Inteligência Artificial para Jogos

Aula 07 - Descoberta de caminho - Parte 01

Apresentação da Aula

Logo nas primeiras aulas deste curso, mais precisamente nas aulas 2 e 3, você aprendeu alguns conceitos e técnicas relacionadas à navegação dos jogadores de futebol no campo. Nessas aulas, foram introduzidos os comportamentos de navegação e apresentados vários desses comportamentos. Lembrando que foram utilizados muitos princípios da matemática e da física, como uso de vetores e inércia, para fazer com que os jogadores que estavam sendo desenvolvidos pudessem ter um movimento mais natural, fazer curvas suaves, parar lentamente, e muitas outras formas de movimentação.

O resultado ficou muito bacana, não foi? 😁

Pois é, vou ser sincero... nem tudo foi resolvido. Para fazer um personagem se deslocar em um ambiente virtual, apenas o uso de comportamentos de navegação (*steering behaviors*) em geral não é suficiente. Para o caso do jogo de futebol, eles até são. Porém, na maioria dos jogos, antes de os personagens saírem fazendo curvas suaves, eles precisam planejar para onde vão e qual caminho tomar para chegar lá. Isso, os comportamentos de navegação não fazem. Eles apenas jogam o personagem para um lado ou para o outro em função da reação a forças que forem aplicadas a eles. Pode-se dizer que os comportamentos de navegação apresentados nas aulas 2 e 3 são “reativos”. Porém, em muitas situações, os personagens precisam ser “proativos”. Eles precisam tomar decisões internamente e agir em função delas, e não apenas em função de eventos externos.

Quer saber como você pode começar a planejar os seus caminhos?

Ao infinito e além... quero dizer... então, continue!

Objetivos

Conhecer as dificuldades de planejar o deslocamento de personagens;

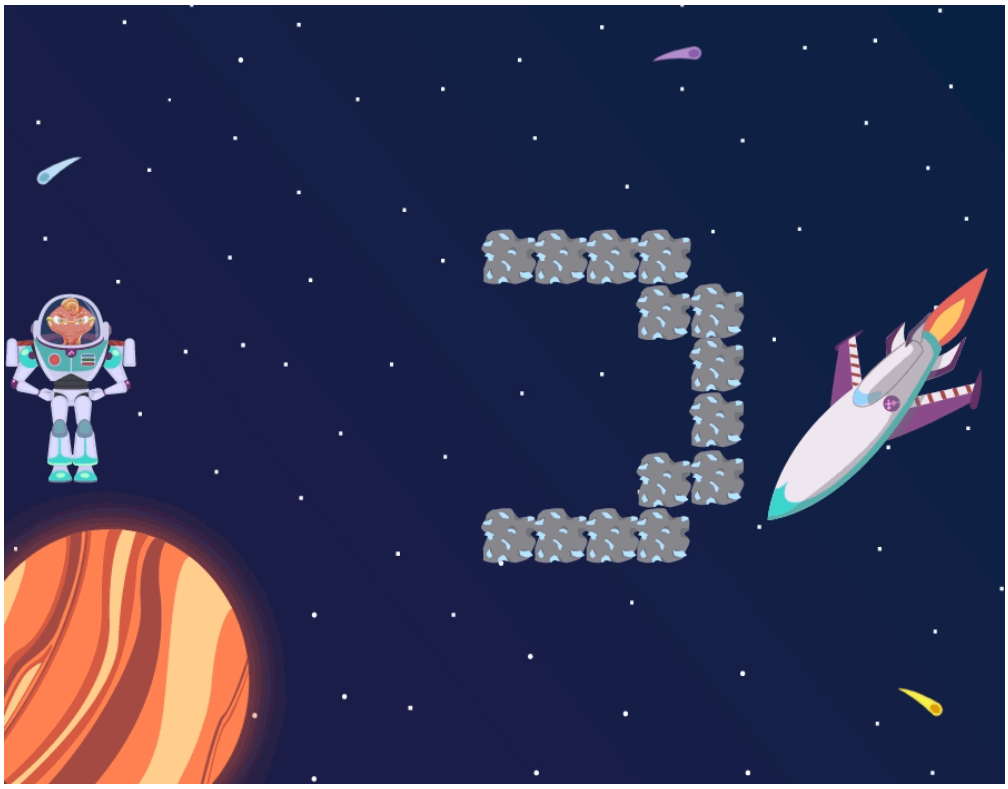
Compreender como representar um cenário para facilitar a movimentação de personagens;

Aprender um algoritmo básico para o planejamento da movimentação de personagens em linha reta.

1. Traçando o caminho!

Quer um exemplo para ilustrar essa necessidade de se pensar o caminho até o local desejado? Imagine um jogo cujo cenário é ilustrado na Figura 01. Nesse cenário, o personagem, representado por um patrulheiro espacial, deve ir até a nave que se encontra após uma barreira, que é feita de Vibranium em formato de U deitado. Se você usar forças de atração para guiar o patrulheiro espacial até a nave, ele ficará preso na armadilha criada pelo formato côncavo do U e não sairá de lá enquanto não fizer um buraco para escapar (#SQN 😁). Você precisará, então, de uma técnica adicional para que o patrulheiro espacial possa saber que a melhor forma de alcançar a nave não é indo direto em sua direção, mas contornando o obstáculo à frente.

Figura 01 - Exemplo de cenário 2D onde o uso de steering behaviors pode não ser apropriado



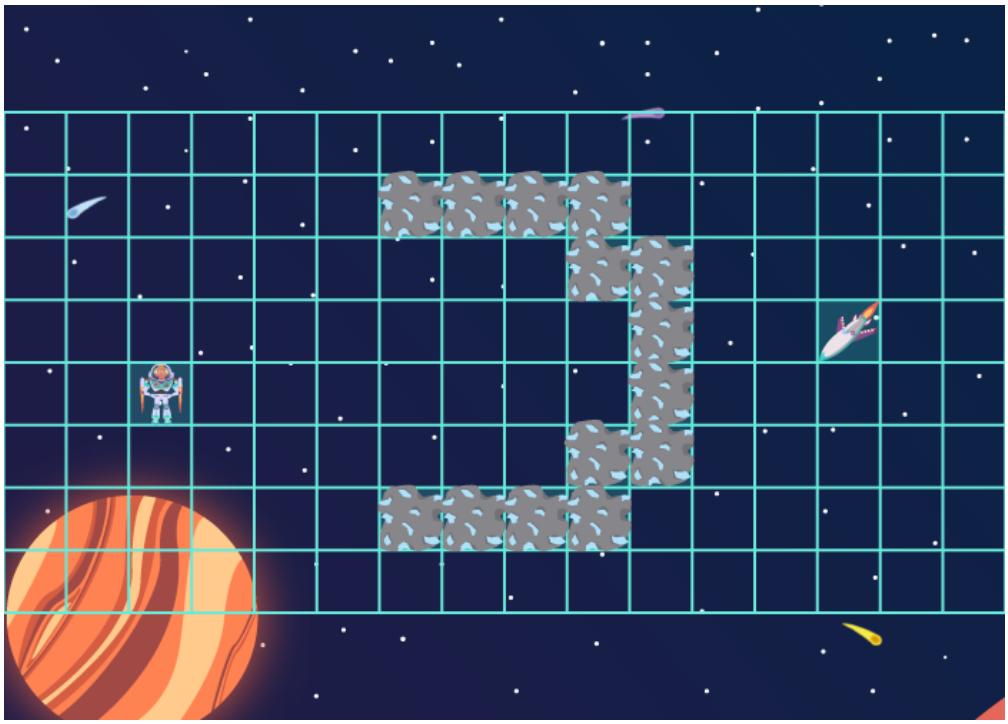
Olhando assim para o exemplo, parece muito simples. Como o computador não é capaz de saber que, nesse caso, a menor distância (a ser percorrida) não é uma reta?

O fato é que o computador precisa **planejar** sua trajetória. Então você fará com que os personagens saibam como planejar para onde ir quando houver obstáculos à frente. Depois que você ensinar ao seu personagem como calcular a melhor rota, você verá que ele conseguirá achar caminhos em labirintos bem mais rápido do que você. Ele será capaz inclusive de calcular o melhor caminho entre dois pontos não apenas evitando os obstáculos, mas também levando em conta inúmeros outros aspectos, como evitar inimigos, reduzir os custos (de gasolina), entre outros.

2. Primeiros caminhos: caminhando em linha reta

Retorne ao exemplo que gerou a necessidade de uma nova técnica de IA para o personagem. Mas simplifique o cenário, deixando-o discreto... você não quer torná-lo tímido. 😉 O objetivo aqui é transformar o espaço de representação do jogo em uma malha (*grid*) de células, como em uma tabela. Assim, o espaço deixaria de ser "contínuo" (sem interrupções) e passaria a ser "discreto" (com posições separadas em intervalos regulares). A Figura 02 ilustra como seria o cenário da Figura 01, mas "discretizado".

Figura 02 - Ambiente discreto do cenário apresentado na Figura 01

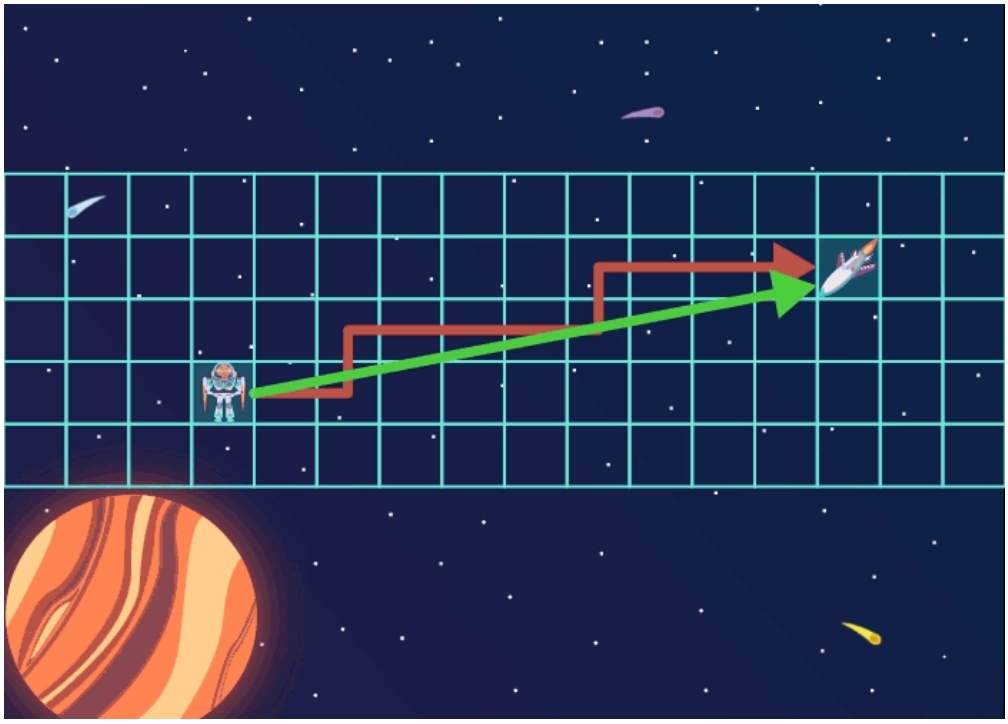


Com certeza você já viu muitos jogos cujos cenários são definidos em malhas regulares, como o apresentado na Figura 02. Na verdade, a maioria dos jogos simplificam a representação contínua do espaço em intervalos regulares. Essa simplificação é útil desde jogos *puzzles* a jogos de RPG. Então, adote também essa simplificação nos seus algoritmos de planejamento de deslocamento dos personagens.

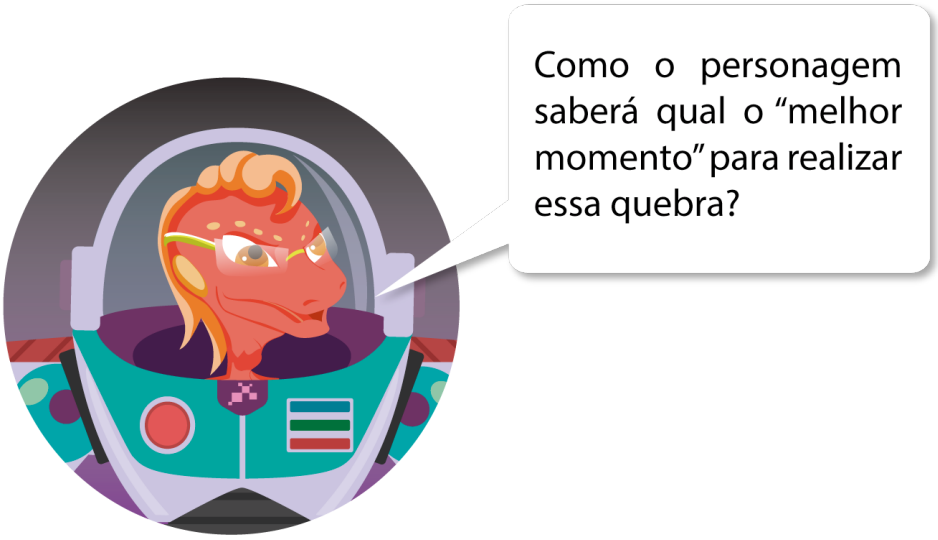
Comece do básico! Ensine o seu personagem a planejar seu deslocamento de um ponto ao outro da malha regular do cenário, mas sem obstáculos! 😁

Se o personagem deseja alcançar uma posição, como ele se encontra em uma malha, não dá para simplesmente ir em linha reta, como se fosse um espaço contínuo. É necessário, em uma das células da malha, alterar a rota gerando uma quebra de direção, como ilustrado na Figura 03.

Figura 03 - Diferença entre o deslocamento desejado e o realizado sobre uma malha regular



Na figura, a **seta verde** representa o caminho desejado pelo personagem e a **seta vermelha** representa um possível caminho dele.



Na verdade, você deve perguntar: quando o personagem saberá qual o “melhor caminho”? Por um momento, pode-se dizer que o “melhor caminho” é aquele que gera menor distância de deslocamento. Você verá que há diferentes formas de calcular a “distância” entre dois pontos, mas, por enquanto, assuma que essa distância **é fruto do número de células visitadas no deslocamento de um ponto ao outro**. E que também o personagem só pode se mover para as células adjacentes horizontais e verticais (as diagonais, não). Quando se usa essas convenções para representar as distâncias, você estará utilizando a **Distância de Manhattan**.

Curiosidade

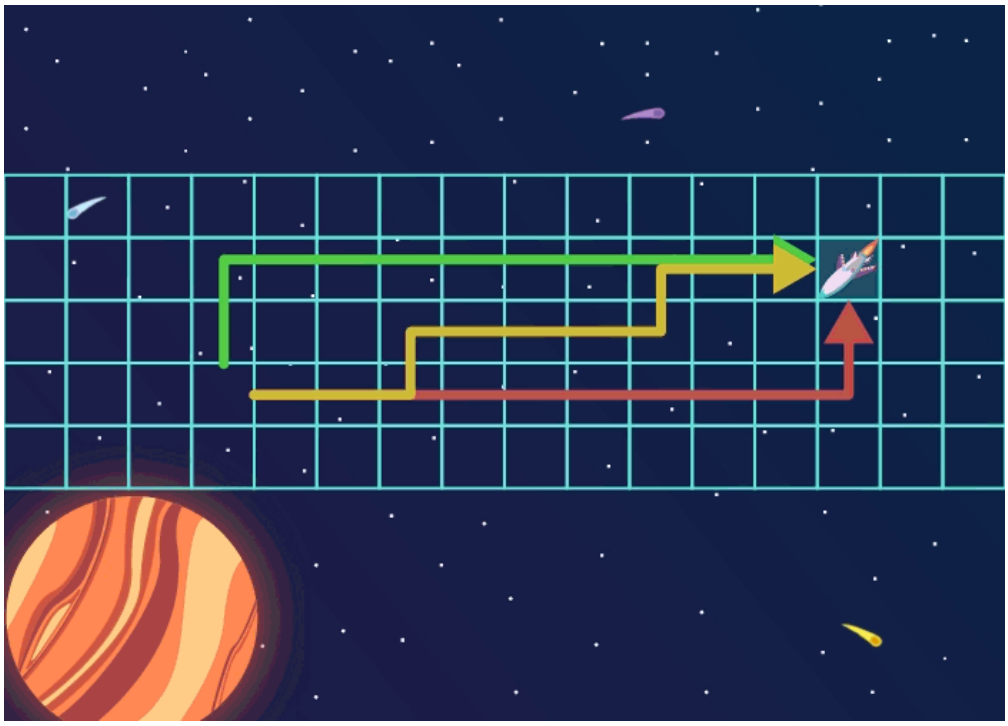
Distância de Manhattan?

Figura 04 - Mapa interativo de Manhattan

Esse nome é usado porque lá em **Manhattan** praticamente todas as ruas são paralelas ou perpendiculares. Não existem ruas “em diagonal”. Assim, as distâncias são medidas em número de quadras.

Considerando o cálculo do melhor caminho realizado através da **Distância de Manhattan**, veja a Figura 05.

Figura 05 - Possíveis “bons caminhos” entre dois pontos, considerando o “melhor caminho” baseado em número de movimentos a células adjacentes



Todos os caminhos representados nessa figura, bem como muitos outros, são caminhos igualmente bons. Pode-se dizer, então que, todos eles são os “melhores caminhos”, pois possuem o número mínimo de deslocamentos a células adjacentes, seja na horizontal ou na vertical.

Se todos são igualmente bons, qual deles o personagem deve adotar?

Uma boa estratégia é usar o que você considera mais “natural”. Em geral, considera-se mais “natural” um ajuste gradual no caminho trilhado, como a sequência de células na rota amarela da Figura 05. Para isso, adote um algoritmo concebido inicialmente para desenhar linhas retas nas telas dos computadores chamado de **Algoritmo de Bresenham**.

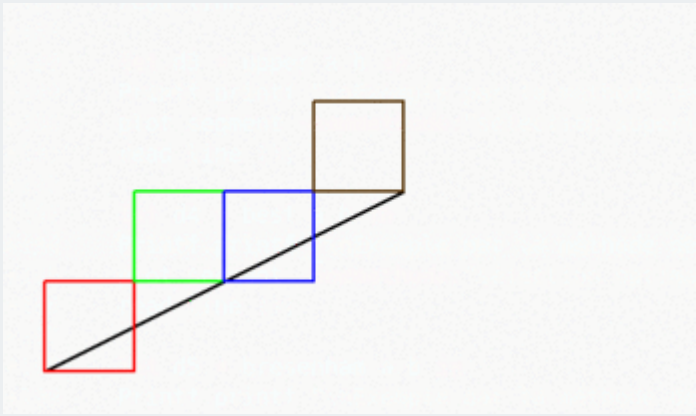
Saiba Mais

Figura 06 - Jack Elton Bresenham

Desenvolvido por Jack Elton Bresenham em 1962, na IBM, o **algoritmo da linha de Bresenham** é responsável por desenhar linhas em uma tela de computador. Ele é um dos primeiros algoritmos descobertos no campo da computação gráfica. Devido à velocidade e simplicidade, ele é implementado em vários hardwares, como plotters e placas gráficas modernas. Atualmente, o nome "Bresenham" é usado para toda uma família de algoritmos estendidos ou modificados do original.

Fonte: IT HISTORY SOCIETY. Dr. Jack Elton Bresenham. Disponível em: <https://www.ithistory.org/honor-roll/dr-jack-elton-bresenham>. Acesso em: 18 maio 2018 (Traduzido e adaptado).

Figura 07 - Exemplo de plotagem de um segmento com o algoritmo de Bresenham



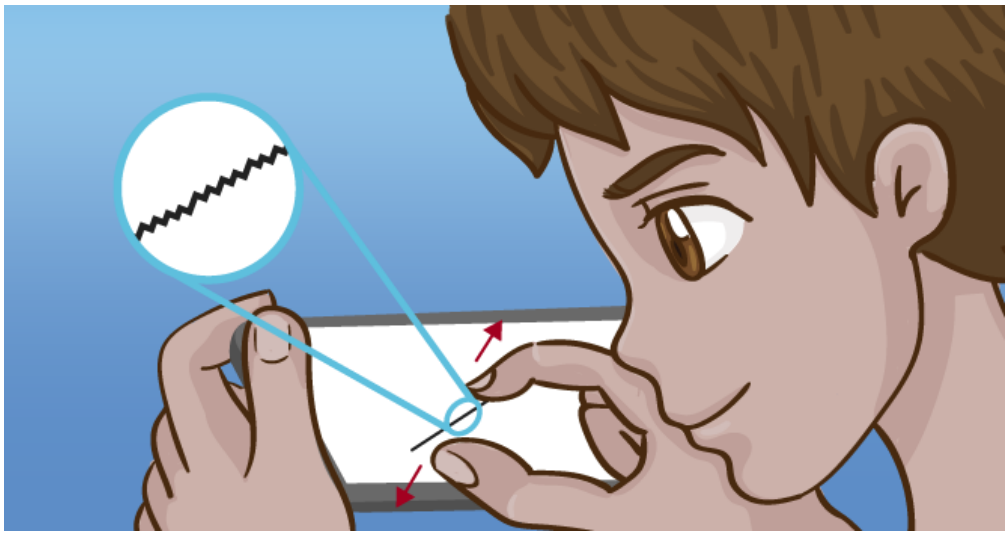
Fonte: REVOLVY. **Bresenham's line algorithm.** https://www.revolvy.com/main/index.php?s=Bresenham%27s+line+algorithm&uid=1575. Acesso em: 16 maio 2018

Para mais informações, acesse aqui. 😉

2.1 Conceitos iniciais do Algoritmo de Bresenham

A imagem que se vê nas telas do computador é definida por uma malha retangular, tal como o cenário da Figura 05. A diferença é que a malha usada em uma imagem é de *pixels* (É a menor unidade de uma imagem digital, que é fruto da combinação de três cores básicas: Red (vermelho), Green (verde) e Blue (azul), RGB), o que torna as quebras menos perceptíveis. Porém, se você for olhar atentamente, irá notar os "degraus" gerados pelos *pixels* pintados, como a Figura 08 ilustra.

Figura 08 - Zoom no desenho de uma reta em uma imagem

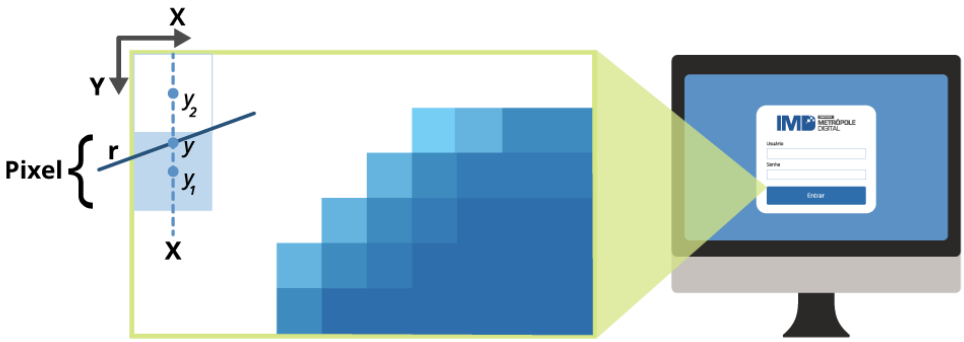


O algoritmo usado para desenhar uma reta de um ponto a outro em uma imagem faz exatamente o que se quer com o personagem. Ou seja, ele segue o rastro gerado pela tinta do **algoritmo de Bresenham**.

Esse algoritmo se baseia na equação da reta definida pelos seus pontos de extremidade. Essa equação define a reta "ideal" (uma reta contínua, sem quebras nem degraus). O algoritmo funciona, então, usando um laço para percorrer as coordenadas de um determinado eixo, por exemplo o eixo X, e calcular o Y correspondente na equação da reta. O pixel da coordenada X que tiver valor em Y mais próximo do Y calculado pela equação da reta será o pixel pintado.

Para ficar mais claro, observe a Figura 09.

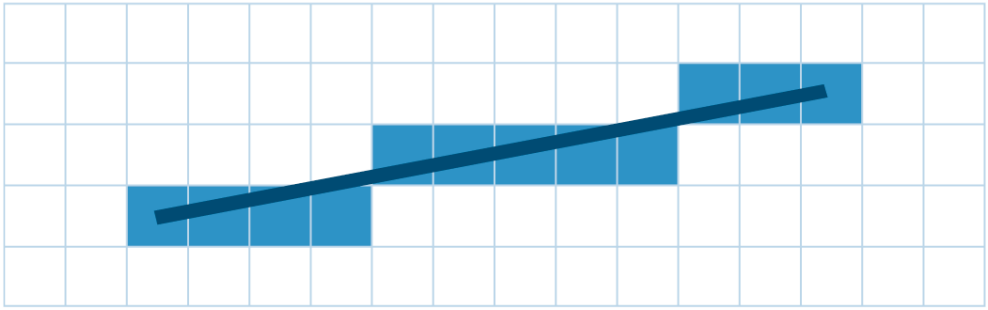
Figura 09 - Comparação entre as distâncias das coordenadas Y na reta calculada pela equação e em dois pixels



Nela, a reta r é calculada a partir da equação da reta tomando base os pontos extremos (no caso do percurso do personagem são os pontos de origem e de destino). Para um determinado valor x, temos y calculado a partir da equação da reta. Então, o pixel no eixo X que possui o Y mais próximo do y calculado será o pixel a ser pintado. No caso do exemplo da Figura 09, será pintado o pixel do y1.

Ao se aplicar o mesmo procedimento em um laço (*loop*) para cada um dos valores em X entre os pontos extremos da reta, tem-se um resultado similar ao apresentado na Figura 10.

Figura 10 - Sobreposição da reta gerada pela equação e a reta gerada pelo algoritmo de Bresenham

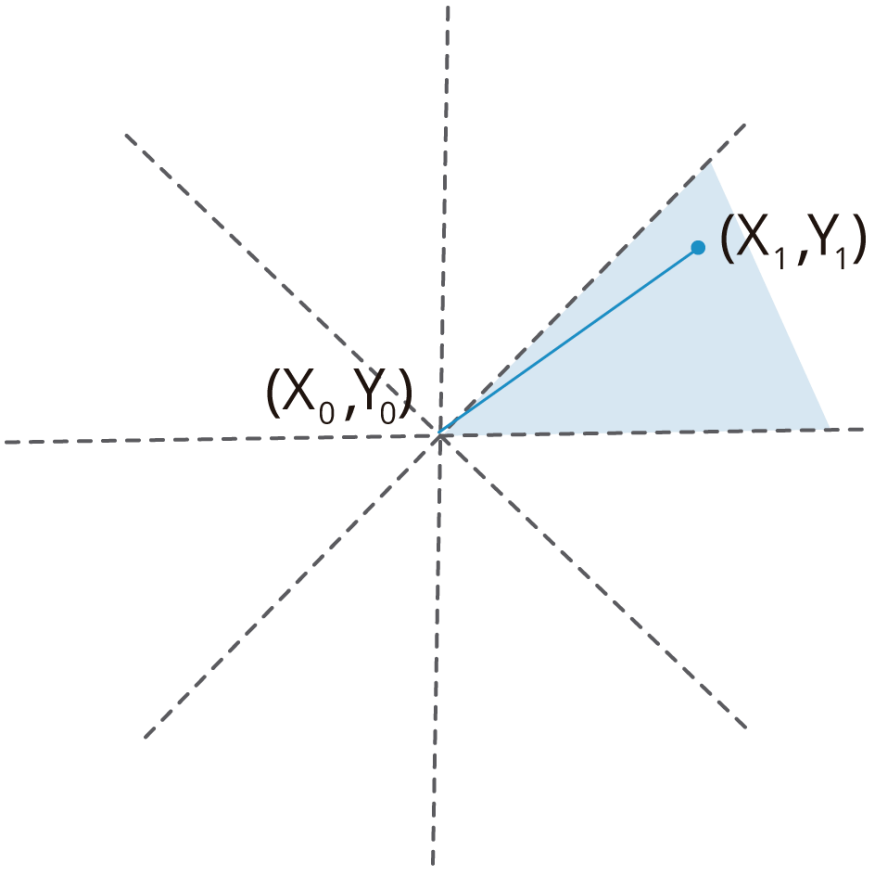


Porém, a princípio, a ideia acima funciona apenas para as retas cujo ângulo de inclinação se encontra no primeiro octante do plano... octante? O que é isso?

Bem, se o plano pode ser dividido em quatro quadrantes, ele também pode ser dividido em oito octantes! 😁

A Figura 11 ilustra melhor o que é um octante. O que está ressaltado na figura é a região em que, para todos os pontos pertencentes a essa região, os valores de *x* são maiores do que 0 e maiores que os de *y*. Ele é o primeiro octante do plano.

Figura 11 - O primeiro octante do plano



A princípio, a ideia funciona apenas nesse octante porque, em vez de ficar procurando o pixel com o *y* mais próximo da reta da equação, você começará com um *y* fixo e irá incrementá-lo apenas quando necessário. Se você observar a Figura 10, os valores de *y* conservam o valor inicial, até o momento em que a distância para a reta da equação ultrapassa o tamanho do pixel. Quando isso ocorre, o valor de *y* é incrementado. Em seguida, ele continua com o valor para os próximos pixels, até o momento em que a diferença ultrapassa novamente o tamanho do pixel e, novamente, ele é incrementado. E assim por diante, até chegar no ponto destino.

Isso só funciona para o primeiro octante, mas não se preocupe. Você definirá mais tarde um procedimento para esse octante e os demais casos serão gerados a partir da aplicação do mesmo procedimento sobre um espelhamento das coordenadas (as coordenadas estarão invertidas). 😱

2.3 Algoritmo completo

Agora que você entendeu a ideia do algoritmo, já pode descrevê-lo em formato de pseudocódigo. A rotina recebe quatro valores inteiros (x_0, y_0, x_1 e y_1) que correspondem às coordenadas da posição na qual personagem se encontra e a posição a qual ele deseja ir. Algumas variáveis são declaradas no início da rotina para facilitar os cálculos. **Inclinação**, que é o coeficiente angular da reta definida pelos pontos passados, **erro**, que é o fator de erro que se adiciona para totalizar o valor da inclinação e **caminho**, que é uma sequência de pontos inicialmente vazia, mas que representa o caminho a ser construído. Antes de iniciar o laço do algoritmo, defina as variáveis x e y que ficarão armazenando a posição a ser inserida no caminho.

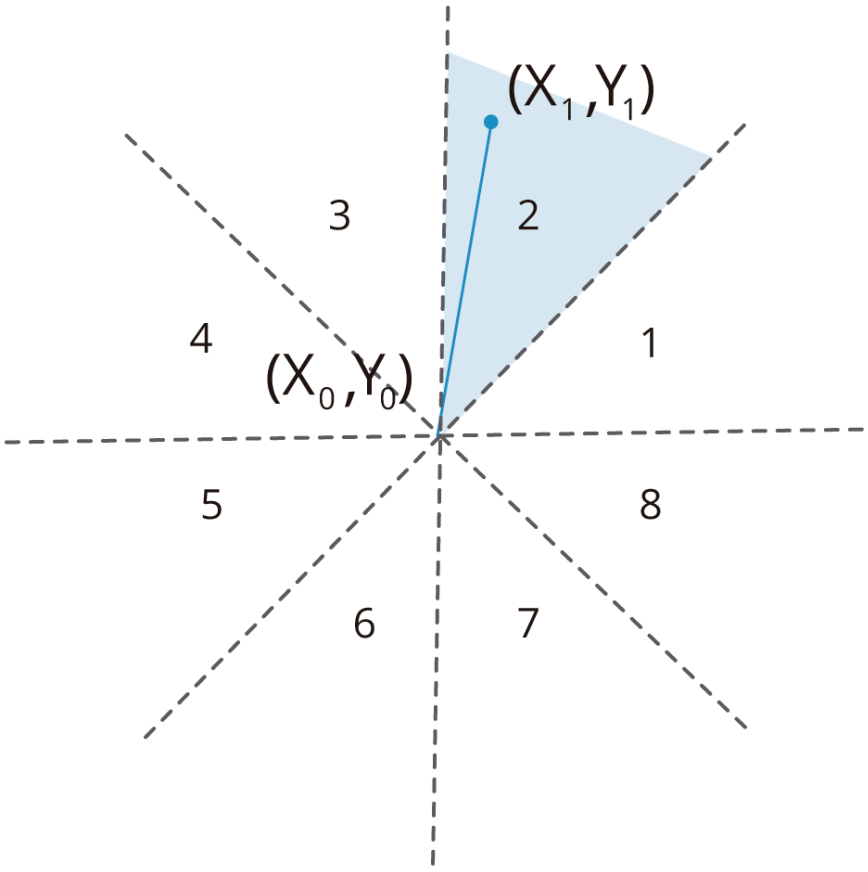
A cada iteração do laço, insira o ponto (x, y) no caminho e atualize o erro em função da inclinação da reta. Se o erro for maior ou igual a 1, então y é incrementado (chegou a hora de gerar uma mudança na rota 😉), o erro é reajustado e o ponto com o novo y é inserido no caminho. Por fim, após o término do laço, a posição do destino também é inserida no caminho, e este retornado.

Código 01 – Pseudocódigo do algoritmo de Bresenham para traçar um caminho reto

```
1   Função CaminhoReto (Int x0, Int y0, Int x1, Int y1)
2     Real inclinação = (x1 – x0) / (y1 – y0);
3     Real erro = -0.5;
4     Arranjo caminho = [];
5
6     Int x, y = y0;
7     Para x de x0 a x1 - 1
8       insere (x,y) em caminho
9       erro = erro + inclinação
10      Se erro ≥ 1 então
11        y = y + 1
12        erro = erro - 1
13        insere (x,y) em caminho
14      Fim_se
15    Fim_para
16    insere (x1,y1) em caminho
17
18    Retorna caminho
19  Fim_procedimento
```

O último detalhe que resta é fazer esse algoritmo funcionar para os demais sete octantes que restam no plano. Como já dito, da forma que foi descrita, funciona apenas para o primeiro octante. Você precisará generalizar para que o personagem siga em qualquer direção. O primeiro passo é descobrir em qual octante o caminho se encontra. Veja a figura 14.

Figura 14 - O segundo octante do plano e referências para os demais



Se você se lembra, o primeiro octante era definido pelo conjunto de pontos tal que $x > 0$, $y > 0$ e $x \geq y$, não é? Raciocine agora sobre o segundo quadrante, apresentado na Figura 14. Esse octante é definido pelos conjuntos de pontos tal que $x > 0$, $y > 0$ e $y \geq x$. Aplicando a mesma ideia para os demais octantes, você terá as condições presentes no Quadro 02. Nele, abs(x) representa o valor em absoluto (valor positivo) de x.

Quadro 02 - Condições de cada quadrante

Octante	Condição
1	$X_1 > X_0$, $Y_1 > Y_0$ e $X_1 \geq Y_1$
2	$X_1 > X_0$, $Y_1 > Y_0$ e $Y_1 \geq X_1$
3	$X_1 < X_0$, $Y_1 > Y_0$ e $Y_1 \geq abs(X_1)$
4	$X_1 < X_0$, $Y_1 > Y_0$ e $abs(X_1) \geq Y_1$
5	$X_1 < X_0$, $Y_1 < Y_0$ e $abs(X_1) \geq abs(Y_1)$
6	$X_1 < X_0$, $Y_1 < Y_0$ e $abs(Y_1) \geq abs(X_1)$
7	$X_1 > X_0$, $Y_1 < Y_0$ e $abs(Y_1) \geq X_1$
8	$X_1 > X_0$, $Y_1 < Y_0$ e $abs(X_1) \geq Y_1$

Agora que você sabe identificar os diferentes casos, falta chamar o procedimento de forma adequada. A estratégia mais simples é “enganar” o algoritmo fazendo-o “pensar” que se trata do primeiro octante, mesmo que não seja. Para isso, inverta as coordenadas, tanto as que você passou para ele, quanto as que ele insere no caminho. Por exemplo, em vez de passar as coordenadas (x_0, y_0) e (x_1, y_1) para ele, você passou (y_0, x_0) e (y_1, x_1), e trocou de volta o caminho gerado fazendo com que as coordenadas (x_0, y_0) e (x_1, y_1) sejam consideradas (y_0, x_0) e (y_1, x_1).

Para levar em conta todos os quadrantes, você precisará realizar as seguintes trocas:

Quadro 03 - Inversão dos dados para calcular **caminho** nos diferentes octantes

Octante	Ordem normal	Alteração na ordem de entrada	Alteração na ordem do caminho gerado
1	x, y	x, y	x, y
2	x, y	y, x	y, x
3	x, y	y, -x	-y, x
4	x, y	-x, y	-x, y
5	x, y	-x, -y	-x, -y
6	x, y	-y, -x	-y, -x
7	x, y	-y, x	y, -x
8	x, y	x, -y	x, -y

Resumo

Os chamados **comportamentos de direção** (*steering behaviors*) vistos por você nas aulas passadas, consistem em usar regras simples tentando deixar os movimentos do personagem o mais realista possível. Apesar de serem bastante úteis, os algoritmos de *steering behaviors* normalmente não consideram os elementos do cenário, ou seja, não levam em conta os obstáculos que possam estar no caminho.

Você precisou compreender as dificuldades de planejar o deslocamento de personagens em um cenário de jogo, ao observar uma situação aparentemente simples, que foi a de levar um personagem “estelar” de um lado para o outro, mas sem a presença de um obstáculo.

Para a movimentação desse personagem, você aprendeu uma técnica de mapear o cenário por meio de uma *grid* (malha) e como alterar a rota dele gerando uma quebra de direção por meio da **Distância de Manhattan**. Já para deslocá-lo em uma reta, num ambiente discretizado, foi utilizado um algoritmo básico denominado de **Bresenham**.

Hum... considere-se quase um patrulheiro estelar.

Quase porque você ainda irá aprender como desviar dos obstáculos... na próxima aula. 😁

Até lá!

Referências

MILLINGTON, Ian; FUNGE, John. **Artificial intelligence for games**. 2. ed. Massachusetts: Morgan Kaufmann Eds, 2009.

MADHAV, Sanjay. **Game programming algorithms and techniques**: a plataform-agnostic approach. São Paulo: Addison-Wesley, 2014.