**UNIVERSIDADE FEDERAL DE PERNAMBUCO**
**CENTRO DE INFORMÁTICA**

**Rodrigo Oliveira Tenório**

# *STEERING BEHAVIORS* E MOVIMENTOS INTERMITENTES: UM CONFLITO ENTRE AS CAMADAS DE LOCOMOÇÃO E DIREÇÃO

RECIFE-PE
2016

**RODRIGO OLIVEIRA TENÓRIO**

# *STEERING BEHAVIORS* E MOVIMENTOS INTERMITENTES: UM CONFLITO ENTRE AS CAMADAS DE LOCOMOÇÃO E DIREÇÃO

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito para obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Giordano Cabral

RECIFE-PE
2016

# RESUMO

Este projeto aborda o conflito entre as camadas de locomoção e de direção (*steering behaviors*) no comportamento de movimento para agentes autônomos quando, estes, apresentam um tipo de movimento intermitente (exemplo: a locomoção das águas-vivas). Também são abordadas, neste trabalho, sugestões de adaptações para o algoritmo de movimento que podem mitigar esses conflitos.

# ABSTRACT

This project discusses the conflict between the layers of locomotion and steering (Steering Behaviors) in the motion behavior for autonomous agents, when the agents present an intermittent type of movement (ex: locomotion of the jellyfish). Also discussed in this paper are suggested adaptations for the motion algorithm that can mitigate these conflicts.

# Sumário

# 1. INTRODUÇÃO

Em 2015, foi desenvolvido, para a disciplina de jogos 2D do Centro de Informática (CIn), um jogo chamado JellyFish Saga. Este jogo foi ambientado no fundo do oceano, onde diversos *Non Playable Characters* (NPCs), representando águas-vivas, tentavam caçar uns aos outros, desviando de pedras e corais que estivessem em seu caminho.

Durante a fase de planejamento do jogo, chegou-se à conclusão que um algoritmo de *steering behavior* seria uma excelente escolha para implementar a inteligência artificial desses (NPCs). O algoritmo é conhecido por permitir uma simulação de movimento natural e convincente para agentes autônomos, principalmente para grupos numerosos de agentes, com o benefício de ser fácil de implementar e possuir um baixo custo de processamento [1]. Essa abordagem teria sido simples, não fosse por um detalhe: o tipo de locomoção das águas-vivas.

A movimentação das águas-vivas ocorre por propulsão a jato, que é um método de locomoção aquática bastante característico de diversos invertebrados no meio aquático. Tais animais enchem de água uma cavidade muscular e, ao esguichar seu conteúdo, conseguem propelir seu corpo em uma direção oposta à água esguichada [2].

A fim de conferir realismo à movimentação das águas-vivas no jogo, realizou-se um estudo do seu movimento no mundo real e em simulações do mesmo em outros jogos. A conclusão desta pesquisa resultou em um movimento que possui características que conflitaram com o algoritmo de *steering behavior*, dificultando sua implementação e diminuindo sua eficiência.

Para tentar mitigar esse problema, sem abandonar as características do movimento em favor de uma solução mais simples e menos realista, foi feita uma extensa pesquisa na área de *steering behaviors* e na área de movimentos com características semelhantes utilizados em agentes autônomos. Constatou-se que parece não existir na literatura e nem mesmo em fóruns da área de desenvolvimento de jogos, artigos ou textos que se referem a esta questão.

Portanto, tendo em vista o contexto abordado, este trabalho teve como objetivo detalhar os desafios que surgem ao adotar os algoritmos de *steering behaviors* para simular agentes autônomos que apresentem uma locomoção

intermitente, bem como, discutir e sugerir adaptações para aumentar a eficiência do algoritmo.

Nos próximos capítulos, serão abordados: a movimentação das águas-vivas tanto no mundo real quanto sua representação no mundo dos jogos; o funcionamento do algoritmo de *steering behaviors* para compreender melhor suas intrincácias; as tecnologias adotadas para implementar os comportamentos desejados; e um relatório de adaptações sugeridas para os comportamentos. Por fim, os resultados serão demonstrados através de cenários de testes que destaquem os impactos das adaptações sugeridas por este trabalho.

# 2. CONTEXTUALIZAÇÃO

## 1. ÁGUAS-VIVAS

Durante a fase de *brainstorming* do projeto, a ideia de usar águas-vivas para representar os personagens de um jogo aquático foi praticamente instantânea. Este animal apresenta características que se adequam, perfeitamente, com as mecânicas que a equipe de desenvolvimento pretendia implementar no jogo: águas-vivas têm a capacidade de engolir umas às outras; elas podem usar veneno em suas vítimas; possuem as mais diversas cores e formatos; e, talvez a característica mais interessante para um jogo, elas possuem um padrão de movimento distinto que ajuda a destacar o jogo como algo original. Tornando-se, assim, a escolha perfeita para o jogo e os algoritmos de *steerings behaviors* podem ajudar a representar o comportamento destes personagens.

Antes de se discutir os detalhes sobre a implementação do algoritmo de *steering behaviors* para os agentes autônomos deste projeto, é interessante conhecer mais sobre as características do tipo de movimento das águas-vivas. Um melhor entendimento do animal e de seu movimento possibilita elaborar uma simulação mais verídica e resolver possíveis conflitos que surgem no caminho.

Ainda neste capítulo, serão abordadas algumas representações de águas-vivas no mundo dos jogos e, por fim, será discutida a implementação adotada pelo jogo JellyFish Saga, que gerou a identificação do problema discutido por este trabalho.

### 1. Propulsão a jato

O tipo de movimento empregado pelas águas-vivas na natureza é a propulsão a jato. Trata-se de um método de locomoção aquática onde os animais enchem de água uma cavidade muscular e esguicham-na para impulsioná-los na direção oposta à água jorrada. O enchimento da cavidade causa um aumento na massa e no arrasto do animal. Devido ao aumento do volume na cavidade muscular, a velocidade do animal varia à medida que ele se move através da água, acelerando enquanto expele a água e desacelerando enquanto aspira a água. Mesmo que estas flutuações no arrasto e na massa possam ser ignoradas, se a frequência dos ciclos

de propulsão a jato for suficientemente alta, ela será considerada um método relativamente ineficiente de locomoção em relação a outros animais aquáticos [3].

## 2. Especificidades do movimento das águas-vivas

Ao assistir uma água-viva nadar, fica claro que seu movimento depende de abrir e fechar seu sino. Esses animais têm um anel de músculos em torno das bordas do sino que ao contraírem estes músculos, ele se comprime. Esta contração força para fora a água que foi armazenada dentro do sino. Depois de se impulsionar para frente na fase de contração, as águas-vivas relaxam os músculos e abrem o sino, antes de contraí-lo novamente repetindo o ciclo.

Um fato, ainda que não influencie diretamente a modelagem adotada neste trabalho, é que por muito tempo se acreditou que o custo metabólico do transporte de água-viva era elevado quando comparado com um peixe de massa igual. Entretanto, um novo estudo mostrou que as águas-vivas são, na verdade, os nadadores mais eficientes do mundo animal. O segredo delas ficou conhecido como sistema de propulsão dupla. Elas possuem simetria radial então, quando contraem seu sino, isso verte um corpo em formato de *donut* fluído, o qual gira em torno de si mesmo. As águas-vivas derramam esses anéis de vórtice atrás delas e esse vórtice termina por criar um segundo impulso “grátis”, sem ser necessário nenhum esforço por parte do animal [4].

## 3. Abordagem em jogos

Saindo do mundo real e entrando no contexto virtual, quando se refere a jogos em ambientes aquáticos, não é raro a presença de águas-vivas em tais cenários. Trata-se de um animal fascinante e detentor de um padrão de movimento bem característico, fato este que os torna excelentes opções para representar NPCs, principalmente em jogos de plataforma.

Talvez, a representação mais característica das águas-vivas, quando usadas como NPCs, seja a abordagem que o jogo Terraria utiliza. Nesta implementação, a água-viva, hostil ao jogador, alinha o topo de sua cabeça/sino em sentido ao *player* e se propele nesta direção, não mudando seu curso até a próxima propulsão. Na natureza, as frequências entre as propulsões são tão curtas que o movimento do animal chega a parecer contínuo e suas mudanças de rotas são suaves e

curvilíneas. Entretanto, nos jogos, essa frequência tende a ser diminuída, visando obter uma espécie de caricatura deste movimento na natureza.

## 4. A abordagem deste trabalho

Após estudar o movimento das águas-vivas na natureza e de observar sua representação em jogos, a equipe de desenvolvimento decidiu adotar uma convenção semelhante à que jogo Terraria adotou, porém, com um ritmo de impulsos mais frequente. Esta convenção do movimento seguiu duas restrições:

- Após um impulso, a água-viva não pode mudar a direção da velocidade (não faz curvas).
- As águas-vivas só se impulsionam na direção que estão encarando no momento do impulso.

Ao adotar essas regras, a equipe de desenvolvimento do jogo ficou satisfeita com o resultado do movimento alcançado. Com a ideia de elaborar este trabalho e o fato de que os desafios abordados pelo mesmo surgiram justamente por causa da implementação do movimento convencionado acima, foi necessária a realização de um teste de validação para justificar a manutenção destas restrições, que tiveram por objetivo tornar o movimento da água-viva mais característico.

Para validar esta afirmação, foi elaborado um teste consistindo de um cenário virtual no qual um objeto vermelho realiza o movimento convencionado de água-viva. Neste teste, foi solicitado a trinta pessoas que ligassem o tipo de movimento a uma das quatro opções contidas em um formulário: Carro, Sapo, Água-Viva, Mosca.

O resultado indicou que 75% das pessoas associaram o movimento com águas-vivas e 20% com o sapo. Durante o teste foi constatado que duas variáveis influenciam diretamente na identificação do movimento de água-viva: o período entre um impulso e outro (um segundo se mostrou ideal) e a variável de deslize do movimento, que dá a impressão que o corpo está submerso em água.

## *2. STEERING BEHAVIORS*

Nesta sessão, é apresentado o algoritmo de *steering behaviors*, sua história e arquitetura, bem como, um exemplo prático de como ele pode ser implementado.

Para melhor compreensão do problema abordado por este trabalho, é fundamental o entendimento de como o algoritmo funciona.

## 2.1. Histórico

Em 1986, o cientista da computação Craig Reynolds criou um modelo computacional chamado de *boids*, capaz de simular movimentos coordenados vistos em bandos de aves ou peixes. O nome *boids* se referia às criaturas simuladas no modelo. Originalmente, esse modelo de grupos de animais era baseado em três comportamentos distintos [5]:

- Separação: foi usado para evitar situações onde os membros do grupo se aglomeravam.
- Alinhamento: usado para dirigir todos os elementos do grupo em uma direção comum.
- Coesão: orientou um único *boid* para a posição média de vizinhos próximos.

Este algoritmo foi capaz de criar uma simulação realista dos comportamentos naturais de grupo de animais através de uma combinação hábil desses três comportamentos simples.

Reynolds continuou aperfeiçoando seu modelo e, em 1988, produziu um artigo intitulado *Not bumping into things*, onde ele apresentava, descrevia e avaliava diversas implementações de algoritmos para desvio de obstáculos utilizados pelos *boids*. Essas estratégias não eram baseadas em planejamentos complexos, nem em algoritmos de busca de caminho, ao invés disso, elas utilizavam apenas informações locais como, por exemplo, objetos num raio próximo ao redor do veículo.

Surgiu, então, o próximo passo lógico no desenvolvimento do modelo de *boids*: Em 1999, foi apresentado na *Games Developer Conference* um trabalho de Reynolds intitulado *Steering behaviors for autonomous characters* no qual ele melhorava os comportamentos já apresentados no modelo original dos *boids*. Novos módulos para sistemas autônomos complexos foram apresentados. Cada um desses novos comportamentos definia apenas uma reação específica do agente autônomo ao ambiente simulado.

Os resultados desses módulos são peças simples que podem construir um sistema complexo. Dependendo da combinação de comportamentos utilizados, o veículo pode ser configurado para lidar com diferentes situações complexas. Os

comportamentos foram então agrupados sob o nome de *steering behaviors*, os quais definem o nível mais baixo para um sistema autônomo [5].

## 2.2. Agentes autônomos

Se no modelo de *flocking* tinham-se os *boids* como as unidades simuladas, nos *steering behaviors* têm-se os personagens autônomos. Estes são um tipo de agente autônomo usado em animação ou mídia interativa, como jogos e realidade virtual. Eles representam um personagem em uma história ou jogo e possuem uma capacidade limitada de improvisar suas ações. Esse fato, os difere tanto de personagens de filme de animação, cujas ações são escritas com antecedência, quanto de “avatares” em um jogo ou realidade virtual, cujas ações são direcionadas em tempo real por um jogador ou participante humano. Nos jogos, os personagens autônomos são às vezes chamados de NPC, apresentando três características-chaves, porém altamente customizáveis [6]:

- Capacidade limitada de perceber o ambiente. Um agente possui um raio limitado de percepção de seu arredor.
- Processa as informações de seu ambiente e calcula uma ação. Esta ação sendo, na prática, uma força que pode afastá-lo de uma ameaça ou o aproximar de uma presa.
- Um agente autônomo não tem líder. No sentido de que as decisões do agente são tomadas por ele mesmo.

## 2.3. Camadas do movimento

Em seu artigo, de 1999, *Steering Behaviors for Autonomous Characters*, Reynolds usa a palavra “veículo” para descrever seus agentes autônomos. Ele, então, descreve o movimento de veículos idealizados (idealizados porque não se está preocupado com a engenharia real desses veículos, mas simplesmente assumi-se que eles existem e responderão às regras) como sendo, dividido em três camadas: Seleção de Ações, Direção e Locomoção [7].

- Seleção de ações: um veículo tem uma meta (ou metas) e pode selecionar uma ação (ou uma combinação de ações) com base nesse objetivo. O artigo de Reynolds descreve muitos objetivos e ações associadas, tais como: buscar um alvo, evitar um obstáculo e seguir um caminho.

- Direção: uma vez que uma ação tenha sido selecionada, o veículo deve calcular seu próximo movimento. No contexto deste trabalho, o próximo passo será uma força, mais especificamente, uma força de direção. Felizmente, Reynolds desenvolveu uma fórmula de força de direção simples que caracterizou o algoritmo: força de direção = velocidade desejada – velocidade da corrente.
- Locomoção: é a parte inferior da hierarquia comportamental de três níveis descrita acima. A camada de locomoção representa a personificação de um personagem. Ele converte sinais de controle da camada de direção para o movimento do corpo do personagem. Este movimento está sujeito a restrições impostas pelo modelo físico do corpo, como a interação de *momentum* e força (limitação de forças que podem ser aplicadas pelo corpo) [7].

A motivação de fazer a distinção abstrata entre a camada de direção e locomoção é antecipar a "conexão" de um novo módulo de locomoção. Imagine trocar o cavalo de um cavaleiro por uma motocicleta. A seleção de ação e o comportamento da direção permanecem os mesmos. O que mudou é o mecanismo que mapeia os sinais de controle (ir mais rápido, virar à direita, etc.) em movimento. O papel do piloto não mudou [7].

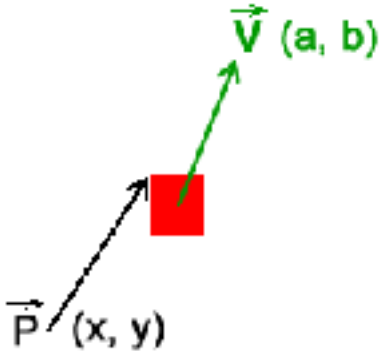
Isto sugere que ao utilizar uma convenção apropriada para comunicar os sinais de controle, os comportamentos de direção podem ser completamente independentes de um esquema de locomoção específica.

É interessante citar que, na maioria dos estudos na área de *steering behaviors*, a camada de locomoção é abstraída. Essa abstração não ocorre neste trabalho, pois o objeto de estudo, neste caso, é justamente os desafios que surgem devido às restrições impostas pela camada de locomoção do nosso veículo (a água-viva).

## 2.4. A força de *steering*

Agora que já se tem um bom entendimento de conceitos-chaves do *steering behavior*, pode-se discutir sobre a força de *steering* propriamente dita, que confere ao movimento do personagem autônomo, o tão desejado efeito de movimento natural que Reynolds cita em seu artigo. Para visualizar melhor como funciona essa força, nada melhor que acompanhar como se dá a implementação de um *steering behavior*.

A implementação de todas as forças envolvidas nos *steering behaviors* pode ser alcançada usando vetores matemáticos.




O movimento é calculado usando a integração de Euler:

A direção do vetor velocidade controlará para onde o personagem se move, já seu comprimento (ou magnitude) irá controlar o quanto ele irá mover cada quadro de tela, quanto maior o comprimento, mais rápido o personagem se move. O vetor velocidade pode ser truncado para garantir que não será maior que um determinado valor, geralmente a velocidade máxima.

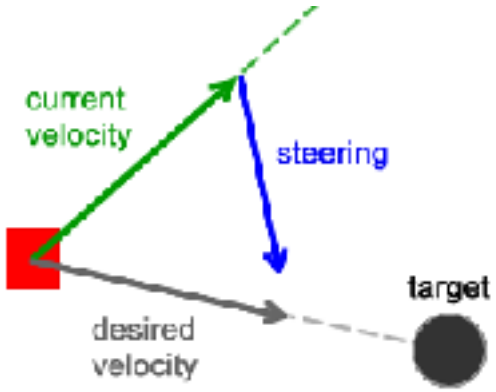
Para fazer o veículo perseguir um alvo, sem o envolvimento de uma força de *steering*, poder-se-ia usar o seguinte cálculo para sua velocidade:

Sem a força de direção, o veículo descreve rotas retas e muda instantaneamente sua direção quando o alvo se move, fazendo assim uma transição abrupta entre a rota atual e a nova.

Então, percebe-se que se houver apenas a força de velocidade envolvida no cálculo, o personagem seguirá uma linha reta definida pela direção desse vetor. Uma das ideias dos *steering behaviors* é influenciar o movimento do personagem através da adição de forças (chamadas forças de *steering*). Dependendo dessas forças, ele se moverá em uma ou outra direção.

Para o comportamento de busca, a adição de forças de direção ao veículo em cada *frame* facilita o ajuste da sua velocidade, evitando mudanças bruscas de rota. Se o alvo se move, o personagem vai mudar, gradualmente, o seu vetor de velocidade, tentando alcançar o alvo em sua nova localização.

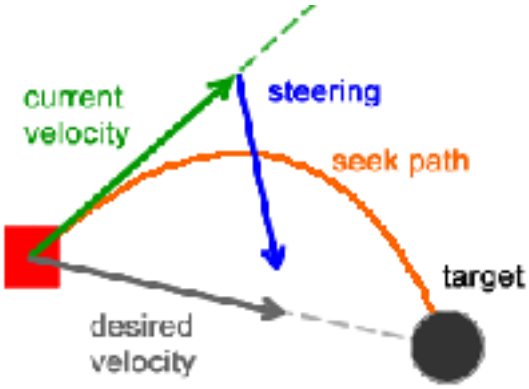
O comportamento de busca envolve duas forças: velocidade desejada e direção:




A velocidade desejada é uma força que guia o personagem em direção ao seu alvo usando o caminho mais curto possível (linha reta entre eles – anteriormente, esta era a única força que atuava sobre o veículo). A força de direção é o resultado da velocidade desejada subtraída pela velocidade atual, a qual empurra também o personagem para o alvo. Estas forças são calculadas da seguinte forma:

Depois que a força de direção for calculada, ela deve ser adicionada ao veículo (ela será adicionada à força de velocidade). A adição da força de direção à velocidade em cada quadro de tela fará com que o personagem abandone com suavidade sua antiga rota reta e siga em direção ao alvo, descrevendo um caminho de busca (curva laranja, na figura a seguir):

A adição dessas forças e o cálculo final da velocidade/posição são:




A força de direção é truncada para garantir que não exceda a quantidade de forças permitidas que o personagem pode manipular. Adicionalmente, a força de direção é dividida pela massa dos veículos, o que produz diferentes velocidades de movimento para diferentes pesos de veículos.

O vetor de velocidade de cada veículo muda cada vez que o alvo se move, no entanto, leva algum tempo para mudar e começar a apontar para o alvo novamente. O resultado é uma transição de movimento suave dando a impressão de naturalidade.

# 3. CONFLITO ENTRE AS CAMADAS

Nos capítulos anteriores, foi possível conhecer a movimentação das águas-vivas no mundo real e em jogos, além da modelagem adotada por este trabalho. Também foi visto como funciona o algoritmo de *steering behavior* e sua arquitetura. Finalmente, de posse desses conceitos, fica mais claro compreender o problema que surge ao tentar simular o movimento destes animais utilizando *steering behaviors*.

Como já mencionado anteriormente, durante a fase de concepção do jogo JellyFish Saga, existia a ideia de fazer um jogo com diversas águas-vivas caçando umas às outras, demonstrando um nível de inteligência que a permitisse navegar pelo cenário, desviando de eventuais obstáculos, enquanto buscavam seus objetivos. Chegou-se à conclusão de que o algoritmo de *steering behavior* era ideal para atingir este objetivo. Porém, na fase de implementação, foi constatado que uma característica-chave do algoritmo, não era compatível com o tipo de movimento que foi convencionado para as águas-vivas do jogo.

No capítulo dois, foi apresentado as convenções adotadas pelo jogo JellyFish Saga (JFS) para representar o movimento de suas águas-vivas, as regras que esse movimento deve respeitar para manter uma simulação característica de tais animais. É possível dividir o movimento adotado em duas fases distintas:

- Fase de contração: na natureza, seria o momento onde a água-viva expele a água e propele seu corpo para a frente. No jogo, é o quadro de tela onde a força resultante dos cálculos do algoritmo é aplicada no corpo do agente e a animação de contração é iniciada.
- Fase de relaxamento: na natureza, corresponde à fase onde a água-viva está em estado de repouso ou sugando água para dentro de sua cavidade. No jogo, é a fase onde o agente está flutuando e alterando a sua direção enquanto aguarda o próximo impulso.

Ao se realizar testes com esta configuração de movimento, notou-se que ao se aplicar as forças de *steering* para movimentar o agente, um problema se torna aparente. Para que o algoritmo de *steering behavior padrão* funcione da maneira esperada, o cálculo da força de *steering*, que influencia a rota do agente, precisa ser realizado e aplicado a cada frame de tela [8]. Porém, com a convenção adotada para

o movimento intermitente, não é possível ajustar a rota das águas-vivas a cada frame, apenas no momento em que o movimento está na fase de contração.

Executando o algoritmo de *steering behavior* padrão [7] ao mesmo tempo em que se respeita a restrição do movimento convencionado, a locomoção dos agentes se torna perceptivelmente ineficiente. Os agentes passam a não desviar de obstáculos com exatidão e a perseguição dos alvos se torna imprecisa. Esse fenômeno ocorre porque as forças de *steering* não possuem uma janela de tempo suficiente para influenciar a rota do agente.

Para melhor visualizar o problema, pode se imaginar dois cenários:

- Cenário sem restrição
  - A água-viva detecta um obstáculo.
  - Forças de *steering* começam a atuar para desviar o agente da rota de colisão.
  - Essas forças começam fracas, pois a água-viva ainda está distante do obstáculo, e aumentam à medida que ele se aproxima.
  - A água-viva descreve uma curva e desvia da pedra suavemente.
- Cenário com restrição
  - A água-viva detecta um obstáculo à sua frente.
  - Forças de *steering* são calculadas, mas não podem ser aplicadas, pois a água-viva está na fase de relaxamento e ela continua se movendo em direção ao obstáculo.
  - Agora bem próximo ao obstáculo, finalmente chega a fase de contração do movimento onde as forças de *steering* podem ser aplicadas. Contudo, por estar bem próximo do obstáculo, a força é enorme e a água-viva realiza um movimento brusco tentando desviar do obstáculo.
  - A água-viva desvia a tempo, mas tem sua rota prejudicada pelo movimento brusco.

Com a identificação do problema, pode-se vislumbrar a possibilidade de se abandonar a restrição do movimento que causa o conflito, permitindo que as forças de *steering* ajam continuamente no corpo do agente. Porém, o abandono de tais regras termina por descaracterizar o movimento intermitente dos agentes, neste caso as águas-vivas, lembrando menos o movimento do animal e mais a movimentação de um veículo genérico.

Ao se optar pela adoção das restrições, buscou-se ajuda na literatura e em fóruns de desenvolvimento para lidar com o conflito oriundo dessa opção, mas essa busca não obteve sucesso. Após extensa pesquisa de artigos na área, não foi encontrado um estudo onde algoritmos de *steering behaviors* são usados em um veículo que utilizasse movimento de propulsão a jato ou movimento intermitente similar.

É interessante notar que o mesmo problema seria enfrentado por um agente que se movesse por pulos, um sapo, por exemplo. Uma vez no ar, o sapo não pode alterar sua rota, dificultando a aplicação do *steering behavior* para seu movimento. Este trabalho decidiu por nomear este tipo de movimento que causa o conflito abordado de movimento intermitente, pois a definição desta palavra descreve muito bem a característica de movimentos como o da agua-viva e do sapo-pulante:

Dizer que algo é intermitente significa dizer que essa coisa cessa e recomeça por intervalos, que se manifesta com intermitências, que não é contínua, que tem interrupções.

# 4. OBJETIVO

Foi discutido no capítulo sobre *steering behavior*, que a camada de direção (*steering*) é independente da camada de locomoção. Isto é correto, porém, nada impede que adaptações sejam feitas na camada de *steering* para melhor se adequar a sua camada de locomoção. Essa é, na verdade, uma das características mais fortes do algoritmo, ele é altamente customizável.

Portanto, ao se decidir manter as restrições do movimento intermitente, torna-se necessário implementar adaptações, algumas de autoria própria, na camada de locomoção e na camada de direção para atingir resultados que sejam eficientes do ponto de vista da jogabilidade, mas que também mantenham os aspectos característicos de uma movimentação intermitente.

Assim, pode-se dizer que este trabalho tem como principal objetivo explicitar os principais desafios oriundos da incompatibilidade encontrada na união de *steering behaviors* com uma locomoção intermitente, e gerar adaptações para mitigar cada um desses desafios, bem como registrar os resultados dessas adaptações em métricas para medir a eficiência do algoritmo.

# 5. JELLYFISH SAGA

Neste capítulo, serão apresentadas as ferramentas utilizadas, bem como o jogo utilizado como base para demonstração dos problemas e soluções abordadas neste trabalho.

## *3.* O JOGO

Inicialmente, desenvolvido em 2015 para a cadeira de jogos 2d do CIn, por uma equipe liderada pelo aluno Rodrigo Oliveira Tenório, a ideia do jogo foi inspirada na fase inicial do jogo Spore (2008). Em JellyFish Saga (JFS), o jogador controla uma água-viva cujo o objetivo principal é devorar águas-vivas menores enquanto foge de águas-vivas maiores. O jogo se ambienta num cenário que simula o fundo do oceano, com diversos tipos de corais formando obstáculos e labirintos por onde as águas-vivas se movem.

Conforme já mencionado, este projeto usa a implementação do jogo JFS como base para demonstrar os desafios já relatados neste trabalho. Para melhor estudar e destacar cada desafio, modificações tiveram que ser feitas no código do jogo, e mecânicas que não eram pertinentes a sua ideia original foram posteriormente adicionadas para dar suporte a este trabalho. Por exemplo, originalmente, não havia motivo para que as águas-vivas desviassem uma das outras, pois a intenção delas era devorar umas às outras. No entanto, esse comportamento foi adicionado ao jogo para realizar alguns casos de testes que serão abordados mais adiante.

## *4.* DETALHES TÉCNICOS

O jogo JFS foi criado usando a *engine* de jogos Unity [9]. Os motivos que levaram a escolha desta plataforma foram vários, entre os principais estavam a simplicidade de portar o jogo para diversas plataformas e a facilidade de se achar tutoriais na internet, bem como uma documentação bastante completa e uma comunidade ativa nos fóruns.

A *engine* fornece duas opções de linguagem para codificar os *scripts* do jogo: C# e JavaScript. Foi definida a utilização da linguagem C# em todo o código do jogo, tanto pela familiaridade com a linguagem como também por conselhos de

desenvolvedores mais experientes no uso da Unity, segundo os quais, usar uma linguagem tipada pouparia eventuais problemas no futuro.

A *engine* possui um modo de teste onde é possível alterar valores de variáveis do código do jogo manualmente em tempo de execução [10], o que se mostra ideal para o objetivo deste trabalho que demonstra o impacto que cada mudança causa do desempenho do algoritmo.

A *engine* também fornece diversos recursos que facilitam a implementação do algoritmo de *steering behaviors*, possibilitando o uso de sua *engine* de física, detecção de colisões e *ray casts*.

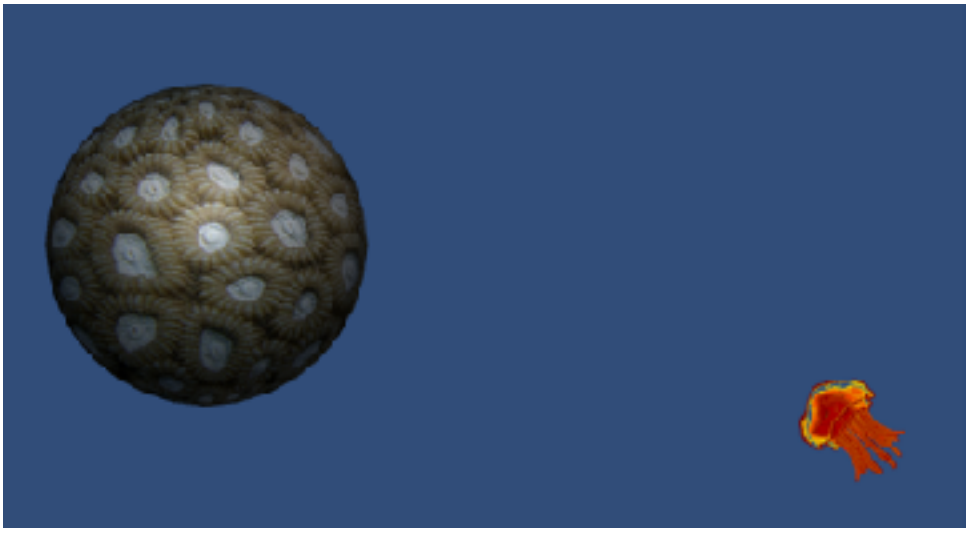
# 5. ELEMENTOS DO JOGO

## 5.1. O agente

Como o agente autônomo deste trabalho tem-se o NPC água-viva que, daqui por diante, será chamado apenas de agente. Ele é representado por um *sprite* 2d que possui animação para os estados em repouso e em movimento.

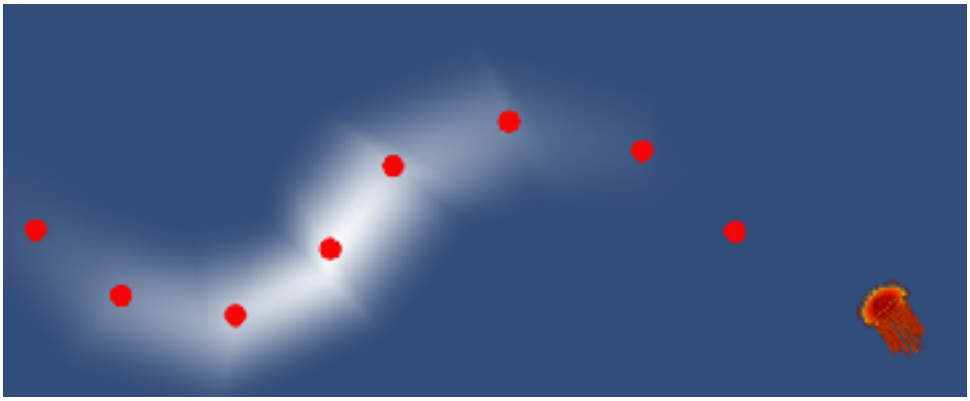
Os agentes possuem uma interface no modo de teste da *engine*, que permite ativar e desativar diversos comportamentos de *steering behaviors* que estão sendo executados, possibilita a alteração dos valores de variáveis como força exercida pela contração, peso do agente, velocidade de rotação e tamanho dos *feelers*.

## 5.2. Os obstáculos



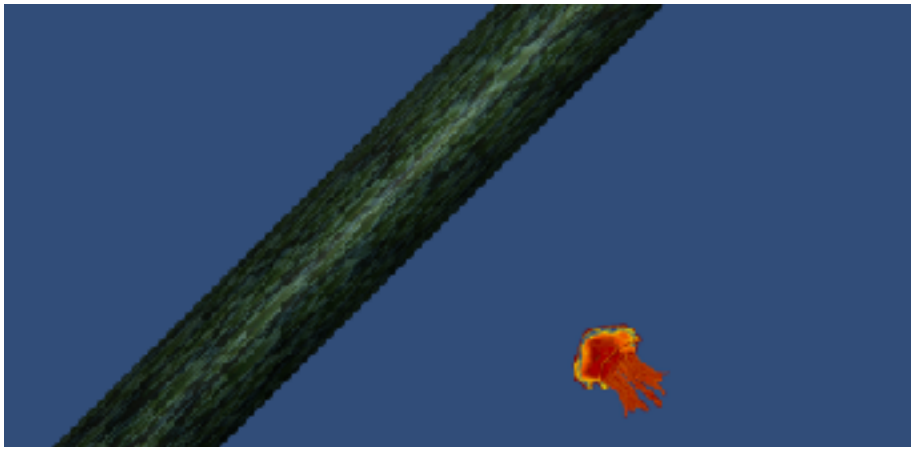
No jogo, os obstáculos são representados por esferas 3d com uma caixa de colisão em formato de círculo 2d fornecida pela *engine* do *unity*. Texturas de corais são utilizadas caracterizar esses objetos.

## 5.3. O Percurso



O objeto responsável por determinar a rota que os agentes devem seguir. É formado por diversos nodos vermelhos, quando o agente toca um, segue em direção ao próximo. O caminho é realçado por uma iluminação branca.

## 5.4. As paredes



Para representar as paredes temos cubos 3d com caixas de colisão retangulares 2d. Estes objetos também usam uma textura de coral.

## 5.5. O ambiente

O cenário simula o fundo do oceano e, para ajudar a implementar este ambiente, usamos a *engine* física do *unity* para exercer uma força de resistência ao movimento, simulando o efeito submersão aquática.

# 6. ARQUITETURA DO PROJETO

Neste momento, será discutida a arquitetura da implementação do algoritmo que possibilita a simulação de movimento dos agentes. Essa arquitetura engloba as três camadas do movimento de agentes autônomos mencionada no capítulo três. Abordam-se elas, dando uma visão geral de suas responsabilidades e as particularidades das adaptações feitas por cada camada para comportar o movimento convencionado para as águas-vivas.

## 6. A CAMADA DE LOCOMOÇÃO

No capítulo de *steering behavior*, viu-se que a camada de locomoção é a responsável por fazer receber os sinais da camada de *steering* e traduzir esses sinais em movimento [7]. A incompatibilidade do tipo de movimento das águas-vivas do jogo com o algoritmo de *steering behavior* é, em grande parte, resolvida com uma solução adotada nesta camada.

Ao invés de empregar as forças de *steering* para influenciar a direção do vetor velocidade do agente, usam-se estas forças para influenciar a direção que o agente encara. Esta mudança de implementação permite que, mesmo que o agente não se mova imediatamente naquela direção, no momento em que ele entrar na janela de impulso, ela possa se mover para a melhor direção avaliada naquele exato momento.

A todo o frame, a força do algoritmo de *steering* é calculada e somada a velocidade atual do agente. O vetor resultante desta soma não é atribuído ao vetor velocidade do agente, como o algoritmo clássico determina. O vetor resultante é usado como meta de direção. O agente passa então a rotacionar seu corpo para encarar esta direção. No jogo, usa-ss a função de *lerp* [11] para realizar esta rotação para que a mudança não seja brusca.

O desacoplamento das forças de *steering* com o vetor velocidade do agente, evita que a restrição do movimento das águas-vivas seja desrespeitada. Seu vetor velocidade não terá a direção modificada entre os impulsos. A interação das forças de *steering* com a direção que o agente encara, que a partir de agora será chamado de vetor face, permite que o agente já esteja pronto para se propelir na melhor direção no momento do impulso.

Essa solução encontrada é um exemplo claro da separação das camadas de movimento que Reynolds descreveu. A camada de *steering* produz seus vetores de maneira independente e repassa esses vetores para a camada de locomoção. A camada de locomoção utiliza estes vetores da maneira que melhor lhe convier.

Matematicamente falando, substituí-se a fórmula clássica do algoritmo de *steering behavior*:

por:

No momento em que o agente se impulsiona, a força de impulso é aplicada com a mesma direção do . Os resultados alcançados com essa simples mudança de aplicação de forças são positivos, pois permitem que os agentes retenham o movimento característico das águas-vivas e permitem que o algoritmo de *steering behavior* os guiem.

Porém, um problema persiste, a janela de impulso limitada ainda causa os problemas já citados no capítulo anterior. Esbarros frequentes em obstáculos e mudanças de rotas bruscas devidos a janela de impulso se iniciar tarde demais após a detecção dos mesmos.

Portanto, vê-se que durante a implementação da camada de *steering*, adotam-se técnicas que permitem a mitigação desses problemas, ainda que não se possa eliminá-los por completo.

## 7. A CAMADA DE *STEERING*

Abordar-se-á, agora, a implementação dos comportamentos de *steering* utilizados por este projeto. Em cada comportamento, é discutido sua aplicação, a implementação adotada por esse projeto, os problemas enfrentados e as técnicas utilizadas para mitigar esses problemas.

É importante destacar uma característica chave da implementação destes comportamentos. Nos cálculos dos vetores de *steerings* deste projeto, todos os

vetores resultantes são “setados” com a magnitude do vetor velocidade atual. Isso é feito com dois objetivos: o primeiro, pelo motivo que não se trunca os valores para uma velocidade máxima. Então para evitar picos de velocidade e controlar a velocidade, essa medida foi adotada; outro motivo é para não aumentar a velocidade do agente fora da janela de impulso, o que seria uma violação do movimento convencionado para os agentes.

## 7.1.COMPORTAMENTO DE BUSCA E FUGA

Os comportamentos de busca e fuga implementam padrões comportamentais complementares. A lógica por trás do algoritmo é a mesma para ambos. A única diferença é a força de direção resultante. *Seek* é usado para orientar o agente para o alvo especificado. Um exemplo seria a água-viva nadando diretamente para o seu alvo. O comportamento de fugir é usado para simular uma forma simples de evasão.

### 7.1.1. Implementação

Não existem diferenças significantes entre a implementação clássica e a utilizada neste trabalho. A força de direção é calculada com base nos valores para a velocidade atual do veículo e a posição do alvo especificado. Em primeiro lugar, a posição do alvo é expressa como um vetor entre o alvo e o agente. Isto representa a velocidade desejada. A força de direção resulta da diferença entre este vetor e o vetor de velocidade atual.

No comportamento de busca a velocidade atual do veículo é subtraída da velocidade desejada. Para obter a força de direção para o comportamento de fuga, a velocidade desejada é subtraída da velocidade atual. Usando esses cálculos simples, o agente pode ser guiado para o seu alvo ou forçado a evitá-lo.

### 7.1.2. Problemas

Os problemas enfrentados na implementação do algoritmo de busca e fuga são os já conhecidos nessa área. Entre eles, o problema de chegada, onde o agente passa direto pelo alvo e precisa dar a volta e, assim, sucessivamente.

## 7.2. COMPORTAMENTO DE CHEGADA

O comportamento de chegada é uma extensão do comportamento de busca. Como o comportamento de busca, ele é usado para direcionar o veículo para um alvo especificado. A diferença importante é a maneira como o agente chega ao destino. O comportamento de busca vai de encontro ao seu objetivo a toda velocidade, viaja um pouco mais na direção atual e depois volta a dançar como uma mariposa em torno de uma fonte de luz. Normalmente, este não é o tipo de resultado que se espera. O comportamento de chegada diminui a velocidade do veículo de uma forma controlada de acordo com uma especificação e para o seu movimento na posição desejada.

### 7.2.1. Implementação

Sua implementação se dá dentro do algoritmo de busca. Cada agente possui um raio de proximidade. Quando o alvo do agente entra nesse raio, o mecanismo de redução de velocidade do agente entra em ação, fazendo com que ele alcance seu alvo com velocidade zero.

### 7.2.2. Problemas

Na implementação clássica, para desacelerar o agente rumo ao seu alvo, uma força de *steering* é gerada e adicionada a velocidade do agente. Porém, no caso dos agentes deste trabalho, essas forças de *steering* não podem ser aplicadas a todo o momento, devido à restrição da camada de locomoção. Isto torna a implementação do algoritmo de chegada problemática.

Para mitigar este problema, foi utilizado o mecanismo conhecido como “*hard stop*”, onde altera-se a velocidade do agente diretamente, sem soma de vetores, fazendo com que a cada frame sua velocidade seja diminuída por um valor.

## 7.3. COMPORTAMENTO PERSEGUIÇÃO E EVASÃO

Esse comportamento é uma evolução dos comportamentos de busca e fuga respectivamente. Quando um agente está a perseguir o seu alvo, que também está em movimento, é necessário que o agente tenha uma capacidade de prever onde seu alvo está indo a fim de interceptá-lo. Analogamente, quando o agente estiver

fugindo do seu perseguidor é interessante que ele possa prever a futura posição do tal para que possa realizar manobras evasivas.

### 7.3.1. Implementação

Obteve-se esse comportamento fazendo com que o agente utilize o algoritmo de busca, porém utilizando a posição futura do alvo no cálculo da força de *steering*. Para isso, é usada a velocidade do alvo para calcular sua posição nos próximos T segundos.

### 7.3.2. Problemas

Um dos problemas enfrentados é o fato de que quando o agente está próximo de seu alvo, não é interessante que ele continue perseguindo uma posição futura, o que terminaria por atrapalhar seu comportamento. Isso não é um problema exclusivo da locomoção intermitente, é um problema geral enfrentado por esse comportamento. Para solucioná-lo usou-se o seguinte mecanismo: ao se aproximar do alvo o valor de T diminui.

Um outro problema, esse característico da implementação adotada neste projeto, é o fato de que um alvo pode estar se movendo em uma direção, porém encarando uma posição oposta, se preparando para o próximo impulso. Seu perseguidor terá como o alvo a posição futura de acordo com o vetor velocidade do mesmo. Isso prejudica a previsão de posição. Não foi encontrada uma solução para este problema.

## 7.4. COMPORTAMENTO DE VAGUEIO

O comportamento de vagueio permite que os agentes se movam de forma autônoma e sem um alvo específico através do cenário. É comparável ao comportamento de formigas à procura de comida. A maioria dos animais, em seu caminho à procura de algo, não explora a área circundante objetivamente. Em vez disso, movem-se em caminhos mais aleatórios e indefinidos.

### 7.4.1. Implementação

Se não houver nenhum objetivo em seu raio de percepção, o agente passa a executar o comportamento de vagueio. A implementação usa a seguinte abordagem: um círculo é posicionado à frente do agente. A cada frame, este círculo descreve um raio dentro de si, e esse raio pode se distanciar angularmente do anterior, limitadamente. A cada frame soma-se o vetor velocidade do agente com o vetor representado pelo raio do círculo. O efeito desse algoritmo é que a força de *steering*, que desvie minimamente o vetor velocidade do agente, nunca oscile bruscamente, fazendo com que a rota descrita por esse comportamento descreva curvas suaves.

## 7.5. COMPORTAMENTO DE DESVIO DE OBSTÁCULOS

Para dirigir um agente através de um cenário predefinido de uma forma realista, é necessário movê-lo da maneira que um espectador esperaria. Portanto, é necessário definir um comportamento que o permita lidar com obstáculos. No mundo real, os seres humanos e os animais evitam colidir de maneira instintiva. Quando uma pessoa quer tomar uma cerveja gelada, ela não vai se mover para a geladeira em linha reta. Inconscientemente, ela escolherá um caminho que a desvia de todos os obstáculos do lugar, como mesas, cadeiras e outras coisas que se encontram espalhadas em uma casa, buscando atingir seu objetivo sem se machucar em algo. O comportamento de desvio implementa esse cuidado evitando obstáculos predefinidos.

Os comportamentos de desvio de colisão não são um algoritmo de "*pathfinding*". O comportamento fará com que os personagens se movam pelo ambiente, evitando obstáculos, e, eventualmente, encontrando uma rota para percorrer os blocos – mas não funciona muito bem com os obstáculos em forma de "L" ou "T", por exemplo.

### 7.5.1. Implementação

A primeira coisa quando se implementa o comportamento de desvio de obstáculos é a definição do obstáculo dentro do ambiente virtual. Uma vez que esta simulação se baseia numa cena bidimensional, utilizando formas geométricas simples como substituição de obstáculos complexos da vida real. A representação, possivelmente, mais simples de um obstáculo é o círculo e usa-se essa abordagem para modelar os obstáculos deste projeto.

A ideia básica, por trás do desvio de obstáculos, é gerar uma força de direção para evitá-los cada vez que um está perto o suficiente para bloquear a passagem. Mesmo que o ambiente tenha vários obstáculos, o comportamento se foca em apenas um deles de cada vez para estimar-se a força de desvio.

Para calcular a força de *steering* usa-se um vetor, comumente chamado de “see_ahead”, o qual detecta obstáculos à frente do agente. Ao entrar em contato com um impedimento, um vetor é avaliado do centro do empecilho até a ponta do vetor “see_ahead”. Esse vetor calculado será a força de *steering* que desviará o agente do obstáculo.

Uma particularidade das implementações dos comportamentos de desvio deste projeto, é que o algoritmo não se pode dar ao luxo de usar valores baixos para essa força de desvio. Como a janela de impulso é pequena, para impedir a colisão, é necessário fazer com que essa força tenha um peso considerável no cálculo da força resultante. Neste projeto é usado para o vetor de desvio oriundo do obstáculo, uma magnitude semelhante ao do vetor velocidade do agente.

### 7.5.2. Problemas

Os comportamentos de desvios deste projeto foram os que mais levantaram desafios devido à natureza do movimento simulado e suas restrições. Para que o agente realize o desvio de maneira eficiente, assim que ele detecta o obstáculo, ele precisa começar a desviar. Porém com as restrições do movimento, por muitas vezes a janela de impulso do agente só inicia quando é tarde demais para fazer um desvio suave ou mesmo para evitar a colisão.

Para mitigar este problema adotam-se as seguintes adaptações:

- Ao invés de um, usam-se três vetores de detecção (see_ahead), cada um com uma inclinação diferente. Isso permite uma melhor detecção de obstáculos nas laterais do agente evitando a detecção tardia delas.
- Comprimento dos vetores de detecção são maiores do que se adotaria para um movimento sem a restrição adotada por este trabalho. Aumenta-se o comprimento do see_ahead para compensar um ocasional desvio tardio.
- Comprimento do vetor de detecção varia de acordo com a velocidade do agente. Quanto mais lento, menor o vetor de detecção. Isso evita que o agente tenha dificuldade em girar em torno de si quando próximo de obstáculos.

- Já que não se pode aplicar as forças de *steering* fora da janela de impulso, adota-se uma mecânica de freio. Ao se aproximar de um obstáculo o agente diminui sua velocidade diretamente.
- Vetores de see_ahead na mesma direção, mas com diferentes comprimentos. Isso aumenta a força de desvio quando muito próximo de um obstáculo, pois as forças se acumulam. Também evita o problema onde a ponta do vetor corta o obstáculo e anula a força de *steering*, apesar do obstáculo ainda obstruir o caminho.
- Separam-se as forças de desvio de obstáculo e de paredes em uma força de *steering* distinta das forças resultantes de outros comportamentos, podendo assim dar um peso diferente para elas no cálculo da força de *steering* final.

Todas essas medidas aumentaram o desempenho do desvio de obstáculos na simulação do movimento. As frequentes colisões tiveram suas ocorrências diminuídas e as rotas dos agentes, ao realizar o desvio, se tornaram mais suaves.

## 7.6. COMPORTAMENTO DE DESVIO DE PAREDES

O algoritmo de desvio de paredes é bastante semelhante ao de desvio de obstáculos, o objetivo é fazer com que os agentes evitem colidir com as paredes que os rodeiam e que, ao invés disso, tenham a habilidade de seguir, paralelamente, a elas.

### 7.6.1. Implementação

Analogamente ao vetor see_ahead do comportamento anterior, tem-se, agora, os vetores de detecção chamados de *feelers*. Quando um desses *feelers* penetra em uma parede pode-se calcular a força de steering. Basicamente, multiplica-se a profundidade a penetração do *feeler* na parede pelo vetor normal da parede. Essa força desviará o agente da parede.

### 7.6.2. Problemas

Os problemas enfrentados aqui são os mesmos que viu-se no comportamento de desvio de obstáculos. O movimento intermitente atrapalha o desvio eficiente das

paredes. Para amenizar este problema, adotam-se soluções semelhantes às anteriores:

- Três *feelers* com inclinações diferentes, mas com aberturas de ângulos entre eles maiores que a dos vetores see_ahead.
- Mecanismo de freio.
- Cumprimento do *feeler* proporcional a velocidade.

# 3. A CAMADA DE SELEÇÃO DA AÇÃO

## 3.1. A mente

Denomina-se de Mente de um agente, a parte do algoritmo que é responsável por avaliar o ambiente do agente e decidir a melhor maneira de combinar seus comportamentos de *steering*. A mente pode decidir quais comportamentos irá executar e quais irão ignorar, que pesos atribuir a cada uma das forças de *steering* produzidas pelos comportamentos. Dessa maneira, a mente do agente consegue gerar uma força resultante extremamente refinada para melhor ajudar o agente a realizar seu objetivo. Pode-se dizer que a mente é a camada de seleção de ação do movimento do agente autônomo.

## 3.2. Implementação

Neste projeto, a mente dos agentes é responsável pelas seguintes tarefas:

- Executar o comportamento de vagueio apenas se não houver alvos na proximidade do agente.
- Separar as forças de desvio das demais e permitir que tenham um peso diferente no cálculo da força de *steering* resultante.
- Escolher qual o melhor alvo para perseguir na vizinhança do agente.
- Lidar com situações em que o ambiente apresenta vários inimigos.
- Em caso de “*deadlock*” não permitir que o agente fique imóvel.

## 3.3. Problemas

Foram encontradas grandes dificuldades para executar, simultaneamente, os comportamentos de busca e fuga. Observou-se que esse problema parece estar

relacionado ao movimento intermitente das águas-vivas. O problema será discutido posteriormente no caso de teste Mind-seek-flee-balance.

# 7. AVALIAÇÃO

Para avaliar os resultados obtidos com as soluções e adaptações sugeridas por este trabalho, são construídos cenários de testes usando a *engine* do Unity e o jogo. Esses testes são inspirados na suíte de testes do framework de benchmark para *steering behaviours* apresentados no estudo de Singh *et al.* (2008)[12].

Pode-se classificá-los em cinco categorias:

- Cenários de validação simples. Ex.: Dois agentes desviando de uma parede para alcançar um alvo.
- Interações um a um, básicas. Ex.: Dois agentes movendo em direções opostas para uma colisão frontal.
- Interações entre agentes incluindo obstáculos. Ex.: Dois agentes que não veem um ao outro até um último momento por causa de obstáculos grandes.
- Interações de grupos. Ex.: Um agente cortando caminho por entre um pequeno grupo de agentes.
- Cenários de larga escala. Ex.: Muitos agentes navegando por um corredor na mesma direção.

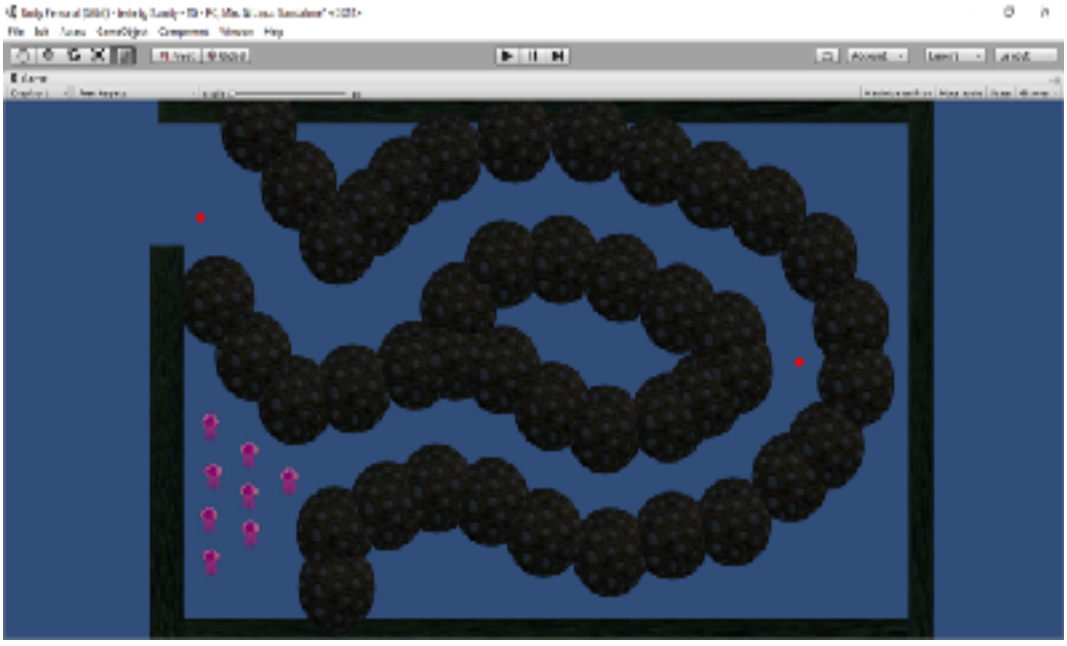
Os resultados desta suíte de testes serão submetidos a uma análise utilizando métricas de avaliação proposta pelo framework citado. Sendo essas métricas:

- Número de colisões: na maioria dos casos, menos colisões revelam um *steering* mais realístico.
- Tempo: quanto mais rápido o agente alcança seu alvo, mais eficiente é o *steering*
- Aprovação: Um grupo de 10 testadores dizem se o resultado é satisfatório ou não satisfatório

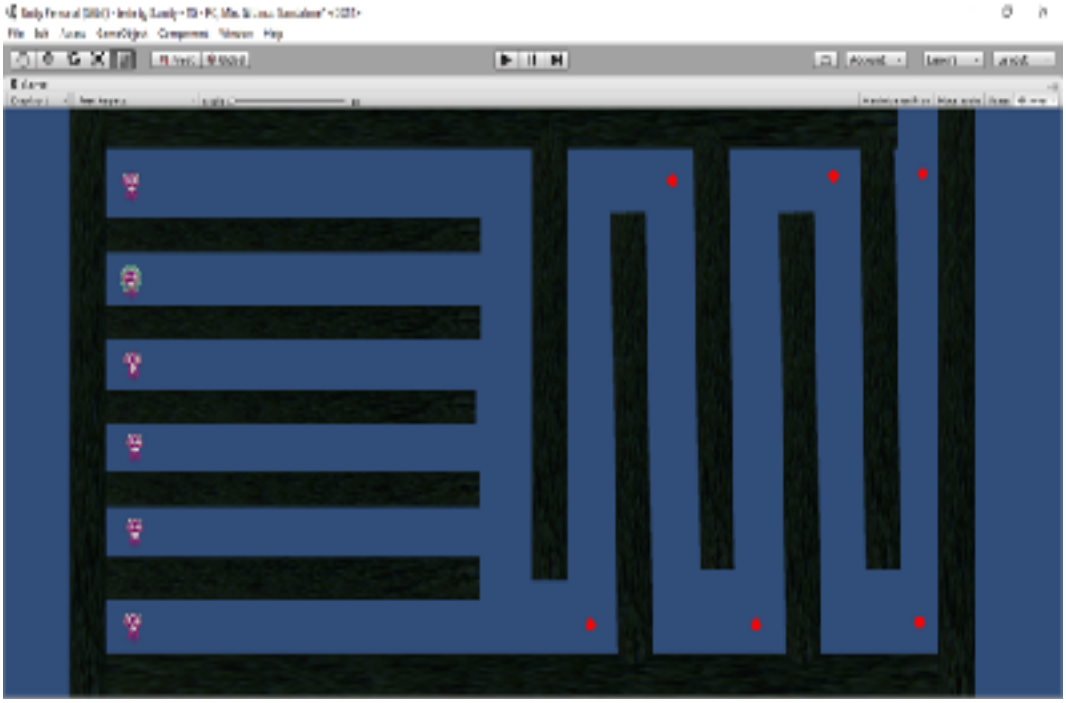
A seguir, será detalhada uma amostra dos cenários de teste realizados, com seus nomes, suas descrições, os resultados das métricas de cada teste e as observações feitas a partir de seus resultados.

# 4. CENÁRIOS DE TESTES

## 4.1. Cenário S-Corridor-Group

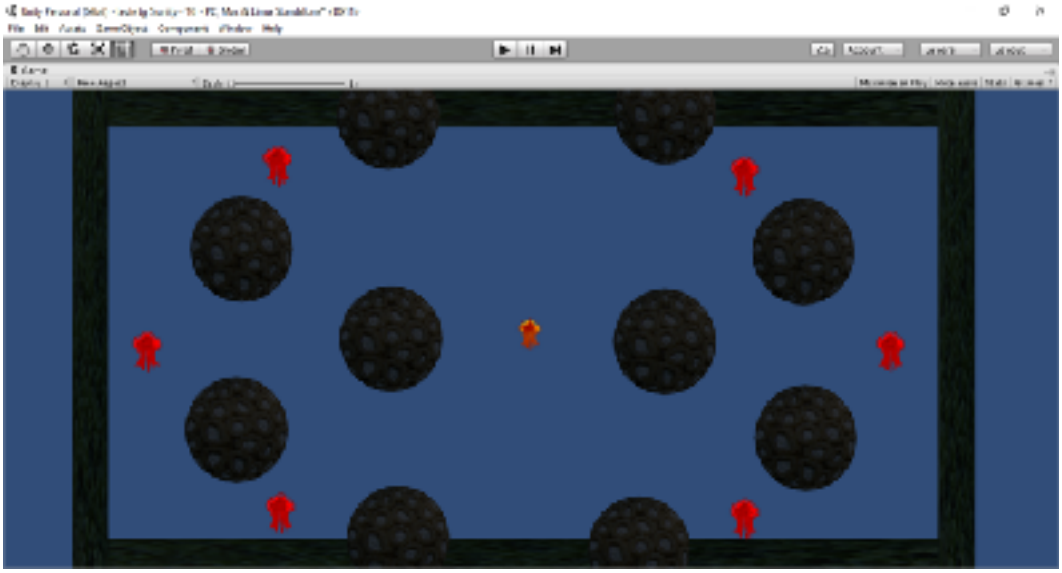


## 4.2. Cenário S-Maze-Group

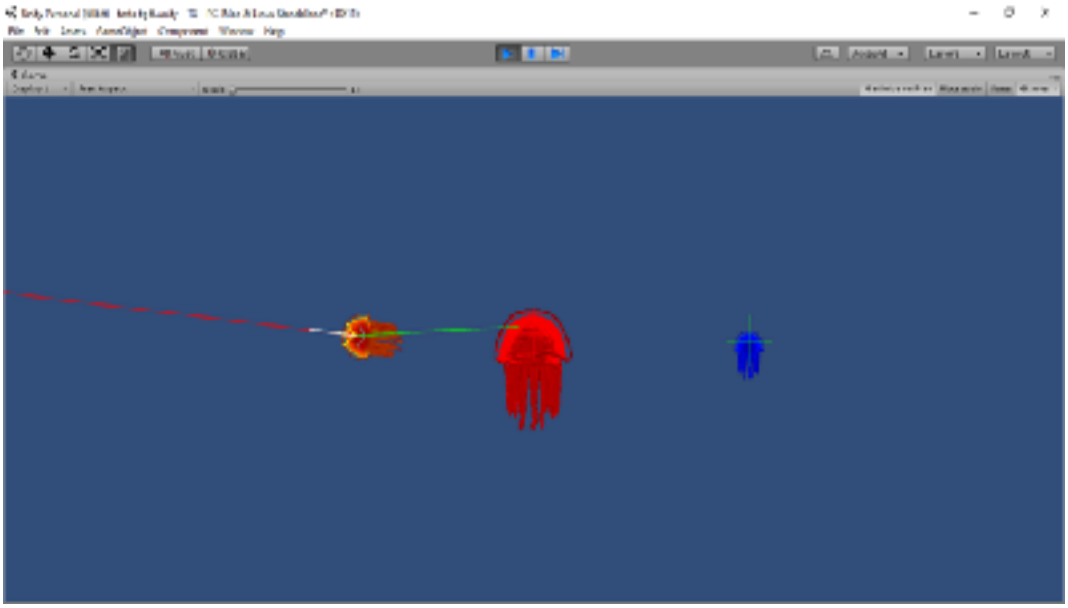


## 4.3. Cenário Pursuit-Group

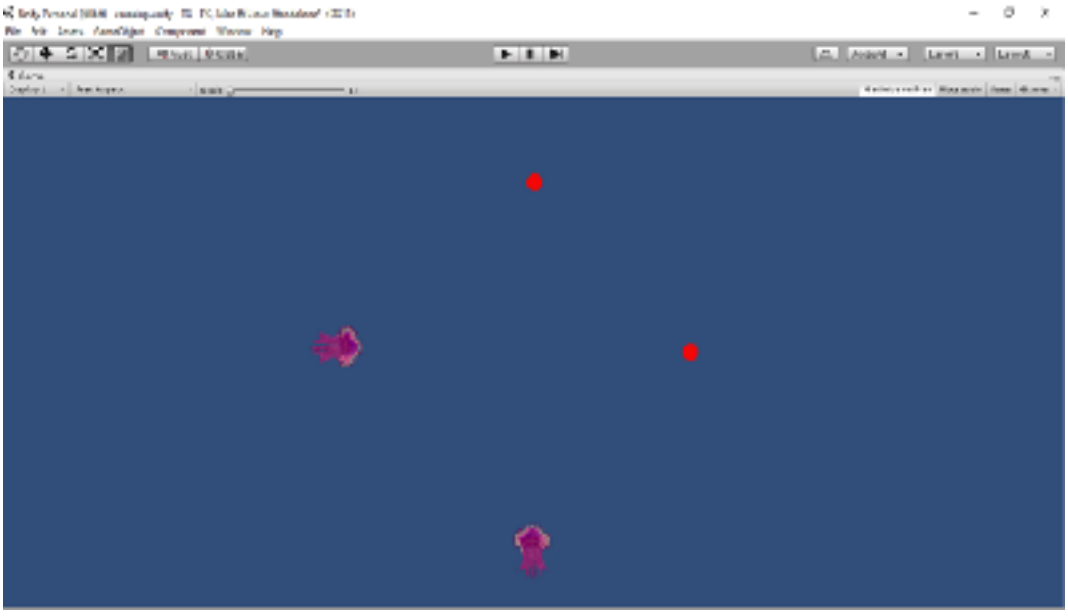
## 4.4. Cenário Evade-Group



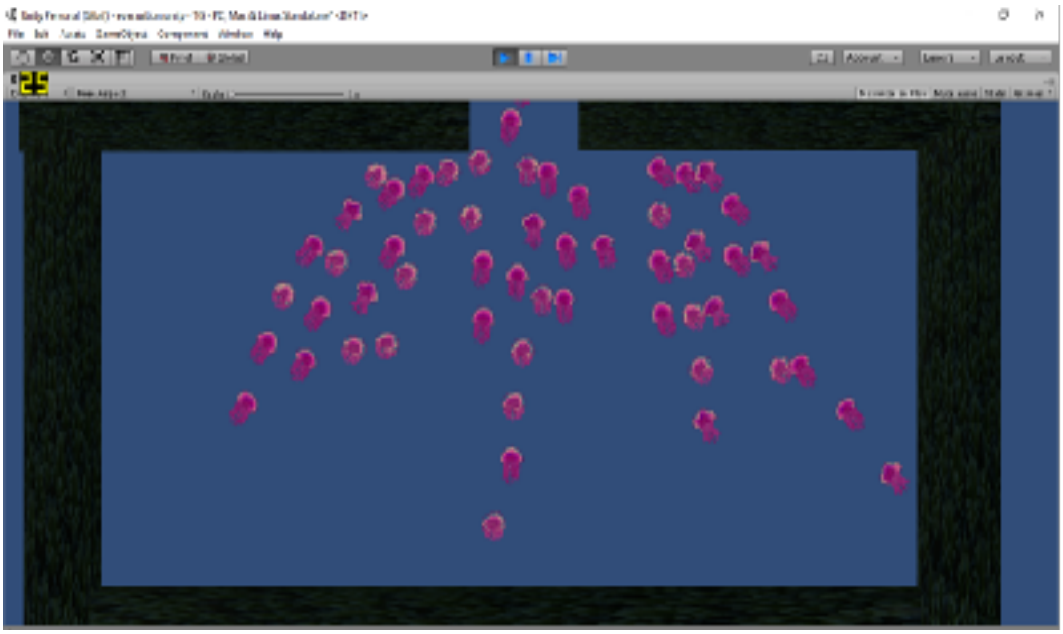
## 4.5. Cenário Mind-Seek-Flee-Balance



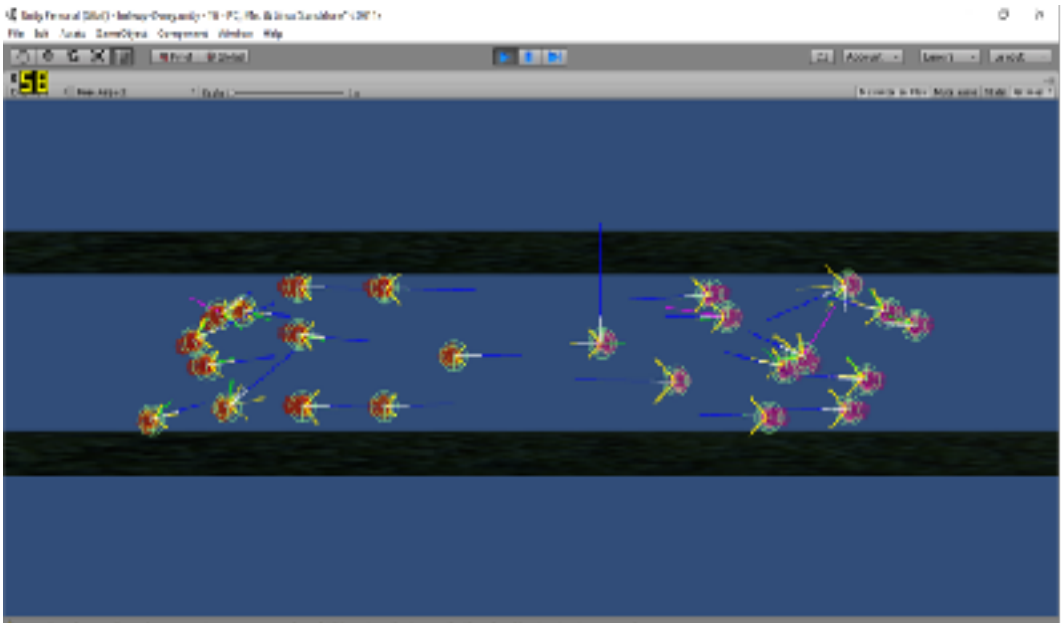
## 4.6. Cenário Crossing



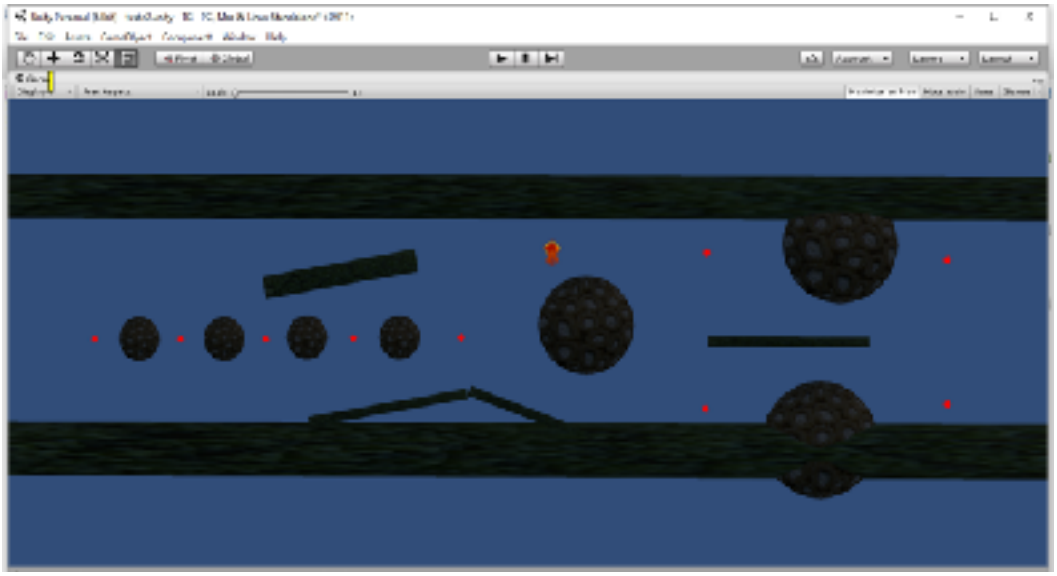
## 4.7. Cenário Evacuation



## 4.8. Cenário hallway-2-way



## 4.9. Cenário Solo-Tricks



7.2 RESULTADOS

**Cenário S-Corridor-Group**

| | |
|---|---|
| Comportamentos utilizados | obstacle-avoidance, path-following, queue. |
| Número de agentes | 8 |
| Número de colisões | 179 |
| Tempo | 42s 58’ |
| Descrição | Neste teste é possível medir a eficiência do comportamento de desvio de obstáculos. Também se vê o comportamento de fila evitando colisões entre agentes. O tempo é medido da largada até o último agente passar pelo último ponto vermelho do percurso. |
| Observações | Um ajuste no controle de força aplicada no impulso diminuiria as colisões nas paredes. Os agentes podem ficar presos nos sulcos entre os obstáculos. |
| Aprovação | 90% |

**Cenário S-Maze-Group**

| | |
|---|---|
| Comportamentos utilizados | wall-avoidance, path-following, queue. |
| Número de agentes | 6 |
| Número de colisões | 82 |
| Tempo | 58s 18’ |
| Descrição | wall-avoidance, path-following, queue Neste teste, é possível medir a eficiência do comportamento de desvio de paredes. Também se vê o comportamento de fila evitando colisões entre agentes. O tempo é medido da largada até o último agente passar pelo último ponto vermelho do percurso. |
| Observações | Neste teste, a adoção de feelers maiores ocasionou dificuldade na rotação dos agentes, fazendo com que eles não se recuperem de uma eventual rota oposta. |

| | |
|---|---|
| Aprovação | 90% |

**Cenário Pursuit-Group**

| | |
|---|---|
| Comportamentos utilizados | wall-avoidance, obstacle-avoidance, pursuit, evade, mind. |
| Número de agentes | 7 |
| Número de colisões | Irrelevante. |
| Tempo | Indeterminado. |
| Descrição | Neste cenário tem-se um agente laranja caçando agentes azuis. A mente se encarrega de escolher qual o melhor alvo, baseado na proximidade. Vê-se também a interação dos comportamentos de desvio e de busca agindo em conjunto. |
| Observações | Para este cenário, métricas como tempo e número de colisões são irrelevantes, pois ao mesmo tempo em que o comportamento de caça pode ser eficiente, o comportamento de fuga pode ser igualmente competente, tornando o período do teste indeterminado.<br>Talvez o mecanismo de predição deva levar em conta a direção para qual o alvo aponta ao invés de sua velocidade atual.<br>O mecanismo de predição da posição futura do alvo também se provou eficiente para o agente perseguido quando encurralado numa parede. Permite que o agente “drible” seu oponente ao invés de ficar encarando a parede parado. |
| Aprovação | 70% |

**Cenário Evade-Group**

| | |
|---|---|
| Comportamentos utilizados | wall-avoidance, obstacle-avoidance, pursuit, evade, mind. |
| Número de agentes | 7 |
| Número de colisões | Irrelevante. |
| Tempo | Indeterminado. |
| Descrição | Neste cenário, tem-se um agente laranja fugindo de seus perseguidores vermelhos. |
| Observações | Da mesma forma que o cenário anterior, métricas como tempo e número de colisões são irrelevantes neste caso de teste. |
| Aprovação | 60% |

**Cenário Mind-Seel-Flee-Balance**

| | |
|---|---|
| Comportamentos utilizados | seek, flee, mind**.** |
| Número de agentes | 3 |
| Número de colisões | Irrelevante. |
| Tempo | Indeterminado. |
| Descrição | Neste teste pode-se ver a Mente trabalhando. Ela faz o balanceamento das forças de steering de busca e fuga baseadas na proximidade. Tem-se um agente laranja que evita o agente vermelho e é atraído pelo agente azul. |

| | |
|---|---|
| Observações | Pode-se ver, também, que os algoritmos de steering behaviors não têm capacidade de traçar rotas rebuscadas. Neste cenário, com estas posições, o agente laranja não consegue desviar do agente vermelho a fim de capturar o azul. Ele simplesmente se afastará da ameaça mais próxima, pois sua força de steering sempre será maior. |
| Aprovação | 40% |

**Cenário Crossing**

| | |
|---|---|
| Comportamentos utilizados | path-following, obstacle-avoidance, queue. |
| Número de agentes | 60 |
| Número de colisões | 1 |
| Tempo | 5s |
| Descrição | Rotas dos agentes se cruzam, o objetivo é que eles não colidam. |
| Observações | O algoritmo criado neste projeto não foi capaz de realizar o desvio entre os próprios agentes de maneira eficaz. Possivelmente, a adoção de um mecanismo de predição no comportamento de desvio proporcione uma melhora neste teste. |
| Aprovação | 30% |

**Cenário Evacuation**

| | |
|---|---|
| Comportamentos utilizados | path-following, obstacle-avoidance, queue |
| Número de agentes | 2 |
| Número de colisões | 551 |
| Tempo | 1m 26s |
| Descrição | Neste teste, um grupo de agentes se dirige a uma única saída. O tempo é contado até o momento em que o último agente deixa o ambiente. |
| Observações | Como esperado, o resultado não foi satisfatório. Em parte, por se usar um mecanismo de hard-stop para o comportamento de fila, e por não estar sendo empregado um comportamento de grupo. |
| Aprovação | 50% |

**Cenário hallway-2-way**

| | |
|---|---|
| Comportamentos utilizados | path-following, obstacle-avoidance, queue**.** |
| Número de agentes | 24 |
| Número de colisões | 180 |
| Tempo | 27s |
| Descrição | Dois grupos de agentes viajam em direções opostas. Este cenário averiguar a capacidade de desvio entre agentes em um corredor apertado e lotado. |

| | |
|---|---|
| Observações | Mais uma vez fica clara a falta que um comportamento de flocking faz. Os agentes possuem a capacidade de desviar, porém não sem antes evitar a colisão.<br>Esse cenário também expõe a problemática do tamanho dos feelers em relação ao tamanho dos obstáculos. Se os feelers forem grandes o suficiente para ultrapassar os limites das caixas de colisão dos obstáculos o comportamento de desvio falhará. |
| Aprovação | 60% |

**Cenário Solo-Tricks**

| | |
|---|---|
| Comportamentos utilizados | path-following, obstacle-avoidance, wall-avoidance |
| Número de agentes | 1 |
| Número de colisões | 9 |
| Tempo | 51s |
| Descrição | Neste teste o agente precisa realizar uma série de manobras para realizar o percurso. O intuito deste teste é evidenciar a qualidade dos algoritmos de desvio do agente em manobras mais precisas. |
| Observações | Neste teste foi necessário um ajuste fino dos comportamentos de chegada para evitar ultrapassar os pontos da rota. Também se fez necessário encontrar um bom equilíbrio entre as forças do comportamento de busca e de desvio. Este é mais um teste que mostra a necessidade de implementar uma mecânica de ajuste na força de impulso na camada de locomoção. |
| Aprovação | 90% |

## 7.3 CONSIDERAÇÕES SOBRE OS RESULTADOS

Esses resultados foram obtidos com as adaptações informadas no capítulo sobre a arquitetura do projeto. Infelizmente, não foi possível obter um conjunto de resultados relativo à abordagem genérica do *Steering Behaviors* pois a versão do projeto que utilizava esta abordagem foi sobre-escrita pela implementação customizada para movimentos intermitente.

Ciente disto, este projeto irá disponibilizar o código fonte bem como todos os recursos necessários para a execução do projeto, a fim de possibilitar a obtenção de outros conjuntos de resultados que poderão ser comparados com os documentados aqui neste trabalho.

# 8. CONCLUSÃO

Espera-se que com a elaboração deste trabalho, o objetivo de se explicitar os desafios de se aliar um movimento intermitente com um algoritmo de *steering behavior* tenha sido alcançado. A contribuição deste tema para a área é destacar esse atrito entre as camadas de *steering* e de locomoção e sugerir algumas adaptações que embora não sejam complexas, podem ajudar a mitigar esses problemas. Assim, futuros desenvolvedores que se deparem com essa situação já podem ter um ponto de partida para lidar com o desafio.

Os resultados dos cenários de teste realizados mostram que ainda existe muita margem para melhorias. Em particular, os desvios entre os próprios agentes mostraram um problema difícil de se resolver no contexto do movimento convencionado. A execução simultânea dos algoritmos de busca e fuga de maneira concorrente também se mostrou um problema desafiador para o qual não foi encontrado uma solução eficaz. No mais, nos testes que focam em desvio de obstáculos estáticos na busca e fuga de objetivos, o desempenho foi bastante satisfatório, dadas as restrições do movimento adotado.

Pensando em trabalhos futuros, existe uma imensa gama de algoritmos mais complexos que podem ser incluídos nos comportamentos dos agentes. Mecanismos de predição para evitar colisões, lógica *fuzzy* e testes de exclusão para desvio obstáculos. Um pipeline para tomada de decisões para auxiliar a mente dos agentes. Tais mecanismos podem melhorar ainda mais os resultados obtidos nos cenários de teste executados. Nesse sentido, os resultados obtidos pelos testes realizados neste trabalho serão de imensa ajuda para avaliar a melhorias obtidas.

# REFERÊNCIAS

[1] Understanding Steering Behaviors. Disponível em: https://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732. Acesso em: 05 dez. 2016.

[2] DEMONT, M. Edwin; GOSLINE John M. **Mechanics of Jet Propulsion in the Hydromedusan Jellyfish, Polyorchis Pexicillatus**: I. Mechanical Properties of the Locomotor Structure. Journal of Experimental Biology 1988 134: 313-332.

[3] Locomotion in Water. Disponível em: <http://w3.marietta.edu/~mcshaffd/aquatic/sextant/loco.htm> Acesso em: 05 dez. 2016.

[4] GEMMELL, Brad J.; COSTELLO, John H.; COLINA, Sean P.; STEWART, Colin J.; DABIRIE, John O.; TAFTID, Danesh; PRIYA, Shashank. **Passive energy recapture in jellyfish contributes to propulsive advantage over other metazoans**.

[5] BEHAVIORS, Steering. Disponível em:<http://www.steeringbehaviors.de/> Acesso em: 05 dez. 2016.

[6] Chapter 6. **Autonomous Agents**. Disponível em http://natureofcode.com/book/chapter-6-autonomous-agents/. Acesso em: 05 dez. 2016.

[7] REYNOLDS, Craig W. **Steering Behaviors for autonomous characters**. 1998.

[8] Understanding Steering Behaviors: Seek. Disponível em: https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-seek--gamedev-849. Acesso em: 05 dez. 2016.

[9] Unity. Disponível em: https://unity3d.com/unity. Acesso em: 05 dez. 2016.

[10] Unity: The Inspector. Disponível em: https://unity3d.com/learn/tutorials/topics/interface-essentials/inspector. Acesso em: 05 dez. 2016.

[11] Vector3.Lerp. Disponível em: https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html. Acesso em: 05 dez. 2016.

[12] SINGH,Shawn; KAPADIA, Mubbasir; REINMAN, Glenn; FALOUTSOS, Petros. **Watch Out!** A Framework for Evaluating Steering Behaviors. 2008.