Inteligência Artificial para Jogos

Aula 02 - Comportamentos de navegação

Apresentação da aula

Na aula passada, foi apresentada uma visão geral acerca da temática da disciplina, para você ter uma ideia do que ainda virá pela frente. Agora, você colocará em prática vários daqueles conceitos, então prepare-se para finalmente pôr a “mão na massa”! Para isso, selecionei um exemplo de jogo, dando preferência a um que tinha um cenário no qual pudesse abordar várias ideias de IA aplicadas em jogos.

Você sabe qual selecionei? Claro que não! rsrs Vou te dar uma pista...

Você já pensou em fazer um Neymar virtual, capaz de lances incríveis, de dar uma "saiota" (No jargão futebolístico brasileiro, é o ato de passar a bola entre as pernas de um adversário.) ou um "banho de cuia" (Termo futebolístico relativo a quando um jogador dribla o outro passando a bola por cima deste.) no adversário, fazer aquele lançamento no pé de Mbappé ou Cavanni? É um desafio e tanto, não é?

Isso mesmo que você deve estar pensando: selecionei um jogo de futebol. Construir um jogo de futebol pode ser o primeiro passo a ser dado para você ajudar a resolver grandes problemas da comunidade de IA.

Vamos, então, realizar o primeiro passo!

Objetivos

Compreender o que são comportamentos de navegação;

Conhecer os comportamentos de navegação *Seek* e *Arrival*.

Aplicar e implementar os comportamentos *Seek* e *Arrival* na ferramenta Unity.

1. Dando os primeiros passos

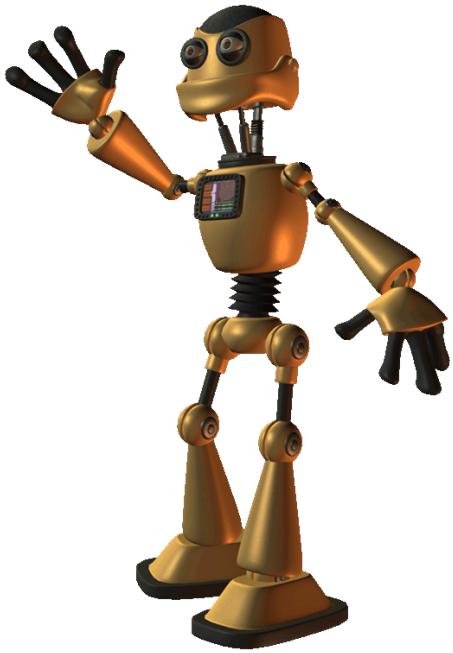
Imagine que você é um técnico de futebol e tem de treinar sua equipe. O que você precisa fazer?

Ops, esqueci de dizer que sua equipe é formada por personagens virtuais, que ainda não se movimentam e que precisarão das instruções a serem dadas por você (programador) para realizarem as tarefas. Para ficar mais claro, vou fazer uma analogia com o desenvolvimento motor de uma criança.

Quando se começa a andar, primeiro se rasteja, depois engatinha, cai, levanta-se e assim por diante. Certamente você não lembra, mas precisou passar por todo esse processo. Percebeu que não é tão fácil?

Agora, imagine um jogo de futebol formado por robôs bípedes que ainda apresentam uma certa dificuldade de se manterem em pé ao andar. Como eles conseguirão correr atrás de uma bola ou fazer jogo de corpo sobre o adversário?

Figura 01 - Robô em movimento



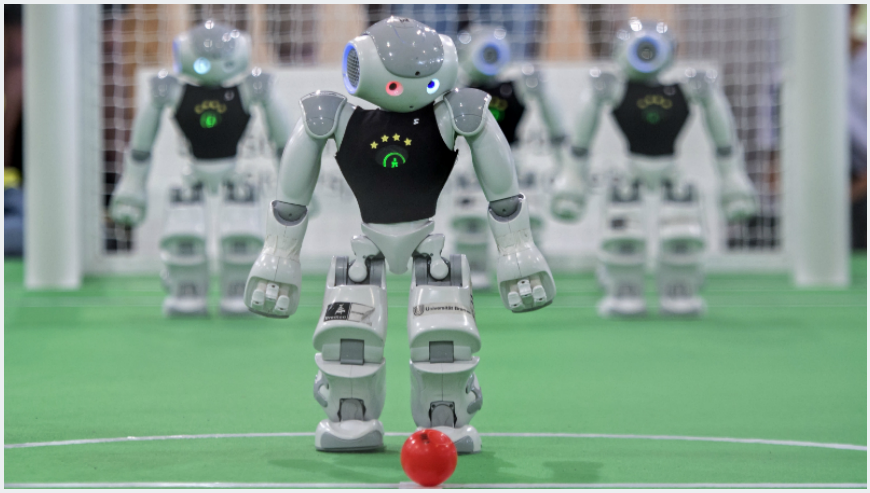
Fonte: PABLO, Juan. Disponível em: <http://gifanimadosyfrasescortas.blogspot.com.br/2013/11/gif-animados-transparentes-variado.html>. Acesso em: 11 abr. 2018.

Bem, esse nem tanto!

Enfim, não é uma tarefa simples, mas desde 1997 um grupo de cientistas japoneses da comunidade de IA idealiza uma competição anual de futebol robótico, na qual equipes de vários países criam os algoritmos de IA de robôs e estes competem entre si, na chamada RoboCup.

Curiosidade

Já pensou se a seleção do Brasil disputasse um jogo com uma seleção de robôs?



Em breve, isso será realidade! É que um dos grandes desafios da comunidade de pesquisadores de robótica e inteligência artificial nas próximas décadas é **fazer com que em 2050 robôs humanoides possam jogar e ganhar uma partida de futebol contra a equipe vencedora da Copa do Mundo**. Para mais detalhes, acesse aqui.

Na RoboCup, há uma liga que não é beeeem de robôs. É uma simulação de robôs em um ambiente virtual, realizada por meio de algumas técnicas que são empregadas nos jogos digitais e vice-versa. Ou seja, é um jogo! No entanto, o objetivo dele não é entreter o usuário (jogador humano), e sim, verificar quem (qual equipe) implementou a melhor IA.

Em uma simulação (jogo digital), o processo de aprendizado é mais fácil e rápido, porque não envolve todas as questões físicas impostas pela gravidade. Desse modo, é possível fazer rapidamente um jogador virtual se deslocar de um lado para o outro no campo de futebol.

Entretanto, o problema aparece quando há vários jogadores correndo no campo e é necessário evitar que eles se atropelem mutuamente. **Como coordenar o deslocamento de vários jogadores em campo?** Caso o algoritmo de IA seja complexo demais, levando muito tempo para coordenar quem vai para onde, isso pode causar *lag* no jogo... e nada é mais chato que *lag*. Precisa-se, então, de um mecanismo simples para resolver esse problema.

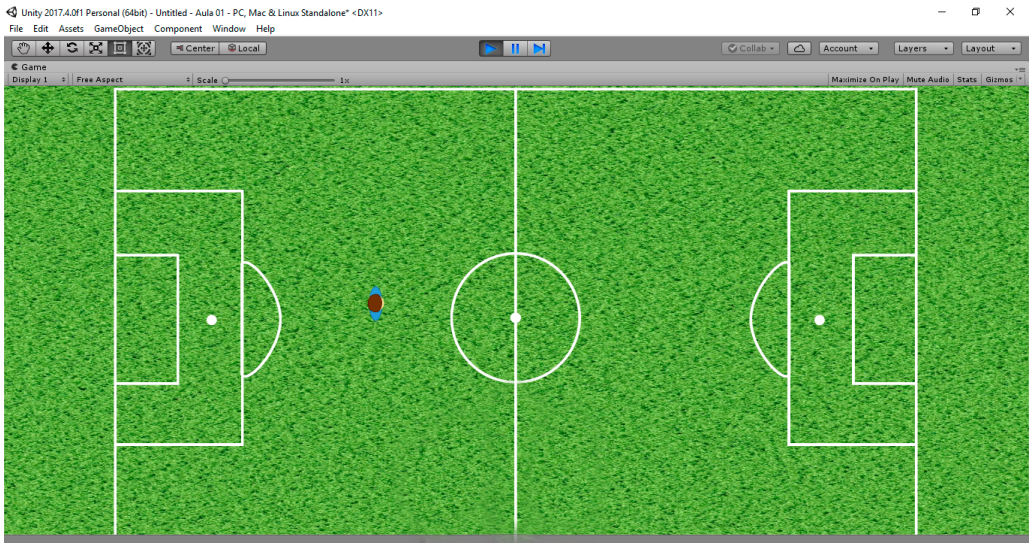
Você verá que o conhecimento adquirido nas disciplinas Física e Matemática poderá ajudar bastante. Mas comece com algo simples, lembre-se de que os jogadores precisarão aprender o básico: movimentar-se em campo, antes mesmo de saber jogar a bola de um lado para o outro. Então, já tem a estratégia do jogo pronta? Não? Vá pensando, pois você começará o processo de construção a partir da próxima seção.

1.1 Cenário inicial

Passo 01 – Construir o campo de futebol

Para começar, você precisa construir o cenário do jogo (que será em 2D) no Unity. Crie, então, o campo de futebol e, por enquanto, apenas um jogador, como ilustra a Figura 02.

Figura 02 - Cenário inicial do jogo de futebol



Os *assets* dessa etapa do cenário podem ser encontrados AQUI. Nesse link, você encontrará as imagens playerA.png, playerB.png, grass.png, ball.png e lines.png, algumas serão utilizadas para montar o cenário inicial desse jogo.

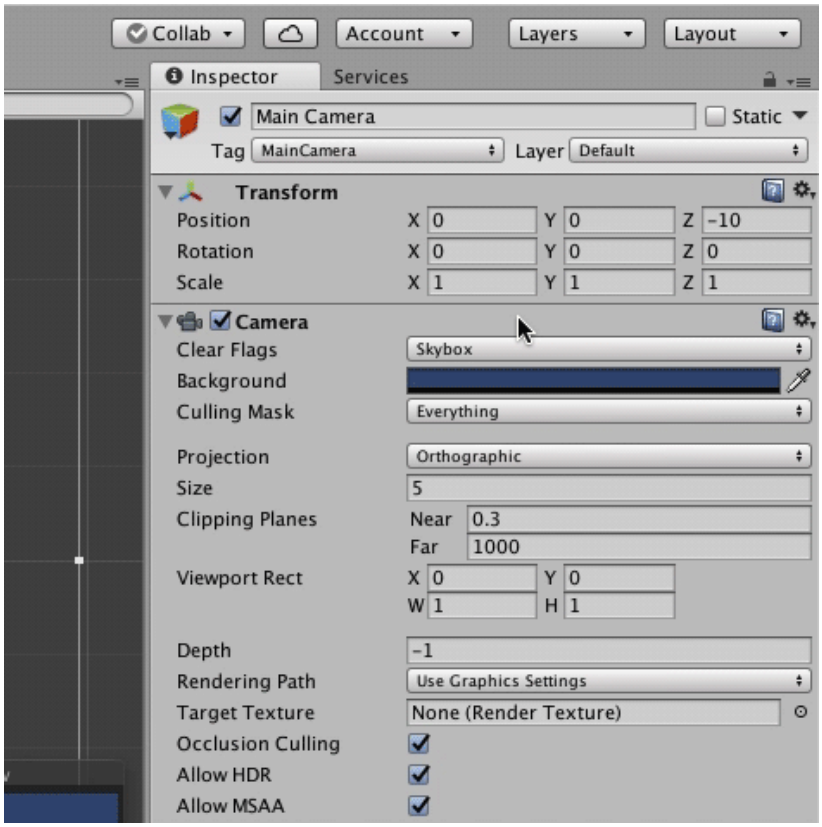
Agora, crie um novo projeto Unity voltado para jogos 2D e organize sua pasta de *Assets* definindo as seguintes subpastas:

- *Scenes:* onde as cenas serão salvas;
- *Sprites:* onde as imagens serão guardadas;
- *Scripts:* onde os códigos de comportamento serão salvos.

Em seguida, copie as imagens do link para a pasta *Sprites* e crie um objeto do jogo. Para isso, você pode utilizar *drag-and-drop*. Denomine o objeto de "player A" e use o componente *Sprite Renderer* para associar a imagem playerA.png a esse objeto. Faça o mesmo para criar dois outros objetos do jogo: as linhas do campo (lines.png) e o gramado (grass.png), respectivamente.

Ao criar esses novos objetos, pode ser que eles fiquem sobre o "player A". Para ter certeza de que os jogadores estarão sempre na frente (sobre) do campo de futebol, você deve criar uma camada de ordenação (*sorting layer*) chamada **Terrain**, a qual associará os objetos *grass* e *lines*. Ao definir que **Terrain** venha antes da camada padrão (*Default*), você garantirá que os objetos *grass* e *lines* sejam sempre desenhados antes dos jogadores. Lembre-se de ajustar a camada que você acabou de criar para vir antes da camada *Default*, conforme exibe a Figura 03.

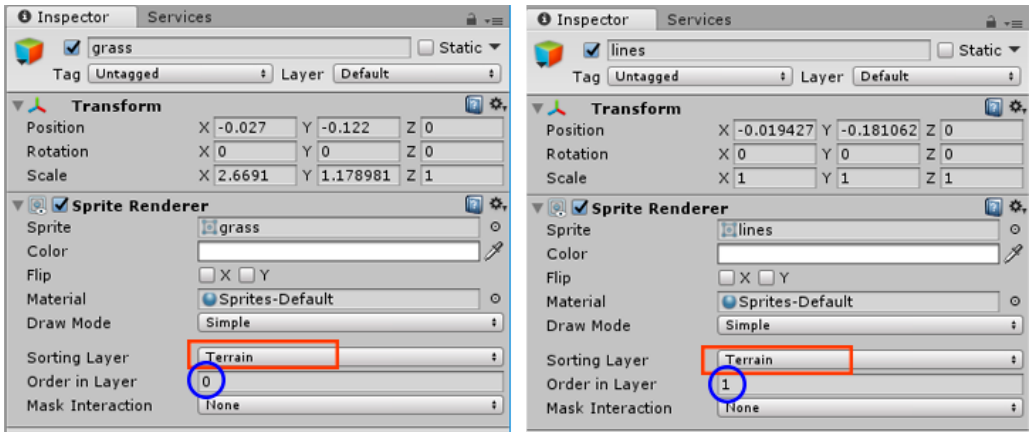
Figura 03 - Definição de uma camada (layer) específica para os elementos do terreno



Após a definição da camada **Terrain**, associe os objetos *grass* e *lines* a ela, especificando a ordem de ***grass*** como **0** e ***lines*** como **1**. Assim, *lines* ficará sempre sobre o campo.

A Figura 04 mostra como as informações devem estar.

Figura 04 - Associação dos objetos na camada de ordenação e a definição de sua ordem na camada



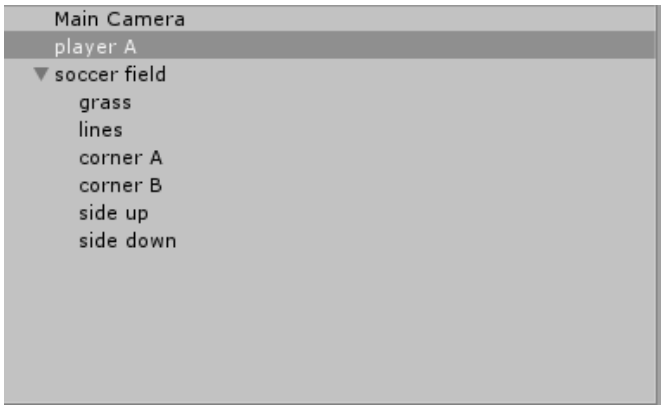
No futebol, há regras específicas para quando a bola sai na lateral ou na linha de fundo, então, para seguir essas regras, insira sensores no cenário, de forma a detectar quando algum elemento sai do campo. Em seguida, crie 4 objetos vazios (Create Empty) para representar as duas laterais e

as duas linhas de fundo e insira o componente ***BoxCollider2D*** em cada um deles. Ajuste os valores de tamanho (Size) e deslocamento (Offset), de maneira que cada um dos sensores capture seu respectivo lado (lateral e escanteio).

> **Atenção**
>
> Lembre-se, também, de especificar que esses *Colliders* são gatilhos de eventos (Is Trigger), caso contrário, eles agirão como objetos de colisão e outros objetos não poderão passar por eles.

Por fim, para melhorar a organização desse cenário, crie um objeto vazio e o renomeie para ***soccer field***. Esse objeto servirá para agrupar os elementos do campo de futebol: o gramado (*grass*), as linhas de delimitação do campo (*lines*) e os sensores de lateral e escanteio. Mova-os, arrastando-os com o mouse, de forma obter a hierarquia apresentada na Figura 05.

Figura 05 - Hierarquia dos objetos criados até o momento



1.2 Movimentando o jogador

Passo 02 – Criando o jogador

Agora que você tem um cenário montado, comece a pensar em como fazer o jogador se deslocar dentro do campo. Sinta-se como o técnico que manda os jogadores se aquecerem, correndo de um lado para o outro. Porém, a ordem a ser dada deve ser no formato de algoritmo. 😉

A melhor forma de começar é definindo um NPC simples, que se desloque de um ponto para o outro. Como você é o técnico, ele precisa te obedecer seguindo os seus comandos e, para isso, você terá de clicar sobre uma posição do campo. Sendo assim, escreva um *script* para definir esse comportamento.

Comece criando uma classe que herda de *MonoBehavior* *(Opa! Está precisando do Google Tradutor? Nada disso, volte à Aula 07 da disciplina Programação Orientada a Objetos (POO) e à Aula 03 da disciplina Desenvolvimento de Motores I)*, especificando nela dois atributos públicos: um que armazena a posição para a qual o NPC deve ir, chamada *goal* *(do inglês, objetivo)*, e outro denominado *speed* *(do inglês, velocidade)*, que especifica a velocidade com a qual o NPC consegue se deslocar da posição em que ele se encontra até seu objetivo. O *goal* precisa ser um atributo da classe *Vector2*, pois ele conterá as coordenadas x e y, enquanto *speed* pode ser um valor real (float).

No Unity, uma das formas de definir os movimentos de um NPC, principalmente quando envolve velocidade, é atribuindo-lhe um corpo físico. Para isso, insira o componente ***Rigidbody2D*** no NPC. Esse componente, vai servir de quebra, ainda, para dar um "chega pra lá" nos outros jogadores. 😁

Como você está vendo os jogadores "de cima", altere a gravidade do componente ***Rigidbody2D*** para 0, caso contrário, o jogador será puxado pela gravidade, "caindo na tela". Além dos elementos já descritos, acrescente os métodos *Start*(), *AdjustSpriteRotation*(), *GoTowardsGoal*(), *FixedUpdate*() e *Update*() no *script*, conforme o exemplo a seguir, a fim de que o NPC se desloque para onde o técnico (você) clicar.

Código 01 – Script inicial do jogador de futebol

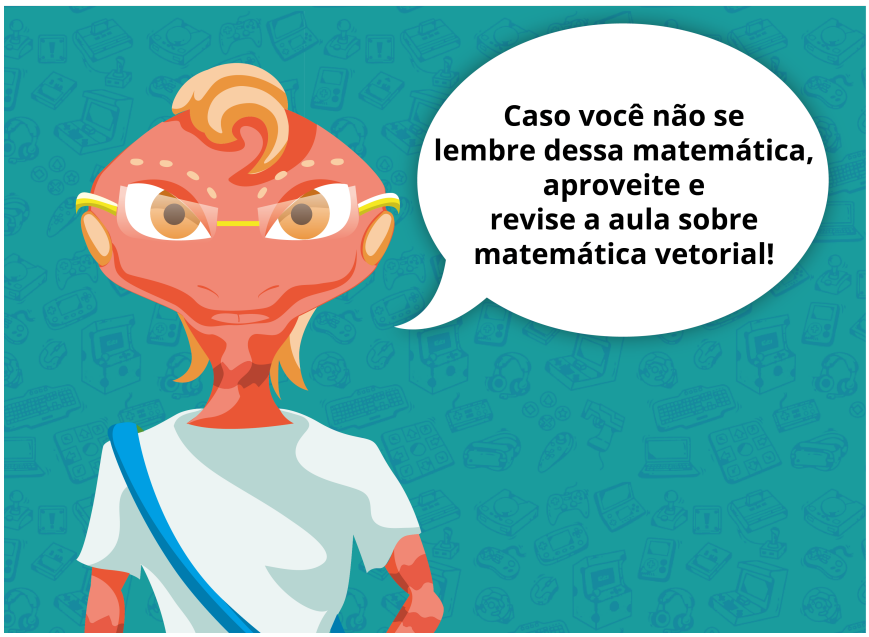
```
using UnityEngine;

public class SoccerPlayerBehavior : MonoBehaviour {

   public Vector2 goal;
   public float   speed;

   private Rigidbody2D body;

   void Start () {
      body = GetComponent<Rigidbody2D>();
      GoTowardsGoal();
      AdjustSpriteRotation ();
   }

   void GoTowardsGoal() {
      Vector2 direction = goal - body.position;
      body.velocity = direction.normalized * speed;
   }

   void AdjustSpriteRotation() {
      float angle = Vector2.Angle(body.velocity, Vector2.right);
      body.rotation = body.velocity.y >= 0 ? angle : 360 - angle;
   }

   void FixedUpdate () {
      GoTowardsGoal();
   }

   void Update() {
      if (Input.GetButtonDown("Fire1")) {
         goal = (Vector2)
                  Camera.main.ScreenToWorldPoint(Input.mousePosition);
         GoTowardsGoal();
         AdjustSpriteRotation();
      }
   }
}

```

Ao ser instanciado, o objeto associado ao *script* inicia (no método *Start*()) armazenando o componente de corpo rígido ***Rigidbody2D*** para posterior uso, calcula a direção a ser seguida até um ponto-objetivo (*GoTowardsGoal*()) e gira a imagem do *Sprite* na direção em que o objeto se encontra (*AdjustSpriteRotation*()).

O método *GoTowardsGoal*() calculará qual a direção que o NPC deve girar para ir ao seu objetivo, através da subtração dos vetores, como ilustra a Figura 06.

Figura 06 - Cálculo da direção a ser tomada pelo NPC

Como você pode ver na Figura 06, o NPC se desloca no sentido apontado pela seta azul. Quando o usuário clica em um ponto da tela, gera um novo objetivo para ele alcançar. Para que a transição da trajetória do novo objeto seja feita de forma mais “natural”, é necessário calcular um vetor que indique o deslocamento da direção atual para a desejada. Logo, o deslocamento é ajustado para a direção que aponta a seta verde, obtida pela subtração do vetor laranja pelo azul.



Em seguida, o vetor resultante dessa subtração é redimensionado, de modo a especificar no *speed* a magnitude da velocidade. Faça isso, multiplicando a velocidade pela versão normalizada (que tem tamanho 1) da direção. Atribuindo ao vetor resultante a velocidade do componente *body*,

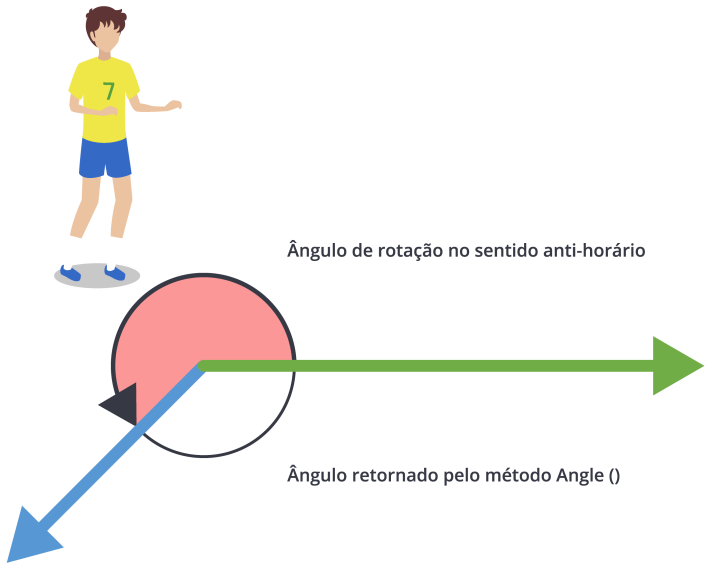
o NPC se deslocará na velocidade desejada. Na verdade, o movimento do NPC é calculado através do método de Euler, que define de forma simples uma aproximação do cálculo da equação de movimento, após uma sequência de atualizações, usando a fórmula a seguir.

Algoritmo 01 – Método de Euler para aproximação do deslocamento

```
posição = posição + velocidade
```

O método *AdjustSpriteRotation*() tem como objetivo girar o *sprite* do NPC na direção já calculada em *GoTowardsGoal*(). Para fazer isso, calcula-se o ângulo entre o eixo X (definido pela constante *Vector2.right*) e a direção que o NPC se encontra, usando a função *Angle*(). Essa função retorna sempre o menor ângulo entre os dois vetores (o ângulo agudo entre eles). Como no ângulo retornado não há informação a respeito do sentido ser anti-horário ou horário e como o *Sprite* precisa ser girado no sentido anti-horário, há necessidade de verificar se o valor da coordenada Y da direção é negativo. Caso seja, gire no sentido anti-horário com mais que 180 graus, fazendo a diferença do ângulo retornado com 360 graus, para conseguir esse resultado.

Figura 07 - Cálculo do ângulo de rotação



Por fim, existem dois métodos de atualização: *UpdateFixed*() e *Update*(). O primeiro serve para atualizar os elementos do jogo em um intervalo fixo de tempo, ideal para simulações físicas e para que os movimentos do jogo não fiquem lentos. O segundo, por sua vez, é chamado sempre que um novo *frame* de visualização deve ser renderizado. Assim, insira a atualização do movimento do NPC em *UpdateFixed*() e a atualização do objetivo (*goal*) para quando o usuário clicar em um ponto do campo em *Update*(). Nesse último, sempre que o botão *Fire1* for pressionado (isso equivale ao botão esquerdo do mouse), o *script* atualiza as coordenadas de seu objetivo e vai em direção a ele.

Ótimo! Você já fez o *script* de comportamento do jogador. Mas onde está a inteligência artificial aí? Bom... por enquanto, em lugar nenhum. 😁 Você vai começar a inseri-la no momento em que quiser adicionar mais realismo à cena.

Por exemplo, imagine que há dois jogadores seguindo o mesmo *script*. Isso significa que quando você clicar na tela, os dois tentarão ir para o mesmo ponto e ficarão um sobre o outro. Aí o juizão vai marcar falta! 😁

Uma solução é adicionar um componente para tratar das colisões dos objetos. Pode-se, então, inserir no objeto ou o componente ***CircleCollider2D*** ou o ***BoxCollider2D***. As opções padrões (*default*) já fazem a configuração necessária para você. Procure fazer isso e você vai ver que não haverá mais sobreposição de jogadores.

Porém, experimente, agora, colocar mais jogadores, cada um com um *goal* diferente. Você perceberá o quanto esses jogadores são... limitados. 😁 O comportamento deles não é nada realista, porque eles tentam ir para os seus destinos por meio de uma reta em direção ao alvo, mesmo que haja obstáculos (outros jogadores) a sua frente e que ninguém em um "estado normal" aja assim. Desse modo, você precisa adaptar os jogadores para que eles saibam **como desviar de obstáculos à frente..**

Você verá como fazer isso nesta... Não! Na próxima aula. rsrs Vou começar com um pouco de teoria, descrevendo como surgiu o que é chamado atualmente de "comportamentos de navegação" ou, em inglês, *steering behaviors*.

2. Comportamentos de navegação (*steering behaviors*)

Os comportamentos de navegação são técnicas de IA para Jogos usadas para definir o movimento de personagens de forma realista (você encontrará muitas referências, mesmo as em português, mencionando *steering behavior*), basicamente por meio das leis da mecânica (Física). Essas técnicas não se utilizam de algoritmos complexos, envolvendo planejamento do deslocamento ou algo que requeira muito tempo de execução e cálculos. Não, nada disso! Elas simplesmente usam o conceito de força e o de combinação de forças para criar movimentos aparentemente complexos.

As técnicas de IA para Jogos começaram a ser usadas por Craig Reynolds, em 1986, para simular o movimento de um grupo de animais, como de peixes ou de pássaros. Preste atenção nos vídeos a seguir!

Vídeo 01 – Revoada de pássaros no estado do RN

Fonte: LEMOS FILHO, RUBENS. **No RN, o lindo balé de arribaçã**. 2018. Disponível em: https://www.youtube.com/watch?v=umJMYfyAi6c
Acesso em: 10 abr. 2018.

Vídeo 02 – Cardume em Fernando de Noronha, PE

Fonte: Viegas, Roberta. **O cardume**. Disponível em: https://www.youtube.com/watch?v=LJC9BlPhdT0
Acesso em: 10 abr. 2018.

Você vai ver que há uma sincronização dos milhares de pássaros e dos peixes. Há alguma entidade controlando eles? Dizendo quem deve ir mais para a direita, mais para a esquerda, quem sobe, quem desce? Não. Há, por trás desse movimento gracioso que você viu no vídeo, a chamada **inteligência coletiva**. O movimento dos pássaros emerge das decisões e colaborações individuais de cada pássaro. Quando um deles vê um pouco mais de espaço à direita, ele vai para direita, mas se ele se afastar muito do bando, ele volta. Esse movimento já faz com que o pássaro à esquerda mude seu trajeto e assim por diante. O mesmo ocorre com o cardume.

Inteligência coletiva é uma área multidisciplinar, que envolve estudos da psicologia, sociologia, filosofia e... também da computação. Na computação, em específico em IA, há estratégias para simular (apesar das simplificações) um tipo de inteligência coletiva, que é chamado de **inteligência de enxames** (do inglês *swarm intelligence*). Esse termo foi definido assim porque as estratégias utilizadas são normalmente inspiradas nos modelos de movimentação do coletivo de abelhas (enxames), de pássaros (bando) ou peixes (cardumes), aplicados a entidades computacionais simples, chamados de agentes ou ***boids*** (pequenos robôs).

No modelo de Reynolds, os ***boids*** são governados por apenas três forças:

- **Separação**: fazendo com que o *boid* siga para uma direção menos "tumultuada" de outros *boids*;
- **Alinhamento**: fazendo com que o *boid* siga a mesma direção que os *boids* ao seu redor;
- **Coesão**: fazendo com que o *boid* tenda a não se afastar de seu bando.

Agora, dê uma olhada na simulação a seguir. É uma simulação de 10.000 *boids*. Além das forças citadas anteriormente, há também forças para evitar que eles colidam com obstáculos (o cenário é em 3D e dá para ver alguns *boids* sobre o obstáculo, mas não é porque eles "entraram" dentro do obstáculo, eles estão acima dele) e que fujam de um predador voando ao redor deles.

Vídeo 03 – Animação com mais de 10.000 criaturas simulando revoada

Fonte: SEUNGWOO, Ji. **Steering Behavior 3D, Flocking**. Disponível em: https://www.youtube.com/watch?v=wWfmrvjd9jY
Acesso em: 10 abr. 2018.

Imagine se você fosse calcular os planos de voo de cada um dos 10.000 boids! Ia dar *lag*, com certeza!

Bom, com essa técnica Craig Reynolds trabalhou não só em jogos, mas também fazendo animação procedural para filmes, como no filme *Batman: o Retorno*, de 1992, no qual um grupo de morcegos é controlado por *scripts* usando a técnica citada. Desde então, a técnica (Figura 08) foi usada em vários filmes, como *Batman Begins*, quando Bruce Wayne enfrenta seu medo no meio de uma revoada de morcegos. O comportamento desses morcegos foi criado usando as técnicas de comportamento de navegação.

Figura 08 - Cena do filme "Batman Begins"

Fonte: THE DISCUSSION. Disponível em:<http://thediscussion.net/main/9-things-not-all-fans-knew-about-batman-begins/>. Acesso em: 04 abr. 2018.

Essa técnica mostrou que é possível criar comportamentos complexos apenas aplicando forças de atração e/ou de repulsão. Assim, vários comportamentos já foram definidos apenas aplicando essas forças ou variações delas. A partir de agora, você estudará os comportamentos mais simples, como a busca por um alvo (*Seek*) e o comportamento de chegada (*Arrival*). Eles serão essenciais na construção de comportamentos mais complexos, que serão abordados na próxima aula.

2.1 Comportamento *Seek* (busca por um alvo)

Vou começar explicando a matemática dos comportamentos de navegação através de um comportamento específico: *Seek*. Esse comportamento é normalmente atribuído aos personagens quando se quer que eles tenham movimentos suaves em direção a uma posição do cenário. Você já percebeu que o jogador criado na seção anterior é muito "robótico"? Por exemplo, quando você clica em uma posição oposta à direção que ele está indo, ele ignora todas as leis da inércia e simplesmente muda de direção conservando a mesma velocidade. Um jogador de verdade não teria capacidade para isso (alguém teria de mudar as Leis da Física para tanto). O mais natural seria que o jogador fosse reduzindo sua velocidade para poder mudar de direção e depois seguir em direção ao clique do usuário.

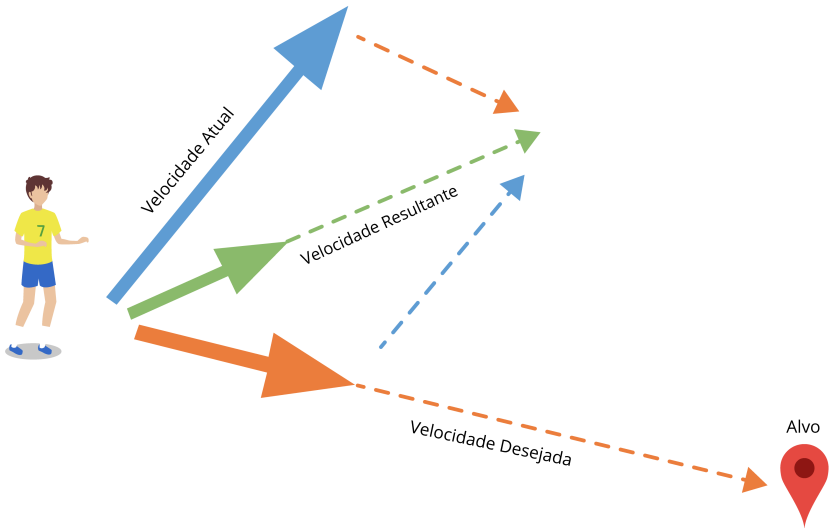
Pois bem, você vai fazer isso. 😁

As forças dos comportamentos de navegação que citei podem ser representadas por vetores 2D ou 3D, dependendo do tipo de cenário com que você estiver trabalhando. Nesse caso, trata-se de um jogo 2D e, portanto, serão utilizados vetores bidimensionais.

Como na matemática vetorial será utilizada (soma, subtração de vetores), é interessante que a posição do personagem também esteja representada com um vetor. No Unity, isso já está implícito, uma vez que todos os objetos do jogo possuem uma posição definida através de um vetor. Detalhe: essa posição é, por padrão, representada com um vetor 3D, mas por enquanto basta ignorar o eixo Z e os objetos já estarão em posições definidas através de vetores 2D.

O que resta a fazer agora é calcular a velocidade adequada do personagem, que sofrerá influência das forças de atração e/ou repulsão que você desejar. O que você precisa é aplicar uma força à velocidade atual na direção do alvo (velocidade desejada). A soma dos dois vetores de velocidade resultará em uma nova velocidade, denominada **resultante**, que deve ser limitada a um valor máximo para evitar que a mesma aumente indefinidamente ao longo do tempo (essa soma ocorre a cada novo frame, lembra?). A Figura 09 ilustra graficamente o cálculo da velocidade resultante. Ela será a nova velocidade do personagem..

Figura 09 - Movimentação baseada na aplicação de forças (vetores)



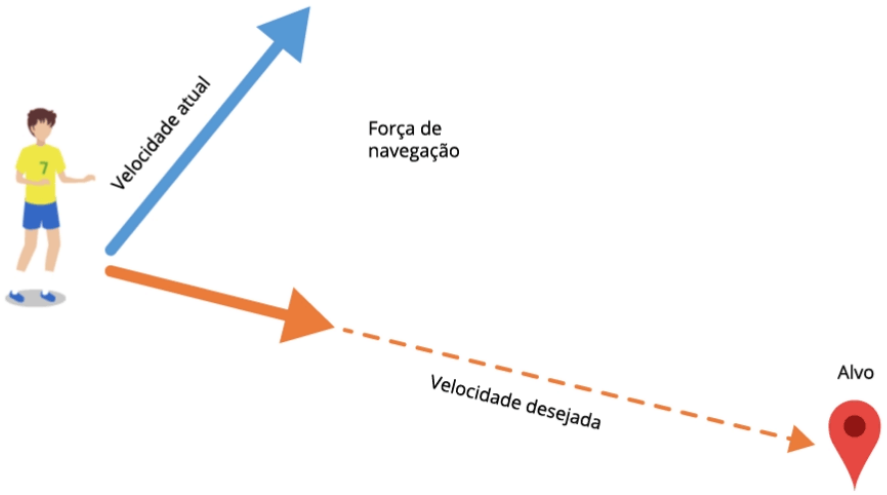
Agora você deve estar se questionando: "*mas a velocidade resultante não direciona o personagem ao alvo*"?. De fato, não. Porém, se você aplicar essa estratégia a cada *frame*, vai ter um movimento suave da posição e velocidade atual até o alvo, baseado no método de Euler (apresentado na seção anterior), conforme ilustra a Figura 10.

Figura 10 - Atualização contínua da velocidade resultante

A atualização contínua da velocidade fará com que o personagem se desloque paulatinamente ao alvo. O cálculo da velocidade resultante tal como foi descrito é intuitivo e fácil de compreender. Porém, demora para convergir. Ou seja, o personagem pode ficar rodando em torno do alvo e

nunca o alcançar. Então, existe uma outra abordagem para remover esse problema. Aplique uma força sobre a velocidade atual, chamada **força de navegação (*steering*)**, que vai "empurrar" o personagem de forma que a velocidade atual vá na direção à velocidade desejada, conforme ilustra a Figura 11.

Figura 11 - A força de navegação (steering force) "empurrando" o personagem para a direção desejada



O algoritmo a ser seguido, baseado no que foi explicado até o momento, encontra-se no pseudocódigo a seguir. Primeiro, calcule a velocidade desejada, através da subtração dos vetores alvo e posição (linha 1), e depois transforme essa velocidade em um vetor de tamanho igual à velocidade máxima. Esse cálculo é feito normalizando a velocidade calculada (o vetor fica com tamanho de 1) e depois multiplicando-a com o valor de velocidade máxima (linha 2).

Com o resultado da velocidade desejada, calcule o vetor de força de navegação através da subtração dos vetores de velocidade desejada e atual (linha 3). Agora, você precisa truncar a força calculada de forma que o tamanho do vetor não exceda um certo limite (força máxima) (linha 4). Isso vai permitir uma certa suavidade nos movimentos. Aplique, então, a força de navegação à velocidade atual, limitando-a igualmente a um tamanho máximo (linhas 5 e 6). Por fim, atualize a posição do personagem com sua nova velocidade (linha 7).

Algoritmo 02 – Comportamento *Seek*

```
1: veloc_desejada = alvo - posição
2: veloc_desejada = normaliza(veloc_desejada) * veloc_máxima
3: força_nav = veloc_desejada – velocidade
4: força_nav = limita(força_nav, força_máxima)
```

```
5: nova_veloc = velocidade + força_nav
6: velocidade = limita(nova_veloc, veloc_máxima)
7: posição = posição + velocidade
```

Se você quiser aprimorar ainda mais esse mecanismo, considerando a massa do personagem para simular a inércia dos jogadores mais "gordinhos", inclua um novo cálculo na força de navegação, inserindo a instrução a seguir depois da linha 4. Ou seja, a força de navegação a ser aplicada é inversamente proporcional à massa do jogador. Quanto mais "gordinho", menor a força a ser aplicada.

Algoritmo 03 – Introdução de inércia no comportamento Seek

```
5: força_nav = força_nav / massa
```

Implementar esse algoritmo no Unity é muito tranquilo porque ele facilita muito os cálculos através dos métodos predefinidos da classe *Vector2* e do componente *Rigidbody2D* inseridos no jogador.

Passo 03 – Implementação do comportamento *Seek*

Inicialmente, é necessário introduzir algumas variáveis públicas na classe, para que o usuário possa facilmente configurar o comportamento do NPC. Já existe uma variável que define a velocidade máxima, que é *speed*. Falta agora uma para especificar a força máxima de navegação. Defina-a, por exemplo, com o nome de *maxSteeringForce*.

Para facilitar, divida o Algoritmo 02 em duas partes: uma para o cálculo da força de navegação e a outra para aplicá-la na velocidade do NPC. Assim, você conseguirá criar e adaptar novos comportamentos de forma mais simples, pois a segunda parte continuará a mesma. Enquanto a primeira parte pode ficar em um método qualquer, por exemplo *Seek*(), a segunda parte ficará no próprio método *UpdateFixed*().

Além desses métodos, é necessário alterar o método *Update*(), uma vez que não faz mais sentido girar o NPC para o alvo depois que o usuário clicar em um ponto. Logo, faz mais sentido atualizar o *Sprite* do NPC no método *UpdateFixed*(), e não no método *Update*(), para que o NPC gire paulatinamente, à medida que as forças vão sendo aplicadas.

Mas como para tudo na vida há um limite, você vai colocar um para essa velocidade também! Se a velocidade do NPC for muito pequena, ao ponto de poder ser considerado um objeto parado, não precisa mais atualizar sua direção. Então crie o atributo *stopVal* para definir esse limite do que se considera "parado". Essas ideias podem ser vistas implementadas no Unity no Código 02 abaixo.

Código 02 – Script com o comportamento *Seek*()

```
using UnityEngine;

public class SoccerPlayerBehavior : MonoBehaviour {

   public Vector2 goal;
   public float   speed;
   public float   maxSteeringForce;
   public float   stopVal;

   private Rigidbody2D body;

   void Start () {
      body = GetComponent<Rigidbody2D>();
      AdjustSpriteRotation ();
   }

   void AdjustSpriteRotation() {
      float angle = Vector2.Angle(body.velocity, Vector2.right);
      body.rotation = body.velocity.y >= 0 ? angle : 360 - angle;
   }

   Vector2 Seek() {
      Vector2 velocity = (goal - body.position).normalized * speed;
      Vector2 steering = velocity - body.velocity;
      return Vector2.ClampMagnitude(steering, maxSteeringForce);
   }

   void FixedUpdate () {
      Vector2 steering = Seek();
      Vector2 newVelocity = body.velocity + steering;
      body.velocity = Vector2.ClampMagnitude(newVelocity, speed);

      if (body.velocity.magnitude > stopVal)
         AdjustSpriteRotation();
   }

   void Update() {
      if (Input.GetButtonDown("Fire1")) {
         goal = (Vector2)
                  Camera.main.ScreenToWorldPoint(Input.mousePosition);
      }
   }
}

```

O vídeo a seguir mostra que os movimentos do personagem agora estão mais suaves. Diferentemente da primeira versão, dá para ver o jogador virando para um lado ou para o outro.

Vídeo 04 - Movimento suave do jogador no movimento *Seek*

Conteúdo interativo, acesse o Material Didático.

Esse comportamento foi gerado com os seguintes valores:

- Speed = 4

- Max Steering Force = 0.3

É interessante brincar um pouco com esses valores. Você vai ver que quanto maior o limite máximo de força, o NPC irá se virar mais rápido para o seu destino. Quanto menor o valor, mais lentamente ele irá se virar, ficando parecido com um carro, que precisa dar uma volta bem maior para fazer a curva.

O vídeo 05 abaixo, ilustra um problema do comportamento que foi implementado e que você já deve ter notado. Ao chegar no destino, o NPC fica indo de um lado para o outro feito barata tonta! 😁

Vídeo 05 - Movimento *Seek* com o vai-e-vem ao passar do destino

Conteúdo interativo, acesse o Material Didático.

O problema que ocorre aí é que o NPC vai a toda velocidade até o ponto destino, e se esquece de frear! Quando chega no ponto desejado, não dá para simplesmente parar usando as forças de navegação previamente descritas. Então, ele passa direto do destino feito um maluco. Depois, volta ao destino em alta velocidade... e passa novamente. E fica nesse vai-e-vem de cachorro atrás da própria cauda. E aí, entendeu!?

Esse comportamento é útil quando os jogadores disputam a bola rolando. Quem chegar primeiro fica com a bola. Nesse caso, eles precisam ir sem pisar no freio. Porém, se o objetivo é só ir até um ponto do campo, o ideal é que eles comecem a reduzir sua velocidade à medida em que eles se aproximam do destino. Faz sentido, não é? 😁

Agora vou lhe apresentar um outro comportamento, chamado *Arrival* (do inglês, chegada), similar ao *Seek*, mas que vai reduzindo a velocidade ao chegar no destino.

2.2 Comportamento *Arrival* (chegada)

Como dito anteriormente, o comportamento *Arrival* é bem similar ao *Seek*. A diferença é que ele deve reduzir a velocidade ao se aproximar do destino. Bom, "se aproximar" é algo vago para o computador. É necessário especificar a que distância ele deve começar a frear. Para isso, defina um novo atributo público na classe, chamado de *arrivalDistance*, que representará a distância do NPC para o destino na qual ele começará a "pisar no freio".

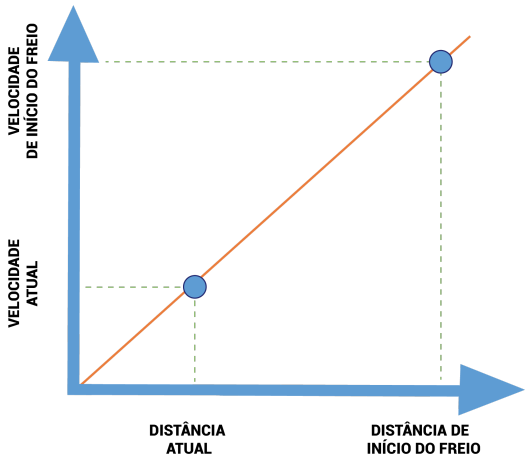
Pisar no freio na matemática vetorial é aplicar uma força contrária à velocidade atual, porém de menor intensidade, pois você não quer que pareça que o jogador bateu numa "parede invisível". Ele precisa ir parando aos poucos.

Quanto de força, então, deve-se aplicar? Se você aplicar um valor fixo a cada atualização, o NPC pode parar antes do destino... ou até depois de ter passado do destino.

Uma forma de resolver esse problema é aplicar menos força em direção ao destino à medida que o NPC se aproxima dele. Nesse caso, você pode usar uma função de atenuação da velocidade desejada em relação à distância, de tal forma que quando a distância chegar a zero a velocidade deve ser também zero (o que significa que o jogador estará parado). Ou seja, quanto mais o jogador se aproxima do destino, mais sua velocidade vai se aproximando de zero.

Há várias maneiras de implementar uma função de atenuação, porém a mais simples, e por isso, a mais usada, é através de uma interpolação linear. Uma regra de três simples entre a distância e a velocidade de início do freio e a distância e a velocidade atual do jogador pode resolver o caso, como ilustra a Figura 12.

Figura 12 - Freio implementado através de uma Interpolação linear



Pela regra de três, o freio é apenas uma questão de multiplicar o vetor de velocidade desejada pela fração das distâncias, especificada abaixo.

$$fração\,de\,distância = \frac{distância\,atual}{distância\,do\,início\,do\,freio}$$

A implementação da ideia descrita encontra-se no Código 03. Nele, introduz-se o método *SetMagnitude*(), uma vez que os vetores estão sempre sendo reajustados com um determinado tamanho. Crie, então, um método para simplificar o código.

Veja que o método *Arrival*() possui três casos. O primeiro é quando o NPC já se encontra muito próximo do alvo (distância menor que a constante *stopVal*). Nesse caso, esqueça as forças de navegação e retorne simplesmente uma força contrária a atual velocidade do NPC. Assim, ao somar essa força com a sua velocidade, ela será zerada, fazendo-o parar. O segundo caso é quando o NPC se encontra na "área de freio" (distância menor do que *arrivalDistance*). Nesse caso, aplique os cálculos descritos anteriormente. Por fim, se nenhum desses casos ocorrer, é porque o NPC precisa manter o comportamento *Seek*.

Código 03 – Implementando o comportamento *Arrival*

```
Vector2 SetMagnitude(Vector2 v, float max) {
   return v.normalized * max;
}

Vector2 Seek() {
   Vector2 velocity = SetMagnitude(goal - body.position, speed);
   Vector2 steering = velocity - body.velocity;
   return Vector2.ClampMagnitude(steering, maxSteeringForce);
}

Vector2 Arrival() {
   float distance = Vector2.Distance(body.position, goal);
   if (distance < stopVal)
      return -body.velocity;
   else if (distance < arrivalDistance) {
      Vector2 velocity = SetMagnitude(goal - body.position, speed);
      velocity *= distance / arrivalDistance;
      Vector2 steering = velocity - body.velocity;
      return Vector2.ClampMagnitude(steering, maxSteeringForce);
   }
   else {
      return Seek();
   }
}

void FixedUpdate () {
   Vector2 steering = Arrival();
   Vector2 newVelocity = body.velocity + steering;
   body.velocity = Vector2.ClampMagnitude(newVelocity, speed);

   if (body.velocity.magnitude > minDist)
      AdjustSpriteRotation();
}

```

Parabéns! 👏🏻 Você conseguiu ensinar dois comportamentos para os jogadores: ir em direção a um alvo e diminuir a velocidade quando ele estiver chegando nele. A partir desses comportamentos, você irá instruir os craques da bola que você criar a realizar manobras mais radicais na próxima aula! Até lá!

Resumo

Espero que todo esse conteúdo de matemática vetorial não tenha te assustado. 😁

À primeira vista, os cálculos podem parecer complicados, mas todas as operações a serem realizadas são operações básicas sobre vetores. Tá enrolado? Volta lá nas aulas de Matemática e Física para jogos e dê uma refrescada no conteúdo, e sempre aperreie seus coleguinhas e tutor para tirar as dúvidas!

Você viu nessa aula que os comportamentos de navegação são estratégias simples para criar comportamentos complexos. Porém, entre correr até um ponto e frear, não tem tanta complexidade... ainda! Prepare-se para fortes emoções ~~no próximo capítulo da novela~~ na próxima aula.

Até lá! 😁

Atividade

Imagine que um personagem do seu jogo se encontra na posição (0, 0) do cenário com velocidade definida pelo vetor (1, 1). O personagem tem como objetivo ir para a posição (6, 6). Considerando que o limite máximo de força de navegação é de 1 unidade e o de velocidade é de 2 unidades, em que posição o personagem se encontrará no próximo *frame* se você aplicar os comportamentos de navegação *Seek*?

Para verificar a resposta, clique aqui.

Resposta

Apesar de, no jogo, não ser você quem irá fazer os cálculos aqui propostos (eles serão feitos pelo computador), o intuito desta atividade é de você acompanhar o raciocínio por trás dos cálculos. Os dados que temos é que $posição = (0; 0)$ e $alvo = (6; 6)$. Seguindo, então, as fórmulas apresentadas, temos:

$$velocidade_{desejada} = alvo - posição$$

$$velocidade_{desejada} = (6; 6) - (0; 0) = (6; 6)$$

$$||velocidade_{desejada}|| = ||\,(6;6)\,|| = \left(\frac{1}{\sqrt{2}};\frac{1}{\sqrt{2}}\right)$$

$$velocidade_{máxima} \times ||velocidade_{desejada}|| = 2 \times \left(\frac{1}{\sqrt{2}};\frac{1}{\sqrt{2}}\right) = (\sqrt{2};\sqrt{2})$$

$$força_{navegação} = velocidade_{desejada} - velocidade = (\sqrt{2};\sqrt{2}) - (1;1) = (\sqrt{2}-1;\sqrt{2}-1)$$

Como $||força_{navegação}|| < força_{máxima}$, ou seja que $(\sqrt{2}-1;\sqrt{2}-1) < 1$, então

$força_{navegação}$ não será alterada (ou seja, truncada no valor máximo). Assim...

$$velocidade_{nova} = velocidade_{atual} + força_{navegação} = (1;1) + (\sqrt{2}-1;\sqrt{2}-1) = (\sqrt{2};\sqrt{2})$$

Como $||velocidade_{nova}|| < velocidade_{máxima}$,ou seja, que $(\sqrt{2};\sqrt{2}) < 2$, não precisamos truncá-la. Assim, a nova posição é calculada a partir dessa nova velocidade.

$$posição_{nova} = (0;0) + (\sqrt{2};\sqrt{2}) = (\sqrt{2};\sqrt{2})$$

A posição do personagem no próximo frame é, portanto, $(\sqrt{2};\sqrt{2})$.

Referências

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

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

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