Inteligência Artificial para Jogos

Aula 03 - Comportamentos de navegação II

Apresentação da Aula

Olá! Pronto para continuar?

O que lhe motivou chegar até aqui foi fazer com que os jogadores da aula anterior não ficassem se esbarrando uns nos outros. Lá, você explorou comportamentos de navegação mais simples, que não exigiam informações adicionais de outros elementos no cenário. Agora, você verá novos comportamentos...

Prepare-se para conhecer dois tipos, o primeiro é um específico denominado de **desvio de obstáculo**, podendo ser encontrado em algumas referências no termo original *Obstacle Avoidance*. E o segundo é o **caminho a seguir**, também chamado de *Path Following*. Ambos os comportamentos serão importantes para você criar jogadas ensaiadas. Com eles, pode-se dizer aos jogadores que em uma determinada jogada eles precisarão passar por alguns pontos de controle.

Então vamos lá? É... vida de técnico não é moleza! 😉

Objetivos

> Conhecer os comportamentos de navegação de desvio de obstáculos (*Obstacle avoidance*) e de caminho a seguir (*Path following*);
>
> Compreender onde e como aplicá-los;
>
> Implementar os comportamentos de navegação de desvio de obstáculos e de caminhos a seguir no Unity.

1. Comportamento *Obstacle Avoidance* (desvio de obstáculo)

Bom... são vários detalhes que você vai precisar pensar. Mas comece por partes. A ideia por trás do comportamento para evitar obstáculos é simples, basta aplicar uma força contrária aos obstáculos que estão logo à frente do personagem. Simples assim! 😁

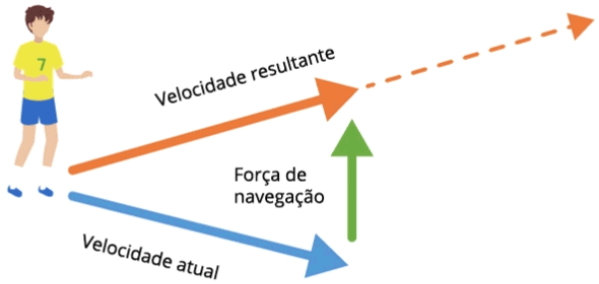
Entretanto, há alguns detalhes que precisam ser levados em conta. Por exemplo, se você tiver com vários obstáculos à frente do NPC, será que a força de navegação levará em conta todos eles? Se não, de qual deles desviar? Como eu vejo quando um obstáculo está na minha frente? Ou melhor, quando um obstáculo se encontra na minha rota de colisão? O quanto de força eu devo aplicar para afastar apenas o suficiente para evitar a colisão? Bem... são muitos questionamentos. rsrs

A solução é a seguinte: se houver vários obstáculos, você irá selecionar aquele que representar maior “ameaça" de colisão. Nesse cenário, bem como na maioria dos cenários, a maior “ameaça" é o obstáculo mais próximo. Mas isso pode ser diferente. Por exemplo, é possível que o índice de ameaça seja uma razão entre o tamanho e a distância. Então, o NPC pode se desviar de obstáculos mais distantes, desde que eles sejam grandes o suficiente para gerarem um desvio maior que um pequeno que está mais próximo. No seu caso, os obstáculos são outros jogadores, todos do mesmo tamanho. Então, fique com a versão mais simples, **tratando de desviar de quem está mais próximo**. 😉

Uma vez escolhido de quem se desviar, aplique uma força de navegação contrária ao obstáculo a cada novo *frame*. O resultado é que o personagem irá se desviar do obstáculo escolhido. Se um novo obstáculo aparecer (ou continuar presente), ele aplicará novamente o mesmo procedimento a cada um dos obstáculos no caminho. E assim por diante...

Nesse cenário, por exemplo, suponha que um jogador esteja indo em direção ao seu alvo, como ilustra a Figura 01.

Figura 01 - Força de navegação para desviar de um obstáculo



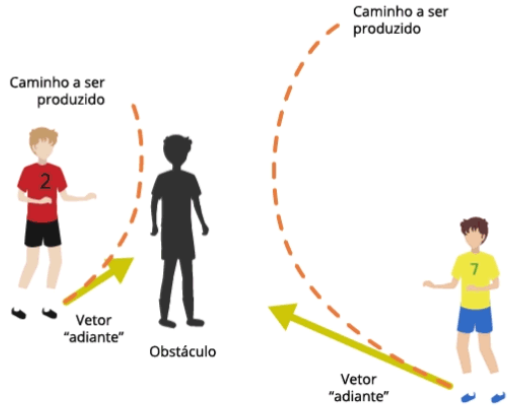
No meio do caminho, aparecerá um jogador adversário para atrapalhá-lo. Ele precisará, então, desviar, aplicando uma força contrária (ou tangencial) à posição do adversário, de forma que a velocidade resultante faça com que ele contorne o obstáculo.

Bom, ideia é boa! Mas como saber quando o adversário se encontrará no caminho do jogador? Veja agora como resolver esse segundo problema.

Para saber se há obstáculos à frente, você precisará definir uma margem (distância) que considera adequada para o jogador se desviar deles. Ou seja, precisa estabelecer em que momento a força de desvio de obstáculo pode ser aplicada, caso contrário, os jogadores podem se desviar de obstáculos que ainda estão muito afastados deles. Por exemplo, da trave que se encontra do outro lado do campo. E não é isso que você quer, correto?

Com o valor dessa margem, crie um vetor na mesma direção do jogador, cujo tamanho seja igual à margem preestabelecida. Pode-se dizer que esse vetor será uma espécie de sensor para ver o que está “adiante", ou seja, que vai detectar se algum objeto (no exemplo, o juiz) do jogo cruza com ele. Se houver alguma interseção desse vetor com o juiz, é porque ele encontra-se no “meio do caminho" do jogador. **Quanto maior o vetor (margem de colisão), mais prematuramente o jogador vai se desviar dos obstáculos**, como ilustra a Figura 02. Faz sentido, não é?

Figura 02 - Exemplo de vetores “adiante” com diferentes tamanhos



No Unity, esse vetor “adiante" pode ser implementado como um sensor, como o **EdgeCollider2D** ou **BoxCollider2D**. Você irá, entretanto, usar uma versão mais genérica, usando um “raio" de colisão.

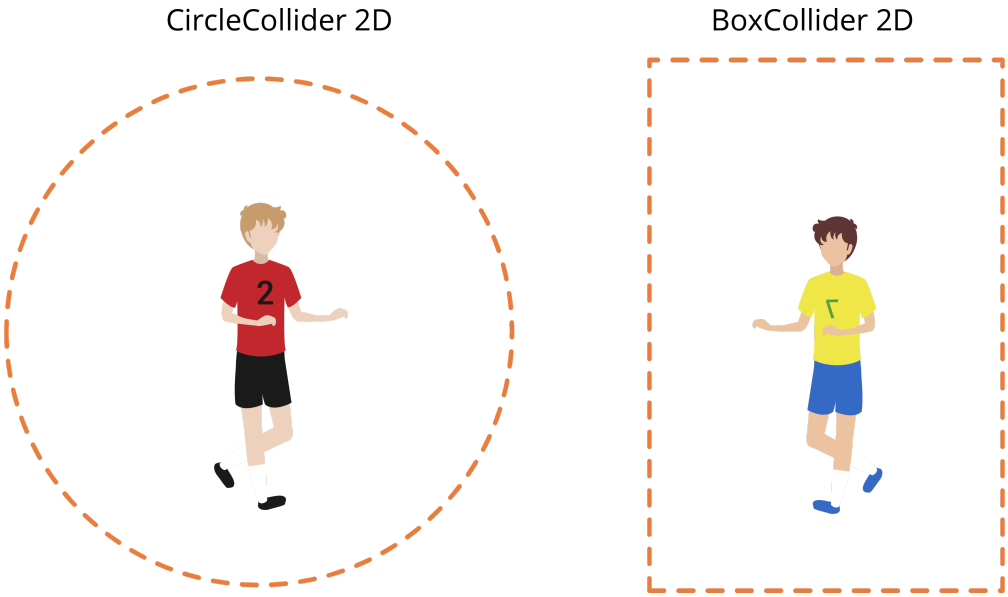
O vetor “adiante", chamado a partir de agora de **ahead**, pode ser calculado projetando a posição do jogador através de um vetor na direção de sua velocidade, porém previamente com um tamanho igual à margem de colisão. Assim, o vetor pode ser calculado através das instruções a seguir.

```
adiante = posição + normaliza (velocidade) * margem_de_colisão
```

Para você saber se há algum obstáculo no caminho do jogador, basta verificar se a semirreta que vai da posição do jogador à sua posição adiante cruza (tem interseção) com algum dos objetos do cenário. Esse cálculo requer que os objetos do jogo possuam uma figura geométrica para representar seus limites, também conhecidos como **regiões** ou **caixas envoltórias** (*bounding box*). Apesar de o termo usar “caixa", nem sempre essa figura geométrica é uma caixa (em jogos 3D) ou retângulo (em jogos 2D). Muitas vezes, é utilizado um círculo (em 2D) ou uma esfera (em 3D).

No Unity, as **caixas envoltórias** são definidas por componentes *Colliders*, os quais podem-se anexar aos objetos do jogo. Nesse caso, pode-se associar um **CircleCollider2D** ou **BoxCollider2D**. A Figura 03 mostra as regiões em que a interseção com outro objeto seria considerada como colisão. Apesar das duas possibilidades, normalmente comportamentos mais realistas são conseguidos usando o círculo como **região envoltória**. Por essa razão, usa-se o **CircleCollider2D** como sensor de colisão.

Figura 03 - Dois tipos de sensores para detectar colisão



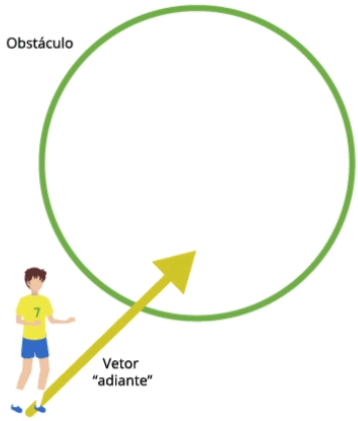
Você aplicará o raio a partir do centro do objeto *CircleCollider2D* (normalmente posicionado no centro da imagem do jogador), e se o jogador cruzar com um *collider* é porque há um obstáculo à frente. Mas lembre-se que se houver mais de um obstáculo à frente, você terá de escolher a “maior ameaça" à colisão. Como visto anteriormente, isso é calculado por meio da distância do personagem ao obstáculo. Então, você pode ordenar pela distância todos os obstáculos do caminho e o primeiro do conjunto ordenado será o mais próximo.

Muito bem! 👏 O segundo problema, que é “de quem você deve se esquivar", foi resolvido.

Você vai calcular qual a força de navegação a ser aplicada para o jogador se desviar do obstáculo encontrado. Como na seção anterior, é preciso definir uma força que “empurre" o jogador a uma direção desejada, nesse caso, para longe do obstáculo (ver Figura 04). O cálculo da força de desvio é similar ao cálculo da força de navegação usado no comportamento *Seek* (que, como você deve lembrar, foi utilizado na aula passada). Calcula-se, então, o vetor força através da diferença de dois outros vetores. Porém, como não é atração em direção ao alvo, mas sim repulsão do obstáculo, faz-se a subtração entre o vetor “adiante" e o centro do objeto de colisão, e depois se aplicará a mesma transformação que se aplicou no *Seek* para definir o tamanho do vetor resultante com força máxima, conforme fórmula abaixo.

```
força_desvio = adiante – centro_obstáculo
força_desvio = normaliza (força_desvio) * força_de_desvio_máxima
```

Figura 04 - Vetor força de desvio de obstáculos



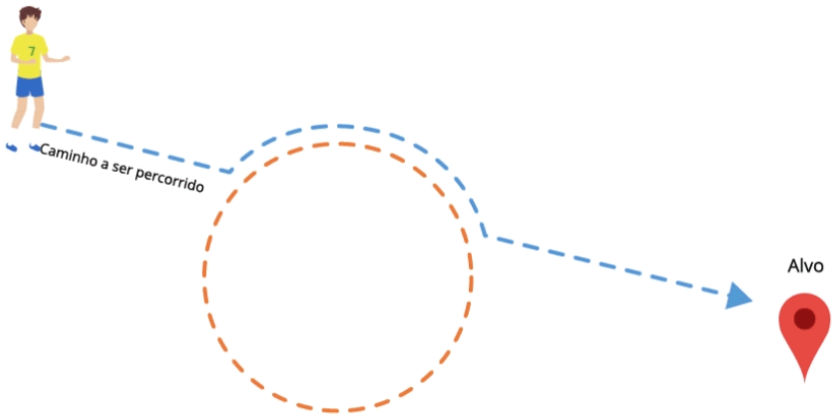
Resta mais uma questão: o comportamento de desvio de obstáculos não funciona sozinho. O NPC só se desvia de um obstáculo se ele estiver indo para algum lugar, perseguindo ou escapando de alguém. Então, os comportamentos precisam ser combinados. Mas como fazer isso?

Uma opção é você combinar as forças, que é simplesmente somar os seus vetores. Isso vai funcionar com vários comportamentos. Porém, com o desvio de obstáculos, essa pode não ser a melhor estratégia. Uma força pode anular a outra e os objetos irão colidir. Nessa condição de os objetos colidirem, a soma de vetores é suficiente. Porém, se quiser evitar isso, uma segunda opção é priorizar um dos comportamentos, que é o NPC **seguir em direção ao seu alvo** ou ele **se desviar**.

Como funcionaria esse esquema? É mais simples do que você imagina! A cada momento do jogo, você vai fazer seu jogador escolher entre que tipo de comportamento (e por consequência, qual força de navegação) ele vai aplicar. Se o caminho estiver livre, ele vai em direção ao alvo (comportamento *Seek* ou *Arrival*). Se aparecer um ~~cachorro~~ jogador adversário no meio do caminho, ele escolherá desviar do obstáculo (comportamento *Obstacle Avoidance*). Até aqui nenhuma novidade, né? Só que para você fazer os desvios ficarem pequenos o suficiente, você vai fazer o jogador se perguntar isso **a cada frame do jogo**, de forma que o desvio “contorne" o obstáculo apenas o suficiente para liberar o seu caminho.

A Figura 05 ilustra o que ocorrerá nesse tipo de combinação de forças.

Figura 05 - Caminho a ser percorrido com desvio de obstáculo e busca pelo alvo



Essa estratégia de **desvio de obstáculo** resolve a questão para cenários mais simples, porém ainda existem situações onde a simulação resultante dele apresentará problemas. Por exemplo, é possível que, ao estar muito próximo do obstáculo, o NPC tente desviar mesmo que esteja quase parado. Outro problema surge quando os obstáculos estão tão próximos do personagem que a manobra de desvio não o impede de colidir com o mesmo. Isso acontece porque os comportamentos de navegação alteram a velocidade do personagem **paulatinamente**, ao longo de várias iterações do jogo, e pode ser que, mesmo desviando, não haja tempo de evitar a colisão (às vezes isso acontece na vida real, não é mesmo?). Nesse caso, uma possível solução é fazer com que a margem do sensor de colisão seja dinâmica, ou seja, que varie em função da velocidade do personagem.

Veja que essa ideia faz muito sentido. Por exemplo, um motorista de um carro prestará atenção nos obstáculos que se encontram a vários metros de distância dele, enquanto alguém passeando de bicicleta concentrará sua atenção aos obstáculos que estarão há apenas alguns metros. Então, você poderá fazer o mesmo com nosso o jogador de futebol. Fazendo a margem de desvio de obstáculo variar, dependendo se ele estiver correndo ou andando em campo.

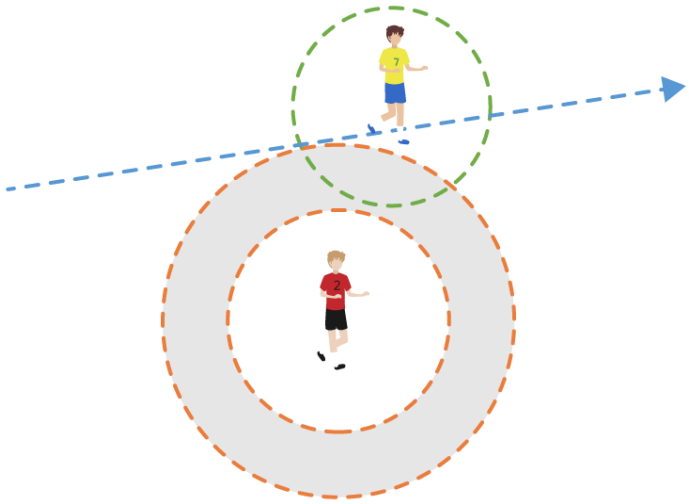
Agora que as ideias foram todas devidamente apresentadas, coloque em prática tudo isso que você viu na teoria! 😉

Passo 01 – Implementar no Unity o desvio de obstáculos

Adapte o cenário da aula anterior para esse novo comportamento, com os mesmos elementos, porém, coloque mais jogadores no campo. Crie um tipo de jogador, porque, por enquanto, você quer que apenas um jogador siga as instruções (deslocar-se dentro do campo em direção ao alvo clicado). É interessante, então, criar o *prefab* de um jogador adversário, defini-lo como obstáculo e ver se o jogador (o que você controla) percebe que há adversários à frente e consegue desviar deles.

Inicialmente, você precisa criar um objeto para ser um modelo de jogador adversário. Para isso, há uma imagem de jogador do time laranja na pasta *Sprites*. Carregue essa imagem para a cena do jogo e crie um objeto com o componente *Sprite Renderer*. Ajuste-o na posição, rotação e escala desejadas (na aula anterior, você utilizou uma escala de 0.5 para o jogador). Para ele ser detectado, insira o componente *CircleCollider2D*. Como você está trabalhando com apenas um raio (pode trabalhar com dois se assim desejar, um para o lado esquerdo e outro para o direito), dobre o raio do *collider* para que leve em conta o corpo do NPC que está desviando e do que será desviado, como ilustra a Figura 06.

Figura 06 - Raio do CircleCollider2D com tamanho dobrado



Além desse ajuste, informe no *collider* que ele é um gatilho (*Is Trigger*). Esse ajuste é apenas para informar que nenhuma física será aplicada quando houver colisão. Você está apenas querendo “prever" uma possível colisão através de um raio a ser disparado a partir da posição do jogador que está sendo controlado.

Uma vez que o objeto foi criado, crie um *prefab* dele para poder replicá-lo em diferentes posições do campo, uma pasta chamada *Prefabs* e, em seguida, um novo objeto *prefab* vazio nela. O próximo passo é arrastar e soltar o jogador recém-construído sobre o *prefab* vazio. Replique o objeto arrastando-o sobre o cenário o número de vezes que desejar (5 é um bom número, pois você fará um time de 5).

Agora, você precisa alterar o *script* de comportamento do jogador que está sendo controlado.

A única variável pública a mais que se precisa é uma para definir a distância em que o jogador vai começar a se desviar dos obstáculos. Essa variável vai delimitar o vetor “adiante", explicado anteriormente. Em seguida, crie um método da classe para calcular a força de navegação para o desvio de obstáculos, seguindo as explicações anteriores. O Código 01 mostra o resultante dessas explicações.

Código 01 – Script com o comportamento de desvio de obstáculos

```
public class SoccerPlayerBehavior : MonoBehaviour {

   ...

   public float aheadDistance = 1.5f;

   ...

   Vector2 ObstacleAvoidance() {
      Vector2 steer = Vector2.zero;

      float ahead = aheadDistance * (body.velocity.magnitude / speed);
      RaycastHit2D[] obs =
         Physics2D.RaycastAll(body.position, body.velocity, ahead);

      if (obstacles.Length > 1) {
         Array.Sort(obs, delegate(RaycastHit2D h1,RaycastHit2D h2){
            return h1.CompareTo(h2);
         });
         RaycastHit2D hit = obs [1];

         Vector2 to = (Vector2) hit.transform.position - body.position;
         Vector2 dir = SetMagnitude(body.velocity, to.magnitude) - to;
         steer = SetMagnitude(dir, maxSteeringForce);
      }
      return steer;
   }
}

```

Preste atenção nos "..." porque eles representam partes omitidas de cada código!

No código, você inicializou a variável *steer* com um vetor nulo (0,0). Se houver colisão à frente, você irá alterar o valor dessa variável antes de retorná-la. Caso contrário, o retorno de um vetor nulo indica que não há força alguma a aplicar para desviar de obstáculos.

Em seguida, calcule a distância que levará em conta no teste do que há à frente, cuja variável **ahead** armazenará. Para isso, use o atributo público da classe **aheadDistance** multiplicado pela fração da atual velocidade pela velocidade

máxima do NPC. Ou seja, caso o NPC esteja na velocidade máxima, essa fração será 1, fazendo com que **ahead** seja o próprio valor de **aheadDistance**. Quanto menor a velocidade do atual, menor será **ahead**, que é utilizado na linha seguinte para computar quais os *colliders* existentes se traçar um raio da posição atual do NPC, na direção de sua velocidade e de tamanho **ahead**. O método **Physics2D.RaycastAll()** retorna um arranjo de informações sobre as colisões desses *colliders*. O arranjo é, então, armazenado na variável **obs** (de obstáculos), sendo cada um dos elementos do tipo **RaycastHit2D**.

O teste em seguida verifica se o número de elementos em **obs** é maior do que 1, ou seja, se há mais de um *collider* à frente. Na verdade, sempre haverá pelo menos um, pois o próprio jogador que o *script* está controlando possui um *collider*, então haverá no mínimo ele. Se houver mais de um, então é porque há obstáculos (além dele) à frente. Nesse caso, como você quer apenas o obstáculo mais próximo, ordene o arranjo retornado. Na ordenação através do método **Sort()**, passe um delegate que recebe dois objetos do tipo **RaycastHit2D** e retorna -1, 0 ou 1, dependendo se a distância do primeiro é menor, igual ou maior do que a do segundo, respectivamente. Apenas com essa informação, o algoritmo do método de ordenação consegue colocar os elementos do arranjo em ordem de distância. Com o arranjo ordenado, o objeto que você precisará desviar, **hit**, é o segundo elemento do arranjo, pois o primeiro é o próprio NPC.

As linhas seguintes seguem a mesma estratégia de cálculo de vetores dos comportamentos apresentados anteriormente (Seek e Arrival), só que agora no sentido contrário. O vetor ***to*** é o que vai da posição atual do NPC à posição do obstáculo a ser desviado. Ou seja, ele indica a direção do obstáculo. Use a mudança de tipo explícita (*cast*) para **Vector2** porque, por padrão, a **transform.position** de qualquer objeto no Unity é um vetor 3D. Para calcular a direção da força que desvia do obstáculo, use a diferença do vetor velocidade (com o tamanho da distância do NPC para o obstáculo) pelo vetor com a posição do obstáculo. Com a direção de força calculada, ajuste para que seja dada a maior força de navegação possível.

Por fim, só precisa alterar a chamada para **ObstacleAvoidance()** no método **FixedUpdate()**. Se não houver força alguma, aplique a busca pelo objetivo. Caso contrário, você irá se desviar, como implementado no Código 02.

Código 02 – Script com o comportamento de desvio de obstáculos

```
...
void FixedUpdate () {
    Vector2 steering = ObstacleAvoidance();

    if (steering == Vector2.zero) {
        steering = Arrival();
    }
}
...

```

Muito bem! Você está fazendo o jogador correr em uma direção desejada (*Seek*). ir parando quando estiver chegando (*Arrival*), desviar de obstáculos (*Obstacle avoidance*), bem como outros comportamentos vistos aula 02 (escapar de um adversário (*Flee*) e perseguir um alvo em movimento (*Pursuit*)). Falta agora ele começar a fazer jogadas ensaiadas. O primeiro passo é fazê-lo seguir uma sequência de pontos de um caminho, veja na seção a seguir.

2. Comportamento *Path Following* (seguir o caminho)

Da mesma forma que *Obstacle Avoidance*, o comportamento *Path Following* **seguir um caminho**, pode ser implementado de várias formas. Veja aqui uma forma simplificada, que difere da original, porém, que apresenta resultados adequados para o cenário de um jogo de futebol.

Quando os comportamentos de navegação foram criados por Reynolds, ele definiu o comportamento para seguir estritamente as linhas que ligam os pontos do caminho. No seu método, ele interpola o segmento de reta entre os pontos através da projeção da posição do personagem na reta, e faz com que a força de navegação empurre o personagem para o ponto correspondente àquela fração do segmento de reta. O resultado era que o personagem seguisse o caminho como se fosse um trem em seu trilho.

Em alguns cenários, isso é ótimo! Por exemplo, se você tivesse fazendo um jogo de corrida, seria necessário que os NPCs, que seriam os carros, percorressem a pista de corrida seguindo suas retas e curvas. Porém, no cenário de jogo de futebol, não

há necessariamente uma “pista" a ser seguida. Então, nesse caso, onde se quer apenas controlar possíveis posições dos jogadores, não é necessário ser tão preciso. Dessa forma, flexibilize mais a estratégia, adotando uma mais simples.

Defina, inicialmente um caminho como uma sequência de pontos de controle. A ação de **seguir um caminho** passa a ser simplesmente ir na direção do primeiro ponto da sequência e, assim que alcançá-lo, ele será retirado.

Fácil de entender, né? 😁

O fato é que quando o ponto alcançado for retirado, o próximo passa a ser o primeiro da sequência. Logo, o personagem irá na direção desse próximo ponto, que, por sua vez, ao ser alcançado, será retirado da sequência e um novo ponto toma seu lugar, e assim por diante até não ter mais ponto na sequência para visitar.

Durante esse percurso, você poderá também acrescentar a possibilidade de que novos pontos sejam inseridos no caminho. Quando isso ocorrer, esse novo ponto deve ser o último a ser visitado, ou seja, deve ir para o final da sequência.

Figura 07 - Caminho como uma sequência de pontos a serem visitados pelo personagem

Como armazenar os pontos dessa sequência?

Em computação, sempre que se lida com uma forma de armazenar dados que possuam operações específicas sobre eles, está lidando com **estruturas de dados**. Existem várias estruturas de dados, cada uma com um propósito diferente e adequada para resolver um problema específico. Nesse caso, as operações que você poderá realizar sobre os pontos que foram dados para formar o caminho, é unicamente inserir um novo ponto no final e retirar um do início. Ou seja, sempre que algo novo é acrescentado na estrutura, ele vai para o final, e sempre que algo sai da estrutura, ele sai do início.

Ora, isso não lembra algo? Em determinados momentos, como quando você vai comprar os bilhetes do cinema, fazer pagamentos em bancos, entre outros. Isso mesmo, uma **Fila**. Pois é, na computação, também acontece algo parecido, mas se utilizando da **estrutura de dados** para inserir informações no final e retirar do início.

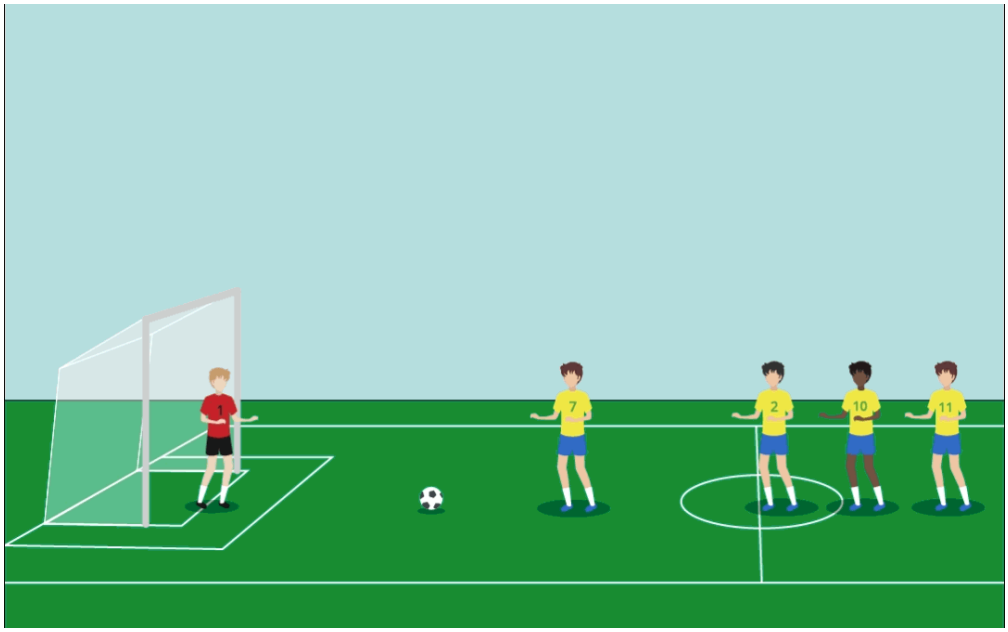
Acho que não foi muito difícil imaginar esse nome, não é? 😁

Imagina um treinamento de cobrança de pênaltis, como o ilustrado na Figura 08. O treinador chama a molecada, e alinha eles numa fila, um atrás do outro. Quem vai bater o primeiro pênalti? Vai ser o primeiro da fila, certo? E se alguém do meio da fila tentar furar para bater o pênalti antes da vez dele?

Opa, aí o treinador não deixa... A estrutura de dados **Fila** funciona dessa forma: o primeiro que chegou vai ser o primeiro que vai bater o pênalti (ou o que o seu programa precisar fazer!), e quando ele bater, vai sair da fila e dar a vez ao próximo.

Isso acontece até todo mundo ter batido os seus pênaltis! Se alguém chegar atrasado, só pode entrar no final da fila, e vai ser o último a treinar sua habilidade. Então sempre que se usa uma estrutura de dados **Fila**, os dados são armazenados em uma sequência, e ela não pode ser quebrada: **o primeiro que for colocado na fila será o primeiro dado processado, e assim por diante sem ninguém pular a vez do outro**.

Figura 08 - Representação de uma Fila



Além da **Fila**, existem outras estruturas de dados cujos nomes também são bem sugestivos. Por exemplo, existe uma estrutura chamada de **Pilha**. Bom... como se refere a um conjunto de dados, não pense que a **Pilha** aqui é sinônimo daquela bateria que você coloca no seu *gamepad*. A **Pilha** que está sendo mencionada aqui é parecida com quela estrutura que normalmente sua mãe reclama de sua mesa de estudo... só que um pouco mais organizada. 😁

Pense em uma pilha de livros...

Normalmente, quando você quer colocar um novo livro nesse “conjunto", você coloca no topo dele. Para retirar um livro desse conjunto... bom... se o que você quer é apenas remover um livro do conjunto (seja ele qual for), é mais sábio você retirar apenas o que se encontra no topo (é verdade que, com uma certa habilidade, você conseguirá remover um que está lá no meio, mas se seguir o “princípio do menor esforço", não será esse do meio que você vai remover).

Voltando ao nosso treinamento, imagina que um garoto cobra o pênalti perfeito, estilo “paradinho com giro 360º e dando cambalhota antes de chutar a bola". Ele corre com emoção e dá um mergulho de peixinho no campo.... Mas não contava que os colegas emocionados iriam querer comemorar com ele! Todos pulam em cima do pobre coitado, numa grande pilha de jogadores acumulados em cima do grande craque. Ele pode até tentar sair, mas não vai conseguir! Para isso, o jogador lá de cima da pilha tem que sair, depois o segundo, o terceiro... até que o último esteja livre do esmagamento e possa ser colocado na maca para atendimento. 😱

Assim se comportam os elementos de uma estrutura **Pilha** em que os dados ainda são colocados sequencialmente para o processamento, mas o primeiro dado que será processado é o último que foi colocado (o último jogador que pulou tem que ser o primeiro a sair).

Existem muitas estruturas de dados na computação, mas **Filas** e **Pilhas** são as mais comuns, independentemente se a ação desenvolvida é um jogo ou aplicação bancária. Nesse caso, por enquanto, será utilizada apenas a **Fila**.

Por ser uma estrutura de dados muito comum, o C# (utilizado no Unity) já definiu uma classe própria, chamada **Queue**, que a implementa. O uso dessa classe irá facilitar a sua vida para implementar o caminho do jogador.

Bom, uma vez que se tem o caminho armazenado em uma **Fila**, o que fazer para calcular as forças de navegação?

Como sempre, a sua estratégia precisa ser adequada ao problema que está enfrentando. Tudo vai depender de quão preciso ou de quão realista você quer que os jogadores sejam, sabendo que precisão e realismo é diretamente proporcional à complexidade. Ou seja, quanto mais precisa e realista, mais complexa deve ser sua estratégia.

Imagine, por exemplo, se você fosse um jogador de futebol e seu técnico dissesse “corra em volta do campo tocando em cada uma das bandeirinhas”, como você faria isso? Obviamente, você iria em direção a uma bandeirinha até tocá-la, em seguida para a próxima até tocá-la e assim por diante. Normalmente, você não iria se preocupar com uma próxima bandeirinha até que ela seja a próxima a ser tocada. Essa é uma estratégia simples. Nela, os próximos pontos do caminho não são levados em conta.

Imagine (novamente) uma situação mais complexa, que você está modelando uma corrida de carros e coloca os pontos de controle do caminho nas curvas da pista. Nesse caso, ao se dirigir a uma curva, o NPC-carro precisa levar em conta as próximas curvas. Por exemplo, se uma curva for para a esquerda, ele precisa encostar à direita para “entrar na curva" no ângulo e velocidade adequados, como ilustra a Vídeo 01, caso contrário ele pode não ser eficiente no trajeto e, pior ainda, não frear adequadamente e passar direto.

Vídeo 01 – Exemplos de Curvas no jogo Forza Horizon

Fonte:VIKINGS1FAN1. **Forza Horizon Ferrari Enzo game play**. 2013. Disponível em: https://www.youtube.com/watch?v=8Rb003uGMlA
Acesso em: 20 abr. 2018.

Bom, isso é para alertá-lo da necessidade de se adequar ao jogo que está sendo desenvolvido e não se limitar a receitas preestabelecidas de como algo deve ser implementado. Como dito anteriormente, tudo depende das características do seu jogo.

No jogo que você está desenvolvendo , a versão da estratégia mais simples é novamente suficiente para modelar o comportamento dos personagens. Assim adote a “visita" aos pontos sem levar em consideração os pontos seguintes. Se houver mais de um ponto no caminho, você poderá fazer com que o jogador siga ao próximo ponto através do comportamento *Seek*, caso o ponto atual seja o último do caminho adote o comportamento *Arrival*.

Por fim, você precisará definir quando o jogador “alcançou" um ponto do caminho. Podendo ser rigoroso, como o técnico que te mandou correr tocando nas bandeirinhas, ou ser mais flexível, adotando uma distância mínima dos alvos que “tocar” neles.

Bom, você é o técnico. É você quem manda! De qualquer forma, note que quanto maior a distância mínima de contato com o ponto, mais suaves serão as curvas do jogador na sequência dos pontos. E quanto menor, menos suaves elas serão.

Passo 02 – Implementar no Unity o *Path Following*

Comece a implementação do *Path Following* pela definição do caminho. Como dito anteriormente, o C# possui uma classe própria para a estrutura de dados **Fila** chamada **Queue**. Essa é uma classe genérica, na qual se precisa detalhar o tipo de dados que ela armazena. Como o caminho é formado por pontos no campo de futebol, o que você irá armazenar são valores do tipo **Vector2**. A definição dessa classe encontra-se em um *namespace* específico. Logo, para usá-la, é necessário declarar seu uso através da palavra-chave *using*.

A interação do usuário com o NPC-jogador pode se dar da seguinte forma: **sempre que o usuário clicar sobre uma parte do campo, um novo ponto é inserido no caminho do jogador**. Assim, você precisa alterar o método **Update()** para que o clique do mouse não mais atualize a variável *goal*, mas adicione o ponto na fila. Para facilitar a visualização dos pontos presentes nessa fila, você precisa também adicionar um código que desenha uma cruz nos pontos.

O Código 03 apresenta parte da solução. Nele, além da variável **path**, contendo os pontos do caminho, tem-se também as constantes (readonly) **crossX** e **crossY**. Esses atributos são utilizados apenas para o desenho da cruz em cada ponto do caminho, através do método de **Debug DrawLine()**.

Código 03 – Script com o comportamento *Path Following*

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

public class SoccerPlayerBehavior : MonoBehaviour {
    ...
    private Queue<Vector2> path;

    private readonly Vector2 crossX = new Vector2(0, 0.1f);
    private readonly Vector2 crossY = new Vector2(0.1f, 0);
    ...

    void Start () {
        body = GetComponent<Rigidbody2D>();
        path = new Queue<Vector2>();
        AdjustSpriteRotation();
    }

    void Update () {
        if (Input.GetButtonDown("Fire1")) {
            Vector2 p = Input.mousePosition;
            path.Enqueue(Camera.main.ScreenToWorldPoint(p));
        }
        foreach (Vector2 p in path) {
            Debug.DrawLine(p - crossX, p + crossX);
            Debug.DrawLine(p - crossY, p + crossY);
        }
    }

    ...
}
```

Você já pode fazer um teste do novo código! Veja o caminho de marcas sobre o campo ao clicar sobre ele. Porém, o jogador ainda não está seguindo os pontos que você marcou. Claro! O método **FixedUpdate()** continua o mesmo e ainda não implementamos o *Path Following* propriamente dito.

No método **FixedUpdate()**, a única alteração que você precisa realizar é dizer que a força de navegação a ser utilizada, agora, é a que retorna de um novo método, que você criará, chamado **PathFollowing()**. Antes de implementar esse método, você precisa definir qual será a distância mínima para considerar que o jogador "tocou" no ponto do caminho. Pode-se colocar esse valor como um atributo público da classe para facilmente ajustá-la. Em seguida, o algoritmo será simples: se houver mais de um ponto no caminho, chame o comportamento *Seek*, se houver apenas um, o comportamento *Arrival*, caso contrário (não há pontos no caminho), retorne

um vetor nulo. Nesse entretempo, se a distância entre o jogador e o ponto do início da **Fila** for menor do que a distância mínima (ou seja, se ele “tocou" no ponto), pode-se retirar o ponto da **Fila**. O Código 04 mostra em mais detalhes a implementação aqui descrita.

Código 04 – Implementação do método *Path Following*

```
...
public float touchDist = 1.5f;
...

void FixedUpdate () {
   Vector2 steering = ObstacleAvoidance();

   if (steering == Vector2.zero) {
      steering = PathFollowing();
   }
   ...
}

Vector2 PathFollowing() {
   if (path.Count > 1) {
      if (Vector2.Distance(body.position, path.Peek()) < touchDist) {
         path.Dequeue();
      }
      goal = path.Peek();
      return Seek();
   }
   else if (path.Count == 1) {
      goal = path.Peek();
      return Arrival();
   }
   else {
      return Vector2.zero;
   }
}
...

```

Agora, você pode testar de verdade e ver se o jogador está, de fato, seguindo suas ordens. 😉

3. Outros comportamentos de navegação

O que você viu até o momento são comportamentos úteis para o jogo de futebol que está desenvolvendo. Porém, os comportamentos de navegação não se restringem a apenas esses. Existem vários outros. Em todos, o princípio segue as mesmas diretivas dos comportamentos apresentados aqui, que é a criação de forças a serem aplicadas sobre personagem... ou naves espaciais... ou carros... ou seja lá o que a IA do seu jogo está controlando.

Há comportamentos específicos, por exemplo, para simular o caminho aleatório de um personagem, para um grupo de personagens seguir um líder, para um grupo de personagens seguir uma formação de fila (um atrás do outro) ou uma barreira (um ao lado do outro). Como dito antes, tudo depende de seu jogo.

Irei, entretanto, apresentar aqui apenas a ideia geral de alguns desses outros comportamentos, a saber: *Wander*, para movimentação aleatória, e *Leader Following*, para seguir um líder.

3.1 Comportamento *Wander*

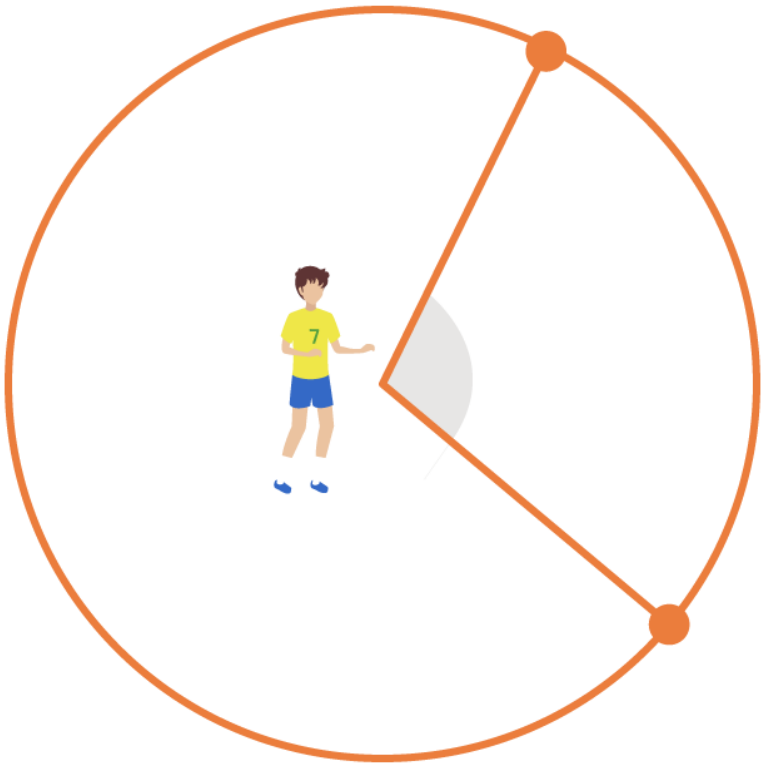
Em muitos jogos, os personagens precisam ficar “perambulando" pela área, como se estivessem distraídos, sem nada para fazer (ou melhor, à espera de ue algo aconteça). Esse movimento precisa ser realista o suficiente para dar a impressão que o personagem está apenas “caminhando sem destino certo".

Uma forma de se fazer isso, na verdade a mais simples, é gerar um ponto aleatório sobre o terreno e aplicar o comportamento *Seek* em direção a esse ponto. Porém, antes de o personagem alcançar o ponto, você sorteia novamente um outro ponto, e assim por diante. O resultado é que o personagem vai ficar perambulando pelo ambiente. O problema, porém, é que essa estratégia pode não gerar um comportamento muito realista, caso o ponto sorteado encontre-se no lado oposto ao que o personagem estava indo. Nesse caso, ele vai precisar dar um giro de 180 graus, o que não é o que se espera de alguém apenas “passeando”.

Uma outra estratégia é ficar sorteando pequenas variações de ângulo no vetor de velocidade do personagem. Dessa forma, ele não se vira de repente. Se houver alguma alteração, será suave.

Bom... para garantir ainda mais a suavidade nos movimentos do personagem, o ângulo aleatório pode não ser aplicado diretamente sobre a velocidade do personagem, mas em um círculo “imaginário" em frente dele, como ilustra a Figura 09. Nessa abordagem, o ângulo sobre o círculo define o alvo para onde a força de navegação deve “puxar” o personagem.

Figura 09 - Representação do ângulo de variação no comportamento *Wander*



3.2 Comportamento *Leader Following*

Imagine que você esteja desenvolvendo um jogo de tiro, no qual uma equipe precisa seguir seu líder. O líder deve, então, seguir na frente, normalmente com um alvo como objetivo, e os demais devem segui-lo. Porém, não dá para todos irem

“logo atrás" dele, senão haverá gente se empurrando e dando cotovelada para todo lado. É preciso seguir a uma distância que dê impressão de que o personagem faz parte do grupo, mas que eles não fiquem se “amontoando" uns sobre os outros.

Dessa forma, não dá simplesmente para usar os comportamentos *Seek, Arrival* ou *Pursuit* que você viu na última aula (e nas atividades). É preciso adaptá-los (pelo menos um deles) e combinar com outros comportamentos para termos algo mais realista.

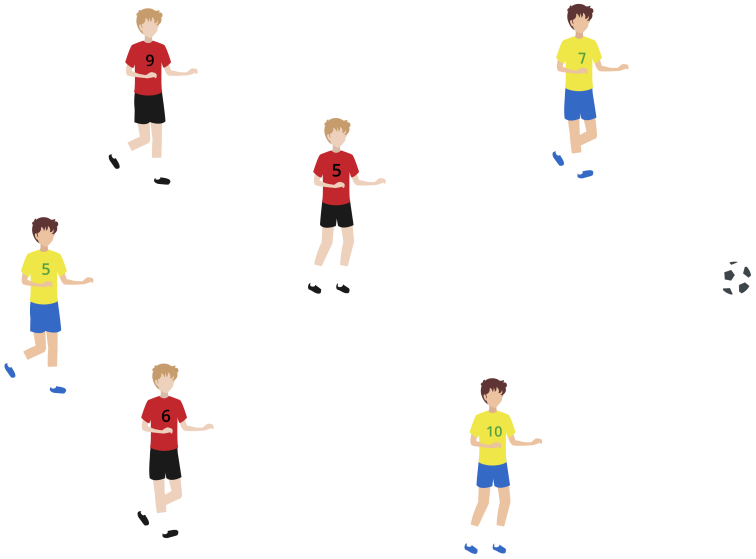
Para implementar o comportamento de *Leader Following*, você pode utilizar a combinação (e adaptação) dos seguintes comportamentos:

- *Arrival*: para ir rapidamente na direção do líder, mas atenuar a velocidade quando estiver chegando perto, chegando a parar, se for o caso;
- *Evade*: para fazer com que o personagem saia rapidamente caso esteja na frente do líder;
- *Separation*: para evitar sobreposição dos membros da equipe.

O uso de cada um desses comportamentos, nos seus respectivos casos, dá a ilusão de que todos os membros da equipe estão seguindo o líder.

Vale salientar que o ponto de atração para o comportamento *Arrival* não deve ser necessariamente a própria posição do líder, caso contrário os membros da equipe podem se amontoar sobre ele. O ideal é que o ponto de atração esteja ligeiramente “atrás" do líder. Pode-se calcular esse ponto, invertendo a velocidade do líder (o vetor ficaria no sentido contrário) e definindo uma distância fixa para o tamanho do vetor resultante. A Figura 10 mostra uma ilustração desse comportamento.

Figura 10 - Representação do ponto de atração no comportamento *Leader Following*



E com isso você viu alguns comportamentos mais interessantes para colocar nos jogadores do seu time! Pelo menos se um caminhão aparecer no meio do jogo (ah, os bons “macetes" dos videogames), o seu jogador vai conseguir desviar dele antes de virar panqueca! Mas até aqui os nossos jogadores estão sempre seguros de suas decisões, e sempre sabem para que lado vão... o lado que você mandar. 😁

Na próxima aula, vou lhe mostrar como colocar um pouco de “dúvida" nos jogadores (ou em você 😱), quando você estudar o conceito de probabilidades!

Até lá!

Resumo

Como você pôde observar, os comportamentos de navegação são elementos importantes na construção de uma movimentação realista nos personagens de seu jogo. Os comportamentos são simples de implementar, bastando conhecer e saber aplicar a matemática vetorial. Pode-se criar também comportamentos a seu bel prazer, sem estar limitado a regras específicas. Apresentei aqui alguns como fontes de inspiração, mas você pode definir sua própria forma de lidar com as forças que atuam sobre os jogadores.

O que você aprendeu nessas duas aulas sobre comportamentos de navegação será útil na criação de novas formas de movimentação.

Resumidamente, nessa aula, você viu:

1. Como fazer com que os personagens desviem de obstáculos, através do comportamento *Obstacle Avoidance*;
2. Como fazer com que os personagens sigam um roteiro preestabelecido de pontos, ou seja, que sigam um caminho predefinido, através do comportamento *Path Following*;
3. A ideia geral de como desenvolver comportamentos para mover-se de forma aleatória, através do comportamento *Wander*, e como seguir um líder, através do *Leader Following*.

Além disso, você viu também os conceitos de estruturas de dados, em que duas foram apresentadas:

- **Fila**: estrutura na qual os novos elementos são inseridos sempre no final e a remoção sempre ocorre com os primeiros. As operações sobre esse tipo de estrutura também são conhecidas como operações do tipo FIFO (do inglês, *First In, First Out*. Ou seja, os primeiros a entrar serão os primeiros a sair).
- **Pilha**: estrutura na qual tanto a inserção quanto a remoção dos elementos ocorrem no início (ou topo) da estrutura. As operações sobre esse tipo de estrutura são conhecidas como operações do

tipo LIFO (do inglês, *Last In, First Out*. Ou seja, os últimos a entrar serão os primeiros a sair).

Bom...

Sobre comportamentos de navegação paro por aqui. Porém, os jogadores ainda estão muito simples. Eles possuem, por enquanto, apenas um comportamento (seguir as forças de navegação). Para que eles fiquem mais “inteligentes" você precisa dotá-los da capacidade de adaptação, para que os jogadores tenham comportamentos diferentes dependendo das condições e situações do jogo. Como diria Charles Darwin, sobrevive aquele que mais se adapta!

Veremos isso na próxima aula.

Até lá! 😉

Referências

MILLINGTON, Ian; FUNGE, John. **Artificial Intelligence for Games**. Morgan Kaufmann Eds. 2ª edição. 2009.

REYNOLDS, Craig. **Steering behaviors for autonomous characters**. Online version, 1999. Disponível em http://www.red3d.com/cwr/boids/. Acesso em: 05 abr. 2018.

BELIVACQUA, Fernando. **Understanding Steering Behaviors**. Online. 2012. Disponível em http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732. Acesso em: 05 abr. 2018.