UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ENGENHARIA DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

CAUÊ CHAVES VICENTE

ALGORITMO DE PATHFINDING APLICADO EM VIDEO-GAMES

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO
2017

CAUÊ CHAVES VICENTE

ALGORITMO DE PATHFINDING APLICADO EM JOGOS DIGITAIS

Trabalho de conclusão de curso de graduação apresentado à disciplina Trabalho de Conclusão de Curso 1, do Curso Superior de Engenharia de Computação da Universidade Tecnológica Federal do Paraná, Campus Cornélio Procópio como avaliação para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Paulo Augusto Nardi

CORNÉLIO PROCÓPIO
2017

LISTA DE ABREVIAÇÕES

Video-Game: VG

PC: Playable Character

NPC: Non-Playable Character

MOBA: Multiplayer Online Battle Arena

LRA*: Local Repair A*

CA*: Cooperative A*

HCA*: Hierarchical Cooperative A*

RRA*: Reverse Resumable A*

WHCA*: Windowed Hierarchical Cooperative A*

LISTA DE FIGURA

SUMÁRIO

1 INTRODUÇÃO

Jogos são uma forma de recreação constituída por um conjunto de regras que especificam um objeto a ser conquistado e as formas de conquistá-lo[2].

Jogos são importantes elementos de cultura e possuem contribuição significativa na formação de cada indivíduo, sendo algo necessário a ser estudado com maior profundidade[1].

Jogos podem ser classificados em diversas categorias: cartas, tabuleiro, atlético, infantis e educativos [2]. Porém em 1961, o jogo SpaceWar permitiu o surgimento de um novo tipo de jogo, o videogame (VG) [3].

Jogos eletrônicos são uma categoria que compreende os jogos que possuem componentes eletrônicos. VG é uma subcategoria dos jogos eletrônicos, pois ambos são dotados de componentes eletrônicos, entretanto nem todos os jogos eletrônicos possuem um monitor como dispositivo de saída, fator predominante nos VG. Jogos como pinball e máquinas caça-níquel, por exemplo, são ditos eletrônicos, porém não são videogames[3].

Um VG pode ser definido como um jogo computadorizado que é jogado com um controle eletrônico que serve de input para um software que manipula símbolos em um anteparo de exibição visual[3].

Um requisito muito importante ao se jogar jogos digitais (e, portanto, VGs) é a imersão que o jogador sente, que no contexto de jogos digitais significa o engajamento que uma pessoa sente ao jogar, se assemelhando à ideia de imersão em água[5].

//falar sobre imersão (Immersion in Digital Games: Review of Gaming Experience Research) Um dos requisitos mais importantes ao se tentar manter o jogador imerso no jogo é o uso de Inteligência Artificial (IA), pois mesmo que se haja gráficos e cenários que se aproximam muito da realidade, toda a experiência de jogo é quebrada se o jogador, ao se mover, bater na parede ao tentar contorná-la [6]. //falar o que é IA

Se a IA for desenvolvida apropriadamente ela fornece ao jogo responsividade e velocidade aos inputs de maneira natural, aumentando a sensação de realidade. O comportamento será mais inteligente e flexível. Inteligente no sentido que a situação será abordada de maneira mais próxima ao pensamento humano, e flexível no sentido que o comportamento pode se adaptar conforme a situação [6].

Uma parte fundamental da IA usada em VGs é conhecida como *pathfinding*, que é o processo pelo qual se define uma série de movimentos, ou caminho, que um objeto de jogo realizará para de ir do ponto A até o ponto B, sem que haja colisão com obstáculos indesejáveis. Um caminho satisfatório deve conter duas propriedades: validação e otimização. Validação indica que o objeto de jogo não colidirá com nada ao percorrer o percurso. Otimização é a qualidade que ele chegará ao destino e pode ser medida em termos de tempo, distância ou qualquer outro parâmetro desejado[7].

Um dos fatores de otimização mais utilizados em VGs é o tempo e garante que o caminho ótimo será aquele que o objeto de jogo percorre mais rápido do que qualquer outro. A escolha do fator de otimização se dá pelo fato de que normalmente, em um VG, existem diversos tipos de terrenos em que o personagem terá diferentes velocidades[7]

1.1 PROBLEMAS E PREMISSAS

Um dos fatores que influencia na quebra da imersão do jogador é o deslocamento não realista do objeto de jogo ao ir de um ponto à outro. Um comportamento não realista muito comum em NPCs é a definição de rotas com base em uma informação prévia dos obstáculos e das dificuldades para se chegar ao destino. Essa técnica cria uma percepção sobre-humana do ambiente e torna impossível o delineamento de rotas em ambientes gerados proceduralmente[4].

1.2 JUSTIFICATIVA

A motivação deste trabalho encontra-se no fato que a imersão é um componente importante na experiência que os jogadores buscam[5].

De acordo com a hierarquia dos elementos principais da experiência de jogo (CEGE), a maneira que o usuário lida com o jogo mesmo não fazendo parte dele (sensação de controle e domínio) são recursos de suma importância para garantir uma maior imersão no jogo[5], por isso a responsividade aos comandos do jogador são importantes para garantir uma boa experiência.

Surge então a necessidade da criação de um sistema de geração de caminho (*Pathfinding*) que identifique, em tempo real, o caminho que é executado em menor tempo de um ponto a outro, calculando obstáculos e se movendo de forma natural, gerando uma maior sensação de realismo e imersão.

1.3 OBJETIVOS

1.3.1 OBJETIVOS GERAIS

Desenvolver um VG do tipo MOBA e um algoritmo de *pathfinding* que será implementado em NPCs do jogo.

1..2 OBJETIVOS ESPECÍFICOS

O trabalho visa os seguintes objetivos:

- Revisar literatura sobre algoritmos de *pathfinding* aplicado a jogos digitais.
- Escolher e desenvolver o algoritmo que mais se assemelha ao problema.
- Produzir um jogo online com requisições em um servidor próprio.
- Realizar buscas e armazenar informações em um banco de dados remoto.

2 FUNDAMENTAÇÃO TEÓRICA

Em jogos digitais é muito comum personagens se moverem em um espaço, e isso pode ser feito de maneira em que e a programação do movimento é inalterável, como em rotas de patrulha ou qualquer rota repetitiva. Porém ocorre o caso em que personagens não possuem rotas prontas por motivos como transformação do ambiente ou não conhecimento prévio do destino. Nesses casos é necessário calcular uma rota com base em destinos e obstáculos impostos em tempo real[10].

Em jogos digitais, *pathfinding* é a geração de caminho que leva um personagem (controlável ou não) da sua posição atual até um local objetivo e pode ser aplicado em qualquer jogo que requeira uma movimentação com entradas variadas, como posição inicial, destino e obstáculos. A ideia é que o percurso seja o mais rápido possível, tanto em questões de movimentação quanto de processamento do computador[10].

Algoritmos de *pathfinding* possuem três elementos principais[11]:

- Representação Gráfica do Mapa
- Algoritmo de Busca
- Função Heurística

Representações gráficas do mapa são a discretização de um espaço para que ele possa ser usado em algoritmos de busca e são feitas a partir da fragmentação do espaço em células (nós). Diferentes representações gráficas possuem diferentes formatos de células[11].

Algoritmos de Busca são, no contexto de *pathfindind* em jogos, algoritmos de inteligência artificial que analisam a representação gráfica do mapa e calculam a melhor rota[11].

Funções heurísticas são funções que avaliam os possíveis nós e determinam valores para que seja possível quantificar e analisar as possíveis rotas[10].

O algoritmo de busca A* é um dos vários algoritmos de busca existentes e se destaca quando o assunto é *pathfinding* em jogos digitais pois a maioria dos algoritmos de *pathfinding* presentes no mercado de jogos utiliza o A* como base[10].

São analisados neste capítulo diversos algoritmos de pathfinding que são comumente aplicados em jogos digitais[11], começando pelo algoritmo A* e depois dando continuidade com as suas variações que mais se assemelham ao propósito deste trabalho, explicando suas vantagens e desvantagens. Por final será escolhido um que melhor se assemelha ao problema.

2.1 A*

A* é um algoritmo de busca do tipo best-first, que é um algoritmo de busca baseado em grafos onde o nó escolhido para ser expandido é escolhido pelo menor valor de uma função de avaliação f(n), que é obtida a partir de uma estimativa do custo para ir de um nó atual até o nó destino[12].

Proposto por Hart, Nilsson e Raphael em 1967, é um dos algoritmos de busca mais utilizados pois garante o menor caminho em um grafo [9] e dentre todos os algoritmos de busca por menor caminho baseados em heurística, o A* é o que expande o menor número de nós, fazendo um uso mais eficiente da função h(n) [8].

A função para o cálculo dos nós é dada por f(n) = g(n) + h(n), onde g(n) é o custo do nó inicial até o nó atual e h(n) é o custo estimado do nó atual até o nó Objetivo[8], conforme explicado na Figura 1.

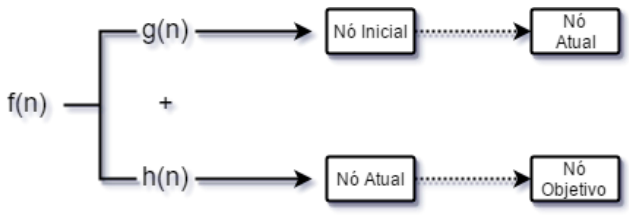

Figura 1 - Cálculo de f(n) para o algoritmo A*
Fonte – Autoria Própria (2017)

Existem certas condições para que o A* seja considerado otimizado, a primeira é que a função h(n) possua uma heurística admissível, isto é, que nunca calcule de forma exagerada o custo para se chegar até o nó destino a partir do nó atual. [12].

O algoritmo dispõe de duas listas, a lista aberta para os nós ditos abertos e a lista fechada para os nós ditos fechados. Isto permite um controle dos nós visitados, evitando releituras desnecessárias[9].

A cada iteração, o nó com menor valor de f(n) é selecionado e expandido, gerando outros nós adjacentes que possuem o nó selecionado como pai. Os valores f(n), g(n) e h(n) dos nós gerados são calculados e armazenados e o nó selecionado é posto na lista fechada, excluindo-o das próximas iterações. Em caso de empate o nó selecionado é aquele com menor valor de h(n)[8]. O pseudocódigo do A* é mostrado na figura 2.

Search Algorithm A*:

1. *Add the starting node to the open list.*
2. *Repeat the following steps:*
   a) *Look for the node which has the lowest f on the open list. Refer to this node as the current node.*
   b) *Switch it to the closed list.*
   c) *For each reachable node from the current node*
      i. *If it is on the closed list, ignore it.*
      ii. *If it isn't on the open list, add it to the open list. Make the current node the parent of this node. Record the f, g, and h value of this node.*
      iii. *If it is on the open list already, check to see if this is a better path. If so, change its parent to the current node, and recalculate the f and g value.*
   d) *Stop when*
      i. *Add the target node to the closed list.*
      ii. *Fail to find the target node, and the open list is empty.*
3. *Tracing backwards from the target node to the starting node. That is your path.*

Figura 2 - Pseudocódigo de A*
Fonte - (MATHEW; MALATHY, 2015)

A qualidade do A* de analisar e armazenar os principais nós possíveis e armazená-los na lista aberta é também o seu maior defeito. Dependendo da complexidade do problema, a lista aberta pode acabar ficando grande demais, o que ocasiona em uma grande demanda de memória, para armazenamento, e processamento, para analisar todos os possíveis nós a cada iteração[8].

Mesmo sendo um dos algoritmos que expande o menor número de nós, ele armazena todos e isto faz com que existam certos problemas na aplicação de *pathfinding* em jogos digitais, visto que velocidade de resposta é algo importante[8].

Outro problema pode ocorrer se utilizado em jogos multijogadores, pois o A* sozinho não resolve conflitos em casos que existem dois ou mais personagens buscando uma melhor rota. Nesses casos é necessária uma solução que leve em consideração os problemas de se possuir mais de um algoritmo sendo executado ao mesmo tempo[13].

Um dos problemas seria quando a rota calculada de dois personagens coincide em um ponto, fazendo com que ocorra colisão entre eles[13], conforme exemplificado na figura 3. Neste caso o personagem P1 traça uma rota até seu destino D1 e o personagem P2 traça uma rota até seu destino D2. Como não há obstáculos presentes no ponto O durante a geração do caminho, ele é incluído na rota. Os dois personagens então possuem o ponto O em comum em suas rotas, se o instante que os dois passarem pelo ponto O for o mesmo, haverá um choque não calculado onde cada personagem serviria como um obstáculo não detectado para o outro, gerando um impasse impossível de ser calculado usando apenas o A*.

As rotas representadas na figura 3 servem apenas de exemplo pois não refletem de forma fiel um caminho gerado por um algoritmo A*.

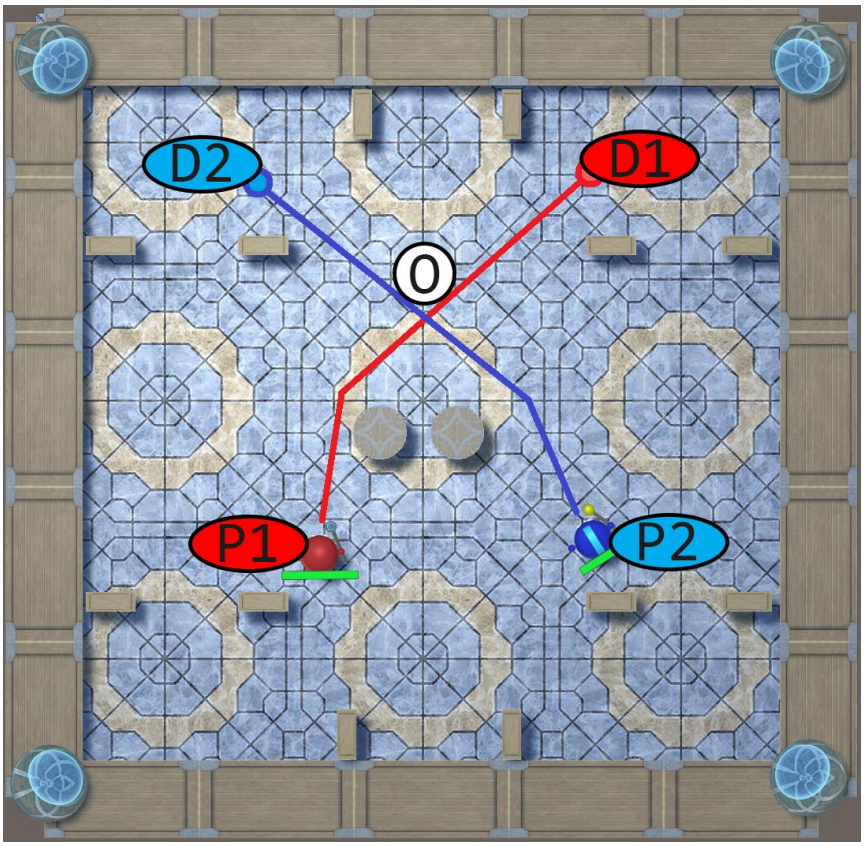

Figura 3 - Problema de rota do A*
Fonte – Autoria Própria (2017)

2.2 LOCAL REPAIR A*

Um algoritmo amplamente usado em VGs são o Local Repair A* (LRA) e surgiu com o propósito de resolver o problema de colisão do A*. O algoritmo consiste em cada personagem traçar sua rota utilizando o A* como se não existissem outros personagens e como explicado na figura 3, uma colisão pode ocorrer. Porém um instante antes da colisão, ou seja, quando o próximo movimento a ser executado possui um espaço ocupado, o algoritmo calcula uma rota alternativa tomando como nó inicial o nó atual[15].

Um dos problemas dessa abordagem é a facilidade com que situações podem se repetir. Por exemplo, a figura 4-a possui o mesmo contexto da figura 3, porém depois do choque, no ponto O, as rotas são recalculadas tomando como nó inicial a posição em que se encontram um instante antes da colisão, porém o novo caminho é gerado para os dois personagens, e possui um novo ponto O2 que resulta em uma colisão. Novamente um novo caminho é gerado para os dois personagens de forma

independente e ocorrerá um novo choque em O. Este tipo de situação gera ciclos infinitos pois nenhum dos personagens possuirá a informação de outro personagem um instante antes do choque[15].

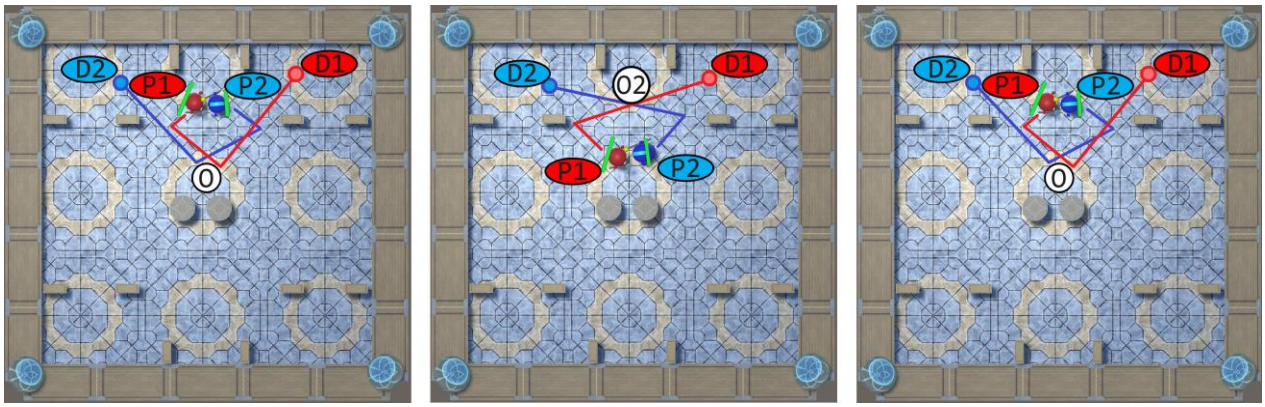

Figura 4 – Colisões do LRA*
Fonte – Autoria Própria (2017)

Uma possível solução para esse problema é a criação de uma variável de nível de agitação, que é simplesmente um valor que é incrementado toda vez que é necessário recalcular uma rota. É gerado então um ruído, que é proporcional ao nível de agitação, porém aleatório. O ruído é então adicionado à heurística de distância, fazendo que quanto mais choques ocorrerem, maior será a probabilidade dos personagens terem heurísticas diferentes e portanto tomarem caminhos diferentes[15].

A solução do nível de agitação pode parecer promissora à primeira vista, porém se o espaço possuir uma grande concentração de agentes e obstáculos, serão necessárias muitas gerações de caminho a cada choque, podendo caindo em loops, o que resulta em demasiadas chamadas do A*[15].

2.3 COOPERATIVE A*

Um algoritmo que visa a solução de maneira cooperativa, ou seja, o funcionamento adequado do Cooperative A*(CA) somente pode ocorrer entre jogadores da mesma equipe. A geração de rotas é dada de maneira individual, seguindo a base do A*. A novidade apresentada é que quando uma rota é calculada, seus dados são armazenados em uma lista de espera. Uma implementação básica do algoritmo é feita por armazenar duas informações da rota: espaço e tempo.

Quando o algoritmo calcula a rota para um personagem, é levado em consideração a lista de espera, que é tratada como um obstáculo de espaço durante o tempo. Cada informação de espaço e tempo presente na lista de espera retrata que um outro personagem estará naquele espaço durante aquele período de tempo, com isso é possível traçar uma rota que não resulte em colisão com os demais personagens.

Um dos problemas apresentados pelo CA* é que as rotas são geradas individualmente, somente levando em consideração caminhos já definidos de outros personagens, o que impossibilita uma coordenação de movimentos mais inteligentes.

Certas situações, como a apresentada na figura 5, podem se tornar grandes problemas. Nela, o personagem S1 traça sua rota até seu objetivo G1 e depois o personagem S2 tenta traçar a sua rota até G2, porém não possui nós para chegar até lá pois S1 deixou os nós necessários na lista de espera. Tal situação poderia ser evitada usando o espaço lateral L.

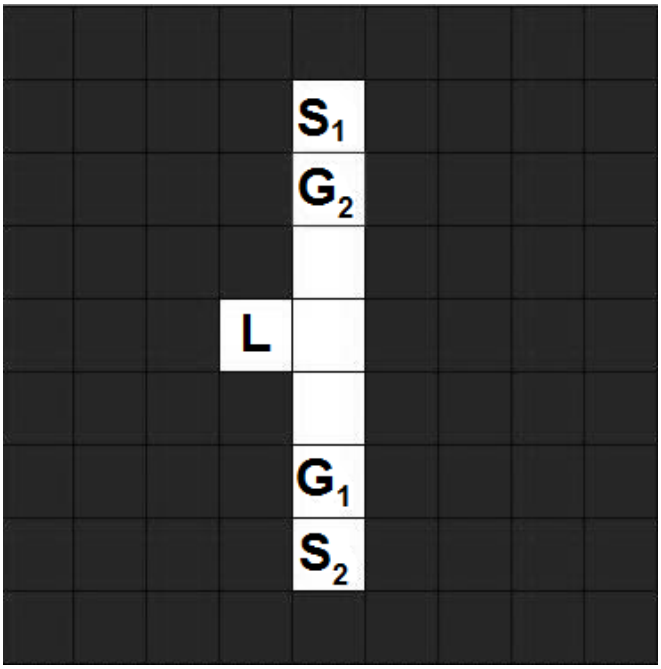

Figura 5 - Problema do Cooperative A*
Fonte - Adaptado de [15]

2.4 HCA*

Hierarchical Cooperative A* (HCA*) é a junção do CA* com um algoritmo chamado Hierarchical A* (HA*). A hierarquia do HA* se refere a abstrações do estado do espaço. É dito que um grande número de abstrações possui uma performance inferior aos algoritmos com menor número de abstrações. Baseado nisso o HCA* utiliza apenas uma abstração, diferente do CA* que utiliza três.

A hierarquia do HCA* é a mais simples possível pois somente é abstraído o mapa, não contando com os outros personagens presentes e nem a lista de reserva, presentes no CA*. Existem diversas maneiras para se obter as abstrações, consideraremos a Reverse Resumable A* (RRA*).

O RRA* por ser uma variação do A*, possui funcionamento muito similar, porém a busca pelo caminho se dá pelo caminho inverso. O algoritmo começa pelo nó destino e traça sua rota até o nó inicial e pausa a sua execução. Toda vez que o algoritmo HCA* requer uma abstração da distância de um nó qualquer N até o nó objetivo, o RRA* checa se este nó está presente na sua lista fechada (significando que esse nó já foi expandido). Se estiver, a distância é retornada para o HCA* utilizar e se não estiver o RRA* é despausado e continua a sua execução até expandir o nó N requisitado. O pseudocódigo do algoritmo RRA* pode ser observado na figura 6.

O HCA* é basicamente o algoritmo CA* sem a informação do tempo e a tabela de reserva, porém a heurística usada é a abstração da distância, calculada pelo algoritmo RRA*. Um problema do HCA* é a finalização da execução dele, pois o personagem que chegou ao seu destino ficará parado nele, o que é desejado pois ele chegou ao seu destino, porém este personagem parado servirá como uma barreira aos outros personagens, fazendo com que talvez seja necessário criar uma rota muito maior do que se o personagem não estivesse parado no destino.

Algorithm 1 Reverse Resumable A*

```
1:  procedure INITIALISERRA*(O, G)
2:      G.g ← 0
3:      G.h ← MANHATTAN(G, O)
4:      Open ← {G}
5:      Closed ← ∅
6:      RESUMERRA*(O)
7:  end procedure

8:  procedure RESUMERRA*(N)
9:      while Open ≠ ∅ do
10:         P ← pop(Open)
11:         Closed ←(add)— P
12:         if P = N then
13:             return success
14:         end if
15:         for all Q ∈ reverse(SUCCESSORS(P)) do
16:             Q.g ← P.g + COST(P, Q)
17:             Q.h ← MANHATTAN(Q, O)
18:             if Q ∉ Open and Q ∉ Closed then
19:                 Open ←(add)— Q
20:             end if
21:             if Q ∈ Open and f(Q) < f(Q in Open)
                    then
22:                 Open ←(update)— Q
23:             end if
24:         end for
25:     end while
26:     return failure
27: end procedure

28: procedure ABSTRACTDIST(N, G)
29:     if N ∈ Closed then
30:         return g(N)
31:     end if
32:     if RESUMERRA*(N) = success then
33:         return g(N)
34:     end if
35:     return +∞
36: end procedure
```

Figura 6 - Pseudocódigo de RRA*
Fonte – (SILVER, 1992)

2.5 WHCA*

Windowed Hierarchical Cooperative A* (WHCA*) é um algoritmo do HCA* e funciona com a premissa de que muitas vezes uma rota calculada não é utilizada até o final, pois no meio da trajetória pode surgir um obstáculo e então toda rota deverá ser recalculada, gerando um gasto computacional desnecessário.

WHCA* utiliza um método de janelas para dividir as buscas em partes menores. Uma janela de tamanho w significa que somente serão calculados w nós para cada personagem. Quando um personagem chega em uma parte considerável da janela (como a metade, por exemplo), a janela é movida para frente e então o é refeito o cálculo da rota.

Para garantir que o personagem realmente estará indo em direção ao objetivo, o WHCA* utiliza uma busca cooperativa (como o CA*) com profundidade w e uma busca abstrata (como o RRA*) com profundidade total, portanto os outros personagens somente serão considerados durante a busca cooperativa, ou seja, durante w nós. Deste jeito, um personagem é capaz de se orientar de maneira cooperativa dentro de uma janela sem perder o caminho até o nó destino.

Um outro diferencial do WHCA* é que a busca não termina quando o personagem chega até o seu nó destino, porém o objetivo do personagem não é mais o de chegar até o nó destino, mas sim completar a janela em que ele se situa, podendo sair de seu local para dar passagem a outros personagens.

2.6 OPERATOR DECOMPOSITION

Operator Decomposition (OD) é uma representação de estado em que um único algoritmo calcula separadamente o próximo passo de cada personagem e depois partindo para os demais personagens que ainda não calcularam seu próximo passo. Cada nó possui não só a informação da posição, mas também do próximo nó da rota.

As etapas do movimento são realizadas da seguinte forma: primeiro calcula-se o movimento do personagem 1 sem levar em consideração se existem outros personagens. Em seguida calcula-se o movimento do personagem 2 levando em consideração apenas os agentes que já possuem um movimento definido, no exemplo, o personagem 1. O movimento do personagem 2 não pode invalidar o movimento já definido do personagem 1. Na figura 6, o algoritmo começa a calcular as rotas a partir do personagem 1 e depois avança para o personagem2. O personagem 1 possui três movimentos: norte, leste e permanecer parado, enquanto o personagem 2 possui apenas o movimento leste. É interessante notar que a técnica OD possui diferentes caminhos dependendo da ordem de personagens escolhida.

Figura 7 - Exemplo do OD
Fonte – Adaptado de (STANDLEY, 2010)

Outra técnica utilizada no OD vem do fato que o próximo nó calculado para um agente nunca será colocado na lista fechada pelo A*, isto ocorre porque o movimento de um personagem só poderá de fato ser definido depois que todos os personagens expandirem os seus nós. Baseado nisso, a estratégia adotada é colocar somente os nós atuais na lista fechada, mantendo as possibilidades em aberto. Isso reduz em aproximadamente 93% da lista fechada, melhorando o tempo de execução.

2.7 INDEPENDENCE DETECTION

O algoritmo de Independence Detection (ID) se baseia na premissa de que não é necessário resolver problemas de todos os personagens, pois nem todos terão conflitos. O algoritmo primeiramente coloca cada personagem presente em um grupo, então calcula normalmente a rota de todos os personagens usando o A*, armazenando os nós necessários para as rotas em uma tabela de conflitos.

Outra premissa analisada pelo ID é que geralmente os personagens possuem mais de um caminho válido para se chegar ao destino. No algoritmo A* esse desempate é realizado por selecionar o caminho que possui menor custo de h(). O ID então observa os personagens que possuem conflitos e tenta calcular rotas alternativas para os personagens presentes, porém com algumas restrições.

O novo caminho a ser calculado não pode ser conflitante com o caminho dos outros personagens e deve ter o mesmo custo do primeiro caminho calculado, senão ocorreria perda na otimização do código. Para atingir esses requisitos, o ID possui um limite de custo e uma tabela de movimentos ilegais, que é alimentada antes do recalculo do caminho.

Em um exemplo que ocorre conflito pela primeira vez entre dois personagens, P1 e P2. Primeiro a tabela de movimentos ilegais é preenchida com os nós presentes na rota de P2, e então são calculadas as rotas alternativas de (P1, e em caso de nenhuma rota alternativa ser livre de conflitos, a tabela de movimentos ilegais é esvaziada e então preenchida com os nós presentes na rota de P1, e então são calculadas as rotas alternativas para P2. Caso os dois personagens não possuam rotas alternativas livre de conflitos, os grupos são unidos, fazendo com que exista um grupo com dois personagens conflitantes. O conflito do grupo é então resolvido sem limite de custo e tabela de movimentos ilegais, porém é escolhida a rota com menor número de conflitos, e em caso de empate nos números de conflitos, o critério de desempate é o mesmo do A*, o custo de h(). O pseudocódigo do ID é mostrado na figura 7.

Algorithm 2 Independence Detection

```
1: assign each agent to a group
2: plan a path for each group
3: fill conflict avoidance table with every path
4: repeat
5:     simulate execution of all paths until a conflict between two
       groups G1 and G2 occurs
6:     if these two groups have not conflicted before then
7:         fill illegal move table with the current paths for G2
8:         find another set of paths with the same cost for G1
9:         if failed to find such a set then
10:            fill illegal move table with the current paths for G1
11:            find another set of paths with the same cost for G2
12:        end if
13:    end if
14:    if failed to find an alternate set of paths for G1 and G2 then
15:        merge G1 and G2 into a single group
16:        cooperatively plan new group
17:    end if
18:    update conflict avoidance table with changes made to paths
19: until no conflicts occur
20: solution ← paths of all groups combined
21: return solution
```

Figura 8 - Pseudocódigo do ID
Fonte – Adaptado de (STANDLEY, 2010)

3 PROPOSTA

A solução apresentada é a utilização de um algoritmo de inteligência artificial para geração de caminho de um ponto A até um ponto B, evitando colisões com obstáculos presentes no cenário. O tempo do percurso deverá ser o menor possível, evitando espera desnecessária do jogador e garantindo fluidez ao jogo.

O algoritmo escolhido dentre os apresentados no referencial teórico é o WHCA* pois ele oferece a possibilidade de se tratar as rotas em janelas, o que reduz a quantidade de iteração do algoritmo.

O jogo em que o algoritmo é executado é um MOBA (Multiplayer Online Battle Arena) com elementos de rouba-bandeira. Cada partida é composta por dois times com quatro jogadores cada.

O mapa é dividido em duas partes, uma para cada time. O objetivo do jogo é pegar a bandeira adversária e transportá-la até o seu lado do campo, conforme explicado na figura 5.

Os jogadores poderão atacar os membros do time inimigo, se o jogador que estiver carregando a bandeira morrer, a mesma cairá no chão, sendo possível qualquer jogador, aliado ou não, pegá-la. Quando um jogador morre, é necessário esperar um tempo até que seja possível jogar novamente. O tempo de espera é aumentado conforme o avanço da partida.

Figura 9 - Representação do Mapa
Fonte – Autoria Própria (2017)

Antes da partida começar os jogadores poderão escolher diversos PCs, chamados de campeões. Os campeões possuem habilidades e atributos próprios e pré-definidos. As habilidades são imutáveis, enquanto os atributos podem ser alterados através de itens comprados nas lojas.

Os atributos que os campeões possuem são:

- Vida - Decrementada quando o campeão recebe dano e se o valor for menor ou igual a zero, o personagem morre. Incrementada proporcionalmente a regeneração de vida.
- Mana - Decrementada quando o campeão utiliza uma habilidade baseada em mana. Incrementada proporcionalmente a regeneração de mana.
- Regeneração de Vida - Valor de vida que será incrementado por segundo.
- Regeneração de Mana - Valor de mana que será incrementado por segundo.
- Defesa Física - Valor utilizado para diminuir o dano físico recebido.
- Defesa Mágica - Valor utilizado para diminuir o dano mágico recebido.
- Ataque Físico - Valor que indica o dano físico a ser infligido na vida do adversário.

- Ataque Mágico - Valor que indica o dano mágico a ser infligido na vida do adversário.
- Redução do Tempo de Recarga - Valor, em porcentagem, que será subtraída do tempo de recarga de cada habilidade.
- Velocidade de Movimento - Fator de multiplicação para a velocidade que o campeão se moverá.
- Velocidade de Ataque - Quantidade de ataque básicos que o campeão consegue usar por segundo.

Existem seis arquétipos de campeões, sendo que eles podem possuir mais de um ao mesmo tempo:

Arquétipo	Característica
Tanker	Maior defesa e/ou vida.
Healer	Habilidades de cura e/ou suporte para o time aliado.
Stealthy	Alta velocidade de movimento e habilidades que interferem na movimentação. Pouca vida.
Assassin	Dano elevado e pouca vida.
Terrain Changer	Habilidades que alteram o mapa, construindo barreiras, pontes, etc.
Summoner	Habilidades que invocam criaturas (summon) para auxiliá-los. O algoritmo de pathfinding será implementado nesses summons.

Tabela 1 - Arquétipos

A movimentação dos campeões é feita apertando as teclas WASD do teclado, e o movimento segue os padrões:

- W – Move o campeão para o norte.
- A – Move o campeão para o oeste.
- S – Move o campeão para o sul.
- D – Move o campeão para o leste.

O botão esquerdo do mouse é responsável por um ataque mais simples, chamado de ataque básico, enquanto o botão direito e as teclas Q, E, Z e C são destinadas à habilidades.

O jogo é *online* e os *summons* podem ser colocados em qualquer momento do jogo, portando quando um novo *summon* é inserido, talvez seja necessário recalcular a rota de outros minions que viriam a ter conflito com ele. Se o WHCA* possuir um tamanho adequado de janela, é possível minimizar esses recálculos pois se a colisão não fizer parte da janela atual, não será necessário refazer nenhuma rota.

3.1 TECNOLOGIAS E FERRAMENTAS

3.1.1 UNITY

O motor gráfico (engine) utilizado é a Unity3D por se tratar de um software de uso gratuito e com material abrangente em fóruns e sites. A linguagem escolhida foi C# pelo fato da universidade fornecer uma boa base na linguagem estruturada C.

3.1.2 MYSQL

O gerenciador de banco de dados utilizado é o MySQL pela facilidade em se encontrar material didático com a sua integração com o motor gráfico Unity e pela base adquirida nas disciplinas de banco de dados.

3.1.3 SERVIDOR

Inicialmente o servidor será local e conforme o desenvolvimento do projeto e implementação de funcionalidades que necessitam de maior processamento, poderá ocorrer uma migração para um servidor externo e contratado.

3.2 ANÁLISE E DESENVOLVIMENTO

3.2.1 REQUISIÇÃO CLIENTE-SERVIDOR

Como o jogo é online, é necessário um servidor para processar problemas que são comuns a todos os jogadores, como a posição e os atributos dos jogadores, gerenciamento dos objetivos (bandeiras), assim como a execução do algoritmo de *pathfinding*.

3.2.2 MODELAGEM DO BANCO DE DADOS

A modelagem do banco de dados se dará de forma simples, sendo necessária apenas para cadastro/login e campeões comprados na loja. O banco de dados estará em um servidor externo e apenas retornará para o jogador quais campeões ele possui, e no caso do cadastro/login, apenas se a operação foi um sucesso ou não.

3.2.3 PROTÓTIPO E TELAS

O protótipo atual possui sistema cliente-servidor, sendo possível vários jogadores se conectarem ao jogo simultaneamente. O jogo possui um menu de seleção de campeões, onde é possível cada jogador jogar com campeões diferentes.

Tal funcionalidade não é nativa da biblioteca para jogos online da Unity (uNet), sendo necessário sobrescrever alguns métodos.

Existem três campeões disponíveis, como mostrada na figura 8, com características distintas como velocidade de movimento, dano, vida e tipo de tiro. Quando um personagem ataca o outro usando o botão esquerdo, a vida do personagem atacado é subtraída, e o valor da variável é sincronizada com todos os clientes. A figura 10 mostra vários jogadores diferentes em cena, enquanto a figura 11 mostra a interação com os ataques.

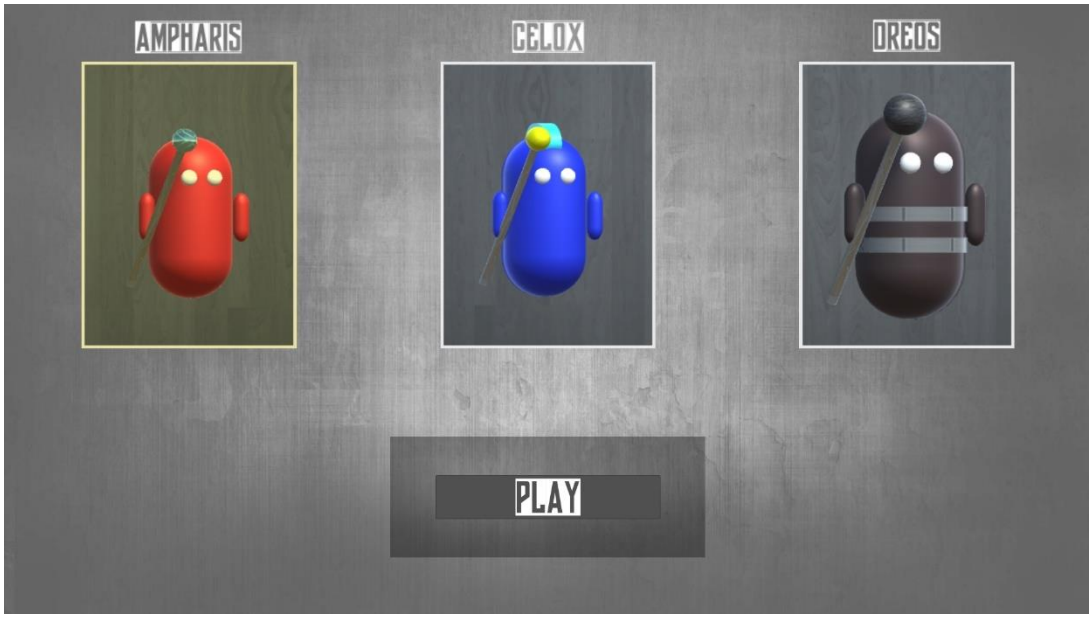

Figura 10 - Menu de seleção de personagens
Fonte – Autoria Própria (2017)

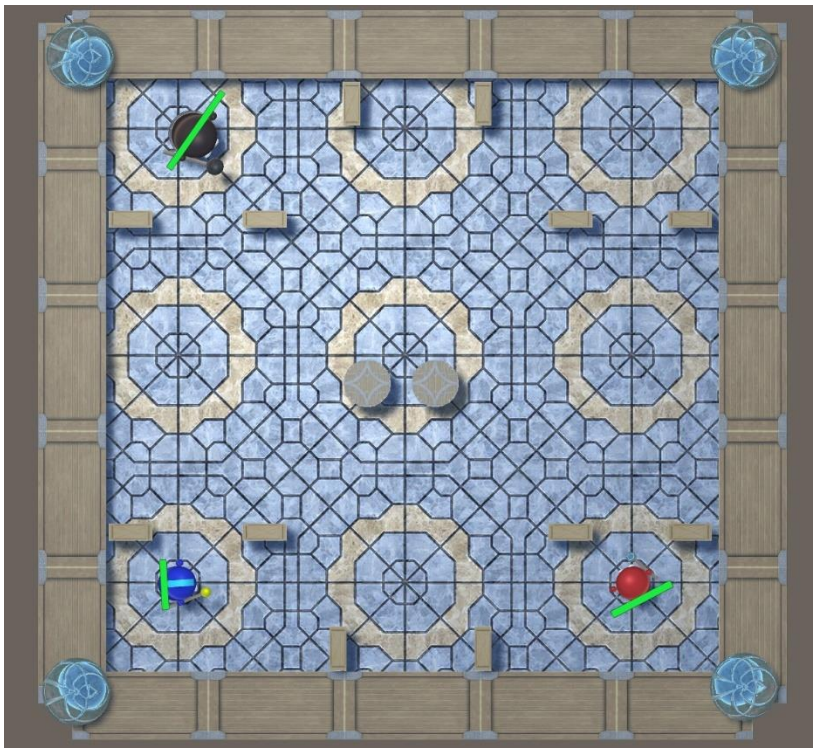

Figura 11 - Personagens dispostos no mapa
Fonte – Autoria Própria (2017)

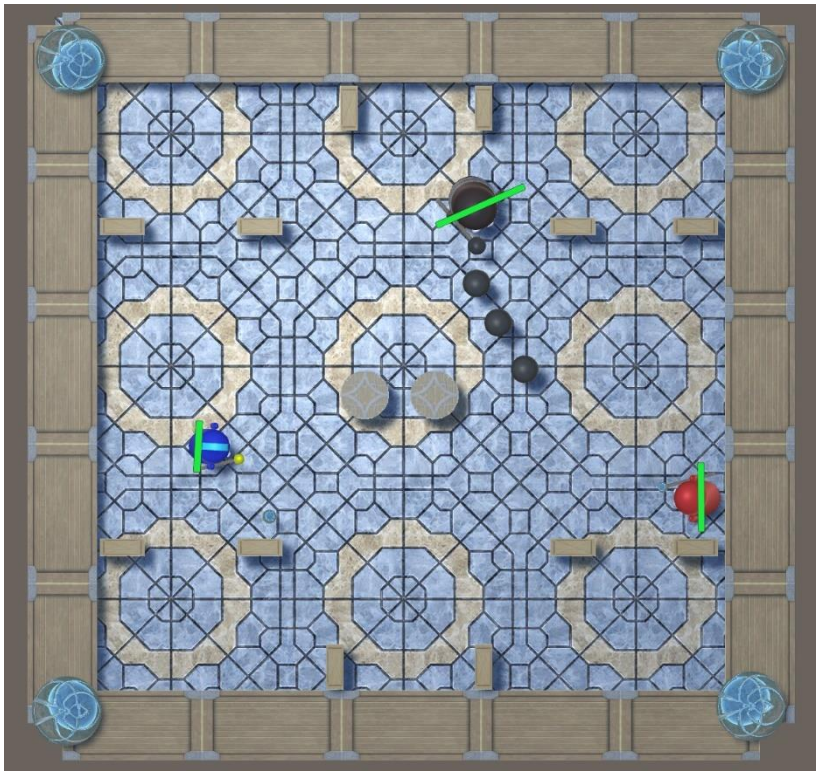

Figura 12 - Personagens atacando
Fonte – Autoria Própria (2017)

4 CRONOGRAMA

	MÊS 1				MÊS 2				MÊS 3				MÊS 4			
	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
ETAPA 1	X	X	X	X	X											
ETAPA 2						X	X	X	X							
ETAPA 3					X					X	X	X	X			
ETAPA 4											X	X	X			
ETAPA 5														X	X	X

Tabela 2 - Cronograma. Fonte: Próprio Autor

- ETAPA 1: Desenvolvimento das mecânicas básicas, skills, loja, inventário e campeões diferentes.
- ETAPA 2: Implementação do algoritmo de *pathfinding* nos *summons*.
- ETAPA 3: Implementação das interfaces de usuário como menu, barra de vida, skills, lobby e loja.
- ETAPA 4: Vinculação com banco de dados.
- ETAPA 5: Polimento e balanceamento do jogo.

5 CONSIDERAÇÕES FINAIS

O trabalho revisa os principais algoritmos utilizados na indústria de VG e é então escolhido o que melhor se adapta ao problema. É uma pesquisa significante pois são explorados diversos métodos e o teste do algoritmo será em um ambiente *online* e em tempo real.

Outro aspecto relevante é que o jogo em que o algoritmo será executado é construído seguindo aspectos de um produto final, ao invés de apenas um protótipo. Tal feito requer certo polimento no jogo, balanceamento entre os campeões para que as partidas sejam justas e mecânicas que não sejam somente base para o algoritmo, mas que sejam divertidas e causem uma boa experiência ao jogador.

Uma possível limitação pode ocorrer devido ao custo computacional necessário para se resolver a abordagem pois o tamanho do mapa e o número de campeões somado ao fato de que as rotas serão calculadas utilizando o sistema cliente-servidor, podem se tornar problemas.

REFERÊNCIAS

[1] J. Huizinga, Homo Ludens: o jogo como elemento da cultura, 6th ed. São Paulo: Perspectiva, 2010.

[2] LUCHESE, FABIANO. "RIBEIRO, Bruno." Conceituação de Jogos Digitais (2009).

[3] BOGDANOWICZ, Marc et al. Born digital/grown digital: Assessing the future competitiveness of the EU video games software industry. Institute for Prospective Technological Studies, Joint Research Centre, 2010.

[4] STAMFORD, John; KHUMAN, Arjab Singh; AHMADI, Jenny Carter & Samad. Pathfinding in partially explored games environments. IEEE, [S.L], out. 2014.

[5] CAIRNS, Paul; COX, Anna; NORDIN, A. Imran. Immersion in digital games: review of gaming experience research. Handbook of digital games, p. 337-361, 2014.

[6] CUI, Xiao; SHI, Hao. An overview of pathfinding in navigation mesh. IJCSNS, v. 12, n. 12, p. 48-51, 2012.

[7] CHARLES, Darryl. Enhancing gameplay: challenges for articifical intelligence in digital games. In: DIGRA Conf. 2003.

[8] KORF, Richard E. Artificial intelligence search algorithms. Chapman & Hall/CRC, 2010.

[9] MATHEW, Geethu Elizebeth. Direction based heuristic for pathfinding in video games. Procedia Computer Science, v. 47, p. 262-271, 2015.

[10] MILLINGTON, Ian; FUNGE, John. Artificial intelligence for games. CRC Press, 2016.

[11] BOTEA, Adi et al. Pathfinding in games. In: Dagstuhl Follow-Ups. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[12] RUSSELL, Stuart; NORVIG, Peter; INTELLIGENCE, Artificial. A modern approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs, v. 25, p. 27, 1995.

[13] STANDLEY, Trevor Scott. Finding Optimal Solutions to Cooperative Pathfinding Problems. In: AAAI. 2010. p. 28-29.

[14] STANDLEY, Trevor; KORF, Richard. Complete algorithms for cooperative pathfinding problems. In: IJCAI. 2011. p. 668-673.

[15] SILVER, David. Cooperative Pathfinding. AIIDE, v. 1, p. 117-122, 2005.