menor valor para ser expandido. Essa função de avaliação visa prever a distância deste nó até o objetivo. No A*, a função de avaliação é definida por f(n) = g(n) + h(n), onde g(n) é a distância real entre o nó origem e o nó candidato à expansão e h(n) (também chamada de função heurística) é um custo estimado do nó candidato até o nó destino (BASTOS; JAQUES, 2010, p. 43).

O algoritmo também mantém um ponteiro normalmente chamado de pai, para cada nó presente no grafo. Este ponteiro é usado para extrair o percurso da busca após a finalização da mesma. Essa extração é realizada acessando, de forma repetida, o pai de cada nó, a partir do nó de destino até atingir o nó inicial (RABIN, 2015, tradução nossa).

Adicionalmente, de acordo com Rabin (2015, tradução nossa) o algoritmo mantém duas listas: a *Open List* e a *Closed List*. A *Open List* é uma fila prioritária

contendo todos os nós que ainda não foram expandidos, e a *Closed List*, armazena os nós no qual o algoritmo já expandiu e consequentemente, já visitou.

A figura 19 representa a implementação do algoritmo A* em pseudocódigo.

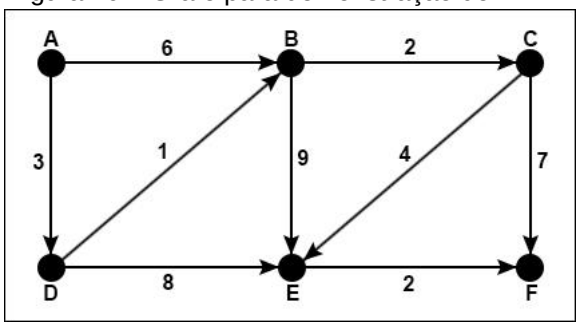
Figura 19 – Pseudocódigo algoritmo A*

```
1  Main()
2      open:=closed:=∅;
3      g(S_start):=0;
4      parent(S_start):=S_start;
5      open.Insert(S_start,S_start)+h(S_start));
6      While open≠∅ do
7          s:=open.Pop();
8          if s=s_goal then
9              return "path found";
10         closed:=closed ∪{s};
11         foreach s'∈neighbor_vis(s) do
12             if s'∉closed then
13                 if s'∉open then
14                     g(s'):=∞;
15                     parent(s'):=NULL;
16                 UpdateVertex(s,s');
17     return "no path found";
18 end
19 Update Vertex(s,s')
20     g_old:=g(s');
21     ComputeCost(s,s');
22     if g(s')<g_old then
23         if s'∈open then
24             open.Remove(s');
25         open.Insert(s',g(s')+h(s'));
26 end
27 ComputeCost(s,s')
28     /* Path 1 */
29     if g(s)+c(s,s')<g(s') then
30         parent(s'):=s;
31         g(s'):=g(s)+c(s,s');
32 end
```

Fonte: Rabin (2015).

A fim de demonstrar o processo utilizado pelo algoritmo A* para encontrar o menor caminho entre dois nós de um grafo, será realizada uma simulação de execução do mesmo. Para isso, será feito o uso do mesmo grafo utilizado na explanação do algoritmo de Dijkstra, cujo modelo está demonstrado na figura 20.

Figura 20 – Grafo para demonstração do A*



Fonte: Do autor

O propósito do algoritmo será descobrir o menor caminho tendo como nó inicial, o nó A, e F como nó de destino, assim como foi demonstrado no algoritmo de Dijkstra.

Como descrito anteriormente, o algoritmo A* utiliza uma heurística, onde realiza o cálculo do custo gasto para chegar a cada nó, somado ao custo da estimativa do nó atual até o nó de destino.

Para os custos de estimativa (h) de cada nó do grafo ilustrado na figura 20, foram usados os valores demonstrados na figura 21.

Figura 21 – Heurística dos nós

Heurística estimada	
Nó	h
A	10
B	5
C	5
D	8
E	1
F	0

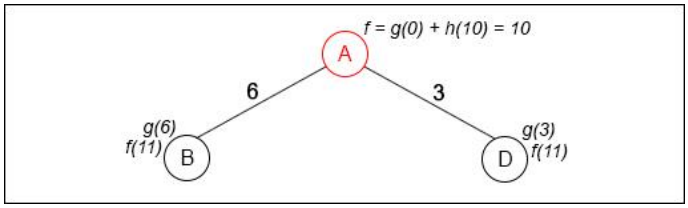
Fonte: Do autor

Os valores da figura 21 foram estimados com base em uma distância em linha reta de cada nó, até o nó de destino. Há outros métodos para determinar estes valores, podendo-se utilizar de valores do mundo real dependendo da situação, um exemplo seria em um grafo entre cidades, neste, a estimativa poderia ser considerada baseando-se em uma média dos limites de velocidade máxima de cada rodovia.

Na primeira iteração do algoritmo A*, o nó A, definido como nó inicial, será também definido como nó atual, expandido e então será calculado o custo do caminho para chegar aos seus nós adjacentes, B e D. Para atingir o nó B, o custo gasto será 0 para o valor de g, uma vez que, por estar no nó inicial, não foi percorrido nenhum caminho ainda, somado à estimativa h de B, e não de A, pois o algoritmo está calculando o valor para atingir B, que equivale a 5.

A figura 22 ilustra o processo descrito no parágrafo anterior. Ao lado superior de cada nó, encontra-se o seu valor de g e f. Os nós do grafo em coloração vermelha representam nós que já foram analisados.

Figura 22 – Algoritmo A* iteração nó A



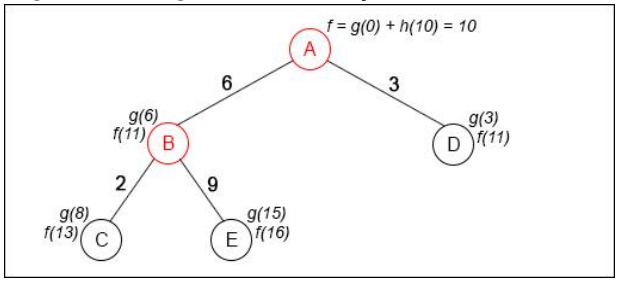
Fonte: Do autor

Em sequência, o algoritmo seleciona o nó com menor custo f. Na figura 22, ambos os nós possuem o mesmo custo f, então será escolhido aleatoriamente o nó B para a próxima iteração.

O nó B é atribuído como nó atual, e a partir dele, são analisados seus nós adjacentes, C e E. O custo para chegar ao nó C, a partir do então nó atual, será o valor g do nó B que vale 6, somado ao custo da aresta que conecta os nós B e C, sendo este valor, 2. Para o valor da heurística (f), será somado o valor total do custo g do nó C (8) com o valor de estimativa h do nó C (5) resultando em 13.

O mesmo processo é feito para o nó E, sendo ilustrado o resultado desta iteração na figura 23.

Figura 23 – Algoritmo A* iteração nó B



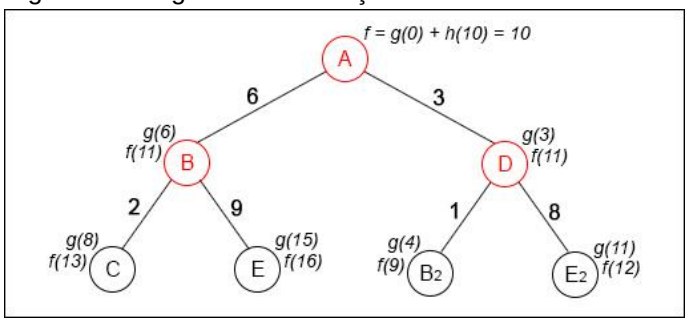
Fonte: Do autor

Dentre os nós que ainda faltam ser expandidos (C, E e D), o algoritmo A* selecionará o nó com menor custo f, sendo este, o nó D.

O nó D é então atribuído como nó atual, e seus nós adjacentes terão seus valores de custo calculados. O primeiro nó adjacente (B), terá 4 como custo g, e para o valor de f, será somado o custo g com o valor h do nó B, sendo este, 5, resultando em um valor f de 9. Para o segundo nó adjacente (E), g terá valor 11 e f, 12.

A figura 24 ilustra o nó D após ser expandido e calculado os valores de seus nós adjacentes.

Figura 24 – Algoritmo A* iteração nó D

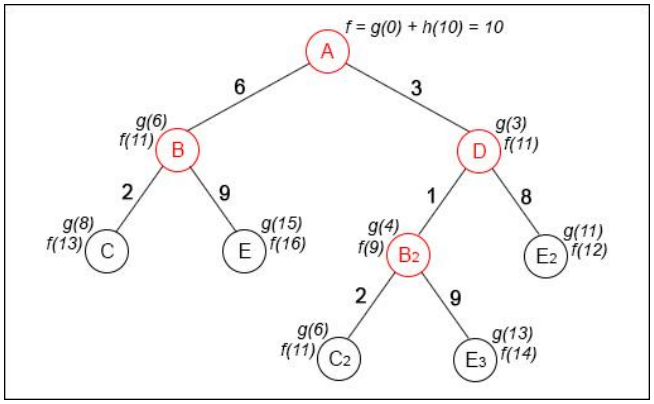


Fonte: Do autor

O próximo nó a ser selecionado será o nó B, por possuir o menor custo de f. O nó B já se encontra na lista fechada, porém como um dos seus nós adjacentes conseguiu melhorar seu caminho, ele precisará ter seus custos recalculados.

O nó B é então tomado como nó atual, e expandido novamente. A figura 25 ilustra os custos resultantes do trajeto até seus nós adjacentes, C e E.

Figura 25 – Algoritmo A* iteração nó B2



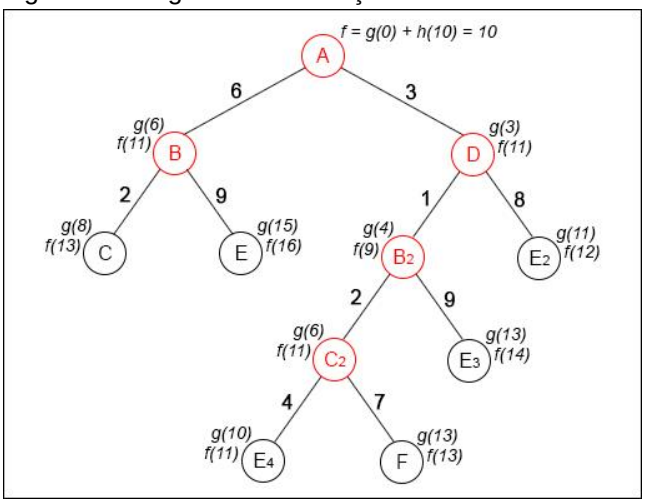
Fonte: Do autor

Em sequência, o algoritmo A* tomará o nó C mais à direita dá árvore como nó atual, que apresenta custo f de 11. Ao expandir o nó C, será calculado os valores g e f de seus dois nós adjacentes, E e F. Para o nó E, o custo g representará o custo g já calculado do nó atual (C) que vale 6, somado ao custo da aresta que conecta os nós C e E, sendo este, 4. Para o nó F, o mesmo processo será feito, com a diferença de que será somado ao valor g do nó C, o custo da aresta que conecta os nós C e F, sendo este valor, 7.

Para a heurística (f), será somado ao valor de g de ambos os nós adjacentes E e F, o valor h apresentado na figura de heurísticas (figura 22), sendo 1 para o nó E e 0 para o nó F.

A figura 26 apresenta o resultado da expansão do atual nó C, localizado mais à direita da árvore.

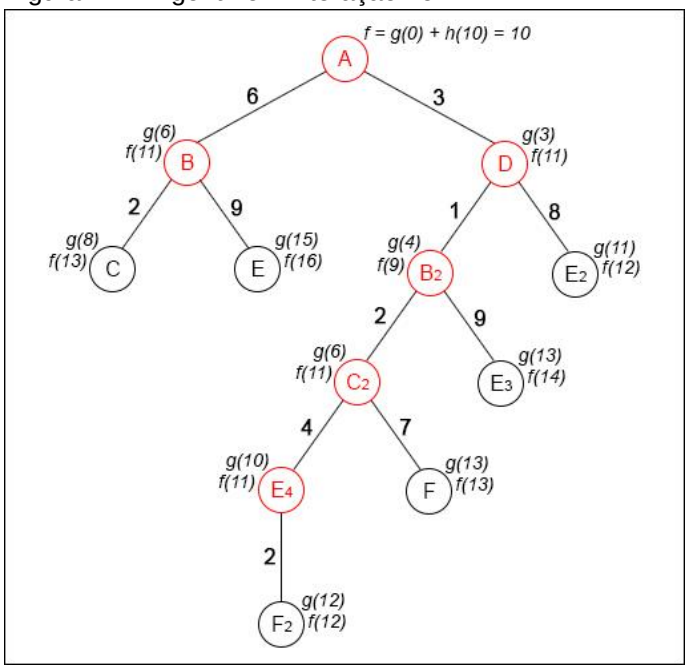
Figura 26 – Algoritmo A* iteração nó C_2



Fonte: Do autor

Na próxima iteração é selecionado o nó E cujo valor de estimativa f equivale a 11. O nó é marcado como nó atual, e seu único nó adjacente é o nó F, que tem seus custos g e f calculados para atingir o nó F. A figura 27 ilustra o resultado da expansão do nó E_4.

Figura 27 – Algoritmo A* iteração nó E_4



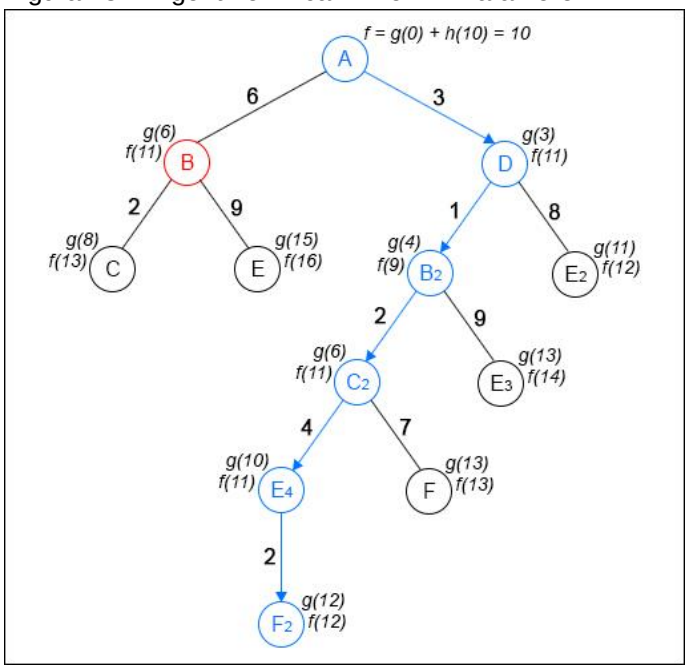
Fonte: Do autor

Em seguida, o algoritmo A* novamente verifica que há dois nós com o mesmo custo f equivalendo a 12, sendo eles os nós E_2 e F_2. Poderia se escolhido qualquer um dos dois para expandir, mas como o nó E_2 precisará expandir mais um nó para chegar ao destino, isso fará com que ele tenha um custo maior do que o nó F_2. Por esta razão, será escolhido o nó F_2 como próximo nó atual. É importante ressaltar que caso fosse escolhido o nó E_2 para expandir, isso não interferiria no resultado final da busca, apenas teria uma iteração a mais.

Ao expandir o nó F_2, o algoritmo identificará que atingiu o nó de destino. Porém, ele ainda precisa verificar se existe algum outro nó na lista de nós abertos cujo valor de f seja menor que o valor f do nó atual (F_2). Não existindo nenhum, o algoritmo finaliza a busca retornando o caminho, cujo percurso é A-D-B_2-C_2-E_4-F_2.

A figura 28 ilustra o caminho final encontrado pelo algoritmo na árvore.

Figura 28 – Algoritmo A* caminho A-F na árvore



Fonte: Do autor

Uma colocação importante sobre o algoritmo A*, é que a velocidade do algoritmo em relação à geração do caminho depende dos valores de estimativa estabelecidos para cada nó. Bastos e Jaques (2010) descrevem que a heurística precisa ser admissível, nunca superestimando o custo real do percurso, ou seja, em um caminho cujo custo real seja 18 unidades, jamais seria recomendado colocar uma estimativa maior que este valor para este mesmo caminho.

No próximo item, serão descritos exemplos de trabalhos que foram utilizados como base para a elaboração deste trabalho.

4 TRABALHOS CORRELATOS

Em jogos digitais, a teoria dos grafos é uma área bastante explorada em consequência dos diversos gêneros de jogos que utilizam seus conceitos para aplicações de algoritmos, principalmente algoritmos de planejamento de caminho. A seguir serão apresentados alguns trabalhos envolvendo estes algoritmos planejadores de caminho.

4.1 ESTUDO COMPARATIVO ENTRE ALGORITMO A* E BUSCA EM LARGURA PARA PLANEJAMENTO DE CAMINHO DE PERSONAGENS EM JOGOS DO TIPO PACMAN

Neste trabalho de conclusão de curso de Ciência da Computação apresentado por Jeanita Bassani da Silva na Universidade Regional de Blumenau no ano de 2005, a autora demonstra a implementação de adaptações para os algoritmos de busca em largura, em profundidade e o algoritmo A*, este que já se utiliza de uma heurística em sua implementação padrão, em um ambiente de jogo do estilo *Pacman*, também desenvolvido pela mesma.

No trabalho, a autora desenvolve um protótipo do jogo *Pacman* construindo o ambiente do jogo com base em um grafo direcionado, e o utiliza como cenário de testes para os algoritmos de busca de caminho, utilizando primeiramente suas implementações padrões e em sequência utilizando os algoritmos com as adaptações propostas pela mesma.

Como resultados, foram obtidos por meio de comparações realizadas pela autora, dados demonstrando os valores de cada execução, como exemplo, o total de movimentos e total de vértices do grafo testados por cada algoritmo.

4.2 IMPLEMENTAÇÃO E TESTE DE UM ALGORITMO PLANEJADOR DE CAMINHOS EM UM JOGO DE ESTRATÉGIA DE TEMPO REAL

Este trabalho de conclusão de curso em Ciência da Computação elaborado por José Antônio Salini Ferreira e apresentado na Universidade Federal do Rio Grande do Sul no ano de 2010, descreve um algoritmo de planejamento de

caminhos desenvolvido pelo mesmo, para a aplicação em jogos do tipo *Real-Time Strategy* (*RTS*).

No trabalho, o autor seleciona um jogo do tipo RTS de código fonte aberto, e utilizando-se do jogo selecionado, ele desenvolve um algoritmo de geração de caminhos que é acoplado aos agentes do jogo para gerar e suavizar um caminho entre pontos distintos do cenário de forma dinâmica, isto é, o caminho é gerado e suavizado enquanto o agente se move pelo ambiente, com o objetivo de fazer o agente possuir movimentos mais parecidos com o de um jogador real. O autor ainda realiza comparações entre o algoritmo gerador de caminho nativo do jogo, e o seu algoritmo proposto no trabalho.

Como resultados obtidos, através de testes de desempenho, qualidade dos caminhos gerados e viabilidade estratégica do caminho, o autor realiza comparações entre ambos os algoritmos.

4.3 AVALIAÇÃO DO ALGORITMO A* NA HEURÍSTICA DE PATH-PLANNING EM AMBIENTES DE JOGOS UTILIZANDO O MOTOR UNITY3D

Neste trabalho de conclusão de curso em Ciência da Computação realizado e apresentado por Brian Castelan Andrade na Universidade do Extremo Sul Catarinense no ano de 2015, é demonstrada a utilização e análise de desempenho do algoritmo de busca de caminho A* atuando em cenários de jogos elaborados pelo mesmo.

Com o auxílio do motor de jogo Unity3D, o autor modelou diferentes cenários de jogos com diferentes níveis de complexidades e tipos de obstáculos, incluindo cenários que contam com elevações, por onde o agente deverá se locomover, entre os obstáculos. Utilizando-se do componente *NavMesh,* componente presente no motor utilizado para a geração de malhas de navegação responsáveis por delimitar a área navegável, isto é, livre de obstáculos, o autor gerou as malhas de navegação dos cenários modelados, para então aplicar a implementação do A* sobre a malha gerada.

Como resultados obtidos, o autor realizou comparações na presença e na ausência do agente A*, nos diferentes cenários modelados, gerando gráficos e tabelas com os dados procedentes das execuções, como a média da taxa de

frames, o consumo de memória RAM, além de gerar gráficos demonstrando as variações destes itens.

4.4 ALGORITMO A-ESTRELA DE ESTADO HÍBRIDO APLICADO À NAVEGAÇÃO AUTÔNOMA DE VEÍCULOS

Esta dissertação desenvolvida por Michael André Gonçalves e apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo no ano de 2013 demonstra o uso do algoritmo A* para a navegação autônoma em ambientes tridimensionais, em que o autor mapeia o ambiente por meio de um grafo, onde determinou-se um ponto inicial para a posição do veículo utilizado no desenvolvimento e várias outras posições consideradas pontos de interesse.

O autor implementou o algoritmo A* utilizando-se de duas heurísticas que, combinadas, foram capazes de acelerar o processo de busca localizando uma rota, considerada como a melhor, entre os pontos de origem e destino escolhidos, evitando os obstáculos presentes no ambiente durante o percurso. A primeira das heurísticas apresentada pelo autor, realiza o reconhecimento do cenário, detectando e desviando dos obstáculos, a segunda por sua vez analisa as limitações do veículo, porém desconsidera os obstáculos.

O autor ainda implementou interfaces de comunicação com sensores, como sensores de mapeamento e de localização permitindo a troca de mensagens entre estes dispositivos, o que torna a utilização do algoritmo mais prática.

Para a obtenção dos resultados, o autor descreve que foram realizados experimentos com o veículo de passeio autônomo desenvolvido pela universidade, onde o mesmo conclui que, por meio de testes tanto em ambientes abertos com menos obstáculos quanto em ambientes complexos, a solução desenvolvida se mostrou viável na utilização.

5 COMPARAÇÃO DOS ALGORITMOS DE BUSCA DE CAMINHO, A* E DIJKSTRA, APLICADO EM CENÁRIOS DE JOGOS

Algoritmos de busca de caminho são amplamente utilizados na área de tecnologia, especialmente para geração de rotas em mapas geográficos, redes de computadores, problemas de logística, etc.

Por contarem com uma simples estrutura, também possuem como característica a fácil implementação, o que resultou no surgimento de diversos algoritmos para a solução do problema do caminho mínimo ao longo dos anos.

Este tipo de algoritmo também é largamente utilizado em jogos digitais, para a determinação e geração de rotas que são utilizadas pelos personagens do jogo para se movimentarem, sem correrem o risco de colidirem com obstáculos do ambiente, e consequentemente, ficarem presos nesses obstáculos.

Neste trabalho, utilizando-se do motor de jogo Unity3D, foram implementados três cenários distintos, com o objetivo de executar os algoritmos Dijkstra e A* nestes cenários e comparar os resultados extraídos de suas execuções, utilizando-se para isso, de uma ferramenta de medição de desempenho incluída no motor utilizado.

Para a geração de rotas em cenários de jogos, se faz necessário o uso de uma ou mais formas de mapear os espaços navegáveis do mesmo, desta forma, o algoritmo tem a capacidade de conhecer o cenário, para determinar as áreas trafegáveis. Dos três tipos básicos de mapeamentos descritos ao longo deste trabalho, foi utilizado na modelagem dos três cenários, o mapeamento por meio de pontos de visibilidade.

5.1 METODOLOGIA

Para a elaboração deste projeto de pesquisa exploratório, foi realizado um levantamento bibliográfico iniciando com foco em jogos digitais, abordando uma breve história, seus principais gêneros, o mercado de jogos e suas principais etapas de desenvolvimento, bem como uma abordagem de motores de jogos com destaque no motor de jogo Unity 3D.

Em sequência, foi dado continuidade no levantamento bibliográfico abordando os problemas de caminho, com ênfase no contexto de jogos digitais.

Após esta etapa foi realizado um relato sobre os algoritmos solucionadores do problema de caminho e em seguida foi explanado detalhadamente, por meio de etapas, as implementações dos algoritmos Dijkstra e A*.

Durante a coleta das bibliografias que foram utilizadas para elaborar este trabalho, deu-se prioridade a livros, artigos nacionais e internacionais, a outros trabalhos de graduações, de mestrado e de doutorado.

5.1.1 *Scripts* de mapeamento do cenário

No desenvolvimento do trabalho, a primeira etapa consistiu na implementação dos *scripts* responsáveis por mapear o cenário, a fim de fazer com que ambos os algoritmos, Dijkstra e A*, tenham a capacidade de reconhecer os pontos navegáveis, também conhecidos como os nós de um grafo.

Para isto, foram criadas duas classes que representam esses nós, utilizando a nomenclatura *Node* e *NodeView*. A classe *Node* é responsável por representar os dados de cada nó individual, como sua distância atual em relação ao nó inicial e suas arestas, enquanto a classe *NodeView* mantém a representação gráfica deste mesmo ponto no cenário, como sua posição em relação ao ambiente.

A figura 28 demonstra a implementação da classe *Node.* Dado um nó qualquer inserido no cenário, a variável *Neighboors,* representada por uma lista de valores em pares, sendo o primeiro valor uma referência para o nó vizinho, e o segundo, representado por um tipo inteiro, a distância desta conexão, armazena todos os nós que possuem conexão com este mesmo nó. A implementação desta classe conta ainda, com as variáveis *DistanceFromStartNode*, responsável por manter a distância entre o ponto inicial até o ponto atual, *ParentNode*, utilizado pelos algoritmos no processo de geração do caminho, para guardar informações do nó vizinho imediato antecessor do nó atual e por fim, a variável *viewTransform*, sendo esta, uma referência para a posição do nó no cenário, oriunda da classe *NodeView*.

Por fim, a classe possui um construtor que recebe as informações visuais do nó por parâmetro, representado pela variável *viewTransform,* e onde são inicializadas a distância entre o nó atual com o nó inicial, utilizando o valor máximo do tipo inteiro, e a lista de nós vizinhos.

Figura 28 – Implementação da classe *Node*

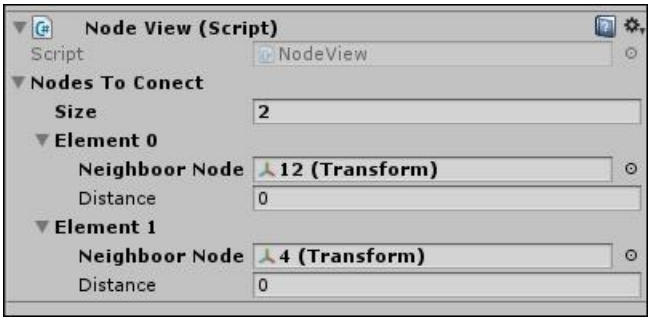
```
1   using System.Collections.Generic;
2   using UnityEngine;
3
4   public class Node
5   {
6       public List<KeyValuePair<Node, int>> Neighboors;
7
8       public int DistanceFromStartNode { get; set; }
9
10      public Node ParentNode { get; set; }
11
12      public Transform ViewTransform { get; set; }
13
14      public Node(Transform viewTransform)
15      {
16          this.ViewTransform = viewTransform;
17          DistanceFromStartNode = int.MaxValue;
18
19          Neighboors = new List<KeyValuePair<Node, int>>();
20      }
21  }
```

Fonte: Do autor.

A classe *NodeView* por sua vez, têm o papel de manter as informações visuais de cada nó inserido no cenário, como sua posição por exemplo.

É nesta classe também que se insere a coleção dos nós vizinhos, e para isto, é utilizada uma classe interna chamada *NeighboorNode*, possuindo duas variáveis públicas rotuladas *neighboorNode* e *distance*. Por meio desta classe interna, é possível visualizar em cada nó, no inspetor do Unity3D, uma coleção de tamanho variável onde é possível inserir os nós vizinhos de cada nó presente no cenário, e sua distância. A figura 29 ilustra esta estrutura, nela o nó demonstrado possui conexão com os nós 12 e 4.

Figura 29 – Representação da estrutura de inserção de nós vizinhos



Fonte: Do autor.

Para ser possível visualizar esta estrutura de nós vizinhos, a classe *NodeView* possui uma variável, rotulada de *nodesToConect* e definida como uma coleção, do tipo *NeighboorNode*, sendo importante destacar que a classe

NeighboorNode precisa possuir o atributo *System.Serializable*, atributo utilizado para que a estrutura permaneça visível no inspetor do Unity3D em modo de edição.

A classe *NodeView* possui ainda uma outra variável que faz referência a um objeto da classe *Node*, que será responsável por receber as informações não visuais de cada nó.

Por fim, a classe possui também implementações de dois métodos responsáveis por ler as informações dos nós vizinhos inseridos no modo de edição, e transferir essas informações para o objeto da classe *Node*, que irá manter esses dados para serem utilizados pelos algoritmos adiante, para gerarem os caminhos.

Ambos os métodos são muito semelhantes, com a única diferença de que um deles, rotulado *InitializeNodeEdges*, utiliza-se dos valores presentes na variável *distance* informadas no inspetor e ilustrado na figura 29, para representar as distâncias dos nós vizinhos, e o outro por sua vez, rotulado de *InitializeNodeEdgesWithUnityDistances*, gera as distâncias por meio do método *Vector3.Distance*, presente no conjunto de *scripts* do motor, que retorna um valor representando a distância direta entre a conexão dos dois pontos no cenário.

A figura 30 ilustra a implementação completa da classe *NeighboorNode*, e a figura 31, a implementação da classe *NodeView*.

Figura 30 – Implementação da classe interna *NeighboorNode*

```
1   using System.Collections.Generic;
2   using UnityEngine;
3
4    [System.Serializable]
5   public class NeighboorNode
6   {
7       public Transform neighboorNode;
8       public int distance;
9   }
10
```

Fonte: Do autor.

Figura 31 – Implementação da classe *NodeView*

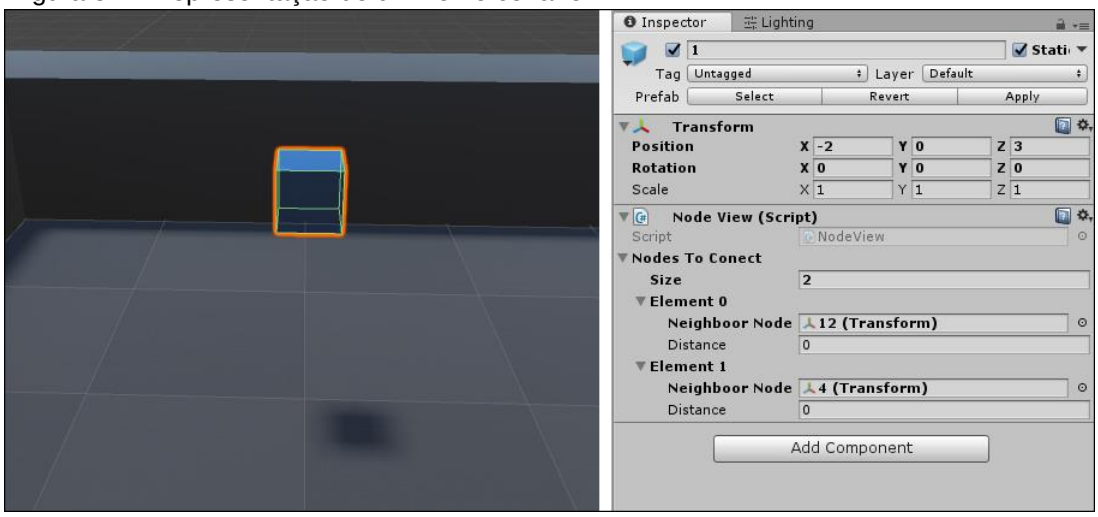
```
11  public class NodeView : MonoBehaviour
12  {
13      public Node node;
14
15      public NeighboorNode[] nodesToConect;
16
17      public void InitializeNodeEdges()
18      {
19          foreach (NeighboorNode node in nodesToConect)
20          {
21              NodeView neighboorNodeView = node.neighboorNode.transform.GetComponent<NodeView>();
22
23              this.node.Neighboors.Add(new KeyValuePair<Node, int>(neighboorNodeView.node, node.distance));
24          }
25      }
26
27      public void InitializeNodeEdgesWithUnityDistances()
28      {
29          foreach (NeighboorNode node in nodesToConect)
30          {
31              NodeView neighboorNodeView = node.neighboorNode.transform.GetComponent<NodeView>();
32
33              //Calculate the distance
34              int distance = Mathf.FloorToInt(Vector3.Distance(this.transform.position, neighboorNodeView.transform.position));
35
36              this.node.Neighboors.Add(new KeyValuePair<Node, int>(neighboorNodeView.node, distance));
37          }
38      }
39  }
```

Fonte: Do autor.

Com a finalização da implementação dos *scripts* responsáveis por representar um nó no cenário, foi criado um objeto do tipo *GameObject*, responsável por representar qualquer entidade, seja esta física ou abstrata, dentro do motor, e anexado a este mesmo objeto o *script NodeView* juntamente com um cubo para se ter uma representação tridimensional do nó.

A figura 32 ilustra a representação física de um nó no cenário, juntamente com o *script NodeView* anexado a si.

Figura 32 – Representação de um nó no cenário



Fonte: Do autor.

Para a finalização da etapa de mapeamento do cenário, há ainda outras duas classes, responsáveis por realizarem a leitura dos nós presente no cenário, e inicializar para cada nó gráfico, um respectivo objeto contendo as informações não visuais do nó, representado pela classe *Node* citada anteriormente, essas duas classes foram rotuladas de *Graph* e *GraphView*.

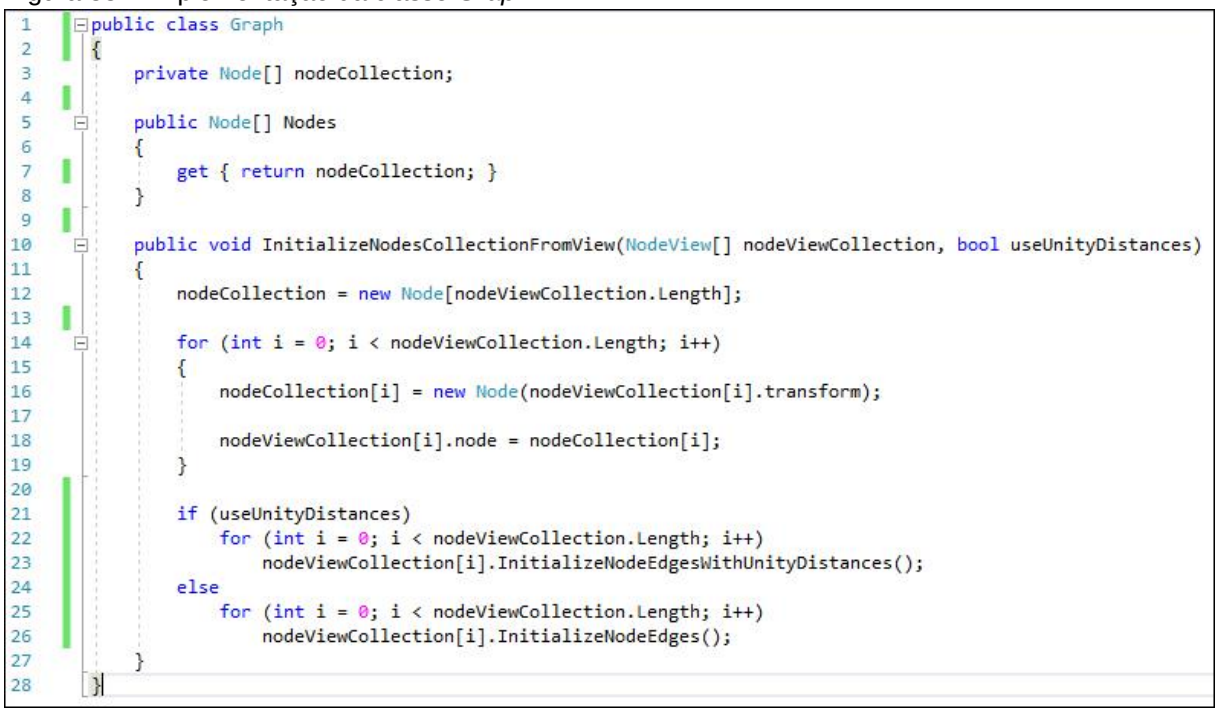
Seguindo o mesmo conceito utilizado nas classes *Node* e *NodeView*, a classe *Graph* lida apenas com dados não visuais, enquanto que a classe *GraphView*, lida apenas com as informações visuais.

A classe *Graph* possui uma variável privada, isto é, acessível apenas por ela mesma, chamada de *nodeCollection*. Esta variável representa uma coleção de objetos do tipo *Node*, e é responsável por guardar as referências dos nós presentes no cenário. Para acessar esta variável de fora desta classe, foi criada a propriedade *Nodes*, que apenas recupera as informações de *nodeCollection*, sem permitir que essa coleção seja alterada por outra classe, uma vez que os nós devem serem apenas lidos, e não alterados.

A classe *Graph* possui ainda um método rotulado *InitializeNodesCollectionFromView* que recebe por parâmetro uma coleção de *NodeView* e uma variável do tipo booleana. Este método percorre a coleção de *NodeView*, e para cada nó da coleção, inicializa um respectivo objeto da classe *Node*, e por fim, percorre novamente a coleção de *NodeView*, invocando o método responsável por inicializar a coleção de nós vizinhos de cada nó. É importante destacar que o método percorre duas vezes a mesma coleção, devido ao fato de que para inicializar os nós vizinhos de cada nó, todos os nós presentes no cenário já necessitam estarem inicializados, evitando erros em tempo de execução do algoritmo.

Abaixo na figura 33, está sendo demonstrada a implementação completa da classe *Graph*.

Figura 33 – Implementação da classe *Graph*

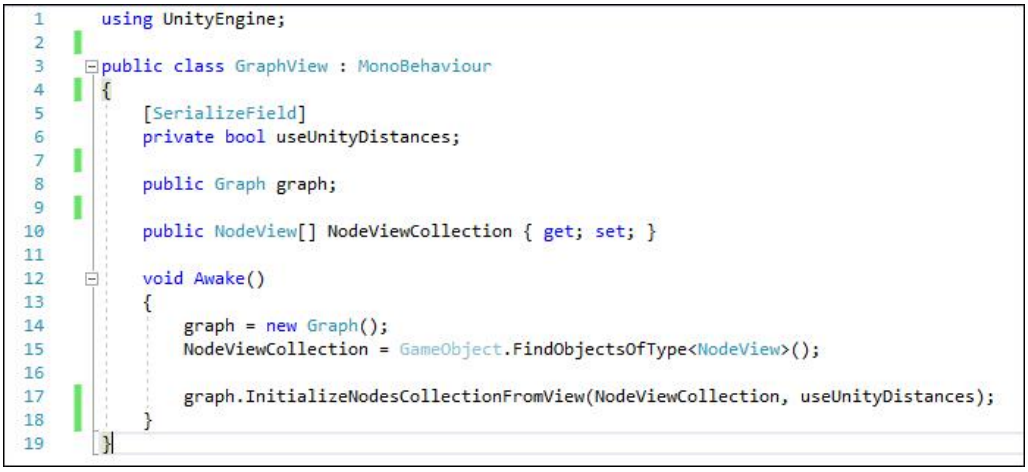


```
public class Graph
{
    private Node[] nodeCollection;

    public Node[] Nodes
    {
        get { return nodeCollection; }
    }

    public void InitializeNodesCollectionFromView(NodeView[] nodeViewCollection, bool useUnityDistances)
    {
        nodeCollection = new Node[nodeViewCollection.Length];

        for (int i = 0; i < nodeViewCollection.Length; i++)
        {
            nodeCollection[i] = new Node(nodeViewCollection[i].transform);

            nodeViewCollection[i].node = nodeCollection[i];
        }

        if (useUnityDistances)
            for (int i = 0; i < nodeViewCollection.Length; i++)
                nodeViewCollection[i].InitializeNodeEdgesWithUnityDistances();
        else
            for (int i = 0; i < nodeViewCollection.Length; i++)
                nodeViewCollection[i].InitializeNodeEdges();
    }
}
```

Fonte: Do autor.

A classe *GraphView* é a última do conjunto de classes responsáveis pelo mapeamento do cenário, responsável por ler os nós visuais (coleção de objetos do tipo *NodeView*) inseridos através do modo de edição do motor Unity3D e popular as informações das classes *Graph* e *Node*. A figura 34 ilustra a implementação desta classe.

Figura 34 – Implementação da classe *GraphView*



```
using UnityEngine;

public class GraphView : MonoBehaviour
{
    [SerializeField]
    private bool useUnityDistances;

    public Graph graph;

    public NodeView[] NodeViewCollection { get; set; }

    void Awake()
    {
        graph = new Graph();
        NodeViewCollection = GameObject.FindObjectsOfType<NodeView>();

        graph.InitializeNodesCollectionFromView(NodeViewCollection, useUnityDistances);
    }
}
```

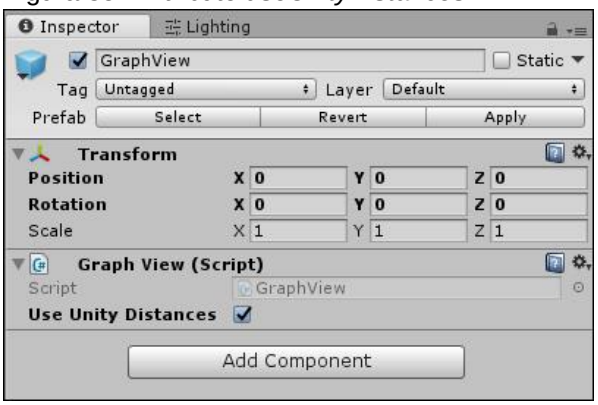
Fonte: Do autor.

Esta classe contém três variáveis, a primeira delas, *useUnityDistances*, determina se as distâncias entre os nós vizinhos serão informadas manualmente (se

o atributo estiver marcado como *false*), ou se o algoritmo determinará estas distâncias automaticamente (se o atributo estiver marcado como *true*), como descrito anteriormente na implementação do *NodeView*.

Esta variável contém ainda, o atributo *SerializeField* pelo fato de ser uma variável do tipo privada, ou seja, ela não é visível de fora desta classe, com isto, este atributo faz com que ela seja visível no inspetor do motor, como demonstrado abaixo na figura 35.

Figura 35 – Atributo *useUnityDistances*



Fonte: Do autor.

A segunda variável, rotulada de *graph*, representa um objeto da classe *Graph*, que será inicializado uma única vez durante todo o processo de execução e comparação dos algoritmos, sendo nesta variável que se encontrarão os dados de cada nó, para serem utilizados pelos algoritmos para gerar os caminhos.

Por fim, a propriedade *NodeViewCollection* é responsável por manter todas as referências para os objetos *NodeView*, que, por sua vez, são as representações gráficas dos nós presentes no cenário.

A classe possui dois métodos, o método *Awake* e o método *InitializeNodesData*. O método *Awake* é um método presente na classe *MonoBehaviour*, que faz parte do conjunto de *scripts* do motor Unity3D. Este método é chamado automaticamente pelo motor no momento em que o mesmo está carregando a instância do *script.*

Neste método, foi inicializado o objeto *graph*, e por meio do método *GameObject.FindObjectsOfType* que também faz parte do conjunto de *scripts* do motor, realiza-se uma busca por todos os nós inseridos no cenário, e por fim, o método *InitializeNodesCollectionFromView,* pertencente à classe *Graph* e descrito

anteriormente, foi chamado para que o objeto *graph* inicialize os objetos do tipo *Node* e manipule esses dados.

Com a implementação do mapeamento do cenário, é possível fazer com que os algoritmos de busca de caminho consigam interagir com o mesmo, reconhecendo os pontos navegáveis e sejam capazes de realizar buscas, com isso, a etapa de mapeamento do cenário é finalizada.

5.1.2 Desenvolvimento do algoritmo de Dijkstra

Na segunda etapa do desenvolvimento, será descrita toda a implementação do algoritmo de Dijkstra, incluindo um breve resumo do seu funcionamento.

O algoritmo de Dijkstra possui um funcionamento consideravelmente simples, podendo ser descrito em poucos passos. O algoritmo inicializa duas listas, uma contendo todos os nós do grafo para serem analisados e outra contendo os nós que já foram visitados e calculados. A lógica do algoritmo pode ser descrita seguindo estes passos:

a) inicialmente, todos os nós são adicionados na lista de nós a serem analisados, uma vez que o algoritmo de Dijkstra analisa todos os nós do grafo;
b) após todos os nós estarem na lista para serem analisados, o algoritmo atribui um valor infinito para o custo de cada nó, significando que o algoritmo ainda não possui conhecimento de nenhum caminho;
c) o terceiro passo consiste em zerar o custo do nó definido como inicial;
d) o algoritmo entra em seu laço de repetição onde analisará o restante dos nós, até que não haja mais nós na lista de nós a serem analisados;
e) dentro do seu laço, o algoritmo seleciona o nó da lista de nós a serem analisados cujo custo ou distância seja o menor, e o toma como nó atual;
f) tendo o nó atual selecionado, o algoritmo percorre todos os nós vizinhos do nó atual. Para cada nó vizinho, o algoritmo verificará se este já foi analisado, se não, o algoritmo o adiciona na lista de nós a serem analisados para ser processado futuramente;

g) o algoritmo então realiza uma comparação entre a distância atual do nó vizinho e a distância do nó atual somado ao custo da aresta que conecta o nó atual ao vizinho. Se a distância do nó atual somado à aresta que conecta o nó atual ao nó vizinho for menor que a distância atual do nó vizinho, o algoritmo substitui o custo do nó vizinho;
h) após completar a verificação de todos os nós vizinhos do nó atualmente selecionado, o algoritmo seleciona o nó da lista de nós a serem analisados com o menor custo;
i) o processo se repete a partir do passo “e”;

Ao fim de todo este processo, o algoritmo de Dijkstra será capaz de retornar o menor caminho do nó inicial para todos os outros nós presentes no grafo.

A implementação do algoritmo de Dijkstra conta com duas classes, uma responsável por receber os dados, como nó inicial e nó de destino, e repassar estes dados para a outra que irá, de fato, resolver o algoritmo e retornar o caminho, além de utilizar as informações da classe *Node* para interpretar os nós do cenário.

A classe *Dijkstra* é responsável por resolver o algoritmo, nela são declaradas algumas propriedades responsáveis por manter os dados do algoritmo, sendo elas, *Iterations*, que representa o número de iterações realizadas, *VisitedNodesQuantity*, responsável pelo número de nós visitados e, por último, *TimeToFinishTheSearch* que armazena o tempo total de execução do algoritmo.

A classe possui o método *ResolveDijkstra* que recebe três parâmetros, *graphNodes*, *startNode* e *destinyNode*, sendo eles, a coleção de nós do grafo, o nó inicial e o nó de destino, respectivamente. Este é o método principal que resolverá o algoritmo.

Ao entrar neste método, são inicializadas as coleções responsáveis por guardar os nós a serem visitados e os nós já visitados. A variável *nodesToVisit*, é responsável por manter a lista de nós a serem visitados e a variável *visitedNodes*, a lista de nós já visitados.

A figura 36 ilustra a seção de inicialização e as configurações iniciais do método que resolve o algoritmo de Dijkstra.

Figura 36 – Seção de inicialização do método que resolve o algoritmo de Dijkstra

```
15      public void ResolveDijkstra(Node[] graphNodes, Node startNode, Node destinyNode)
16      {
17          List<Node> nodesToVisit = new List<Node>();
18          List<Node> visitedNodes = new List<Node>();
19
20          ResetInformations(graphNodes);
21
22          Stopwatch stopwatch = Stopwatch.StartNew();
23
24          startNode.DistanceFromStartNode = 0;
25
26          nodesToVisit.AddRange(graphNodes);
```

Fonte: Do autor.

Ainda é possível notar outras rotinas sendo executadas nesta seção. A chamada para o método *ResetInformations*, é responsável por reiniciar as informações dos nós, como as propriedades *ParentNode* e *DistanceFromStartNode* de cada objeto do tipo *Node*, provenientes da coleção que é recebida por parâmetro deste método, além de zerar as propriedades *Iterations*, *VisitedNodesQuantity* e *TimeToFinishTheSearch,* para que outras execuções do método *ResolveDijkstra* não apresente resultados errôneos.

A figura 37 ilustra a implementação do método *ResetInformations.*

Figura 37 – Implementação do método *ResetInformations*

```
57      private void ResetInformations(Node[] graphNodes)
58      {
59          foreach (Node node in graphNodes)
60          {
61              node.ParentNode = null;
62              node.DistanceFromStartNode = int.MaxValue;
63          }
64
65          Iterations = 0;
66          VisitedNodesQuantity = 0;
67          TimeToFinishTheSearch = 0f;
68      }
```

Fonte: Do autor.

Além da chamada para o método *ResetInformations*, há ainda a inicialização de um objeto do tipo *Stopwatch*, responsável por realizar a contagem de tempo durante a execução do método, e em seguida é atribuído o valor zero para a distância do nó inicial, e por fim, são adicionados todos os nós na lista de nós para serem verificados.

O algoritmo então entra no seu laço principal, onde verificará cada nó e todos os seus vizinhos. A figura 38 demonstra a implementação do restante do método que resolve o algoritmo de Dijkstra.

Figura 38 – Análise dos nós no método *ResolveDijkstra*

```
28          while (nodesToVisit.Count > 0)
29          {
30              Node currentNode = GetNodeToAnalyze(nodesToVisit);
31
32              visitedNodes.Add(currentNode);
33              nodesToVisit.Remove(currentNode);
34
35              VisitedNodesQuantity++;
36
37              foreach (KeyValuePair<Node, int> neighboor in currentNode.Neighboors)
38              {
39                  if (visitedNodes.Contains(neighboor.Key))
40                      continue;
41
42                  if ((currentNode.DistanceFromStartNode + neighboor.Value) < neighboor.Key.DistanceFromStartNode)
43                  {
44                      neighboor.Key.DistanceFromStartNode = currentNode.DistanceFromStartNode + neighboor.Value;
45                      neighboor.Key.ParentNode = currentNode;
46                  }
47
48                  Iterations++;
49              }
50          }
51
52          stopwatch.Stop();
53          TimeToFinishTheSearch = stopwatch.Elapsed.TotalMilliseconds;
54      }
```

Fonte: Do autor.

Nesta etapa do método, enquanto houver nós para serem visitados, o algoritmo selecionará o nó cuja distância seja a menor, adicionará este nó na lista de nós já visitados e removerá da lista de nós a serem visitados. Logo após isto, é incrementado o valor da propriedade *VisitedNodesQuantity*, e o algoritmo passará a analisar todas os nós vizinhos do então nó atual.

Para cada nó vizinho, o algoritmo primeiro verifica se o nó que está sendo visitado a partir do nó atual já está na lista de nós visitados, se estiver, o algoritmo simplesmente ignora esta aresta, uma vez que, no algoritmo de Dijkstra, um nó que já foi visitado já possui o menor caminho possível.

Se o nó ainda não foi visitado, o algoritmo fará o cálculo de distâncias, e, caso a distância a partir do nó atual for menor que a distância atual do nó vizinho, o algoritmo faz a troca dos valores e, além disso, atribui o nó atual como referência à propriedade *ParentNode* do nó vizinho. No final deste processo, o número de iterações é incrementado por meio da instrução na linha 48 da figura 38.

Este processo se repete para todos os nós, até que não haja mais nenhum nó na lista para ser analisado. No fim do algoritmo ainda é possível notar duas instruções (representadas nas linhas 52 e 53, ainda da figura 38) responsáveis por parar a contagem de tempo e atribuir o resultado à propriedade *TimeToFinishTheSearch*.

Com isto, finaliza-se a implementação da classe *Dijkstra*. A outra classe relacionada à implementação do algoritmo de Dijkstra, chamada *TestDijkstra* é responsável apenas por receber e repassar os dados do modo de edição e chamando o método *ResolveDijkstra,* pertencente à classe *Dijkstra*.

A figura 39 ilustra a implementação completa da classe *TestDijkstra*.

Figura 39 – Implementação da classe *TestDijkstra*

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

public class TestDijkstra : MonoBehaviour
{
    [SerializeField]
    private GraphView graphView;

    [SerializeField]
    private LineDrawer lineDrawer;

    [SerializeField]
    private NodeView startNode;

    [SerializeField]
    private NodeView destinyNode;

    private Graph graph;

    private Dijkstra dijkstra;

    private void Start()
    {
        dijkstra = new Dijkstra();
        graph = graphView.graph;
    }

    public void CallDijkstra()
    {
        if (startNode == null || destinyNode == null)
        {
            Debug.LogWarning("É necessário atribuir um nó de início e um nó de destino para realizar a busca!");
            return;
        }

        dijkstra.ResolveDijkstra(graph.Nodes, startNode.node, destinyNode.node);

        // Results
        string report = "Distância do percurso: " + destinyNode.node.DistanceFromStartNode +
            "\tNúmero de iterações: " + dijkstra.Iterations +
            "\tNúmero de nós visitados: " + dijkstra.VisitedNodesQuantity +
            "\tTempo total de execução (ms): " + dijkstra.TimeToFinishTheSearch;
        print(report);
    }
}
```

Fonte: Do autor.

Na implementação desta classe, mantém-se uma referência para o objeto do tipo *GraphView,* por meio da variável *graphView*, uma variável *lineDrawer*, sendo esta, um utilitário desenvolvido para testes, não influenciando nas comparações, além de receber o nó inicial e o nó de destino por meio das variáveis *startNode* e *destinyNode*. Todas essas variáveis são privadas, porém pelo fato de precisarem serem usadas no modo de edição e serem visíveis no inspetor do motor, elas possuem o atributo *SerializeField*. A classe contém ainda, outras duas variáveis,

chamadas *graph* e *dijkstra*, sendo a primeira utilizada para acessar as informações dos nós, e a segunda para calcular o algoritmo de Dijkstra.

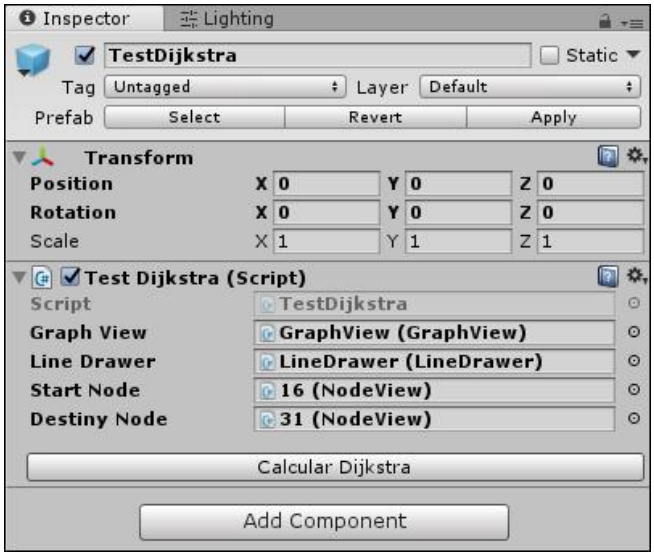
A implementação desta classe conta ainda com dois métodos, o método *Start*, pertence ao conjunto de métodos provenientes do motor, que é chamado ao executar o primeiro *frame* do jogo, e a função *CallDijkstra.*

No método *Start* está sendo inicializado um novo objeto do tipo *Dijkstra*, permitindo assim que seja possível acessar as funcionalidades desta classe, e logo em seguida, está sendo atribuído à variável graph a referência para o objeto *Graph*, inicializado anteriormente na classe *GraphView*. Esta referência precisa ser única durante todo o processo de comparação, e por conta disso, não foi inicializado outro objeto do tipo *Graph*.

O método *CallDijkstra*, apenas verifica se foram informados os nós de início e de destino, e chama o método para calcular o caminho, reportando alguns resultados da busca como a distância total do percurso e o tempo de execução no console de informações do motor.

A figura 40 ilustra um objeto contendo o *script TestDijkstra* anexado a si, demonstrando a forma de inserção dos valores referentes às variáveis *startNode* e *destinyNode*, além do objeto *GraphView* responsável por fornecer as informações do nós presentes no cenário, a partir do inspetor do motor no modo de edição.

Figura 40 – Inserção dos valores para realizar a busca



Fonte: Do autor.

Com os valores devidamente preenchidos e o cenário corretamente mapeado, é possível executar buscas de caminhos utilizando o algoritmo de Dijkstra, e com isto, a etapa de implementação deste algoritmo está finalizada.

5.1.3 Desenvolvimento do algoritmo A*

A terceira etapa do desenvolvimento compreende a implementação do algoritmo A*. O algoritmo A* é muito conhecido na área de jogos por possuir um bom desempenho aliado a uma boa qualidade nos caminhos por ele gerado.

A implementação do A* é muito semelhante ao algoritmo de Dijkstra, e este algoritmo foi, de fato, construído utilizando como base o algoritmo de Dijkstra. O A* também utiliza em sua implementação duas listas, uma contendo os nós a serem analisados e a outra contendo os nós que já foram verificados. Seguindo a mesma metodologia de explanação da implementação do algoritmo de Dijkstra, a lógica do algoritmo A* segue os seguintes passos:

a) inicialmente, somente o nó inicial é adicionado na lista de nós a serem visitados;
b) o algoritmo entra em seu laço de repetição onde analisará o restante dos nós, até que não haja mais nós na lista de nós a serem analisados, ou no caso do A*, até que o nó de destino seja atingido;
c) o algoritmo calcula o custo estimado de todos os nós presentes na lista aberta em relação ao nó de destino, e seleciona o nó cujo custo estimado seja o menor entre todos os outros;
d) Tendo selecionado o nó atual, o algoritmo expande seus nós vizinhos, adicionando todos os que ainda não foram analisados, na lista de nós a serem analisados;
e) o próximo passo é realizar a comparação entre a distância atual do nó vizinho com a distância do nó atual somado ao custo da aresta que conecta o nó atual ao vizinho. Se a distância do nó atual somado à aresta que conecta o nó atual ao nó vizinho for menor que a distância atual do nó vizinho, o algoritmo substitui o custo do nó vizinho;
f) após completar a verificação de todos os nós vizinhos do nó atualmente selecionado, o processo se repete a partir do passo c;

A diferença do algoritmo A* em relação ao algoritmo de Dijkstra se dá no passo c, descrito acima. Nesta etapa, o algoritmo A* utiliza uma heurística, que é calculada utilizando o custo atual do nó, somado a um valor gerado manualmente ou automaticamente, que representa uma distância estimada do custo para atingir o nó de destino. Desta forma, o algoritmo consegue direcionar a busca, analisando os nós sempre mais próximos do nó de destino, diferentemente do algoritmo de Dijkstra, que expande os nós para todos os lados igualmente.

Assim como os passos do algoritmo A* são parecidos com os do Dijkstra, a sua implementação neste trabalho também é bastante semelhante. Para a implementação do A*, foram utilizadas duas classes, rotuladas de *AStar* e *TestAStar*, a primeira, responsável por calcular o algoritmo em si, e a segunda, responsável por receber e enviar os dados para a classe *AStar*, assim como foi realizada na implementação do algoritmo de Dijkstra. O algoritmo A* também utiliza as mesmas classes de mapeamento do cenário utilizadas pelo Dijkstra.

A figura 41 ilustra a implementação das configurações de inicialização dos dados utilizados pelo algoritmo A*.

Figura 41 – Configurações iniciais do A*

```
16      public void ResolveAStar(Node[] graphNodes, Node startNode, Node destinyNode)
17      {
18          List<Node> nodesToVisit = new List<Node>();
19          List<Node> visitedNodes = new List<Node>();
20
21          ResetInformations(graphNodes);
22
23          Stopwatch stopwatch = Stopwatch.StartNew();
24
25          startNode.DistanceFromStartNode = 0;
26
27          nodesToVisit.Add(startNode);
```

Fonte: Do autor.

Na figura 41, a diferença em relação à implementação do algoritmo de Dijkstra está na etapa de adição dos nós na variável *nodesToVisit*, onde no Dijkstra, eram adicionados todos os nós, diferentemente do A* que só é adicionado o nó inicial.

Em seguida, a figura 42 ilustra o laço principal de execução do algoritmo.

Figura 42 – Implementação do laço principal do algoritmo A*

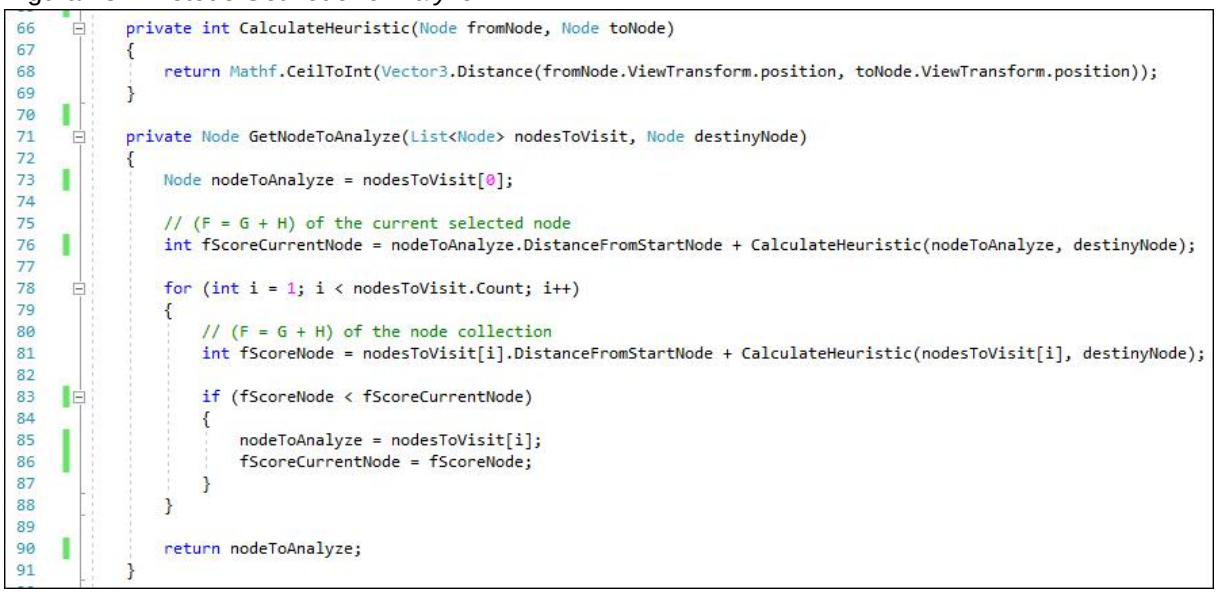
```
29          while (nodesToVisit.Count > 0)
30          {
31              Node currentNode = GetNodeToAnalyze(nodesToVisit, destinyNode);
32
33              // If reached the destiny node, the search is done
34              if (currentNode == destinyNode)
35              {
36                  stopwatch.Stop();
37                  TimeToFinishTheSearch = stopwatch.Elapsed.TotalMilliseconds;
38                  return;
39              }
40
41              visitedNodes.Add(currentNode);
42              nodesToVisit.Remove(currentNode);
43
44              VisitedNodesQuantity++;
45
46              // Calculate the edges for the current node
47              foreach (KeyValuePair<Node, int> neighboor in currentNode.Neighboors)
48              {
49                  if (visitedNodes.Contains(neighboor.Key))
50                      continue;
51
52                  if (!nodesToVisit.Contains(neighboor.Key))
53                      nodesToVisit.Add(neighboor.Key);
54
55                  if ((currentNode.DistanceFromStartNode + neighboor.Value) < neighboor.Key.DistanceFromStartNode)
56                  {
57                      neighboor.Key.DistanceFromStartNode = currentNode.DistanceFromStartNode + neighboor.Value;
58                      neighboor.Key.ParentNode = currentNode;
59                  }
60
61                  Iterations++;
62              }
63          }
64      }
```

Fonte: Do autor.

A distinção do algoritmo A* em relação ao algoritmo de Dijkstra na sua implementação encontra-se na chamada para o método rotulado *GetNodeToAnalyze*, ilustrado na figura 42. Este método é responsável por buscar o próximo nó a ser analisado com base no cálculo da heurística, utilizando a lista de nós ainda não visitados e o nó de destino, este que é recebido como parâmetro do método *ResolveAStar* como demonstrado na figura 41.

O método *GetNodeToAnalyze* está ilustrado na figura 43 abaixo, juntamente com o método auxiliar rotulado *CalculateHeuristic*, responsável apenas por calcular a distância em linha reta entre dois pontos distintos do cenário, que no atual contexto, são os nós do grafo, fazendo isso utilizando o método *Vector3.Distance*, descrito anteriormente, assim gerando o valor da heurística para cada nó.

Figura 43 – Método *GetNodeToAnalyze*



```
66  private int CalculateHeuristic(Node fromNode, Node toNode)
67  {
68      return Mathf.CeilToInt(Vector3.Distance(fromNode.ViewTransform.position, toNode.ViewTransform.position));
69  }
70
71  private Node GetNodeToAnalyze(List<Node> nodesToVisit, Node destinyNode)
72  {
73      Node nodeToAnalyze = nodesToVisit[0];
74
75      // (F = G + H) of the current selected node
76      int fScoreCurrentNode = nodeToAnalyze.DistanceFromStartNode + CalculateHeuristic(nodeToAnalyze, destinyNode);
77
78      for (int i = 1; i < nodesToVisit.Count; i++)
79      {
80          // (F = G + H) of the node collection
81          int fScoreNode = nodesToVisit[i].DistanceFromStartNode + CalculateHeuristic(nodesToVisit[i], destinyNode);
82
83          if (fScoreNode < fScoreCurrentNode)
84          {
85              nodeToAnalyze = nodesToVisit[i];
86              fScoreCurrentNode = fScoreNode;
87          }
88      }
89
90      return nodeToAnalyze;
91  }
```

Fonte: Do autor.

Ainda na figura 43, o método *GetNodeToAnalyze* percorre toda a lista de nós a serem analisados, representado pela variável *nodesToVisit* que é recebida por parâmetro do método, calculando a heurística de cada nó e verificando qual deles possui o menor custo, retornando para o método *ResolveAStar* que será responsável por fazer a análise do nó.

Por fim, a classe *TestAStar* possui o mesmo comportamento da classe *TestDijkstra*, descrita anteriormente, com a diferença de que utiliza um objeto do tipo da classe *AStar* para fazer a chamada ao método *ResolveAStar*.

Com a finalização da implementação do algoritmo A*, o último passo relacionado à desenvolvimento, diz respeito à modelagem dos cenários que serão utilizados como base para as comparações.

5.1.4 Modelagem dos cenários tridimensionais

Ao iniciar o processo de modelagem dos ambientes, foi tido como primeiro passo, a definição dos níveis de detalhamento responsáveis por caracterizar a complexidade de navegação de cada cenário.

Os algoritmos de busca de caminho utilizam os cenários como base para a geração dos caminhos, fazendo a leitura dos mesmos para obter conhecimento e desviar dos obstáculos, navegando pelos pontos livres do ambiente.

Para aplicar a comparação dos algoritmos de Dijkstra e A*, foram desenvolvidos três cenários tridimensionais distintos, cada um possuindo um nível de detalhamento diferente do outro. Para a geração do detalhamento, foram utilizadas elevações no terreno, responsáveis por invalidar algumas áreas dos cenários, além de serem utilizados objetos 3D, que foram baixados gratuitamente da loja de recursos presente no motor Unity3D, para a geração de outros obstáculos menores.

Foram desenvolvidos três cenários, nomeados de *Level1, Level2* e *Level3,* com o propósito de criar três níveis diferentes de detalhamento, seguindo uma metodologia de modelagem de um cenário mais simples, outro apresentando um nível de detalhamento intermediário, e o último cenário apresentando um terreno mais complexo dentre todos. Para cada cenário, foram selecionados três pares de nós de início e de destino, que serão utilizados para realizar três comparações em cada ambiente.

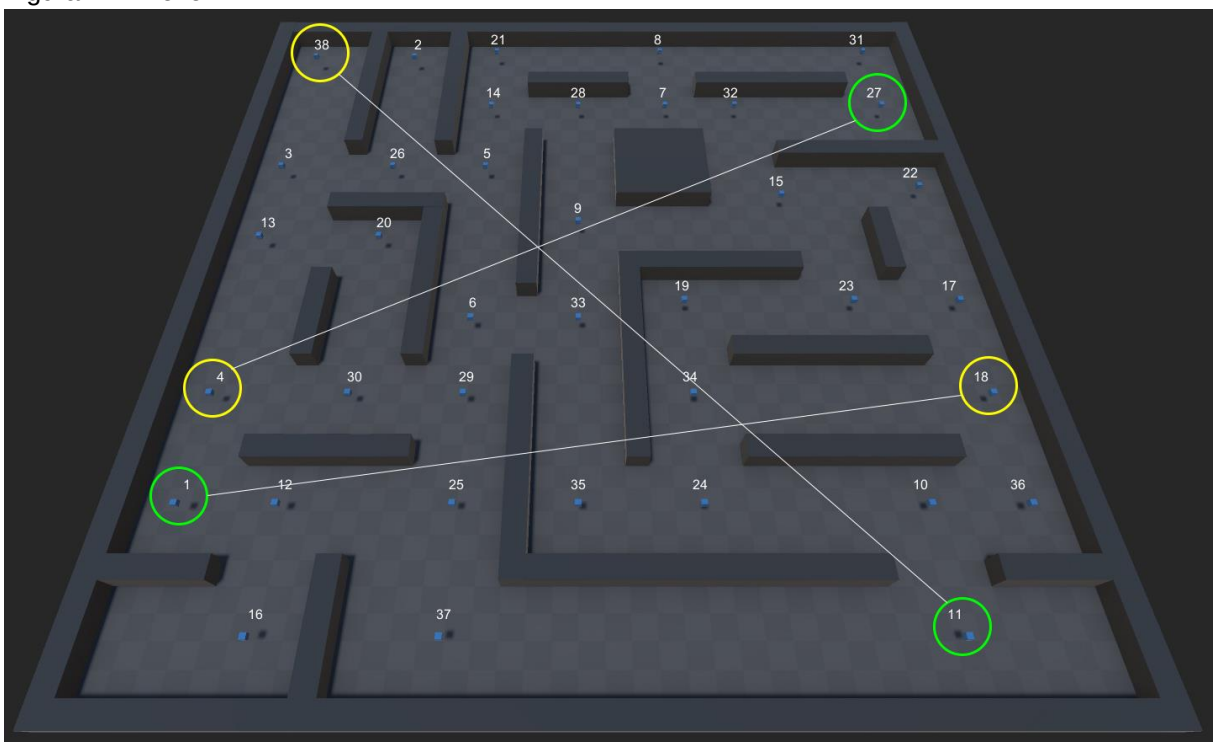
5.1.4.1 Modelagem do *Level1*

O ambiente nomeado *Level1* é, dentre todos os três cenários desenvolvidos, o mais simples. Ele foi desenvolvido utilizando um plano contendo objetos tridimensionais utilizados para representarem paredes que impossibilitam algumas rotas.

A figura 44 representa o cenário como um todo, onde os objetos em cor cinza escuro representam as paredes, e os cubos em cor azul, os nós. Foi utilizado ainda, textos localizados acima de cada nó para representar sua nomenclatura.

Ainda na figura 44, para as três comparações realizadas neste cenário, foram utilizados como nós de início, destacados em cor verde, os nós 1, 11 e 27 e seus nós de destino correspondentes, destacados em cor amarela, foram definidos como 18, 38 e 4, respectivamente. As linhas de cor branca no cenário estão inseridas apenas para auxiliar na visualização de cada nó de início com seu nó de destino correspondente.

Figura 44 – *Level1*



Fonte: Do autor.

5.1.4.2 Modelagem do *Level2*

O segundo cenário modelado, nomeado de *Level2*, conta com uma complexidade um pouco maior em relação ao *Level1*. Nele, foi inserido uma quantidade consideravelmente maior de nós, além da utilização de mais relevos no terreno, com a inserção de dois lagos no cenário, bloqueando algumas rotas.

A figura 45 ilustra o segundo ambiente modelado. Nele, além da maior exploração do recurso de elevação do terreno, como já descrito, foram inseridas também árvores que também são responsáveis por bloquear os caminhos atuando como obstáculos. Para as comparações neste cenário, foram utilizados como nós de início, os nós 8, 68 e 72 e como seus nós de destino correspondentes, os nós 63, 27 e 21.

Figura 45 – *Level2*

Fonte: Do autor.

5.1.4.3 Modelagem *Level3*

Por fim, foi modelado o último cenário onde foram aplicadas as comparações. Nele, os recursos de alteração de terreno foram altamente utilizados, dando origem à três grandes ilhas que geraram grande parte dos obstáculos presentes no ambiente. Além disso, foram utilizados também outros modelos tridimensionais, como rochas e construções, para representar obstáculos menores em cada uma das ilhas.

A figura 46 ilustra uma visão completa do cenário, englobando todas as três ilhas modeladas, com suas conexões por meio de pontes. Este cenário, conta com o maior número de nós dentre todos os três desenvolvidos, resultando em uma quantidade consideravelmente maior de conexões entre os mesmos, o que também contribui para o aumento da complexidade do cenário. Para as comparações neste cenário, foram utilizados como nós de início, os nós 15, 118 e 86 e como seus nós de destino correspondentes, os nós 65, 26 e 51.

Figura 46 – *Level3*

Fonte: Do autor.

Com a finalização da modelagem dos três cenários, foi dado início aos procedimentos referentes à comparação dos algoritmos de Dijkstra e A*, definindo as métricas a serem comparadas, para assim, coletar os dados referentes a essas métricas e aplicar as comparações.

5.1.5 Testes Realizados

Para a realização da etapa de testes, foi utilizado uma máquina com a seguinte configuração: Processador *Intel* Core i7-8700 X12 3,20 G*hz*, 16Gb de memória *RAM*, 1 TB de armazenamento interno e sistema operacional *Windows 10 Pro*.

Foram realizadas três execuções de ambos algoritmos em cada ambiente, como descrito na seção de modelagem dos cenários, utilizando os mesmos nós de início e de destino e analisando as seguintes métricas:

a) número de iterações realizadas pelo algoritmo;

b) tempo gasto para atingir o nó de destino após a busca ter sido iniciada;

c) impacto da execução nos *frames* do jogo;

d) o tempo de frame tomado por cada algoritmo.

Após a finalização das execuções dos algoritmos e da obtenção dos dados foi realizada a comparação dos algoritmos Dijkstra e A*.

A figura 47 ilustra um exemplo de um caminho gerado no cenário *Level3*, onde foi utilizado o nó 120 como nó de início e o nó 3 como nó de destino.

Figura 47 – Exemplo de um caminho gerado durante as comparações

Fonte: Do autor.

Durante a etapa de coleta dos resultados, foi utilizado ainda o software Excel para realizar a tabulação e geração dos gráficos, desenvolvidos com o objetivo de demonstrar os resultados de forma visual, a fim de facilitar a leitura dos mesmos.

5.2 RESULTADOS OBTIDOS

Os resultados referentes ao número de iterações realizadas por cada algoritmo e seus tempos de execução foram obtidos por meio de uma média aritmética das três buscas em cada cenário.

Em relação aos dados do impacto de cada algoritmo nos *frames* do jogo, foi realizado uma execução do jogo na ausência de ambos os algoritmos, para assim, capturar a média de *frames* sem que os algoritmos impactassem no resultado. Logo após esta execução, foram estabelecidos 100 agentes, que juntos, executaram várias buscas durante cinco segundos de forma ininterrupta, e com o auxílio de um *script* de captura de *frames*, foi calculado a média final de *frames* na presença de cada algoritmo, que foram executados de forma individual. Nesta etapa da comparação, foi levado em consideração a média de *frames* por segundo incluindo os processos de renderização gráfica que fazem parte da geração de um *frame* de jogo.

Por fim, para a medida do tempo de *frame*, ou seja, o tempo médio que cada algoritmo consome em um *frame* do jogo, foram executadas as buscas seguindo a mesma metodologia usada na medida do impacto nos *frames*, executando os algoritmos contando com 100 agentes durante o tempo de cinco segundos, e com o auxílio do *Profiler*, foi realizada a leitura dos dados.

5.2.1 Ambiente *Level1*

O primeiro ambiente onde foram testados os algoritmos possui 38 nós distribuídos entre os obstáculos. Analisando os dados obtidos neste cenário, nota-se que a diferença tanto em relação à média de iterações, quanto em relação à média de tempo de execução de cada algoritmo, se manteve próxima, como está sendo demonstrado nas tabelas 1 e 2, respectivamente.

Tabela 1 – Média de iterações por algoritmo no *Level1*

Algoritmo	Média de iterações
Dijkstra	49
A*	31

Fonte: Do autor.

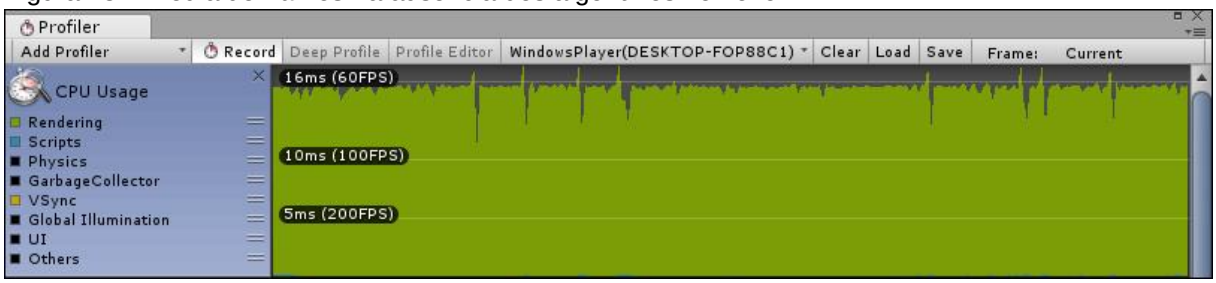
Tabela 2 – Média de tempo de execução por algoritmo no *Level1*

Algoritmo	Média de tempo
Dijkstra	0,1006ms
A*	0,0647ms

Fonte: Do autor.

Em relação aos resultados referentes ao impacto dos algoritmos nos *frames* do jogo, foi executado o mesmo por cinco segundos sem a presença dos algoritmos para assim obter uma média dos frames sem nenhuma interferência. A média de frames obtida desta execução foi de 60.27 *frames* após a finalização da captura, como ilustrado na figura 47.

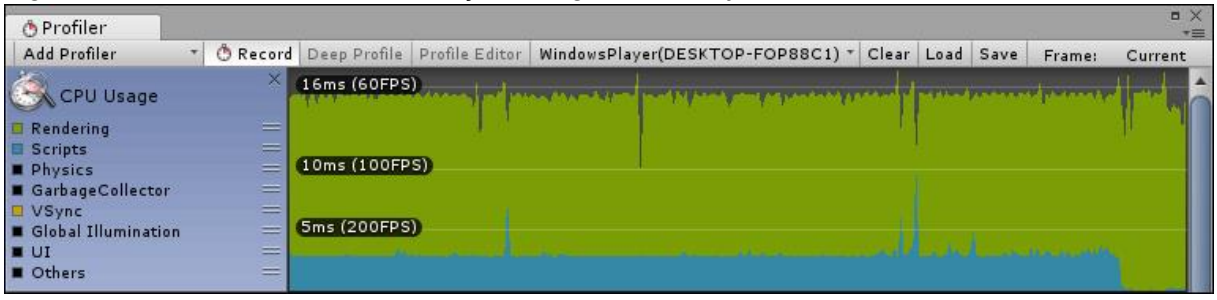
Figura 48 – Média de *frames* na ausência dos algoritmos no *Level1*



Fonte: Do autor.

Após a captura da média de *frames* sem a interferência dos algoritmos, foi executada a busca com o algoritmo de Dijkstra por cinco segundos, capturando a média de frames dos 100 agentes que executaram ininterruptamente durante o tempo determinado. Como resultado desta execução, foi obtido uma média de 60.20 *frames*. A figura 48 demonstra o gráfico da média de *frames*.

Figura 49 – Média de *frames* na execução do algoritmo de Dijkstra no *Level1*

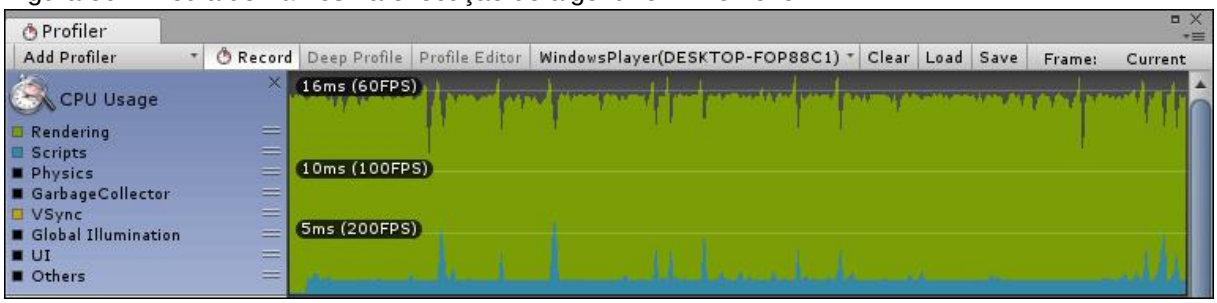


Fonte: Do autor.

Observou-se que a média de *frames* obtida com a execução do algoritmo de Dijkstra permaneceu muito próxima à execução onde não houve interferência dos algoritmos, demonstrando apenas uma leve queda, sendo esta queda menor que um *frame*, o que torna imperceptível ao jogador.

Em seguida, foi executado o algoritmo A*, seguindo a mesma metodologia da execução do algoritmo de Dijkstra. A figura 49 ilustra a média de *frames* obtidos, que após o tempo de captura, foi de 60.23 *frames*.

Figura 50 – Média de *frames* na execução do algoritmo A* no *Level1*



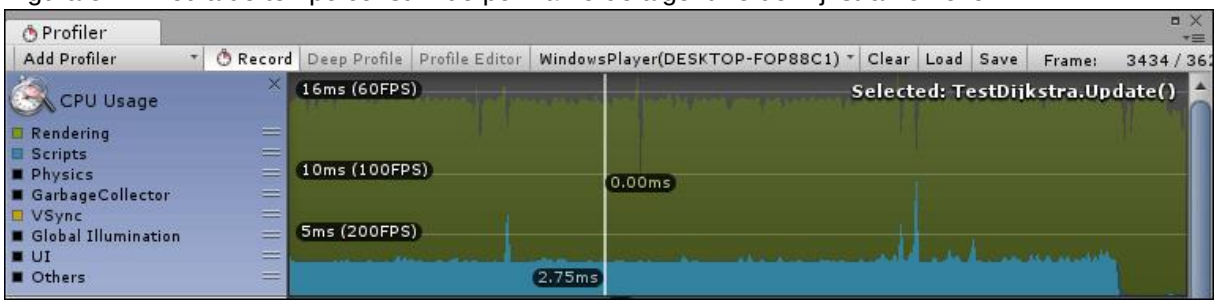
Fonte: Do autor.

Seguindo o mesmo comportamento do algoritmo de Dijkstra, o algoritmo A* também não afetou os *frames* do jogo de forma perceptível, já que se manteve estável na mesma média da execução obtida na ausência dos algoritmos.

Após a captura da média de *frames*, foi então analisado o tempo de frame que cada algoritmo consumiu, utilizando-se para isto, as mesmas execuções anteriores dos algoritmos de Dijkstra e A*.

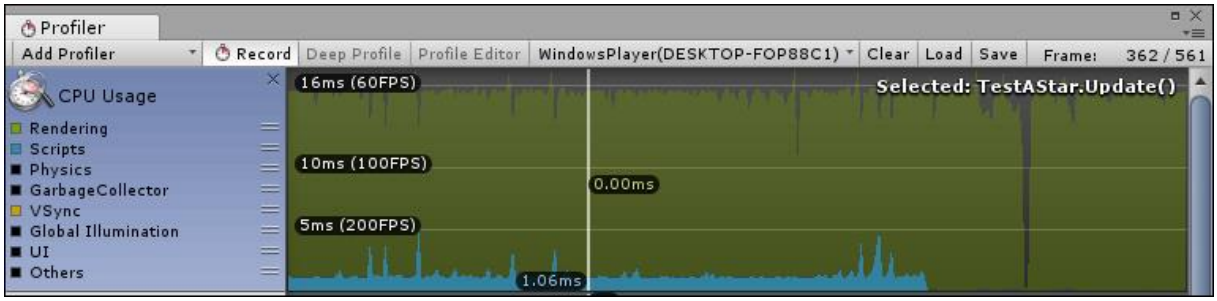
As figuras 50 e 51 ilustram os dados referentes à média de tempo que cada algoritmo consumiu de cada frame ao longo do período de captura.

Figura 51 – Média de tempo consumido por *frame* do algoritmo de Dijkstra no *Level1*



Fonte: Do autor.

Figura 52 – Média de tempo consumido por *frame* do algoritmo A* no *Level1*



Fonte: Do autor.

Analisando as figuras 50 e 51 observa-se que, mesmo os algoritmos de Dijkstra e A* se mantendo com uma mesma média de *frames* ao longo do tempo capturado, tendo uma diferença média de apenas 0.03 *frames*, o algoritmo A* consumiu um tempo por *frame* consideravelmente menor em relação ao algoritmo de Dijkstra.

5.2.2 Ambiente *Level2*

Seguindo os mesmos modelos de representações dos dados do primeiro ambiente, no segundo ambiente observou-se um aumento no número de iterações de ambos os algoritmos, bem como um aumento nos tempos de execuções. Isso se deve por dois principais motivos, o primeiro pelo fato de que o *Level2* possui uma maior quantidade de nós em relação ao primeiro, contando com 74 nós, o que também aumenta o número de conexões entre estes mesmos nós, elevando o número de cálculos que os algoritmos precisam realizar para encontrar o caminho. O segundo motivo deve-se ao fato de que o *Level2* possui um tamanho consideravelmente maior e por consequência, um número maior de obstáculos.

As tabelas 3 e 4 demonstram, respectivamente, os dados da média de iterações dos algoritmos e seus tempos de execuções gastos até encontrarem o menor caminho entre os nós, após as três execuções.

Tabela 3 – Média de iterações por algoritmo no *Level2*

Algoritmo	Média de iterações
Dijkstra	124
A*	48

Fonte: Do autor.

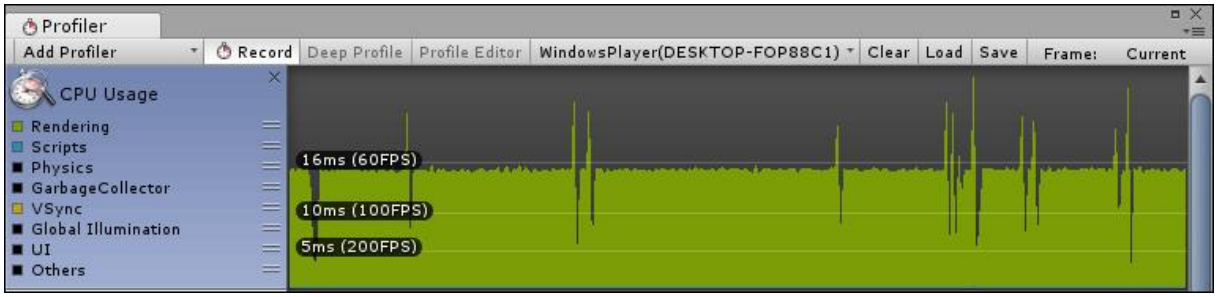
Tabela 4 – Média de tempo de execução por algoritmo no *Level2*

Algoritmo	Média de tempo
Dijkstra	0,2955ms
A*	0,0837ms

Fonte: Do autor.

Seguindo a metodologia de comparação aplicada ao primeiro cenário, foi capturado a média de frames na ausência dos algoritmos a fim de capturar a média de frames, sendo esta de 61.68 frames ao longo dos cinco segundos, como demonstrado abaixo na figura 52.

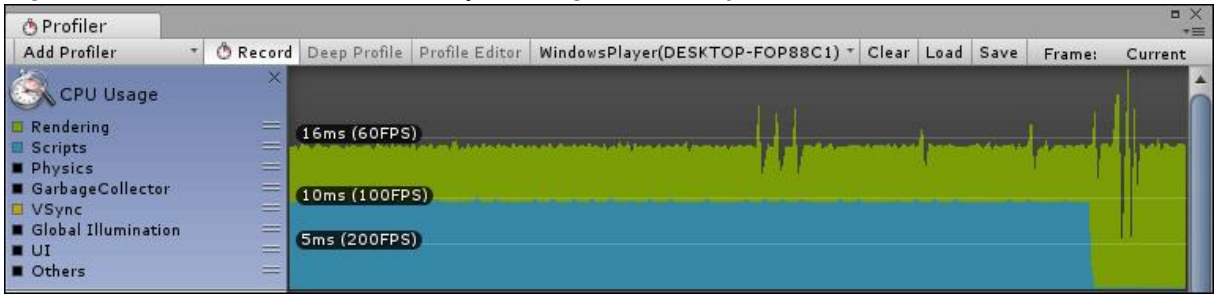
Figura 53 – Média de *frames* na ausência dos algoritmos no *Level2*



Fonte: Do autor.

Em seguida, deu-se continuidade à captura da média de *frames* executando os algoritmos de Dijkstra e A*. O primeiro, apresentou uma média de 60.20 *frames* ao final da execução, enquanto o segundo demonstrou uma média de 60.45 *frames*. Abaixo, as figuras 53 e 54 ilustram os gráficos referentes a estes dados.

Figura 54 – Média de *frames* na execução do algoritmo de Dijkstra no *Level2*



Fonte: Do autor.

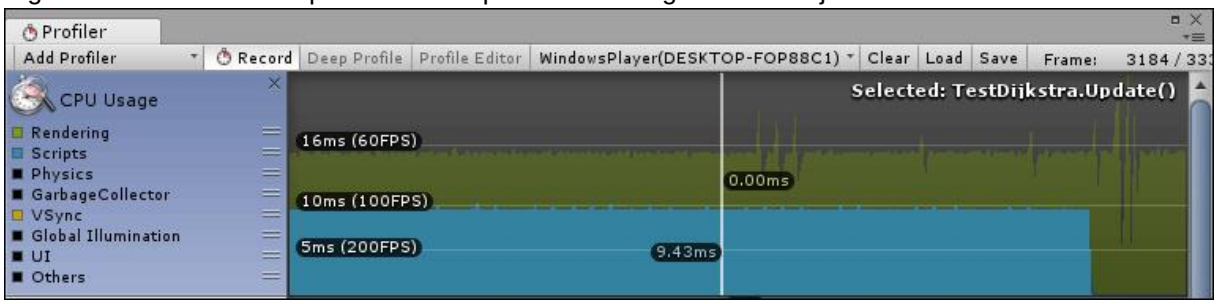
Figura 55 – Média de *frames* na execução do algoritmo A* no *Level2*

Fonte: Do autor.

Como demonstrado nos gráficos das figuras 53 e 54, nenhum dos algoritmos afetou os *frames* em um nível notável ao jogador. O algoritmo de Dijkstra demonstrou uma queda de apenas 1.48 *frames* em relação à execução na ausência dos algoritmos, e o A*, por sua vez, apresentou uma queda de 1.23 *frames*.

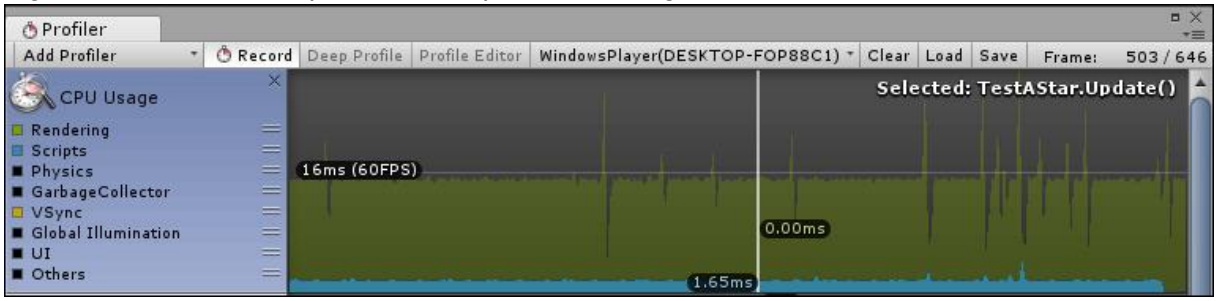
Por último, verificou-se o tempo médio consumido por *frame* para a execução das buscas, de ambos os algoritmos. As figuras 55 e 56 ilustram, respectivamente, os dados obtidos dos algoritmos Dijkstra e A*.

Figura 56 – Média de tempo consumido por *frame* do algoritmo de Dijkstra no *Level2*



Fonte: Do autor.

Figura 57 – Média de tempo consumido por *frame* do algoritmo A* no *Level2*



Fonte: Do autor.

Observando os tempos consumidos por frame em cada algoritmo, é possível notar um grande aumento no tempo que o algoritmo de Dijkstra consumiu para executar as buscas, tendo uma diferença de 6,68 milissegundos a mais em

relação à sua execução no cenário anterior, enquanto que o algoritmo A* manteve um resultado mais próximo à sua execução no primeiro cenário, com uma diferença de apenas 0,59 milissegundos.

5.2.3 Ambiente *Level3*

No último dos três ambientes desenvolvidos, a taxa de iterações bem como o tempo de busca, continuaram a subir, uma vez que este cenário possui uma quantidade de nós ainda maior em relação aos anteriores, contando com 121 nós. Uma outra característica que impactou no aumento destes resultados, é o fato de que o cenário possui, além dos obstáculos menores distribuídos pelas três ilhas, a divisão das mesmas por meio das elevações geradas no terreno, o que obriga os algoritmos a realizarem mais cálculos para localizarem as pontes que conectam as ilhas.

As tabelas 5 e 6 mostram, respectivamente, a média dos dados do número de iterações e do tempo de execução de cada algoritmo nas três execuções.

Tabela 5 – Média de iterações por algoritmo no *Level3*

Algoritmo	Média de iterações
Dijkstra	227
A*	80

Fonte: Do autor.

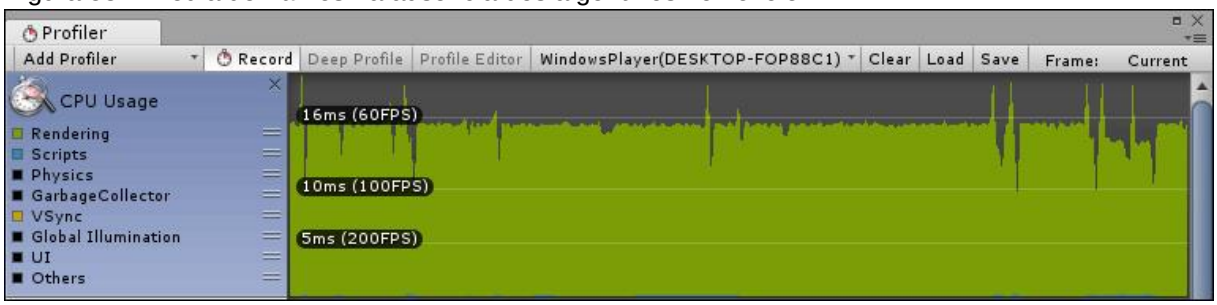
Tabela 6 – Média de tempo de execução por algoritmo no *Level3*

Algoritmo	Média de tempo
Dijkstra	0,7778ms
A*	0,1759ms

Fonte: Do autor.

No *Level3* a captura da média de *frames* na ausência dos algoritmos resultou em um valor de 61.15 *frames* ao longo dos cinco segundos de execução. A figura 57 ilustra o gráfico gerado ao final da captura neste cenário.

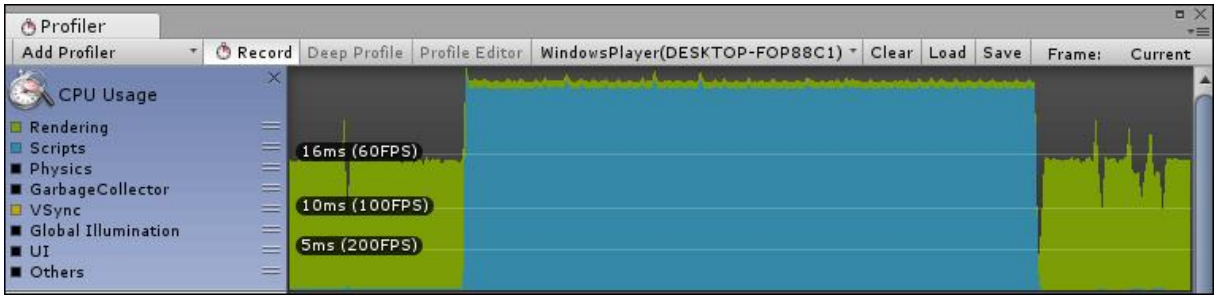
Figura 58 – Média de *frames* na ausência dos algoritmos no *Level3*



Fonte: Do autor.

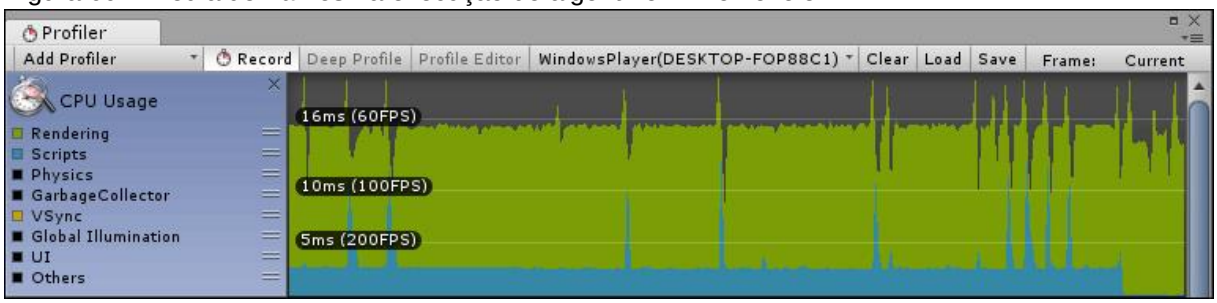
Em seguida, foi capturado a média de *frames* em ambos algoritmos. As figuras 58 e 59 ilustram os gráficos gerados após o tempo de captura, para o algoritmo de Dijkstra e A*, respectivamente.

Figura 59 – Média de *frames* na execução do algoritmo de Dijkstra no *Level3*



Fonte: Do autor.

Figura 60 – Média de *frames* na execução do algoritmo A* no *Level3*



Fonte: Do autor.

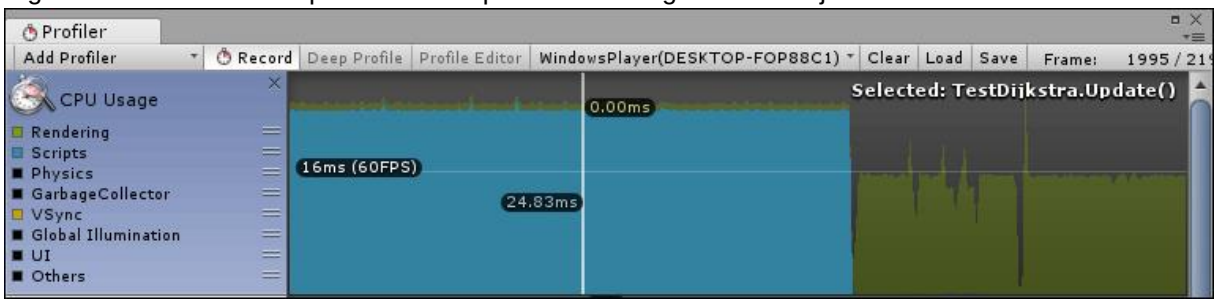
Neste cenário, o algoritmo de Dijkstra acabou impactando na média de frames do jogo, obtendo um resultado médio de 38.56 frames durante as execuções das buscas. O algoritmo A*, por sua vez, conseguiu manter a média de frames em 60.93 *frames* ao longo de todo o tempo de busca.

Com estes resultados, o algoritmo de Dijkstra influenciou os *frames* do jogo de modo que houvesse uma queda média de 22.59 *frames* em relação à execução onde não houve a presença de nenhum dos dois algoritmos, enquanto o

algoritmo A* não impactou os frames de modo com que seja perceptível ao jogador, obtendo uma queda média de apenas 0.22 *frames,* também em relação à execução na ausência dos algoritmos.

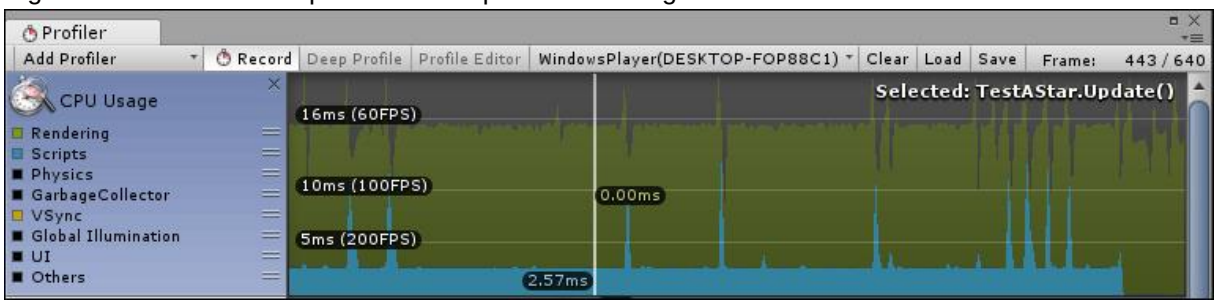
Por fim as figuras 60 e 61 abaixo ilustram os dados relacionados ao tempo médio consumido por *frame* que os algoritmos gastaram para realizarem as buscas.

Figura 61 – Média de tempo consumido por *frame* do algoritmo de Dijkstra no *Level3*



Fonte: Do autor.

Figura 62 – Média de tempo consumido por *frame* do algoritmo A* no *Level3*



Fonte: Do autor.

Analisando as figuras 60 e 61 é possível notar que o algoritmo de Dijkstra utilizou um tempo de *frame* médio de 24.83 milissegundos, o que acabou ocasionando a queda na média de *frames* por segundo, uma vez que quanto maior o tempo gasto para gerar um *frame*, menor é a quantidade de *frames* renderizados por segundo.

Por outro lado, o algoritmo A* conseguiu se manter próximo à média de tempo gasta na sua execução no cenário anterior, com uma média de 2.57 milissegundos por *frame*.

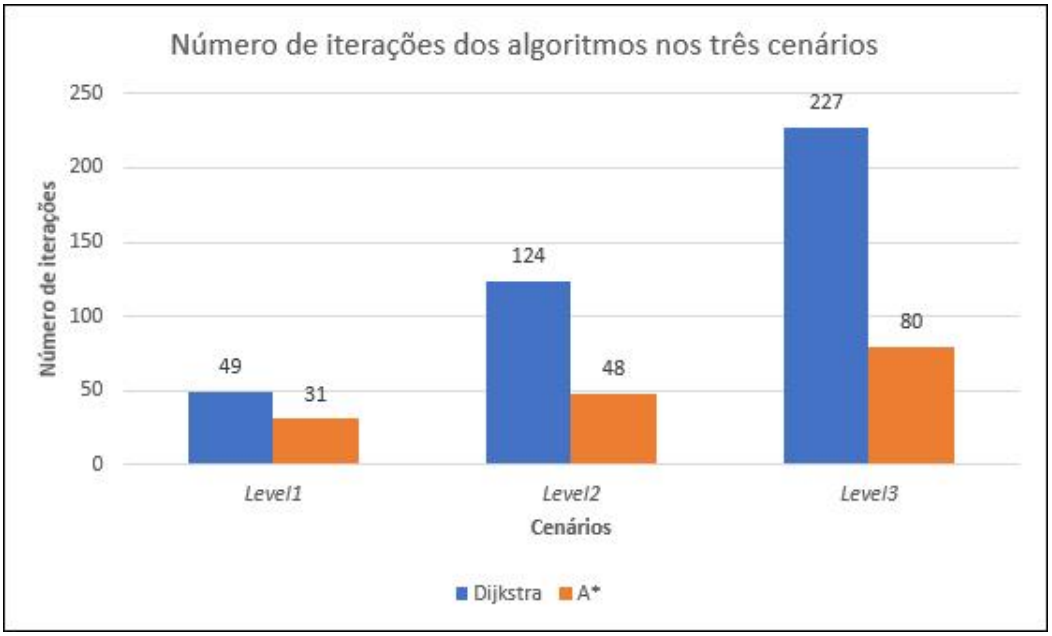
5.3 DISCUSSÃO DOS RESULTADOS

Ao analisar os resultados obtidos, é possível notar que a presença do algoritmo A* causou um impacto mínimo no fluxo do jogo, que para o jogador, é imperceptível. Considerando os três cenários desenvolvidos, que apesar de possuírem diferentes níveis de detalhamento podem ser considerados cenários simples e de baixa complexidade gráfica, nem mesmo o terceiro ambiente, que conta com a divisão em ilhas, foi capaz de impactar de forma relevante a execução do algoritmo A*. Por outro lado, o algoritmo de Dijkstra encontrou problemas neste último cenário, obtendo uma queda notável na taxa de *frames* do jogo.

Para as comparações das médias de *frames*, foi executado o jogo na ausência dos algoritmos para assim, obter a média de frames sem interferência dos mesmos. Nos três cenários, a média de *frames* na ausência de ambos os algoritmos se manteve em 60 *frames*, logo, este valor foi utilizado como base de um bom resultado nas execuções dos algoritmos.

Analisando primeiramente os resultados referentes ao número de iterações, o gráfico demonstrado na figura 62 dispõe estes dados em relação a ambos os algoritmos nos três cenários, a fim de facilitar a visualização da taxa de variação desta métrica.

Figura 63 – Variação no número de iterações dos algoritmos nos cenários

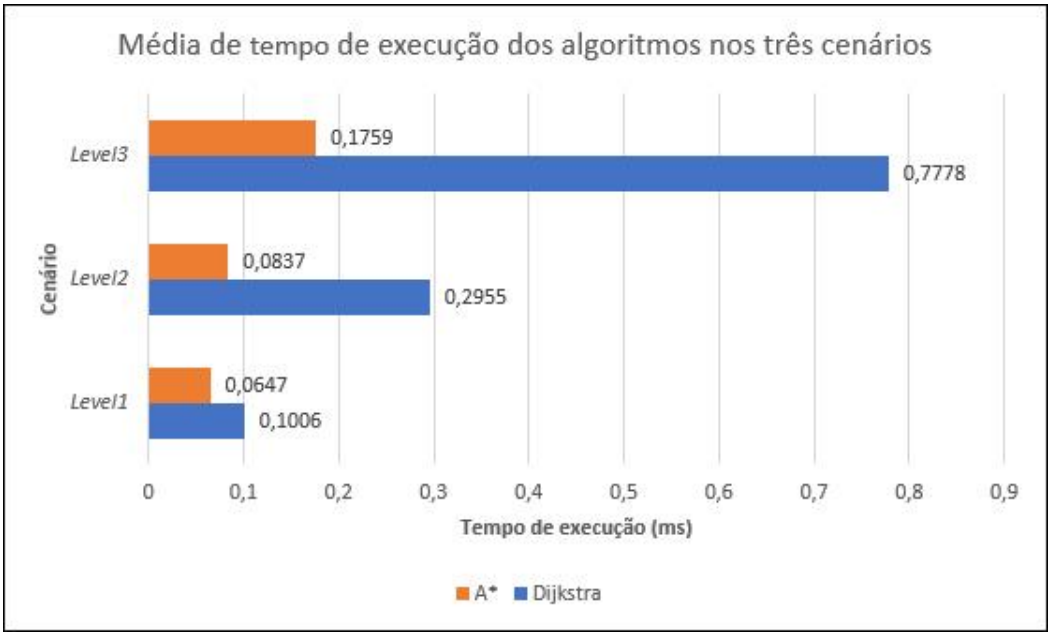


Fonte: Do autor.

Observando os dados da figura 62, é notável que o algoritmo de Dijkstra realiza um número consideravelmente maior de iterações em relação ao algoritmo A*, executando mais que o dobro de iterações no *Level2* e quase o triplo no *Level3*. Isso deve-se ao fato de que o algoritmo de Dijkstra calcula o menor caminho a partir do nó inicial para todos os outros nós do cenário, o que acaba fazendo com que haja a necessidade de visitar todos os nós e suas conexões com os nós vizinhos, diferentemente do A*, que conta com a sua heurística responsável por guiar a busca e visitar apenas os nós que estão em direção ao nó de destino.

Por conta de o algoritmo de Dijkstra realizar mais iterações, o mesmo acaba afetando uma outra métrica avaliada, o tempo total para a realização completa de uma busca, que de acordo com os resultados, também foi maior no algoritmo de Dijkstra em todas as execuções, como ilustrado no gráfico 63 abaixo, onde está sendo demonstrado a variação desta métrica ao longo dos três cenários.

Figura 64 – Variação de tempo de execução de uma busca dos algoritmos nos cenários



Fonte: Do autor.

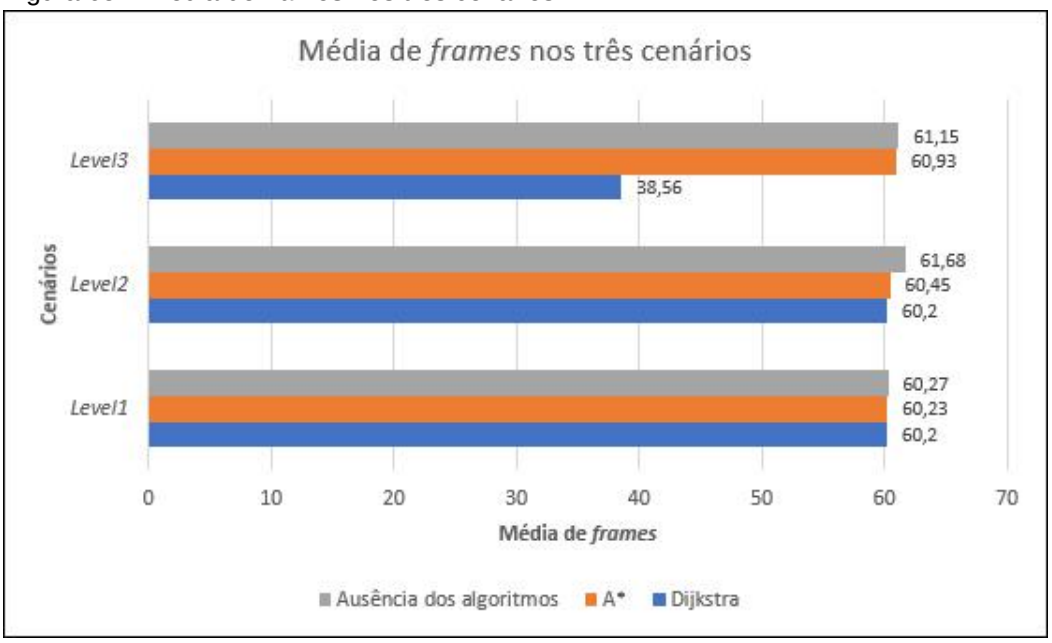
Observando a figura 63, a média de tempo para a realização de uma busca do algoritmo de Dijkstra, além de se apresentar maior nos três casos de comparação, também apresentou uma maior taxa de crescimento de um cenário para outro, em relação ao algoritmo A*, onde sofreu uma variação de tempo maior que o dobro no *Level2* em relação ao *Level1*, e do *Level3* em relação ao *Level2*. Por

outro lado, o algoritmo A* quase não apresentou aumento no tempo de execução do *Level2* em relação ao *Level1*, e, apesar de o tempo de execução ter crescido o dobro no último cenário em relação ao *Level2,* o algoritmo ainda manteve um tempo de execução consideravelmente menor em relação ao algoritmo de Dijkstra.

Com relação ao impacto dos algoritmos nos *frames* do jogo e suas respectivas médias de tempo consumidos por cada *frame*, ambos os algoritmos obtiveram uma boa performance nos dois primeiros cenários, apesar de o algoritmo de Dijkstra consumir mais tempo de *frame,* e conseguiram estabelecer uma média bastante similar. A diferença maior ocorreu no último ambiente modelado, neste o algoritmo de Dijkstra acabou consumindo um tempo de *frame* notavelmente maior, e por consequência, obteve uma menor taxa de *frames* ao longo da execução, se comparado ao A*.

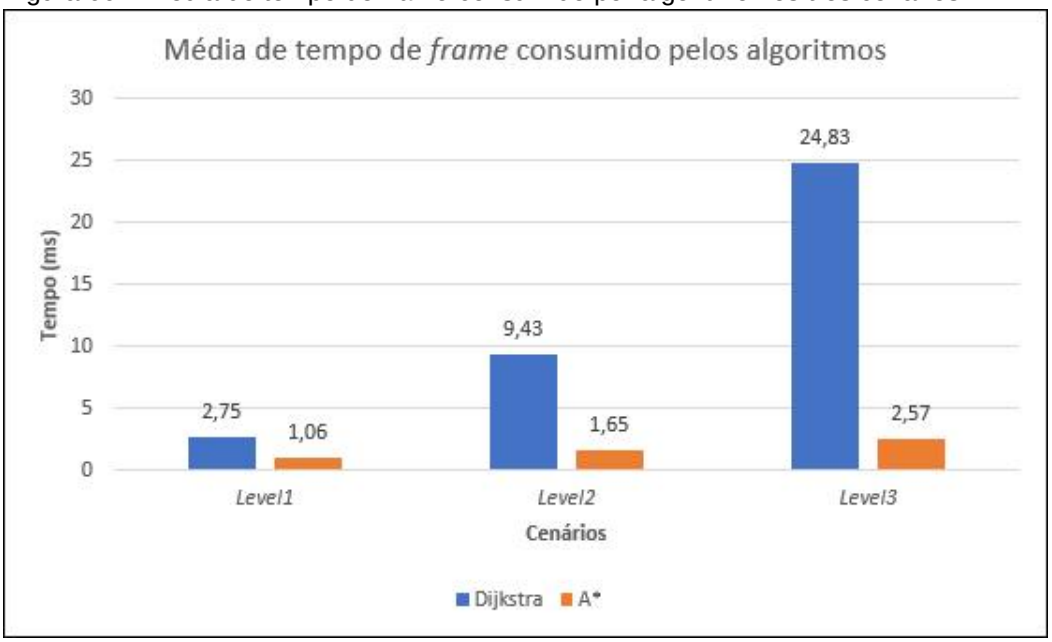
As figuras 64 e 65 demonstram os gráficos relacionados a estes dados, a fim de facilitar a leitura.

Figura 65 – Média de *frames* nos três cenários



Fonte: Do autor.

Figura 66 – Média do tempo de *frame* consumido por algoritmo nos três cenários



Fonte: Do autor.

Analisando os dados da figura 65, é possível observar que o algoritmo de Dijkstra demonstrou uma maior média de tempo consumido por *frame* nos três cenários, bem como uma taxa de crescimento consideravelmente maior conforme a complexidade dos cenários aumentou, onde os valores chegaram a triplicar na troca entre o *Level1* para o *Level2*. Porém, mesmo consumindo mais tempo por *frame*, nos dois primeiros ambientes não foi possível notar uma diminuição anormal na média de *frames* com a execução do algoritmo de Dijkstra, onde as duas execuções, sendo elas na presença do algoritmo de Dijkstra e na presença do algoritmo A*, demonstraram valores próximos e estáveis em 60 *frames*, sendo este valor, o mesmo que também foi obtido ao executar o jogo na ausência dos algoritmos.

Em relação ao último cenário testado, o algoritmo A* conseguiu manter uma média estável de 60 *frames* durante todo o tempo em que executou as buscas, e demonstrou uma média de consumo de tempo por *frame* de apenas 0.92 milissegundos superior à sua execução no ambiente *Level2*. O algoritmo de Dijkstra por outro lado, impactou na taxa de *frames* fazendo com que este valor sofresse uma grande queda, mantendo uma média de 38.56 *frames* durante a sua execução. A principal causa deste impacto no fluxo do jogo, como já foi descrito na discussão dos resultados do número de iterações e o tempo de execução por busca, é a necessidade que o algoritmo de Dijkstra têm de executar a busca de um nó do cenário para todos os outros, o que faz com que o aumento de número de nós de

um cenário ocasione um maior número de iterações, incrementando o tempo necessário para a finalização da busca e, consequentemente, o tempo consumido por *frame*.

Com relação à qualidade dos caminhos gerados, nenhum dos algoritmos demonstrou comportamentos imprevistos ou falhas na geração das rotas selecionadas.

É importante ressaltar que, mesmo o algoritmo A* tendo se mostrado mais performático nos cenários testados, o algoritmo de Dijkstra apresenta pontos positivos em relação ao A*. Um exemplo que pode ser destacado está na qualidade do caminho gerado, o algoritmo A* não garante encontrar o melhor caminho sempre, enquanto que o algoritmo de Dijkstra, pelo fato de realizar uma análise de todos os nós, apresenta uma maior confiabilidade de que o caminho gerado será o menor caminho possível. Além disso, quando o algoritmo de Dijkstra está realizando uma busca, pelo fato de esta busca não ser direcionada, ele consegue identificar atalhos que podem levar diretamente ao ponto de destino. Um exemplo seria algum tipo de tele transporte localizado um pouco atrás do nó de partida, na direção oposta ao nó de destino. Como o algoritmo A* realiza busca direcionada, este ignoraria o atalho, uma vez que ele se encontra na direção oposta ao ponto de destino, neste caso, o algoritmo de Dijkstra expandiria a busca até identificar este atalho, que poderia levar a entidade do jogo diretamente ao nó de destino, de forma instantânea, e neste caso, apresentando uma maior velocidade de busca.

Adicionalmente, um dos aprendizados extraídos destas comparações é que os ambientes modelados com o uso do motor Unity3D devem, preferencialmente, possuírem estruturas mais complexas, bem como mapas maiores, o que impactaria ainda mais nos resultados de ambos os algoritmos. Uma segunda possibilidade seria a inserção de agentes dinâmicos no cenário, ou seja, objetos que contam com movimentação e que também representam um nó no conjunto de nós do grafo, que ocasionaria em uma maior complexidade na geração dos caminhos.

6 CONCLUSÃO

Por meio da realização desta pesquisa, foi possível perceber a relevância dos algoritmos planejadores de caminho e o quão impactante a escolha destes tipos de algoritmos podem ser no processo de desenvolvimento de um jogo digital, influenciando diretamente seu desempenho, e consequentemente, a experiência do usuário.

A área da teoria dos grafos é bastante conhecida por solucionar problemas complexos de geração e manipulação de rotas em redes de internet, mapas geográficos, dentre outros, e no contexto de jogos digitais esta área também é de suma importância, fazendo com que a inteligência artificial acoplada a um agente do jogo tenha a capacidade de interpretar e se movimentar pelos cenários de forma inteligente e independente de um jogador o controlando.

Para o desenvolvimento das comparações apresentadas neste trabalho foram utilizados os conhecimentos de motores de jogos, com destaque para o motor Unity3D, além dos conhecimentos de desenvolvimento de jogos digitais, e também da área da teoria dos grafos por meio das implementações dos algoritmos solucionadores do problema do caminho mínimo. Cada etapa do estudo realizado foi essencial no desenvolvimento das comparações e cada tema teve sua relevância durante este processo. O conhecimento em motores de jogos permitiu que os cenários fossem modelados no Unity3D para atuarem como base para as comparações. O conhecimento adquirido de jogos digitais foi utilizado para a elaboração de uma estrutura básica para o desenvolvimento de um jogo. Por fim, o conhecimento em teoria dos grafos contribuiu com o protótipo desenvolvido por meio da estruturação do mapeamento do cenário e também para a implementação dos algoritmos de Dijkstra e A*.

O levantamento bibliográfico referente as áreas de jogos digitais e motores de jogos foi realizado por meio de títulos obtidos através da Universidade e de artigos internacionais, já nas áreas de grafos, como nos problemas de caminho e nas formas de mapeamento de cenários de jogos, foram utilizados como prioridade títulos de artigos acadêmicos nacionais e internacionais.

Após a finalização das comparações, foi possível verificar que todos os objetivos propostos nesta pesquisa foram alcançados, constatando as diferenças geradas nos dados através dos resultados obtidos.

Adicionalmente, por meio dos resultados das comparações, desenvolvedores de jogos digitais, principalmente os considerados independentes, podem se utilizar dos dados extraídos para que no processo de implementação de um jogo, consigam atingir uma melhor performance por meio da escolha correta do algoritmo de busca de caminho, quando este tipo de algoritmo for requisitado no processo de criação do jogo.

Durante a etapa de desenvolvimento, uma das dificuldades encontradas foi durante a inserção dos nós nos cenários, uma vez que não foi implementado um sistema automatizado para dispor os nós de forma distribuída nos terrenos modelados, tendo assim que serem inseridos manualmente. Uma outra dificuldade encontrada foi durante a etapa de coleta dos resultados, no momento em que se realizou a captura da média de *frames* das execuções dos algoritmos. Isto aconteceu porque, inicialmente seria realizada uma única execução dos algoritmos, porém, devido ao fato de que uma execução ocorre de forma muito rápida, consumindo um único *frame* de jogo, não foi possível obter uma média seguindo esta metodologia, isto porque para capturar a média de *frames* de um jogo, se faz necessário considerar um determinado intervalo de tempo para assim ser possível realizar a captura de vários *frames* e calcular a média dos mesmos ao final do tempo decorrido.

Como uma proposta de trabalho futuro, as comparações dos algoritmos poderiam serem realizadas em um motor de jogo diferente, remodelando os cenários neste outro motor e possivelmente utilizando outras formas de mapeamento nestes ambientes, como por exemplo o mapeamento por malha de navegação, analisando as diferenças com relação ao motor Unity3D e ao mapeamento por *waypoints*, bem como a utilização de outros algoritmos de busca de caminho, sejam esses algoritmos tradicionais ou algoritmos modificados pelo autor.

Uma outra proposta seria a utilização de objetos dinâmicos nos cenários, e não somente pontos estáticos distribuídos pelo ambiente, inserindo um agente que estaria em constante movimento representando um nó dinâmico, desse modo os algoritmos determinariam o caminho de um determinado nó até este agente em movimento, sendo possível realizar uma análise da qualidade do caminho.

REFERÊNCIAS

AHUJA, Ravindra K.; MAGNANTI, Thomas L.; ORLIN, James B.. **NETWORK FLOWS:** Theory, Algorithms, and Applications. New Jersey: Prentice-hall, 1993. 846 p.

BASTOS, Rodrigo; JAQUES, Patrícia A.. ANTARES: Um sistema Web de consulta de rotas de ônibus como serviço público. **Revista Brasileira de Computação Aplicada,** Passo Fundo, v. 2, n. 1, p.41-56, mar. 2010. Disponível em: <http://seer.upf.br/index.php/rbca/article/view/650/519>. Acesso em: 19 nov. 2017.

BATTAIOLA, André Luiz - **Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação** - XIX Jornada de Atualização em Informática/SBC – 2000.

BENIN, Max Ricardo. **Evolução de npc's e adversários em jogos de computador usando algoritmos genéticos.** 2007. 114 f. TCC (Graduação) - Curso de Sistemas de Informação, Faculdades Barddal, Florianópolis, 2007. Disponível em: <http://www.gsigma.ufsc.br/~popov/aulas/Publicacoes/TCC_MaxRicardoBenin_VersaoFinal.pdf>. Acesso em: 04 nov. 2017.

BRASILIENSE, Fabricio. **Desenvolvimento de um Framework de Jogos 3D para Celulares.** 2006. 93 f. TCC (Graduação) - Curso de Ciência da Computação, Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis, 2006. Disponível em: <https://projetos.inf.ufsc.br/arquivos_projetos/projeto_427/TCC.pdf>. Acesso em: 05 nov. 2017.

CANSIAN, Matheus. **A* with navigation meshes.** 2011. Disponível em: <https://medium.com/@mscansian/a-with-navigation-meshes-246fd9e72424>. Acesso em: 21 set. 2017.

CARDOSO, Ana Maria Dias Soares. **Teoria dos grafos:** uma reflexão sobre a sua abordagem no ensino não universitário. 2009. 102 f. Dissertação (Mestrado) - Curso de Matemática, Departamento de Inovação, Ciência e Tecnologia, Universidade Portucalense, Portugal, 2009. Disponível em: <http://repositorio.uportu.pt/xmlui/bitstream/handle/11328/568/TMMAT%20107.pdf?sequence=2&isAllowed=y>. Acesso em: 15 nov. 2017.

CLUA, Esteban W. G.; BITTENCOURT, João R.. **Desenvolvimento de Jogos 3D: Concepção, Design e Programação.** In: XXIV Jornada De Atualização Em Informática (JAI), 2005. p. 1 - 49. Disponível em:

<http://www2.ic.uff.br/~esteban/files/Desenvolvimento%20de%20jogos%203D.pdf>. Acesso em: 04 nov. 2017.

COUTINHO, André et al. **Jogos Digitais 3D:** Ferramentas e processos. Portugal, 2014. 26 p. Disponível em: <http://www.marcelohsantos.com.br/aulas/downloads/2Semestre_2017/fmu/programacao_integracao_jogos_I/Aula01_artigo.pdf>. Acesso em: 05 nov. 2017.

CUI, Xiao; SHI, Hao. **An overview of pathfinding in navigation mesh.** IJCSNS International Journal of Computer Science and Network Security, 12, 48–51, 2012. Disponível em: <http://paper.ijcsns.org/07_book/201212/20121208.pdf>. Acesso em: 23 set. 2017.

_____. **Direction Oriented Pathfinding In Video Games.** International Journal of Artificial Intelligence & Applications. 2011. Disponível em: <https://www.researchgate.net/publication/267405818_Direction_Oriented_Pathfinding_In_Video_Games>. Acesso em: 01 out. 2017.

FLEURY, Afonso; NAKANO, Davi; CORDEIRO, José H. D. O.. **Mapeamento da indústria brasileira e global Jogos Digitais.** São Paulo: Gedigames, 2014. 267 p. Disponível em: <http://www.abragames.org/uploads/5/6/8/0/56805537/mapeamento_da_industria_brasileira_e_global_de_jogos_digitais.pdf>. Acesso em: 28 out. 2017.

JUUL, Jesper. **Half-real:** Video Games between Real Rules and Fictional Worlds. Massachusetts: The Mit Press, 2005. 248 p.

KALANI, Chris. **Fixing Pathfinding Once and For All.** 2008. Disponível em: <http://www.cs.uu.nl/docs/vakken/mpp/other/path_planning_fails.pdf>. Acesso em: 21 set. 2017.

KARLSSON, Börje Felipe Fernandes. **Um Middleware de Inteligência Artificial para Jogos Digitais.** 2006. 127 f. Dissertação (Mestrado) - Curso de Informática, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2006. Disponível em: <https://www.maxwell.vrac.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=7861@1>. Acesso em: 24 maio 2017.

LUCCHESE, Fabiano; RIBEIRO, Bruno. **Conceituação de Jogos Digitais.** Campinas: Universidade Estadual de Campinas, 2009. 16 p. Disponível em: <http://www.dca.fee.unicamp.br/~martino/disciplinas/ia369/trabalhos/t1g3.pdf>. Acesso em: 21 maio 2017.

MARQUES, Gabriel Cavalcanti. **Introdução ao desenvolvimento de jogos digitais utilizando o motor de jogo UDK.**2015. 119 f. Dissertação (Mestrado) - Curso de Tecnologias da Inteligência e Design Digital, Pontífica Universidade Católica de São Paulo, São Paulo, 2015. Disponível em: <https://sapientia.pucsp.br/bitstream/handle/18168/1/Gabriel Cavalcanti Marques.pdf>. Acesso em: 05 nov. 2017.

MELLO, Gustavo; ZENDRON, Patricia. **BNDES Setorial 42.** Rio de Janeiro: Bndes Setorial, 2015. 498 p. Disponível em: <https://web.bndes.gov.br/bib/jspui/handle/1408/9374>. Acesso em: 29 out. 2017

OSÓRIO, Fernando. et al. **Inteligência artificial para jogos:** agentes especiais com permissão para matar... e raciocinar! 2007. 4 f. Trabalho de Conclusão de Curso (Graduação em Jogos Digitais) - Universidade do Vale do Rio dos Sinos, São Leopoldo, 2007. Disponível em: <http://osorio.wait4.org/publications/Osorio-et-al-SBGames07-Tutorial.pdf> Acesso em: 24 set. 2017.

PASSOS, Erick B. et al. **Tutorial: Desenvolvimento de Jogos com Unity 3D**. In: Brazilian Symposium on Games and Digital Entertainment, 8., 2009, Rio de Janeiro. Disponível em: <https://www.sbgames.org/~sbgameso/papers/sbgames09/computing/tutorialComputing2.pdf>. Acesso em: 06 nov. 2017.

PATEL, Amit. **Map representations.** 2017. Disponível em: <http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html>. Acesso em: 10 set. 2017.

_____. **Introduction to A*.** 2017. Disponível em: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm>. Acesso em: 21 out. 2017.

PAZERA, Ernest. **ISOMETRIC GAME PROGRAMMING WITH DIRECTX 7.0.** California: Prima Tech, 2001. 720 p.

PELECHANO, Núria G.; FUENTES, C. **Hierarchical path-finding for Navigation Meshes (HNA*).** Computers & graphics. 2016, vol. 59, p. 68-78. Disponível em: <https://www.cs.upc.edu/~npelechano/Pelechano_HNAstar_prePrint.pdf>. Acesso em: 26 ago. 2017.

POTTINGER, Dave. **Coordinated Unit Movement.** 1999. Disponível em: <http://www.gamasutra.com/view/feature/131720/coordinated_unit_movement.php>. Acesso em: 27 maio 2017.

POZZER, Cesar Tadeu. **Planejamento de Caminho.** Rio Grande do Sul: Universidade Federal de Santa Maria, 2007. 39 p. Disponível em: <http://www-usr.inf.ufsm.br/~pozzer/disciplinas/pj3d_path.pdf>. Acesso em: 20 maio 2017.

QING, Guo; ZHENG, Zhang; YUE, Xu. Path-planning of automated guided vehicle based on improved Dijkstra algorithm. 2017 29th Chinese Control And Decision Conference (ccdc), [s.l.], p.7138-7143, maio 2017. IEEE. http://dx.doi.org/10.1109/ccdc.2017.7978471. Disponível em: <http://ieeexplore.ieee.org/document/7978471/>. Acesso em: 15 nov. 2017.

RABIN, Steve. **AI Game Programming Wisdom.** Boston: Charles River Media, 2002. 658 p.

_____. **GAME AI PRO:** Collected wisdom of game ai professionals. Londres: Crc Press, 2014. 580 p.

_____. **Introdução ao Desenvolvimento de Games Vol I.** São Paulo: Cengage Learning, 2012. 192 p.

_____. **Introdução ao desenvolvimento de games Vol II.** 2. ed. São Paulo: Cengage Learning, 2012. 637 p. Tradução de: Opportunity Translations.

_____. **Introdução ao Desenvolvimento de Games Vol III.** 3. ed. São Paulo: Cengage Learning, 2013. 190 p.

REDDY, Pooja et al. Research on the optimization of dijkstra's algorithm and its applications. International Journal Of Science, Technology & Management, Ghaziabad, v. 04, n. 01, p.304-309, abr. 2015. Disponível em: <http://www.ijstm.com/images/short_pdf/1430806082_453.pdf>. Acesso em: 02 out. 2017.

SANTOS, Raul Joaquim C. dos; SANTOS, Selan Rodrigues dos. **Bricking: Modelagem Tridimensional de Cenários de Jogos em Camadas**. Natal, p.33-36, out. 2009. Disponível em: <http://www.sbgames.org/papers/sbgames09/computing/short/cts9_09.pdf>. Acesso em: 07 set. 2017.

SCHUYTEMA, Paul. **Design de Games - Uma Abordagem Prática.** São Paulo: Cengage Learning, 2008. 472 p.

SILVA, Fernandes D.; SANCHES, Leme A.. **Aplicação conjunta do método de Dijkstra e otimização combinatória para solução do problema do caixeiro viajante.** SegeT Simpósio de Excelência em Gestão e Tecnologia, 2009. Disponível em:

<https://www.aedb.br/seget/arquivos/artigos09/224_224_224_Artigo_Seget.pdf>. Acesso em: 07 out. 2017.

SUN-GI, Hong; SUNG-WOO, Kim; JU-JANG Lee. **The minimum cost path finding algorithm using a Hopfield type neural network.** *Proceedings of 1995 IEEE International Conference on Fuzzy Systems.*, Yokohama, 1995, pp. 1719-1726 vol.4.

VERMETTE, Jonathan. **A Survey of Path-finding Algorithms Employing Automatic Hierarchical Abstraction**. University of Windsor, Canadá, 2011. Disponível em: <http://richard.myweb.cs.uwindsor.ca/cs510/vermette_survey.pdf> Acesso em: 24 set. 2017.

XAVIER, Thomaz C.. **Estudo e desenvolvimento de jogos para internet utilizando Unity 3d.** 2011. 75 f. Monografia (Especialização) - Curso de Sistemas Para Internet, Instituto Federal Sul-rio-grandense, Passo Fundo, 2011. Disponível em: <http://painel.passofundo.ifsul.edu.br/uploads/arq/20150522100544196177278l.pdf>. Acesso em: 08 nov. 2017.

ZAMBIASI, Saulo P.; PINHEIRO, Patricia L.B. **Os Jogos de Computador e a Experiência Interativa: Do Espaço Virtual ao Real.** Computer on the Beach 2010. Florianópolis - SC. 2010. Disponível em: <http://www.gsigma.ufsc.br/~popov/aulas/Publicacoes/ZambiasiPinheiro_ComputerOnTheBech2010.pdf>. Acesso em: 22 out. 2017.

APÊNDICES

APÊNDICE A – ARTIGO CIENTÍFICO

Comparação dos algoritmos de busca de caminho, A* e Dijkstra, aplicado em cenários de jogos

Guilherme Corrêa Milak[1], Luciano Antunes[2], Matheus Leandro Ferreira[3]

[1]Universidade Do Extremo Sul Catarinense (UNESC) – Criciúma, SC - Brasil

guilherme.milak@hotmail.com, luc@unesc.net, mlf@unesc.net

Abstract. *The game design market has been expanding in a spiking fashion. Each year the game industry launches a variety of titles relying more and more on technologically advanced graphic resources. The increasing use of these resources oftentimes depend on the graphics processing hardware unit. However, it is directly linked to how the game is conceived. In a variety of genres, the behaviors of the characters and other entities can also further impact the overall performance and, consequently, the player experience. One of the methods responsible for providing intelligence to a game entity is known as pathfinding. In this project, the Unity 3D game engine was in charge of three-dimensional terrain modelling, in which the pathfinding algorithms A* and Dijkstra were implemented and the paths were generated, further allowing them to be compared to each other. Among the scenarios that were subject to testing, the gathered results demonstrate that as the obstacle complexity rises, the impact of the algorithms in the game's flow increases as well.*

Resumo*. O mercado de Jogos Digitais cresce de forma constante e acelerada. A indústria de jogos lança todo ano diversos títulos que contam com, cada vez mais, recursos gráficos de alto poder tecnológico. O uso destes recursos depende muitas vezes do hardware de processamento gráfico, mas também está diretamente ligado à forma como um jogo digital é idealizado e implementado. Em diversos gêneros de jogos, o comportamento dos personagens e de outras entidades relacionadas ao jogo também impactam na performance do mesmo, e consequentemente, na experiência do jogador. Uma das técnicas responsáveis por prover inteligência à uma entidade de um jogo é conhecida como busca de caminho. Nesta pesquisa, utilizando-se do motor de jogo Unity3D para a modelagem de cenários tridimensionais, foram implementados os algoritmos de busca de caminho Dijkstra e A* que foram utilizados para a geração de caminhos dentro dos cenários modelados, viabilizando uma comparação entre ambos. Dentre os cenários testados, os resultados obtidos demonstraram que, conforme a complexidade dos cenários aumenta, os algoritmos tendem a impactar de forma crescente o fluxo do jogo.*

1. Introdução

Um jogo eletrônico é uma atividade lúdica formada por ações e decisões que resultam numa condição final. Tais ações e decisões são limitadas por um conjunto de regras e por um universo, que no contexto dos jogos digitais, são regidos por um programa de computador (Schuytema, 2008).

Atualmente, com o surgimento de motores de jogos gratuitos e poderosos como o Unity3D e o Unreal Engine o uso de recursos como Inteligência Artificial (IA), que se referindo a jogos digitais, incorpora funções essenciais para o funcionamento do mesmo, foi altamente facilitado. Funcionalidades como som, cálculos e iluminação já veem implementados e previamente configurados.

Um dos desafios de um jogador em um determinado jogo é encontrar o melhor caminho que o personagem deverá percorrer até chegar ao seu ponto de destino. Os motores de jogos permitem que algoritmos de busca de melhor caminho como o A*, Floyd-Warshall, Johnson, Dijkstra, Bellman-Ford, entre outros, sejam implementados. Porém, dentre estes algoritmos, a dúvida é, qual deles é o mais adequado para uso nessas situações.

Neste artigo, é realizada uma comparação empírica entre os algoritmos A* e Dijkstra, utilizando o motor de jogo Unity3D e cenários específicos a fim de realizar uma avaliação de ambos.

2. Jogos digitais

Jogos desenvolvidos digitalmente podem ser descritos como uma ramificação de jogos não digitais. Dentre os usuários de computadores, grande parte já passou horas se divertindo em algum tipo de jogo digital (BATTAIOLA, 2000).

Um jogo digital, ou eletrônico, como também pode ser chamado, é descrito por Schuytema (2008), como uma atividade lúdica elaborada por ações e decisões que, juntas, resultam em uma condição final. Neste contexto, um programa de computador limita essas ações por meio de um universo criado virtualmente, e um conjunto de regras. O universo concretiza as ações dos jogadores, enquanto as regras definem o que pode e o que não pode ser realizado, assim como geram as consequências das ações do jogador.

2.1. Motor de Jogo

Um motor de jogo é um programa de computador projetado para abstrair funcionalidades do desenvolvedor oferecendo módulos e funcionalidades prontas e otimizadas para serem usadas pelo criador do jogo. Em essência, as características de um bom motor envolvem possuir um sistema de renderização 3D, simulador de física, suporte para uma ou mais linguagens de programação para a criação dos scripts, editor de cena integrado e capacidade de importação de dados como imagens, sons, modelos, entre outros. (PASSOS et al., 2009).

2.1.1 Unity 3D

O motor de jogo Unity 3D tem se tornado cada vez mais famoso entre os desenvolvedores de jogos. A ferramenta oferece as tecnologias mais atualizadas em relação à renderização de modelos 3D, imagens, texturas, materiais, partículas e iluminação. O motor conta ainda com um criador de terrenos em tempo real, motor de física, suporte para mais de uma linguagem de programação, áudio e rede, para criação de jogos multiplayer (XAVIER, 2011).

Há três tipos de licença presentes na Unity, sendo elas a versão *Personal*, *Plus* e *Pro*. A versão *Personal* é gratuita, e disponível para quem está aprendendo ou começando a desenvolver sem querer ter gastos, podendo inclusive publicar seus jogos sem ter custos. Já as versões *Plus* e *Pro* são para desenvolvedores que desejam mais recursos do motor (MARQUES, 2017).

3. Mapeamento de cenários

Para realizar uma busca de caminho em um cenário, um agente ou o sistema de busca de caminhos precisa entender o nível, não necessitando de um modelo completo e detalhado, mas uma representação que leve em consideração as informações mais importantes e relevantes (RABIN, 2012).

3.1 Mapeamento por *waypoint*

Os gráficos de *waypoints* representam o mundo como um gráfico abstrato, e não possuem um mapeamento explícito entre os nós no gráfico e o espaço a percorrer (RABIN, 2014).

Neste esquema, a navegação é realizada mapeando-se pontos de visibilidade ou *waypoints*, inseridos nas partes mais importantes do cenário. Cada ponto necessita ter ao menos uma conexão ou link, como também pode ser chamado, direta com um outro ponto do cenário. Um link conecta exatamente dois pontos entre si, indicando que um agente pode se movimentar sem encontrar grandes obstáculos entre esses dois pontos (RABIN, 2012).

A figura 1 exemplifica um cenário utilizando mapeamento por *waypoints*.

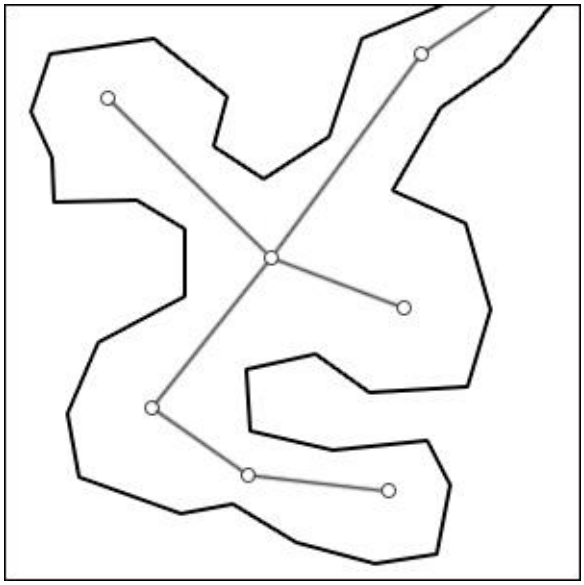


Figura 1. Mapeamento de cenário por waypoints

4. Problema de caminho mínimo em jogos

Em muitos jogos é necessário simular milhares ou milhões de agentes interagindo com o ambiente, e dentre essas interações, a navegação é de grande importância, pois é por meio desta que os agentes podem perseguir, procurar ou interceptar outros agentes no cenário do jogo. A busca de caminho fornece aos personagens a capacidade de navegar em um ambiente de forma autônoma (PELECHANO; FUENTES, 2016, tradução nossa).

4.1. Algoritmo de Dijkstra

O algoritmo de Dijkstra foi introduzido pelo cientista da computação Edsger Dijkstra em 1956, e publicado em 1959, com o objetivo de calcular o menor caminho em uma rede, partindo de um nó inicial e calculando as menores distâncias para todos os outros nós presentes na rede (REDDY et al., 2015).

Tendo escolhido um ponto, ou nó, como origem, o algoritmo calcula o menor caminho para todos os outros nós presente no grafo, partindo de uma estimativa inicial para cada nó. Conforme a iteração acontece entre esses nós, o algoritmo vai ajustando essa estimativa,

sempre com o objetivo de diminuir o custo de navegação entre um ponto e outro (SILVA; SANCHES, 2009).

4.2. Algoritmo A*

O algoritmo A*, assim como o Dijkstra, também encontra um caminho entre dois pontos distintos em um determinado mapa. Embora haja diferentes tipos de algoritmos que realizam este mesmo trabalho, o A* encontrará o caminho mais curto entre dois pontos, se houver um caminho, e fará isso relativamente rápido (RABIN, 2002, tradução nossa).

Adicionalmente, de acordo com Rabin (2002, tradução nossa), este algoritmo é considerado direcionado, ou seja, ao decorrer da busca, utilizando-se de heurísticas, o algoritmo avalia a melhor direção para ir ao encontro do local de destino, não gastando tempo avaliando todas as direções possíveis, o que o torna um algoritmo bastante flexível.

5. Comparação dos algoritmos de busca de caminho, A* e Dijkstra, aplicado em cenários de jogos

Algoritmos de busca de caminho são largamente utilizado em jogos digitais, para a determinação e geração de rotas, que são utilizadas pelos personagens do jogo para se movimentarem sem correrem o risco de colidirem com os obstáculos do ambiente.

Em consequência disso, utilizando-se do motor de jogo Unity3D, serão implementados três cenários distintos, com o objetivo de executar os algoritmos Dijkstra e A* nestes cenários e comparar os resultados extraídos de suas execuções, utilizando-se para isso, de uma ferramenta de medição de desempenho incluída no motor utilizado.

5.1 *Scripts* de mapeamento do cenário

Os scripts responsáveis por mapear o cenário fazem com que ambos os algoritmos, Dijkstra e A*, tenham a capacidade de reconhecer os pontos navegáveis, também conhecidos como os nós de um grafo.

Para isto, foram implementadas duas classes que representam esses nós, utilizando a nomenclatura *Node* e *NodeView*. A classe *Node* é responsável por representar os dados de cada nó individual, como sua distância atual em relação ao nó inicial e suas arestas, enquanto a classe *NodeView* mantém a representação gráfica deste mesmo ponto no cenário, como sua posição em relação ao ambiente.

Foram criadas também duas classes que realizam a leitura desses nós ao carregar os cenários, rotuladas de *Graph* e *GrahpView*. A classe *GraphView* realiza a leitura visual dos nós e envia os dados de cada nó lido para a classe *Graph*, que mantém os dados de todos os nós visíveis para os algoritmos utilizarem nas buscas.

5.2 Desenvolvimento dos algoritmos de Dijkstra e A*

A implementação dos algoritmos de Dijkstra e A* foram realizadas utilizando a linguagem de programação C#, no motor Unity3D.

Cada algoritmo foi desenvolvido na sua forma tradicional e em classes separadas nomeadas de Dijkstra e A*. Foram implementadas também outras duas classes rotuladas de *TestDijkstra* e *TestAStar* responsáveis por chamar os respectivos métodos de busca de cada algoritmo, demonstrando o caminho gerado no ambiente por meio de linhas visuais.

5.4 Modelagem dos cenários tridimensionais

Foram desenvolvidos três cenários, nomeados de *Level1*, *Level2* e *Level3*, com o propósito de criar três níveis diferentes de detalhamento, seguindo uma metodologia de modelagem onde o

primeiro cenário possui um nível de detalhamento mais básico, o segundo apresentando um nível de detalhamento intermediário, e o último por sua vez, contando com o terreno mais complexo dentre todos.

Na figura 2 é demonstrado o cenário nomeado de *Level1*, sendo este o mais simples dentre os três. Nele, os objetos em cor cinza escuro representam as paredes, e os cubos em cor azul, os nós. Foi utilizado ainda, textos localizados acima de cada nó para representar sua nomenclatura. Para as três comparações realizadas neste cenário, foram utilizados como nós de início, destacados em cor verde, os nós 1, 11 e 27 e seus nós de destino correspondentes, destacados em cor amarela, foram definidos como 18, 38 e 4, respectivamente. As linhas de cor branca no cenário estão inseridas apenas para auxiliar na visualização de cada nó de início com seu nó de destino correspondente.

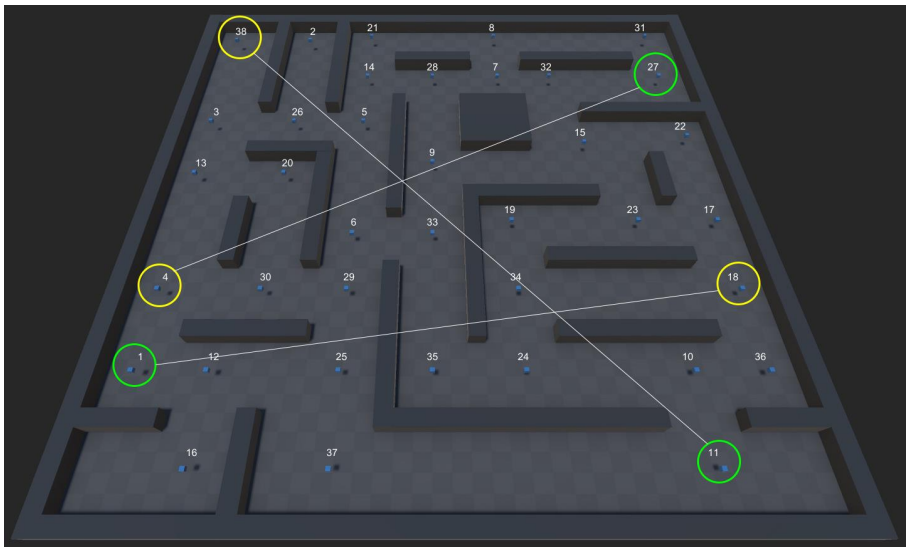


Figura 2. Representação do cenário Level1

O segundo cenário modelado, nomeado de *Level2*, apresenta uma complexidade um pouco maior em relação ao *Level1*. Nele, foi inserido uma quantidade consideravelmente maior de nós, além da utilização de mais relevos no terreno, com a inserção de dois lagos no cenário, bloqueando algumas rotas.

A figura 3 ilustra o segundo ambiente modelado. Nele, além da maior exploração do recurso de elevação do terreno, como já descrito, foram inseridas também árvores que também são responsáveis por bloquear os caminhos atuando como obstáculos. Para as comparações neste cenário, foram utilizados como nós de início, os nós 8, 68 e 72 e como seus nós de destino correspondentes, os nós 63, 27 e 21.

Figura 3. Representação do cenário Level2

Por fim, foi modelado o último cenário onde foram aplicadas as comparações. Neste, os recursos de alteração de terreno foram altamente utilizados, dando origem à três grandes ilhas que geraram grande parte dos obstáculos presentes no ambiente. Além disso, foram utilizados também outros modelos tridimensionais, como rochas e construções, para representar obstáculos menores em cada uma das ilhas.

A figura 4 ilustra uma visão completa do cenário, englobando todas as três ilhas modeladas, com suas conexões por meio de pontes. Este cenário, conta com o maior número de nós dentre todos os três desenvolvidos, resultando em uma quantidade consideravelmente maior de conexões entre os mesmos, o que também contribui para o aumento da complexidade do cenário. Para as comparações neste cenário, foram utilizados como nós de início, os nós 15, 118 e 86 e como seus nós de destino correspondentes, os nós 65, 26 e 51.

Figura 4. Representação do cenário Level3

5.5. Testes Realizados

Foram realizadas três execuções de ambos algoritmos em cada ambiente, como descrito na seção de modelagem dos cenários, utilizando os mesmos nós de início e de destino e analisando as seguintes métricas:

a) número de iterações realizadas pelo algoritmo;
b) tempo gasto para atingir o nó de destino após a busca ter sido iniciada;
c) impacto da execução nos frames do jogo;
d) o tempo de frame tomado por cada algoritmo.

Após a finalização das execuções dos algoritmos e da obtenção dos dados foi realizada a comparação dos algoritmos Dijkstra e A*.

5.6. Resultados obtidos

Os resultados referentes ao número de iterações realizadas por cada algoritmo e seus tempos de execução foram obtidos por meio de uma média aritmética das três buscas em cada cenário.

Em relação aos dados do impacto de cada algoritmo nos *frames* do jogo, foi realizado uma execução do jogo na ausência de ambos os algoritmos, para assim, capturar a média de *frames* sem que os algoritmos impactassem no resultado. Logo após esta execução, foram estabelecidos 100 agentes, que juntos, executaram várias buscas durante cinco segundos de forma ininterrupta, e com o auxílio de um script de captura de *frames*, foi calculado a média final de *frames* na presença de cada algoritmo, que foram executados de forma individual.

Por fim, para a medida do tempo de *frame*, ou seja, o tempo médio que cada algoritmo consome em um *frame* do jogo, foram executadas as buscas seguindo a mesma metodologia usada na medida do impacto nos *frames*, executando os algoritmos contando com 100 agentes durante o tempo de cinco segundos, e com o auxílio do *Profiler*, foi realizada a leitura dos dados.

O gráfico demonstrado na figura 5 dispõe estes dados referentes ao número de iterações em relação à ambos os algoritmos nos três cenários.

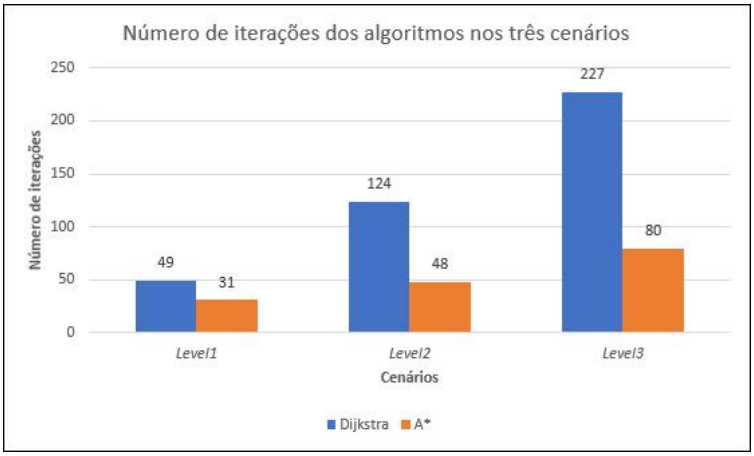


Figura 5. Média de iterações por algoritmo nos três cenários

O gráfico ilustrado na figura 6, demonstra a variação do tempo total necessário para a execução de uma única busca por cada algoritmo, em cada um dos três cenários.

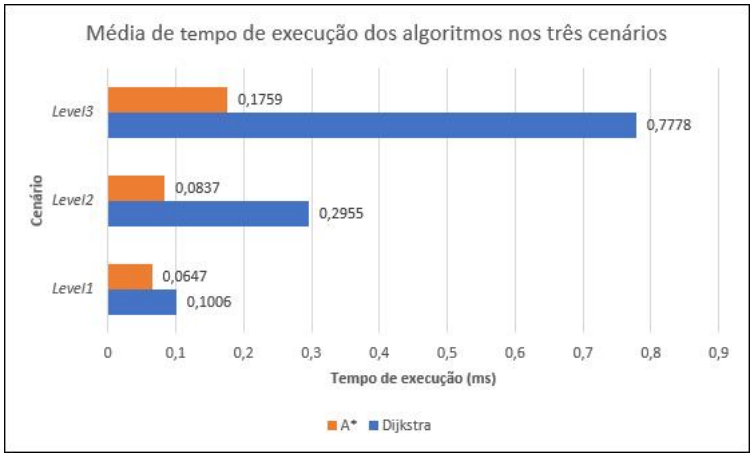


Figura 6. Variação de tempo de execução de uma busca dos algoritmos nos cenários

As figuras 7 e 8 ilustram a média de *frames* obtida durante as execuções nos três cenários, bem como a média de tempo de *frame* gasto por cada algoritmo.

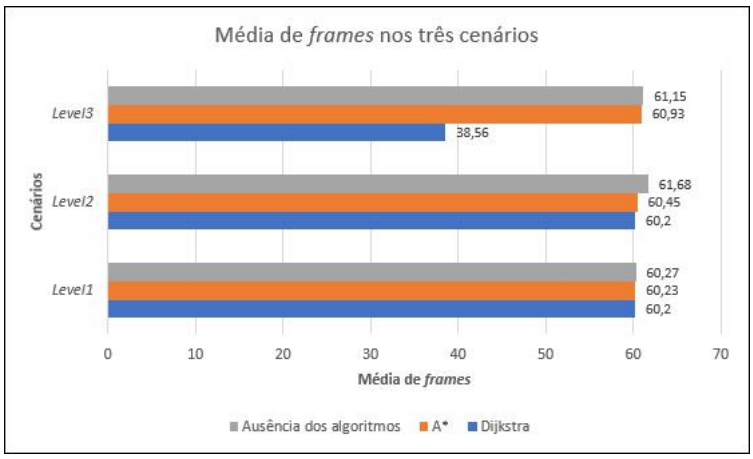


Figura 7. Média de frames nos três cenários

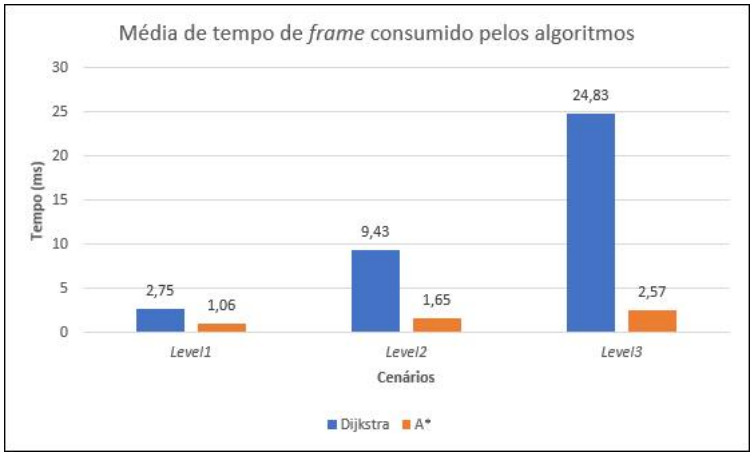


Figura 8. Média do tempo de frame consumido por algoritmo nos três cenários

6. Conclusão

Analisando os resultados obtidos é possível observar que o algoritmo de Dijkstra demonstrou uma maior média de tempo consumido por *frame* nos três cenários, bem como uma taxa de crescimento consideravelmente maior conforme a complexidade dos cenários aumentou, onde

os valores chegaram a triplicar na troca entre o *Level1* para o *Level2*. Porém, mesmo consumindo mais tempo por *frame*, nos dois primeiros ambientes não foi possível notar uma diminuição anormal na média de *frames* com a execução do algoritmo de Dijkstra, onde as duas execuções, sendo elas na presença do algoritmo de Dijkstra e na presença do algoritmo A*, demonstraram valores próximos e estáveis em 60 *frames*, sendo este valor, o mesmo que também foi obtido ao executar o jogo na ausência dos algoritmos.

Em relação ao último cenário testado, o algoritmo A* conseguiu manter uma média estável de 60 *frames* durante todo o tempo em que executou as buscas, e demonstrou uma média de consumo de tempo por *frame* de apenas 0.92 milissegundos superior à sua execução no ambiente *Level2*. O algoritmo de Dijkstra por outro lado, impactou na taxa de *frames*, mantendo uma média de 38.56 *frames* durante a sua execução. A principal causa deste impacto no fluxo do jogo, é a necessidade que o algoritmo de Dijkstra têm de executar a busca de um nó do cenário para todos os outros, o que faz com que o aumento de número de nós de um cenário ocasione um maior número de iterações, incrementando o tempo necessário para a finalização da busca e, consequentemente, o tempo consumido por *frame*.

Com relação à qualidade dos caminhos gerados, nenhum dos algoritmos demonstrou comportamentos imprevistos ou falhas na geração das rotas selecionadas.

Por meio da realização desta pesquisa, foi possível perceber a relevância dos algoritmos planejadores de caminho e o quão impactante a escolha destes tipos de algoritmos podem ser no processo de desenvolvimento de um jogo digital, influenciando diretamente seu desempenho, e consequentemente, a experiência do usuário.

Referências

BATTAIOLA, André Luiz - **Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação** - XIX Jornada de Atualização em Informática/SBC – 2000.

MARQUES, Gabriel Cavalcanti. **Introdução ao desenvolvimento de jogos digitais utilizando o motor de jogo UDK.**2015. 119 f. Dissertação (Mestrado) - Curso de Tecnologias da Inteligência e Design Digital, Pontífica Universidade Católica de São Paulo, São Paulo, 2015. Disponível em: <https://sapientia.pucsp.br/bitstream/handle/18168/1/Gabriel Cavalcanti Marques.pdf>. Acesso em: 05 nov. 2017.

PASSOS, Erick B. et al. **Tutorial: Desenvolvimento de Jogos com Unity 3D**. In: Brazilian Symposium on Games and Digital Entertainment, 8., 2009, Rio de Janeiro. Disponível em: <https://www.sbgames.org/~sbgameso/papers/sbgames09/computing/tutorialComputing2.pdf >. Acesso em: 06 nov. 2017.

PELECHANO, Núria G.; FUENTES, C. **Hierarchical path-finding for Navigation Meshes (HNA*).** Computers & graphics. 2016, vol. 59, p. 68-78. Disponível em: <https://www.cs.upc.edu/~npelechano/Pelechano_HNAstar_prePrint.pdf>. Acesso em: 26 ago. 2017.

RABIN, Steve. **AI Game Programming Wisdom.** Boston: Charles River Media, 2002. 658 p.

_____. **Introdução ao Desenvolvimento de Games Vol I.** São Paulo: Cengage Learning, 2012. 192 p.

_____. **GAME AI PRO:** Collected wisdom of game ai professionals. Londres: Crc Press, 2014. 580 p.

REDDY, Pooja et al. Research on the optimization of dijkstra's algorithm and its applications. **International Journal Of Science, Technology & Management,** Ghaziabad, v.

04, n. 01, p.304-309, abr. 2015. Disponível em: <http://www.ijstm.com/images/short_pdf/1430806082_453.pdf>. Acesso em: 02 out. 2017.

SCHUYTEMA, Paul. **Design de Games - Uma Abordagem Prática.** São Paulo: Cengage Learning, 2008. 472 p.

SILVA, Fernandes D.; SANCHES, Leme A.. **Aplicação conjunta do método de Dijkstra e otimização combinatória para solução do problema do caixeiro viajante.** SegeTSimpósio de Excelência em Gestão e Tecnologia, 2009. Disponível em: <https://www.aedb.br/seget/arquivos/artigos09/224_224_224_Artigo_Seget.pdf>. Acesso em: 07 out. 2017.

XAVIER, Thomaz C.. **Estudo e desenvolvimento de jogos para internet utilizando Unity 3d.** 2011. 75 f. Monografia (Especialização) - Curso de Sistemas Para Internet, Instituto Federal Sul-rio-grandense, Passo Fundo, 2011. Disponível em: <http://painel.passofundo.ifsul.edu.br/uploads/arq/20150522100544196177278l.pdf>. Acesso em: 08 nov. 2017.