UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ERIK DE SOUZA SCHEFFER

Controle de Movimento em Grupo e Coordenação de Dispersão Autônomos de Veículos para Trechos de Animações no Simulador Virtual Tático do Sistema de Simulação do Projeto ASTROS 2020

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
2016

RESUMO

O avanço da tecnologia nos últimos anos, assim como o avanço das ferramentas de desenvolvimento, vem tornando ferramentas de simulação cada vez mais complexas, assim como mais comuns. Tais ferramentas de simulação são usadas em diversas áreas, muitas vezes para treinamento virtual antes do real, visando diminuir custos e riscos do treinamento real. Sistemas complexos ou perigosos, como treinamento de equipamentos militares, se beneficiam de simuladores uma vez que o uso de equipamentos custa caro assim como há riscos à saúde dos envolvidos em casos de erros, como erros causados por inexperiência e falta de treinamento. Buscando se aproveitar dessas vantagens de economia e segurança trazidas por simuladores, o Exército Brasileiro está desenvolvendo um simulador tático para o treinamento do sistema ASTROS 2020. O objetivo desse trabalho é realizar a prototipação de um simulador virtual tático para uma parte do sistema de simulação de lançadores de mísseis e foguetes, ASTROS 2020. São estudadas técnicas para simular comportamentos do mundo real de veículos autômatos. O comportamento estudado busca a criação de uma inteligência para que os veículos autômatos sejam capazes de se deslocarem em comboio ordenado e desviar de obstáculos ao longo do percurso, assim como se dispersarem ao chegarem ao destino. O simulador para o sistema ASTROS 2020 está sendo desenvolvido na plataforma Unity, uma das ferramentas que facilitam o desenvolvimento de jogos e simuladores. O comportamento foi, desse modo, implementando na plataforma Unity e, para tal, as ferramentas da plataforma usadas para o desenvolvimento são, também, estudadas nesse trabalho.

Palavras-chave: Simulação. Inteligência artificial. Unity. Veículos autômatos.

Virtual simulator to simulation system of ASTROS 2020's project

ABSTRACT

The advance of technology in the last years, like the advance of development tools, have made simulation tools more complex and more common. Such simulation tools are used in many areas, usually for virtual training before real training, seeking reduce costs and risks of real training. Complexes or dangerous systems, like military equipment training, benefits from simulation since the use of such equipment is expensive and there are health risks to the ones involved in case of mistakes, like mistakes caused by inexperience and lack of training. Seeking take advantage of this economic and safety advantages brought by simulators, the Brazilian Army is developing a tactical simulator for the training of ASTROS 2020 system. The goal of this paper is to make the prototyping of a tactic virtual simulator for a part of the simulation system of missile and rocket firing system, ASTROS 2020. It is studied techniques to simulate the behavior of real world automatons vehicles. The studied behavior seeks the creation of an intelligence that makes the automatons vehicle capable of move orderly and avoid obstacles along the way as well disperse when arriving in the destination. The simulator for the ASTROS 2020 system is being developed in the Unity platform, one of the tools that make the development of games and simulators easier. So, the behavior was implemented on Unity platform and the platform's tools are also studied in this paper.

Keywords: Simulation. Artificial intelligence. Unity. Automatons vehicles.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artificial
REOP	Reconhecimento, Escolha e Ocupação de Posição
VANT	Veículos Aéreos Não-Tripulados
PEE	Projeto Estratégico do Exército

SUMÁRIO

1 INTRODUÇÃO

Um jogo eletrônico ou simulador gráfico é composto de muitas partes que trabalham em conjunto. É preciso tratar as entradas fornecidas pelo usuário assim como dar respostas adequadas, fazer a renderização da parte gráfica, fazer uma inteligência artificial para eventuais agentes autômatos, entre outros. Todas essas partes também precisam trabalhar juntas de maneira adequada para dar ao usuário a experiência desejada.

Com o avanço da tecnologia nos últimos anos, esses jogos e simuladores têm se tornado cada vez mais complexos. Com o avanço das ferramentas para o desenvolvimento desses, no entanto, está mais fácil fazer com que todas essas partes trabalhem juntas e essas ferramentas automatizam muito do processo de desenvolvimento, tornando o desenvolvimento mais rápido, fácil e barato. Essa facilidade em desenvolvimento de jogos e simuladores vem tornando-os cada vez mais comuns. Diversas áreas, como medicina e militar, estão utilizando simuladores para treinamentos, pois é mais barato e seguro fazer treinamento virtual antes de colocar a atividade em prática. O Exército Brasileiro está se aproveitando disso e está desenvolvendo um simulador para treinamento militar para o Projeto Estratégico ASTROS 2020.

Simulação de física e iluminação são exemplos de processos que são automatizados por ferramentas de desenvolvimento para aplicações gráficas, não sendo necessário o programador implementar tais funções. Mesmo modelos e animações, que muitas vezes são específicos de cada aplicação, podem, muitas vezes, ser encontrados disponíveis na internet. O que fica a cargo do desenvolvedor é, principalmente, a lógica por trás da aplicação. Essa lógica inclui elementos como a interação com o usuário e como os elementos presentes na aplicação reagem às entradas do usuário, como algum tipo de inteligência.

A significância da interação com o usuário em relação à significância da reação dos elementos é dependente do tipo de aplicação. Em jogos eletrônicos, por exemplo, é esperado que os elementos reajam a cada ação do usuário, mas a interação do usuário com a aplicação é significativa e constante, sendo muito importante. Já em simuladores táticos, como o simulador para o projeto ASTROS 2020, a entrada do usuário consiste em um conjunto de ações as quais os elementos do simulador devem executar de maneira adequada, se assemelhando o melhor possível a comportamento de elementos reais; desse modo a inteligência desses elementos para cumprir tais ações é muito importante.

Esse trabalho foca em um aspecto dessa inteligência para o simulador do projeto ASTROS 2020. Entre as muitas ações que são executadas por elementos do simulador, o trabalho se concentra em buscar uma inteligência para o deslocamento de grupos ordenados de veículos e sua dispersão ao chegar no destino. São estudada s maneiras de fazer com que os veículos se locomovam em um comboio tentando se manter em formação enquanto desviam de obstáculos e evitam colidirem uns com os outros.

O simulador está sendo desenvolvido em cima da plataforma Unity. Essa plataforma é uma das ferramentas que facilitam o trabalho do desenvolvedor para a criação de jogos e simuladores. Nesse trabalho, as principais ferramentas que ela oferece que são utilizadas para o desenvolvimento da inteligência para os veículos autômatos são estudadas. Busca-se entender como essas ferramentas funcionam e explica-se como elas foram usadas para o desenvolvimento da inteligência artificial para os veículos.

2 TRABALHOS RELACIONADOS

Movimentação de grupos ordenados é algo comum em jogos de estratégias, nos quais há a necessidade de coordenar grupos de entidades autômatos (podendo variar em entidades como bípedes, quadrupedes, veículos motores, entre outros) para se moverem de um ponto a outros. O trabalho de SILVEIRA et al. (2010) apresenta uma solução para esse problema usando campo potencial. O trabalho propõe a criação de um campo potencial onde obstáculos possuem um potencial alto e lugares livres, contendo o objetivo, possuem um potencial baixo. Para calcular esse potencial, o espaço de solução é discretizado em uma grade regular onde cada célula é associada a uma região quadrada do ambiente e armazena o potencial calculado nessa região.

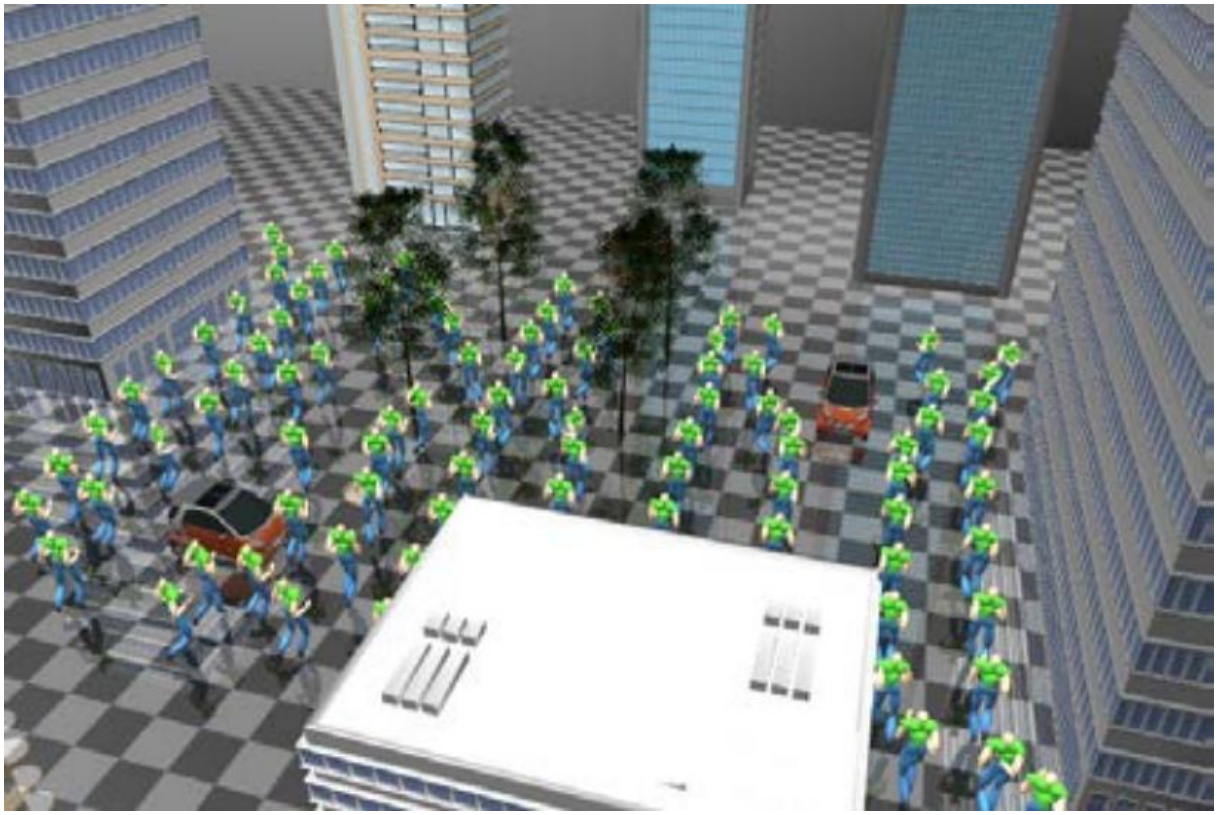
O algoritmo usa um mapa global para encontrar um caminho livre e mapas locais para cada agente para tratar do *steering* e obstáculos dinâmicos. Usando esses mapas, o trabalho mostra bons resultados em encontrar um caminho no ambiente semelhante a um caminho gerado por um agente real, como uma pessoa. Para lidar com grupos, é criado um mapa para os grupos e os agentes são mapeados para posições nesses grupos, sendo atraídos por essas posições, mantendo os agentes na formação e evitando os obstáculos.

A solução proposta por esse trabalho apresenta bons resultados, mas é dependente do mapa potencial criado para o ambiente. O mapa do simulador para o projeto ASTROS 2020 é muito grande e a geração do mapa potencial pode levar tempo demais, então foi buscada uma solução que seja independente do tamanho do mapa.

Outro trabalho que se propõe resolver esse problema de movimentação em grupos é o trabalho HENRY; SHUM; KOMURA (2012). A formação do trabalho é definida por uma malha deformável computada pelo esquema de deformação *as-rigid-as-possible* (mais rígido possível, em tradução livre); é o formato dessa malha define o formato da formação.

Para que os agentes evitem os obstáculos do ambiente, os obstáculos que devem ser evitados possuem campos potenciais. Esses campos potenciais exercem força em cima dos pontos da malha que representam as posições dos elementos da formação, fazendo com que os elementos se afastem dos obstáculos. Conforme os campos potenciais forçam os elementos da formação a se afastarem dos obstáculos, a malha faz com que os elementos se mantenham unidos da melhor maneira possível. A Figura 1 mostra um grupo de agentes em uma cidade, mostrando que eles tentam se manter em formação e evitam os obstáculos próximos.

Figura 1 – Resultado de um grupo em uma cidade do trabalho de HENRY; SHUM; KOMURA (2012).



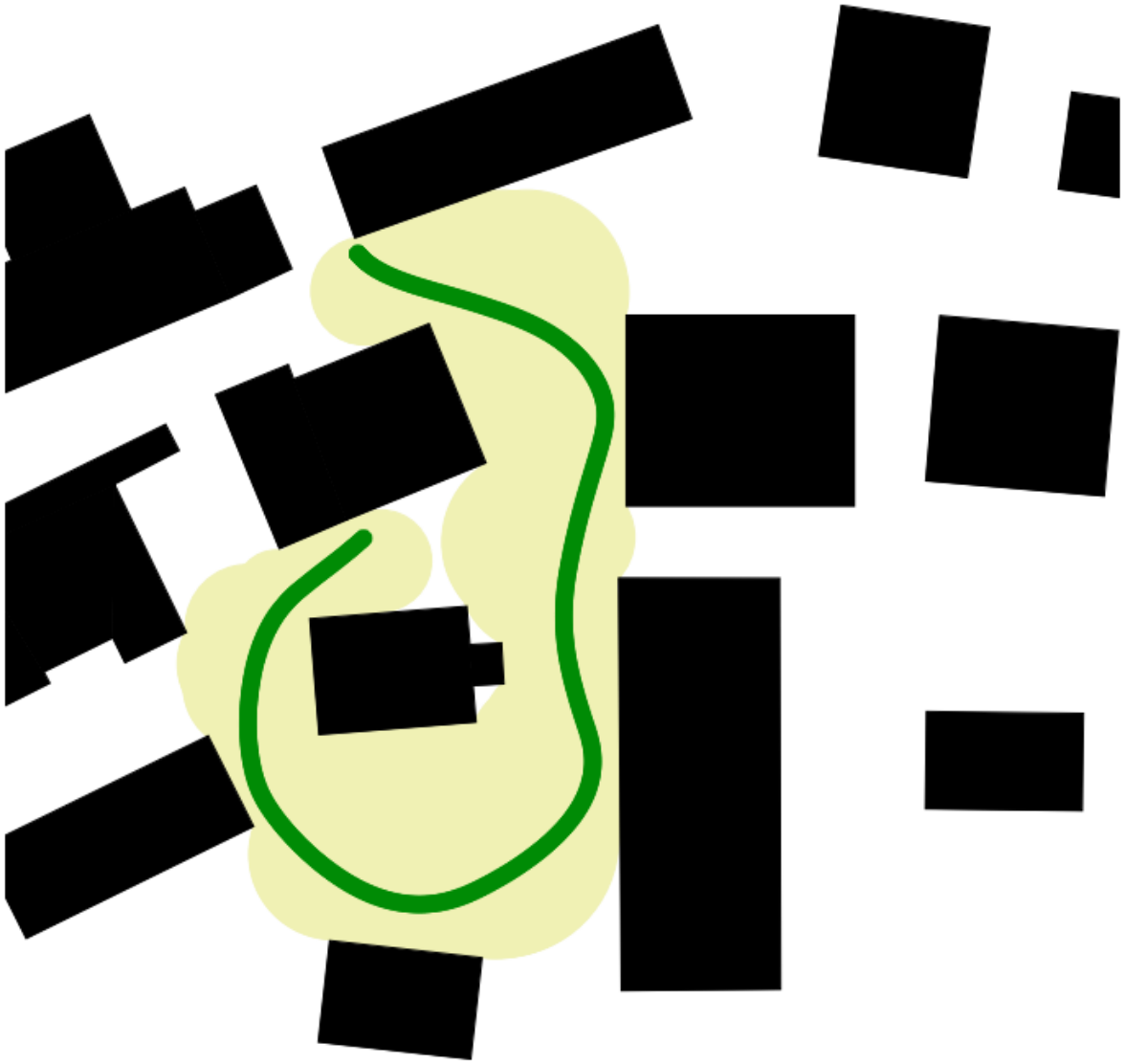
Fonte: (HENRY; SHUM; KOMURA, 2012)

Para calcular essas forças dos obstáculos, o mapa é dividido em uma grade de células de mesmo tamanho e é calculado o campo potencial de cada célula. Isso traz de volta o problema do tamanho do mapa, ficando custoso demais calcular o campo potencial de cada célula. E, também, o trabalho foca na interação do usuário para a manipulação da malha para definir a formação, apresentando ideias e técnicas de interação com usuário em conjunto com uma tela sensível ao toque para permitir ao usuário definir o formato final da malha. O trabalho aqui desenvolvido foca na inteligência para o deslocamento de veículos em uma formação pré-definida.

O trabalho de KARAMOUZAS; GERAERTS; OVERMARS (2009) propõe uma solução de três etapas para o planejamento de caminho. As etapas são a criação de uma rota, a criação de um corredor ao redor da rota e a extração de um caminho do corredor usando campos potenciais para gerar um caminho natural. O trabalho foca nas duas últimas etapas.

Para a criação dos corredores, o trabalho cria um esqueleto de área livre para o mapa extraído do Diagrama Generalizado de Voronoi. Esse esqueleto é usado para criar o corredor e esse corredor é usado para criar um caminho mais natural para o agente percorrer. Para tornar esse caminho mais natural, o trabalho também adiciona um erro nesse caminho, evitando que o caminho tenha partes completamente retas muito longas, tornando-o mais natural. Para fazer o desvio de obstáculos menores, o caminho é calculado de modo que os obstáculos geram forças que fazem o caminho passar em volta e para evitar colisões entre os elementos em deslocamento. Para evitar colisão entre os elementos em deslocamento, cada elemento gera uma força que afasta os demais. A Figura 2 mostra o caminho final gerado pelo trabalho, com suas correções para torná-lo mais natural.

Figura 2 – Caminho final do trabalho de KARAMOUZAS; GERAERTS; OVERMARS (2009)



Fonte: (KARAMOUZAS; GERAERTS; OVERMARS, 2009)

O trabalho mostra que suas técnicas funcionam, mas traz, mais uma vez, o problema da necessidade de ter uma estrutura pré-calculada para o mapa e também não apresenta a definição de uma formação, necessária para o caso do simulador. As técnicas usadas no trabalho para desvio de obstáculos e evitar colisão entre elementos próximos são semelhantes às utilizados no trabalho aqui desenvolvido.

Esses três trabalhos citados mostram resolver o problema ao qual se propõem a resolver. No entanto, esses trabalhos, assim como outros nessa mesma área, necessitam de uma estrutura pré-calculada para o mapa no qual os agentes se deslocarão. Em muitos casos isso não representa problemas. Em jogos de estratégias, por exemplo, os mapas podem ser grandes, mas seu tamanho, ainda assim, é limitado uma vez que quaisquer pré-configurações necessárias para o mapa, como as descritas nesses trabalhos, não podem demorar demais, pois um tempo de demora grande demais não vai agradar o usuário. Outro fator limitante nesses casos é a capacidade do usuário. Em jogos assim, o usuário precisa ter o maior controle possível de todas as suas tropas espalhadas por todo o mapa, de modo que, se o mapa for grande demais, o usuário pode não conseguir dar conta de cuidar de todo o mapa. No caso do simulador para o projeto do sistema ASTROS 2020, o mapa na qual se é trabalhado é muito grande, de modo que fazer cálculos para preparar o mapa pode levar tempo demais. Para contornar esse problema, o trabalho aqui desenvolvido utiliza de modelos propostos por Craig W. Reynolds para criar uma inteligência para o deslocamento que não precise de uma estrutura calculada em cima do mapa, sendo independente do tamanho do mapa.

O trabalho AOYAGI; NAMATAME (2005) busca fazer algo semelhante ao aqui proposto. Também utilizando de base o modelo proposto por Craig W. Reynolds, o trabalho propões adaptações e complementos ao modelo de Reynolds para atingir o seu objetivo, no entanto, o trabalho visa simular o comportamento de um grande número de veículos aéreos, enquanto o trabalho aqui desenvolvido é para veículos terrestres em um grupo menor.

3 EMBASAMENTO TEÓRICO

Para o desenvolvimento do simulador desse projeto, foi estudado sobre simulação assim como técnicas para a criação de uma inteligência artificial para simular o deslocamento ordenado de um grupo de veículos. Esse capítulo apresenta o embasamento teórico para o que foi implementado.

3.1 SIMULAÇÃO

Certas atividades podem ser muito custosas para se treinar, seja em custo material ou custo em vidas. O treinamento de soldados é um exemplo que causa desgaste do material e um risco para a vida dos soldados em caso de acidentes. Apesar do custo, tal treinamento é necessário, pois, sem ele, o custo pode vir a ser ainda maior quando a atividade viesse a ser posta em prática e viesse a fracassar devido à falta de treinamento. Simulação é um modo de reduzir tais custos e, assim como as guerras, a necessidade de simular não é recente. O xadrez, que agora é apenas um jogo, é um exemplo de simulador, por mais simplificado que seja, que permitia a prática de estratégias e criatividade para batalhas.

Durante a primeira guerra mundial, ficou-se ainda mais evidente a necessidade de simuladores, pois era necessário o treinamento de muitos pilotos em pouco tempo e cada acidente causava prejuízo com a perda do avião, a morte de um futuro piloto e o desperdícios das horas gastas no treinamento com esse piloto. Como o número de acidentes era grande, o prejuízo também ficava muito grande. Para diminuir o número de acidentes causados por inexperiência, foram criados simuladores mecânicos para oferecer treinamento prático para os pilotos antes de usarem as naves reais.

Foi durante a segunda guerra mundial, com o Mark I e o ENIAC, que simulação eletrônica mais parecida com o que temos hoje começou a ser empregada. Muito inferiores aos computadores atuais, os computadores da época eram usados para cálculos balísticos buscando simular o lançamento de mísseis. No entanto, ainda era muito caro a criação de simuladores, pois apenas grandes corporações e universidades possuíam máquinas suficientemente potentes para tais simulações e os desenvolvedores e operadores necessitavam possuir um conhecimento elevado, o que encarecia ainda mais a produção de simuladores. Além de todo o custo, as

limitações tecnológicas da época não permitiam a criação de simuladores com a flexibilidade que se tem hoje.

Com o aumento no poder de processamento e barateamento dos computadores ao longo dos anos, a simulação foi se difundindo, mostrando-se útil em muitas áreas além das áreas militares. Áreas como medicina e automobilística estão usando simuladores para práticas e testes antes de pôr tudo em prática real e a simulação na área militar vai muito além de apenas cálculos balísticos (BALADEZ, 2012).

De forma mais técnica, essas simulações podem ser classificadas em três estilos: viva, virtual e construtiva, podendo também ser uma combinação de dois ou mais desses estilos. Dentro desses estilos, as simulações podem ser científicas, onde a interação consiste apenas em observações e medições, ou envolver interação com humanos.

A simulação viva envolve humanos e equipamentos em uma atividade real, em tempo real, como um jogo de guerra em que os soldados vão ao campo praticar. A simulação virtual envolve humanos e equipamentos em um ambiente computacional controlado, onde o tempo não necessariamente corresponde ao tempo real, permitindo ao usuário concentrar-se nos pontos mais importantes da simulação. Um exemplo de simulação virtual é um simulador de voo. Simulação construtiva, ao contrário dos outros estilos, não costuma envolver humanos ou equipamentos e em vez de passagem do tempo, ele é comandado por uma sequência de ações. A antecipação do caminho de um furacão construído por meio da aplicação de temperatura, pressão, correntes de vento e outros fatores climáticos é um exemplo de simulação construtiva (INSTITUTE FOR SIMULATION & TRAINING, 2014).

Dentro desses estilos, o simulador em desenvolvimento pelo Exército Brasileiro se enquadra no estilo de simulação virtual. Os usuários que estão usando o simulador para treinamento precisam interagir com o simulador para que haja um resultado dessa ação no simulador.

3.2 INTELIGÊNCIA ARTIFICIAL APLICADA AO CONTROLE DE MOVIMENTO E DISPERSÃO DE VEÍCULOS AUTÔNOMOS

“Inteligência Artificial (IA) é um ramo da ciência da computação que se propõe a elaborar dispositivos que simulem a capacidade humana de raciocinar, perceber, tomar decisões

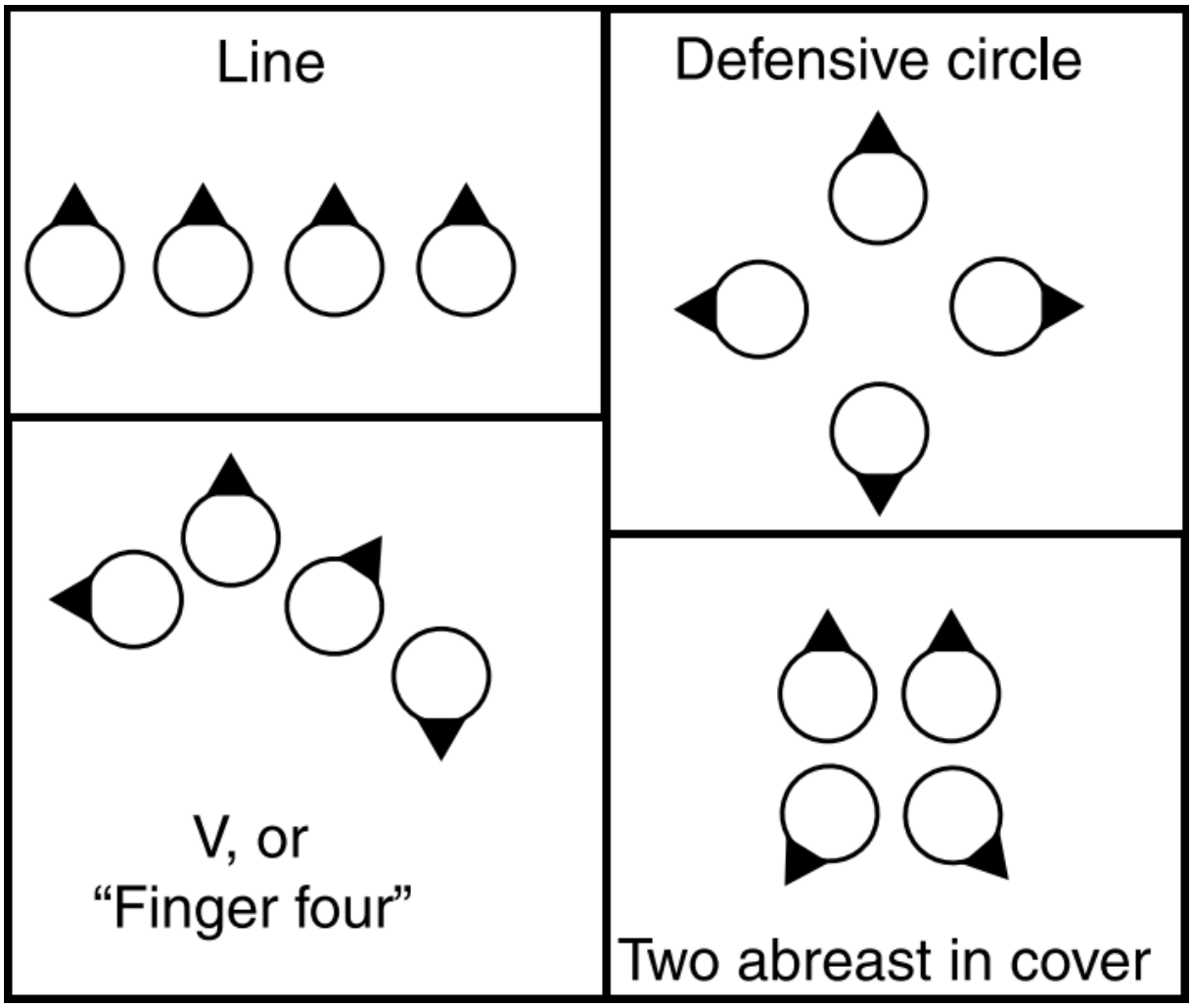
e resolver problemas, enfim, a capacidade de ser inteligente. [...] ” (CIRIACO, 2008) Na área de jogos eletrônicos e simuladores, a inteligência artificial é utilizada para que agentes atuem de maneira que pareçam possuir inteligência. Esses agentes podem ser simples, como um inimigo que persegue o personagem de um jogador, ou complexos, como uma inteligência para comandar um exército em jogos de estratégia, dependendo do contexto. O que será estudado neste projeto será o segundo caso: a criação uma IA para veículos autômatos que devem ser capazes de andar juntos de forma ordenada.

3.2.1 Formação

A forma ordenada com a qual o comboio de veículos anda juntos é uma formação. A formação do comboio é simples, consistindo de um conjunto de pontos que estão relacionados entre si. A relação desses pontos tem como base um ponto escolhido como líder. Se um ponto está localizado a r_s em relação ao slot líder, então a posição do personagem que deve ficar nesse ponto é $p_s = p_l + \Omega_l r_s$ onde p_s é a posição final do ponto *s*, p_l é a posição do líder e Ω_l é a orientação do líder na forma de matriz. Similarmente, a orientação do personagem nesse ponto é $\omega_s = \omega_l + \omega_s$ onde ω_s é a orientação do ponto *s*, relativa à orientação do líder, e ω_l é a orientação do líder. Dependendo de como essa relação entre o líder e os demais pontos da formação é definida, a formação pode possuir diversos formatos. Alguns exemplos de formatos de formação são mostrados na Figura 3, não sendo limitado a apenas essas formações nem a essa quantidade de posições mostradas na figura. Dessa forma, basta o líder se mover pelo espaço que os demais veículos do comboio se movem seguindo a formação (MILLINGTON, 2006).

Tendo definido a relação entre o líder e o resto da formação, é preciso, então, definir o líder. Uma opção seria escolher um dos veículos como líder, desse modo, basta o veículo líder se mover ao longo do terreno que o resto dos veículos da formação seguirão em suas respectivas posições na formação dentro do comboio. No entanto, essa opção pode causar efeitos indesejados. Se o líder encontrar um obstáculo, por exemplo, ele precisará fazer um movimento para desviar desse obstáculo e, como todos os veículos da formação estão se mantendo a uma mesma distância e rotação em relação ao líder, todos os veículos farão esse mesmo movimento de desvio, mas é somente o líder quem precisa desviar do obstáculo. Sendo assim, é preciso encontrar uma outra solução.

Figura 3 - Exemplos de formações.



Fonte: (MILLINGTON, 2006)

A solução mais adequada é escolher um ponto arbitrário, como, por exemplo, o centro de todas as posições da formação, e fazer com que este ponto seja o líder. Esse ponto não é um veículo, então ele pode atravessar obstáculos, mantendo o curso de maneira desimpedida, deixando que cada um dos veículos cuide de desviar os obstáculos que cada um encontrar.

3.2.2 Veículo

Com a formação definida, é preciso, então, definir uma inteligência para os veículos que ocupam as posições na formação. Cabe aos veículos se manterem em suas posições na formação enquanto essa se move, assim como evitar obstáculos ao longo do caminho e evitar colidir com

outros veículos. Para isso, foi usado de base um modelo computacional criado por Craig W. Reynolds.

Em 1987, Craig W. Reynolds apresentou um modelo computacional para simular a movimentação de grupos de animais, como bandos de pássaros e cardumes de peixes (REYNOLDS, 1987). Reynols chamou a criatura genérica sendo simulada pelo modelo de *boid*. O modelo consiste em três comportamentos de direção desejado para cada *boid* seguir. Esses comportamentos são: separação, coesão e alinhamento.

Quando se movimentando em grupo, os membros do grupo procuram evitar colisões (pássaros, por exemplo, podem cair caso colidam com outros membros do bando). Para evitar essa colisão entre membros do grupo, o comportamento de separação diz para cada *boid* evitar os *boids* do grupo que estiverem próximos, movendo-se para longe, em direção oposta aos *boids* próximos, evitando se colidir com esses *boids*. Caso haja mais de um *boid* próximo, pega-se o ponto médio de todos os *boids* como referência.

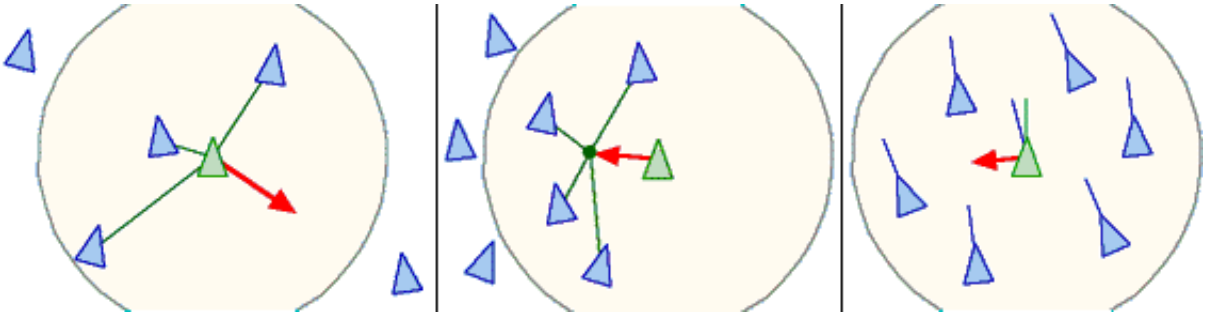
Apesar de haver necessidade de evitar colisões entre os membros do grupo, os membros ainda desejam fazer parte do grupo e se manterem próximos. O comportamento de coesão diz para cada *boid* ir em direção ao ponto médio dos *boids* próximos, buscando ficar a mesma distância de todos os *boids*.

Uma vez que a colisão é evitada e o grupo não se separa, este grupo precisa ir para algum lugar. O comportamento de alinhamento pega a direção atual de todos os *boids* próximos ao *boid* atual e obtém uma direção média de todas essas direções. O comportamento, então, diz para o *boid* mover-se nessa direção média dos *boids* próximos, buscando fazer com que o *boid* atual mova-se na mesma direção dos demais *boids* próximos, fazendo com que o grupo, como um todo, mova-se em uma mesma direção.

O cálculo de cada um desses comportamentos é feito separadamente para cada um dos *boids* do grupo e gera um vetor com uma direção e essa direção é para onde cada *boid* deve se mover para respeitar o comportamento calculado. A direção de cada um desses vetores resultantes vai depender dos *boids* que estiverem próximos, na vizinhança. Um *boid* está na vizinhança de outro *boid* se a distância entre os esses *boids* for menor do que uma distância limite. Essa distância limite é arbitrária, podendo ser pequena ou se estender para incluir todos os membros do grupo, podendo, também, ser diferente para cada um dos comportamentos. O tamanho dessa vizinhança pode ser diferente para cada comportamento depende da aplicação.

A Figura 4 demonstra um exemplo de cada um dos comportamentos. Os triângulos são os *boids*, sendo o triângulo verde o *boid* cujo comportamento está sendo representado e os triângulos azuis são outros *boids* do grupo, usados ou não no cálculo dos comportamentos. O círculo representa a vizinhança do comportamento que diz quais *boids* considerar; *boids* fora desse círculo estão fora da vizinhança e não são considerados para o cálculo do comportamento. A seta vermelha representa a força que deve ser aplicada ao *boid*, em verde, para que este passe a respeitar o comportamento.

Figura 4 - Comportamentos de um *boid*. Da esquerda para direita: Separação, Coesão e Alinhamento.



Fonte: (REYNOLDS)

Com o tamanho das vizinhanças (que varia dependo do comportamento e da aplicação) definido e cada um dos três vetores calculados, é preciso combinar esses vetores, uma vez que o *boid* pode seguir em apenas uma direção. Somando os três vetores, obtém-se um vetor com a direção desses três vetores combinados, que é a direção final que o *boid* deve se mover para seguir o modelo. Como a soma leva em consideração o tamanho dos vetores, também é possível dar pesos a cada comportamento, multiplicando o vetor do comportamento por um escalar real antes da soma, aumentando ou diminuindo seu tamanho e, consequentemente, aumentando ou diminuindo, respectivamente, a sua influência no vetor final.

O comportamento, no entanto, não é calculado uma única vez. Conforme os *boids* vão se movendo com o passar do tempo, os comportamentos precisam ser recalculados, uma vez que a quantidade de *boids* dentro da vizinhança pode mudar, assim como a posição dos *boids* na vizinhança, mudando o vetor de cada comportamento e, consequentemente, mudando o vetor final do modelo. A frequência com a qual os comportamentos são recalculados é variável, podendo ser a cada quadro de uma animação ou em intervalos de tempos fixos regulares, dependendo da aplicação.

Colocando o modelo descrito acima em conjunto com um *steering behavior* (descrito adiante) em objetos que podem ser animados e fazendo com eles se movam na direção que cada um obtém ao calcular os comportamentos em cada interação, é possível obter uma boa simulação de movimento de bando. O simulador em desenvolvimento, no entanto, não é para simular o movimento de animais, mas para o treinamento militar, então o que deverá se mover em grupo são veículos motores e não pássaros ou peixes. Dessa forma, é preciso fazer modificações no modelo para se adequar às necessidades do simulador. As modificações são descritas adiante, onde é explicado tudo o que foi feito para atender as necessidades do simulador.

3.2.3 *Steering Behavior*

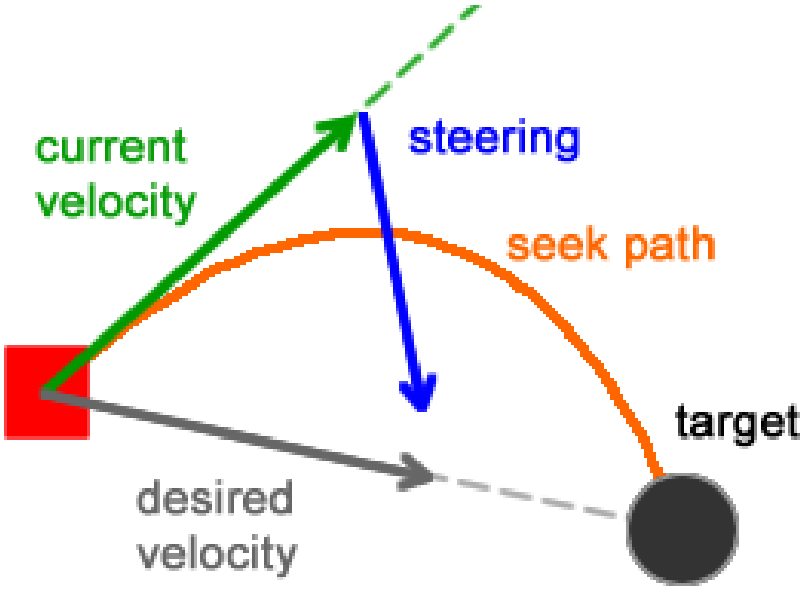
O vetor final gerado pelo cálculo dos comportamentos descrito acima, com as modificações que são descritas adiante, resulta em um vetor com a direção desejada que o veículo deve seguir. No entanto, fazer com que o veículo siga puramente esse vetor pode causar artefatos indesejáveis na hora de mudar a direção atual do veículo.

Quando um obstáculo aparece no caminho, outro veículo se aproxima ou há uma mudança na posição alvo do veículo de modo que o veículo precisa fazer uma curva para mudar de direção. Um veículo real, dada suas limitações físicas, faz uma curva suave; mesmo objetos que não sejam veículos motores, como pessoas ou outros animais, tem limite do quão fechada é a curva que conseguem fazer quando estão em movimento. Desse modo, o veículo simulado precisa fazer curvas suaves, semelhantes às curvas executadas por veículos reais, ao mudar de direção. No entanto, quando há uma mudança no vetor com a direção desejada, o ângulo entre o vetor da direção atual e a nova direção desejada pode ser grande ao ponto de que, se o veículo mudar a rotação instantaneamente para que sua direção seja a direção do novo vetor, a mudança pode não ser natural.

Para corrigir esse problema e fazer com que o veículo faça curvas mais naturais, o vetor com a direção e velocidade atual é combinado com o vetor com a direção e a velocidade desejada. O objetivo dessa combinação é gerar um novo vetor que o veículo possa seguir e que vá, aos poucos, se igualando ao vetor com a velocidade desejada. Desse modo, o veículo se moverá, no final, na direção e velocidade desejada, mas atingirá essa velocidade e direção aos poucos, ao longo do tempo com uma curva suave.

A Figura 5 exemplifica o que se tem e o que se quer. "*Current velocity*" é o vetor que indica a direção e a velocidade atual do veículo (a velocidade é dada pela magnitude do vetor, quanto maior o vetor, mais rápido o veículo se move). "*Desired velocity*" é o vetor com a direção que o veículo quer se mover. Esse é o vetor final gerado pelos comportamentos do modelo Craig W. Reynolds descrito acima e o valor da sua magnitude é limitada com a velocidade máxima que o veículo pode atingir. Como pode ser visto no caso da imagem, a diferença entre a direção do vetor da velocidade atual e a direção do vetor com a velocidade desejada é tal que, se o veículo mudar sua direção atual para a direção desejada em um único quadro, não será um movimento realístico.

Figura 5 – Demonstração do vetor de *steering* e do caminho desejado que ele gera.



Fonte: (BEVILACQUA, 2012)

O que se deseja, então, é o "*Seek path*", que é um caminho mais realista, mas que o vetor calculado pelos comportamentos não gera. Para fazer com que o veículo faça o caminho mais amplo e natural, calcula-se o vetor *steering*: $steering = desiredVelocity - currentVelocity$. Esse vetor é uma força que fará com que o veículo gradualmente mude sua direção até que esta seja igual à direção desejada.

A magnitude desse vetor é a força que será aplicada de modo que, quanto maior a magnitude do vetor *steering*, mais fechada será a curva. Sendo assim, é preciso tratar essa magnitude. Normalmente há uma força máxima que pode ser aplicada para fazer a curva. Um veículo, por exemplo, tem um limite de o quanto suas rodas podem girar e o quão fechado pode

ser a curva, então a magnitude do vetor *steering* tem que ser limitada à um valor máximo de força que pode ser aplicada. Outro fator que influencia na força do *steering* é a massa do objeto. Objetos maiores e mais pesados tendem a ter mais dificuldades em fazer curvas, fazendo-as mais abertas. Desse modo, reduz-se o vetor *steering* baseado na massa do objeto: $steering = \frac{steering}{massa}$.

Com o vetor de *steering* calculado, ele é somado ao vetor da velocidade atual para obter um novo vetor. Esse novo vetor é o vetor que passa a ser a nova velocidade atual do veículo. O veículo passa a ficar voltado na direção desse novo vetor e se moverá na velocidade dada por esse vetor. Como o veículo passa a se mover, ou seja, sua posição muda, e o vetor com a velocidade desejada também muda, uma vez que os comportamentos são constantemente recalculados, o vetor de *steering* também precisar ser recalculado junto com o cálculo dos comportamentos.

3.2.4 Desvio de obstáculos

O vetor de *steering* permite ao veículo fazer curvas mais suaves e realistas, mas outro ponto que o veículo precisa levar em consideração ao se locomover é a presença de obstáculos. O que se deseja nesse projeto não é uma simulação de colisão, mas uma simulação do deslocamento de veículos e, como tal, o obstáculo deve ser evitado sem que haja colisão. Uma vez que o importante não é a colisão, mas evitá-la, os obstáculos são aproximados a esferas para facilitar a detecção e desvio dos obstáculos.

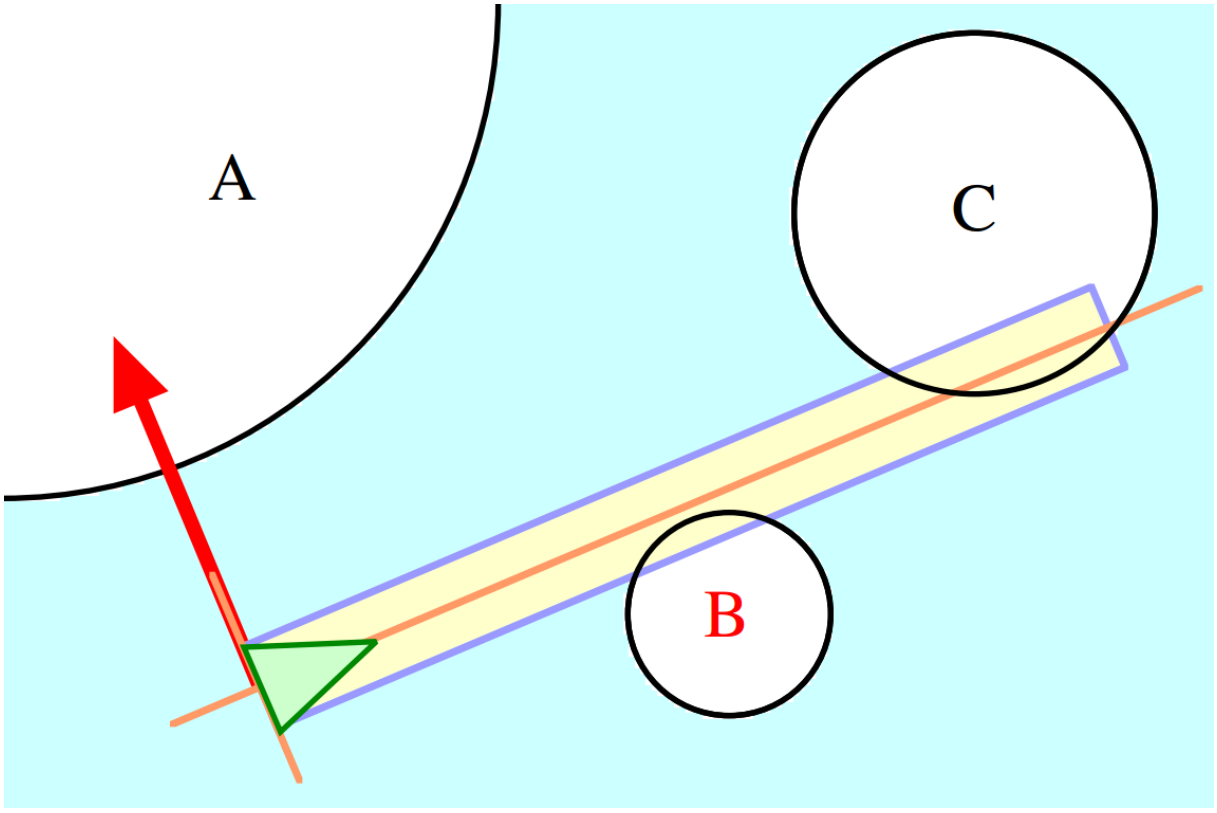
Para evitar obstáculos, o veículo tenta manter uma área retangular livre de obstáculos à sua frente. A largura dessa área deve ser, pelo menos, a largura do veículo e o comprimento da área é a distância com a qual o veículo começa a considerar obstáculos para evitar. Obstáculos fora dessa área não são uma ameaça e são ignorados, desse modo, obstáculos muito distantes, que estão além da área livre e não são uma ameaça imediata, ou obstáculos com os quais o veículo não colidirá caso mantenha a sua rota atual, como obstáculos ao lado ou atrás do veículo, não são considerados.

Com a área que será considerada para o desvio definida, todos obstáculos fora dela podem ser descartados. Dentre os demais obstáculos que não foram descartados, o mais próximo do veículo é selecionado como sendo o mais perigoso e que deve ser evitado

imediatamente. Os demais obstáculos dentro da área serão considerados após esse obstáculo mais perigoso ter sido desviado.

A Figura 6 exemplifica um caso com obstáculos. O triângulo verde representa um objeto que está se locomovendo e precisa desviar de obstáculos ao longo do caminho. Os círculos marcados A, B e C são obstáculos que precisam ser evitados. O obstáculo A não está à frente do objeto, de modo que, se o veículo continuar na direção atual, ele não colidirá com esse obstáculo, sendo assim esse obstáculo não é uma ameaça, desse modo é ignorado. Os obstáculos B e C, no entanto, estão na frente do objeto e estão intersectando a área amarela, que representa a zona que deseja se manter livre de obstáculos. Nesse caso da figura, o obstáculo B é o mais próximo do objeto, então é o obstáculo considerado para se fazer o desvio. No caso da figura, o objeto precisa desviar para a esquerda, como indica a seta vermelha, para desviar do obstáculo B.

Figura 6 – Exemplo de caso de detecção de obstáculo



Fonte: (REYNOLDS, 1999)

Com um obstáculo selecionado para ser evitado, é preciso calcular uma maneira de fazer com que o veículo evite o obstáculo. Para isso, é subtraído o vetor que representa a direção do

obstáculo até o veículo (a direção é do obstáculo para o veículo para que o vetor final faça o veículo se afastar do obstáculo) do vetor que representa a direção e velocidade atual do veículo. Dessa forma, obtém-se um vetor com a direção que deve ser somado ao vetor de *steering* para que o veículo desvie do obstáculo.

3.2.5 Chegada

Um último ponto que precisa ser tratado é a chegada do veículo ao seu destino. Quando o veículo se move, ele busca se mover a uma velocidade máxima e, ao chegar, ele precisa parar, reduzindo sua velocidade a zero. No entanto, o natural é que essa velocidade se reduza lentamente a zero e não instantaneamente.

Para tratar da chegada do veículo e fazer com que a velocidade reduza gradualmente é calculada uma velocidade de corte com a fórmula: $ramped\_speed = maxSpeed * (\frac{targetDistance}{slowingDistance})$. Nessa fórmula "*ramped_speed*" é a velocidade de corte calculada, "*maxSpeed*" é a velocidade máxima definida para o veículo se locomover, "*targetDistance*" é a distância atual do veículo até o destino final e "*slowingDistance*" é a distância determinada na qual o veículo deve começar a desacelerar.

A velocidade máxima que os algoritmos devem usar é, então, o menor valor entre a velocidade máxima definida e a velocidade de corte calculada. Isso garante que a diminuição na velocidade só ocorra quando a distância entre o veículo e o alvo for menor do que a distância determinada para o início da desaceleração. Essa garantia é dada pela divisão $(\frac{targetDistance}{slowingDistance})$ onde o número calculado é maior que 1 quando a distância do veículo até o alvo for maior que a distância de desaceleração (fazendo com que *ramped_speed* tenha um valor maior que *maxSpeed*), e menor que 1 quando a distância entre o veículo e o objetivo final for menor do que a distância de desaceleração (fazendo com que *ramped_speed* tenha um valor menor que *maxSpeed*). Desse modo, pegando o menor valor entre a velocidade de corte e a velocidade máxima, o veículo só começa a desacelerar quando a distância fica menor do que a distância de corte.

Quando a distância entre o veículo e o alvo finalmente fica menor do que a distância determinada para o início da desaceleração, a velocidade máxima que do veículo passa a ser a velocidade de corte calculada. Como a velocidade de corte fica cada vez menor conforme o

veículo vai se aproximando do alvo, a velocidade de corte também vai diminuindo gradualmente, fazendo com que o veículo perca a velocidade aos poucos, sem uma parada brusca.

4 UNITY

A empresa Unity Technologies, com sede em San Francisco, Califórnia, Estados Unidos, foi fundada em 2004 por David Helgason, Joachim Ante e Nicholas Francis. A empresa é criadora da plataforma Unity (CRUNCHBASE). A Unity é uma plataforma para o desenvolvimento de jogos e é o principal software de desenvolvimento de jogos em nível global (UNITY TECHNOLOGIES). A plataforma Unity oferece muitas ferramentas que facilitam o desenvolvimento de jogos e é a plataforma que é utilizada para o desenvolvimento do simulador. A quantidade e variedade de ferramentas que a plataforma Unity oferece para o desenvolvimento do jogo é vasta e pode ser encontrada na sua documentação. Algumas dessas ferramentas fornecidas pela Unity são discutidas a seguir.

4.1 *Game Objects*

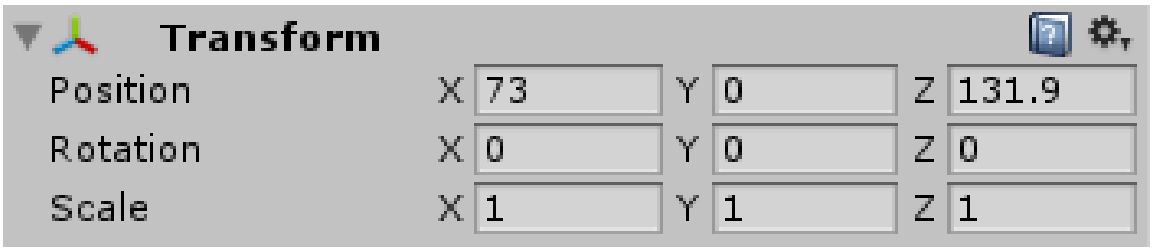
No menu *Game Object – Create Empty* da plataforma Unity é possível criar um *Game Object* vazio. O *Game Object* é a base para o que estará em cena, mas esse não faz nada sozinho. *Game Objects* são containers para outros componentes do jogo (UNITY TECHNOLOGIES). Tudo o que está em cena (visível ou não para o usuário) na Unity está dentro de algum *Game Object*. *Game Objects* podem (ou não) conter elementos como *colliders*, que tratam de colisão, *rigidbodys*, que tratam da simulação física, *scripts*, que são códigos em C# ou JavaScript criados para executar as mais diversas ações, entre outros. Todo *Game Object*, no entanto, deve sempre conter um *Trasnform*, sendo assim um *Game Object* nunca pode estar realmente vazio, mas quando o único componente que ele contém é o seu *Trasnform*, ele é considerado como vazio.

4.2 *Transform*

Transform é um componente que contém a posição, rotação e a escala de um objeto, cada um representado por um vetor 3D, como mostra na Figura 7. Todo o objeto em cena precisa ter uma posição no mundo, assim como uma rotação e um fator de escala e, por essa razão, todo *Game Object* precisa ter um *Trasnform*. É por meio desse componente que os objetos em cena

se movem. Quando, por exemplo, a Unity simula gravidade sobre um objeto e o faz cair, a Unity altera a posição do *Transform* do objeto para fazê-lo cair e aplica eventuais mudanças na rotação do objeto caso o esteja girando durante a queda. Quando o desenvolvedor precisa fazer um objeto se mover, é alterando a posição do *Trasnform* que ele faz o objeto se movimentar pelo mundo do jogo.

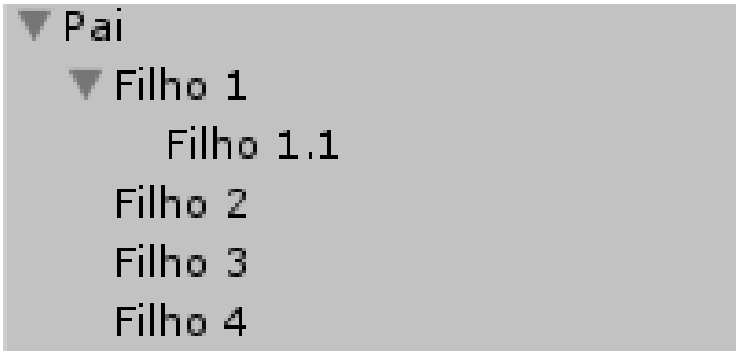
Figura 7 – Exemplo de *Transform* com sua posição, rotação e escala



Fonte: Elaborada pelo autor

Outra propriedade do *Trasnform* é a possibilidade de ter um pai. Todo *Trasnform* pode possuir um pai, permitindo que as manipulações de posição, rotação e escala sejam aplicadas hierarquicamente. Todas as mudanças aplicadas sobre um *Transform* pai são hierarquicamente aplicadas aos *Trasnforms* dos filhos. A Figura 8 demonstra um exemplo de hierarquia de objetos na Unity onde "Pai" é o objeto mais alto na hierarquia de modo que qualquer mudança feita sobre o *Transform* de "Pai" será aplicada, também, a todos os filhos, incluindo o "Filho 1.1". Como as mudanças são aplicadas apenas nos filhos, uma mudança no *Transform* do "Filho 1" mudará o *Transform* do "Filho 1.1", mas não mudará os *Transfoms* dos filhos "Filho 2", "Filho 3" nem "Filho 4", seus irmãos. Da mesma forma, essas mudanças no "Filho 1" não são aplicadas ao "Pai", seu elemento pai.

Figura 8 – Exemple de hierarquia na Unity



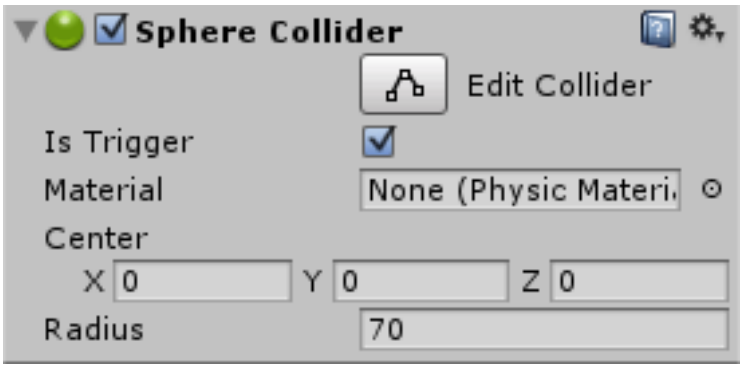
Fonte: Elaborada pelo autor

4.3 *Collider*

Collider é um componente invisível que define a forma de um objeto para o propósito de colisões (UNITY TECHNOLOGIES). Os *Colliders* que podem ser criados na Unity são *Box Collider*, *Capsule Collider*, *Mesh Collider*, *Sphere Collider*, *Terrain Collider*, *Wheel Colldier* e *World Particle Collider*. Esses *Colliders* possuem formas diferentes e alguns são feitos para funções específicas dentro da área de colisão.

A Figura 9 mostra um exemplo de um *Sphere Collider* (um *Collider* em formato esférico) com os campos que podem ser configurados. "*Is Trigger*" define se o *Collider* é ou não uma *Trigger Zone*, "*Material*" indica o material que a Unity vai usar para simular a colisão (madeira, metal, borracha, entre outros), "*Center*" indica o centro do *Collider* em relação ao *Transform* do *Game Object* ao qual esse *Collider* foi adicionado (o centro em (0,0,0) indica que o centro do *Collider* é igual a posição do *Transform*) e "*Radius*" indica o valor do raio da esfera do *Collider*, de modo que quanto maior é esse raio maior é o tamanho do *Collider*. Cada *Collider* possui os seus campos, sendo alguns campos podendo aparecer em mais de um *Collider*. O *Box Collider* (um *Collider* paralelepipédico), por exemplo, possui esses mesmos campos, com exceção do campo "*Radius*" que é substituído pelo campo "*Size*" que consiste em um vetor 3D com o tamanho de cada uma das dimensões do *Box Collider*.

Figura 9 – Exemplo de um *Sphere Collier*



Fonte: Elaborada pelo autor.

Quando dois *Colliders* entram em contato ocorre, então, uma colisão. Essa colisão pode ser tratada manualmente pelo desenvolvedor por meio de *scripts* ou a Unity pode usar o seu motor de física para simular a física de colisão. No entanto, para que a Unity trate a da simulação

de física da colisão, o objeto também precisa possuir um *Rigidbody*. É o *Rigidbody* que define propriedades como massa do objeto; o *Collider* define, apenas, o formato do objeto para a colisão.

Outra forma com a qual os *Colliders* podem ser usados é como "*Trigger Zone*" (zona de gatilho em tradução livre). Ao marcar um *Collider* com a propriedade "*Is Trigger*", o *Collider* passa a não interagir fisicamente com outros objetos, atravessando-os ou sendo atravessado, permitindo usar o *Collider* como uma "*Trigger Zone*" (UNITY TECHNOLOGIES). Com o *Collider* transformado nessa *Trigger Zone*, é possível detectar sempre que outro objeto contendo um *Collider* entra ou sai da *Trigger Zone*, pois, quando outro *Collider* entrar e sai da *Trigger Zone*, funções especificas para cada caso são chamadas contendo o *Collider* que disparou o evento como parâmetro. Isso permite ao desenvolvedor decidir o que deve acontecer entre os *Colliders* colidindo quando ocorre a colisão.

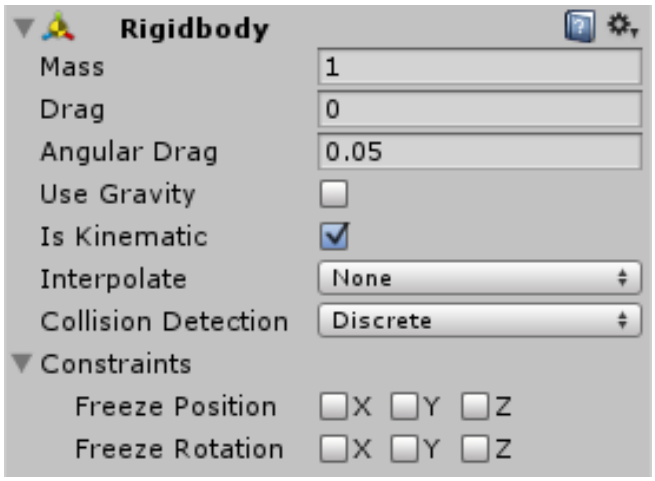
4.4 *Rigidbody*

O *Rigidbody* é um componente que controla a posição de um objeto por meio de simulação física. Um *Game Object* que possui um *Rigidbody* passa a ter seu movimento controlado pelo motor de física da Unity. Desse modo, o objeto passa a ser influenciado pela gravidade e, caso ele também possua um *Collider*, colidirá com outros objetos simulando interações físicas, como deformação na sua velocidade causada por uma colisão. O *Rigdbody* também possui funções que permitem que sejam adicionadas forças manualmente ao objeto, como uma maneira alternativa à modificação direta do *Transform* para fazer a movimentação do objeto.

A Figura 10 mostra um exemplo de um *Rigidbody* com os campos que podem ser configurados. "*Mass*" define a massa do objeto, em quilogramas por padrão. "*Drag*" indica o quanto a resistência do ar afeta o objeto quando movido por outras forças (0 significa sem resistência do ar e infinito faz o objeto para imediatamente). "*Angular Drag*" indica o quanto a resistência do ar afeta a rotação causada por torção (0 significa que não há resistência do ar). "*Use Gravity*" faz com que, se habilitado, o objeto seja afetado pela gravidade. "*Is Kinematic*" faz com que, se habilitado, o objeto não seja afetado pelo motor de física da Unity, podendo ser apenas manipulado pelo seu *Transform*. "*Interpolate*" possui as opções "*Interpolate*", que faz

o *Transform* ser suavizado baseado no *Transform* quadro anterior, "*Extrapolate*", que faz o *Transform* ser suavizado baseado no *Transform* estimado para o próximo quadro, e "*None*", que não aplica nenhuma interpolação. "*Collision Detection*" define o tipo de detecção usada para a colisão, podendo ser "*Discrete*", para usar detecção de colisão discreta, "*Continuous*", para usar detecção de colisão discreta para *Colliders* dinâmicos e colisão continua para *Mesh Colliders* estáticos, ou "*Continuous Dynamic*", para usar detecção de colisão continua com objetos configurados com colisão *Continuous* ou *Continuous Dynamic* ou objetos com *Mesh Colliders* estáticos.

Figura 10 – Exemplo de *Rigidbody*



Fonte: Elaborada pelo autor.

Se um objeto possuir apenas um *Rigidbody*, sem um *Collider*, ele poderá ser influenciado pela gravidade e pelas forças aplicadas manualmente pelo programador, mas não colidirá com nenhum outro objeto. O objeto, por exemplo, atravessará o chão e cairá indefinidamente enquanto sofre efeito da gravidade, se está estiver habilitada, mas não irá parar pôr colidir com outro objeto, como o terreno. Desse modo, ter um *Collider* no objeto junto com o *Rigidbody* é essencial para que o motor de física da Unity trate colisão, uma vez que é o *Collider* que determina o formato do objeto para ser simulado na colisão e o formato de um objeto influência no movimento físico gerado pela colisão. De maneira semelhante, um objeto precisa ter um *Rigidbody* para que o seu *Colllider* seja detectado pela *Trigger Zone* quando há contato.

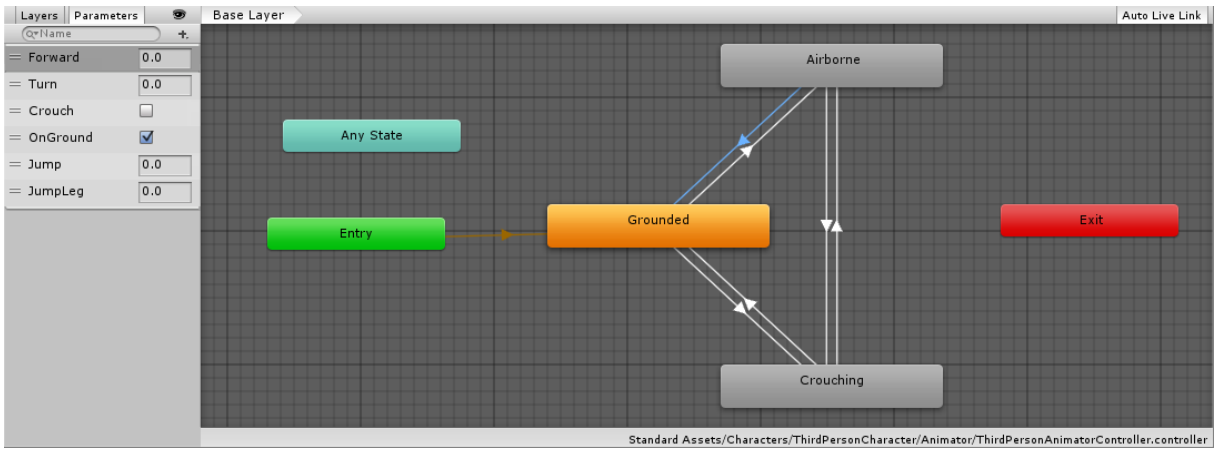
4.5 *Raycast*

Raycast é uma função da Unity que simula um raio sendo lançado em linha reta a partir de um ponto em uma direção, permitindo detectar *Colliders* nessa direção. Passando como parâmetro um ponto de origem e uma direção, a função retorna verdadeiro caso haja um *Collider* nessa direção a partir desse ponto. Também é possível definir uma distância máxima para esse ponto assim como passar um parâmetro de saída na função que conterá informação sobre o objeto que entrou em contato com o raio.

4.6 *Animator*

O *Animator* é uma ferramenta da Unity que permite controlar animações dos modelos. Ele é uma máquina de estados onde cada estado é uma animação e a transição entre cada estado é dada por variáveis de controle do *Animator*. A Figura 11 mostra um exemplo de *Animator*. O estado "*Entry*" é o estado inicial do *Animator* e o estado "*Exit*" é o estado final do *Animator*. "*Grounded*", "*Airborne*" e "*Crouching*" são os estados das animações. Enquanto um objeto estiver com o seu *Animator* em algum desses estados, ele ficará repetindo a animação desse estado.

Figura 11 – Exemplo de *Animator*



Fonte: Elaborada pelo autor

À esquerda na Figura 11 há as variáveis para controlar o *Animator*. É usando essas variáveis que se muda o estado atual do *Animator*. Cada transação possuiu uma ou mais

condições baseadas nessas variáveis para que a transação ocorra. Quando todas as condições para que uma transação ocorra são atendidas, o *Animator* muda de estado e a Unity mistura as animações do estado atual com a do estado seguinte fazendo com que a transação entre os estados seja continua.

Outro detalhe que pode ser observado na Figura 11 é a ausência de transação para o estado final. Essa transação não é obrigatória e em muitos casos não desejada. Uma pessoa, por exemplo, quando não está fazendo nada, não fica completamente parada; ela se movimenta um pouco. Desse modo, mesmo que o objeto contendo o *Animator* não esteja fazendo nenhuma ação, uma animação de descanso pode ser necessária para um melhor realismo.

4.7 *Scripts*

A plataforma Unity permite a criação de *Scripts* nas linguagens de programação C# e JavaScript. É por meio desses *Scripts* que o desenvolvedor controla o jogo e a interação com o usuário. É pelos *Scripts* que o desenvolvedor, também, acessa e altera os outros componentes do objeto. É por *Script* que, por exemplo, se obtém a posição do objeto contida no *Transform* ou se altera as variáveis de controle do *Animator*. Os *Scripits* podem, também, conter funções especiais, como, por exemplo, as funções de quando um objeto entra ou sai de um *Trigger Zone*. As mais importantes dessas funções especiais são as funções "*Start*" e "*Update*".

Por padrão, sempre que um novo *Script* é criado, a classe do *Script* estende a classe "*MonoBehaviour*" que permite implementar as funções *Start* e *Update*. A função *Start* é chamada pela Unity uma única vez quando o objeto, que contém um *Script* cuja classe estende a classe *MonoBehaviour* e implementa a função *Start*, é criado. Um objeto é criado, disparando a função *Start*, quando o jogo inicia e o objeto já está em cena ou quando o objeto é criado por meio de um *Script* de outro objeto. Essa função é usada para inicializações necessárias ao objeto. A função *Update* é chamada uma vez a cada quadro (ou *frame*, em inglês). É nessa função que se põe a lógica que será executada ao longo do jogo. Uma vez que a função *Update* é chamada a cada quadro, é importante cuidar a complexidade do código que será executado dentro dessa função. Um código que leva muito tempo para executar vai aumentar o tempo do quadro, fazendo com a taxa de quadros por segundo diminua, comprometendo a qualidade do jogo.

5 SISTEMA DE SIMULAÇÃO PARA O SISTEMA ASTROS 2020

Em 2012, o Exército Brasileiro lançou o Projeto Estratégico ASTROS 2020 (PEE ASTROS 2020) com o objetivo de modernizar a tecnologia de lançamento de foguetes existente (veículos lançadores, de planejamento e apoio, entre outros), dispor de sistema de Lançamento de Mísseis Táticos de Cruzeiro e munições guiadas de precisão e a aquisição de uma bateria de busca de alvos baseada no uso de Veículos Aéreos Não-Tripulados (VANT) e a implantação do Forte Santa Bárbara em Formosa, GO. Para auxiliar no adestramento de militares em instruções de emprego tático-operacional do sistema ASTROS foi proposto um sistema de simulação. O sistema de simulação proposto para o sistema ASTROS 2020 possui três partes: Treinamento Baseado em Computador, Simulação Virtual e Simulador Tático REOP.

O Treinamento Baseado em Computador permite o treinamento de procedimentos básicos de diferentes viaturas ASTROS 2020 com a utilização de computador ou dispositivo móvel, como tablets. Esse treinamento consiste em uma sequência de informações animadas visando facilitar o adestramento do militar, tendo como objetivo ser o primeiro contato do militar antes da realização do treinamento com simulação virtual ou ter aulas práticas. A Simulação Virtual tem como objetivo adestrar o militar no uso de uma viatura específica utilizando um simulador diferente para cada viatura. O Simulador Tático REOP tem como objetivo adestrar militares em doutrinas táticas relativas ao emprego de uma bateria ASTROS 2020.

O Simulador Tático REOP visa o treinamento no processo de Reconhecimento, Escolha e Ocupação de Posição (REOP) de um Grupo de Artilharia de Mísseis e Foguete. REOP é o conjunto de operações executadas com a finalidade de deslocá-lo de uma Posição de Tiro, Posição de Espera, Zona de Reunião, Coluna de Marcha ou Zona de Embarque para uma outra posição a fim de iniciar ou manter o Apoio de Fogo adequado ao mais alto Escalão.

O simulador para o treinamento de REOP consiste em três partes principais. Uma parte é uma mesa sensível ao toque que será usada por quem está fazendo o treinamento. Nessa parte é onde quem está sendo treinado dá as ordens para as viaturas se moverem ao longo do mapa para executar o processo de REOP. A segunda parte é a tela do instrutor. Essa tela o instrutor usa para avaliar as ordens dadas para a execução do processo REOP. A última parte é uma tela gráfica que mostra, em um ambiente virtual 3D, o movimento das viaturas baseado nas ordens dadas na primeira parte. O foco desse trabalho está nessa última parte

Quando uma rota é traçada pelo usuário, os veículos na parte gráfica precisam se mover ao longo dessa rota. Uma vez que ao usuário cabe apenas traçar a rota, cabe a uma inteligência dos próprios veículos andar de maneira ordenada em um comboio ao longo dessa rota e, quando chegam ao destino, se dispersar para baixo das árvores próximas usando-as como cobertura. O objetivo do trabalho, então, é criar essa inteligência que permita aos veículos se desloquem por essa rota, desviando de obstáculos e evitando colidir com os outros veículos durante o caminho, enquanto tentam se manter na formação, simulando o comportamento real.

Para o desenvolvimento do trabalho, foi considerado que: os veículos são homogêneos; a formação e o caminho são pré-definido antes da execução do algoritmo; a quantidade de veículos esperada varia entre 10 e 15 veículos.

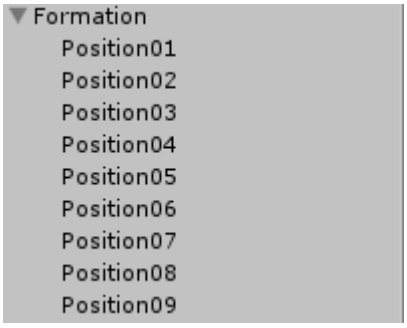
6 DESENVOLVIMENTO

O algoritmo de inteligência para o deslocamento dos veículos foi implementado na plataforma Unity, na qual o simulador está sendo feito. A linguagem dentro da Unity escolhida foi a linguagem C#. Uma vez que tanto a linguagem C# quanto a linguagem JavaScript são compiladas pela Unity para uma linguagem interna, e essa linguagem interna é compilada para a plataforma de destino desejada, a escolha da linguagem faz pouca ou nenhuma diferença no desempenho, de modo que a linguagem C# foi escolhida por familiaridade pessoal com essa linguagem em relação a JavaScript.

6.1 Formação

Para criar a formação não foi preciso fazer o cálculo manual da relação entre cada posição e o líder. Para isso, foi utilizado o sistema de hierarquia de objetos da Unity. O líder arbitrário da formação é um *Game Object* vazio e cada posição da formação é outro *Game Object* vazio filho do *Game Object* líder. A Figura 12 mostra como fica a hierarquia de objetos que compõe a formação. "*Formation*" é o pai e o líder arbitrário da formação e os itens "*PositionXX*" (XX sendo de 01 a 09) são as posições da formação que podem variar de apenas uma posição até quantas forem necessárias. Com a formação crida dessa maneira, basta mover o objeto "*Formation*" que todas as posições se moverão junto, mantendo sempre as mesmas relações entre si, uma vez que a hierarquia dos objetos vai garantir que as mesmas operações de translação e rotação aplicada sobre o *Transform* do objeto pai será aplicada sobre os *Transforms* de todos os objetos filhos.

Figura 12 – Hierarquia da formação na plataforma Unity



Fonte: Elaborada pelo autor.

Para, então, fazer com que a formação efetivamente se mova, é adicionado um *Script* que mudará a rotação e a posição da formação ao longo do tempo. Para isso, calcula-se um vetor de *Steering* para a formação e usa-se o comando "$transform.rotation = Quaternion.LookRotation(finalVector);$" para fazer a rotação e o comando "$transform.position += finalVector * Time.deltaTime;$" para fazer com que a formação se desloque. Nesses comandos, o "*finalVector*" é o vetor obtido com o vetor de *Steering* calculado e "*Time.deltaTime*" é uma variável da Unity que indica o tempo, em segundos, que o último quadro demorou para completar. Ao somar a posição atual da formação com o *finalVector* a formação se move na direção do *finalVector* por uma distância da magnitude desse vetor. A multiplicação por *Time.deltaTime* no comando de deslocamento é importante, pois faz com que a formação se mova a uma velocidade de X/seg em vez de X/quadro, onde X é a magnitude de *finalVector*.

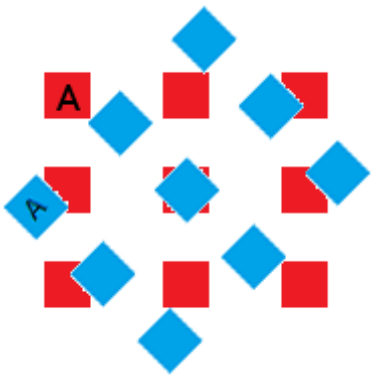
A direção desejada para formação, que é combinada com a velocidade atual para gerar o vetor de *Steering* para o *finalVector*, é simples, sendo apenas a direção da posição atual da formação até o próximo alvo. Como o foco do trabalho foi a inteligência para o deslocamento, esses pontos alvos da formação foram criados manualmente para testes.

O vetor de *steering* é calculado para que se possa combiná-lo com o vetor de direção atual e obter uma curva mais suave e natural. Como a formação é um objeto para controle e não é visto pelo usuário, esse vetor de *steering* não seria necessário, pois não haveria a necessidade de obtenção dessa curva natural, podendo fazer curvas bruscas sem que o usuário viesse a ver essas curvas. No entanto, ele foi introduzido no movimento da formação para evitar um artefato observado durante o desenvolvimento.

Esse artefato pode ser exemplificado na Figura 13. Quando a formação se move e precisa fazer uma curva, como, por exemplo, uma mudança no destino alvo, temos o caso da Figura 13. Sejam os quadrados vermelhos posições na formação em um momento e os quadrados azuis essas posições rotacionadas para a esquerda em um segundo momento, se a rotação for instantânea (em um único quadro), a posição A passa a ficar mais atrás de onde estava. Desse modo, se o veículo está na posição A, ele passa, de um quadro para o outro, a ter a sua posição atrás de onde estava no quadro anterior, o forçando a dar ré no meio do deslocamento que deveria ser apenas para frente. O vetor de *steering* adicionado ao movimento da formação evita que haja uma mudança brusca na posição de algum ponto na formação, causando problemas para o algoritmo de deslocamento do veículo. Esse artefato da mudança

brusca na posição, no entanto, ocorre porque o caminho considerado para a formação é um conjunto de pontos vindo de um grafo que deve ser seguido um a um conforme a formação atinge o ponto de destino atual. Caso o caminho que a formação deve percorrer for passada de tal maneira que não haja curvas bruscas, esse artefato não ocorre.

Figura 13 – Exemplo de artefato na rotação da formação



Fonte: Elaborada pelo autor.

6.2 Veículo

Com a formação criada, é preciso, então, criar a inteligência dos veículos. Para criar a inteligência dos veículos, foi utilizado de base o algoritmo de *boids* de Craig W. Reynolds. Partindo dos três comportamentos originais do algoritmo de *boids* (separação, coesão e alinhamento), cada comportamento foi analisado e adaptado, uma vez que o algoritmo original foi pensado para simular o movimento de grupos de animais e no simulador deseja-se simular o movimento de um comboio de veículos.

Assim como os animais que andam em grupo devem ficar próximos, os veículos de um comboio devem se locomover juntos, no entanto, os veículos não devem ficar a uma mesma distância dos membros próximos como diz o comportamento de coesão do algoritmo original. O comboio possui uma formação fixa que diz em que posição cada veículo deve ficar em relação aos demais do comboio. O comportamento de coesão também gera um artefato indesejado nas bordas do comboio. Veículos nas extremidades da formação tem todos os veículos vizinhos somente em uma direção, desse modo, o centro da vizinhança acaba sempre

apontando para o centro da formação, fazendo com que esses veículos se movam em direção ao centro. Ao se mover em direção ao centro da formação, esses veículos se aproximam de outros, fazendo com o comportamento de separação atue para forçá-los a se afastar novamente. Com essa combinação de se aproximar e se afastar, o comportamento de coesão faz com que os veículos das bordas da formação fiquem ondulando. Esse comportamento de ondulação pode ser interessante para a simulação de grupo de animais, mas esse movimento não é desejado para a simulação de grupo de veículos. Dessa forma, o peso dado para o comportamento de coesão é zero. Para manter os veículos juntos em um comboio, o comportamento de alinhamento foi adaptado e combinado com a formação do comboio.

A formação do comboio é um conjunto de pontos que estão sempre a mesma distância entre si. Cada um desses pontos é a posição de um veículo na formação do comboio e é movendo essa formação que faz o comboio mover-se. Para manter cada veículo na sua posição na formação, o comportamento de alinhamento foi adaptado para que o vetor obtido seja a direção da posição atual do veículo até a sua posição na formação, no lugar de fazer a média das direções dos veículos próximos. Desse modo, basta a formação se mover pelo terreno que cada veículo se moverá junto com a formação tentando se manter na sua posição na formação.

O comportamento de alinhamento modificado descrito acima fará com os veículos movam-se para suas respectivas posições na formação, mas isso não garante que eles não venham se colidir uns com os outros. Durante o deslocamento os veículos podem vir a se colidir caso um veículo precisa passar pela posição de outro para desviar de um obstáculo. O comportamento de separação é, então, utilizado para evitar que os veículos colidem uns com os outros. Desse modo, sempre que dois veículos se aproximarem, eles tentarão se afastar um do outro, mas, por causa do comportamento de alinhamento, também tentarão ir em direção a sua posição da formação do comboio, buscando se afastar, apenas, quando necessário. É, no entanto, importante observar o tamanho da área que o veículo buscará os outros veículos próximos na hora da criação desse comportamento. Se essa área for maior do que a distância entre duas posições da formação, os veículos não conseguirão se manter na formação corretamente, uma vez que os veículos tentarão se afastar antes de atingir sua posição na formação.

Com o cálculo desses comportamentos e a combinação dos vetores que cada um gera, obtém-se um vetor com a direção desejada para o veículo seguir, do mesmo modo que o algoritmo de *boids* original. Esse vetor é, então, combinado com o vetor da direção atual para criar o vetor de *steering* e esse, por sua vez, é combinado ao vetor gerado pelo cálculo do desvio

de obstáculos descrito acima. O vetor final gerado por essas combinações é, então, usado pelo veículo para se movimentar. Os comandos para fazer o veículo se movimentar e rotacionar são os mesmos comandos usados para mover e rotacionar a formação, mas o *finalVector* é obtido com os comportamentos adaptados descritos acima.

6.2.1 Detecção

O algoritmo de *boids* adaptado consegue, então, simular a movimentação dos veículos em um comboio. Cada veículo tenta se manter em sua posição na formação enquanto, ao mesmo tempo, evita colidir com outros membros do grupo e desvia de obstáculos ao longo do caminho. Mas o algoritmo de *boids* e o algoritmo de detecção de obstáculos precisam encontrar quais outros veículos estão próximos e quais obstáculos estão na frente, respectivamente. Uma possibilidade seria fazer com que cada veículo conhecesse todos os outros veículos e obstáculos e a cada execução do algoritmo verificar quais estão dentro da área de interesse de cada algoritmo. A quantidade de veículos é pequena o bastante para que essa solução seja viável, no entanto, o mapa do simulador é muito grande e a quantidade de obstáculos também é grande. Desse modo, fazer com que cada veículo verifique cada um dos obstáculos a cada quadro é muito custoso. Para contornar esse problema, foi usado as *Trigger Zones* da Unity para as detecções.

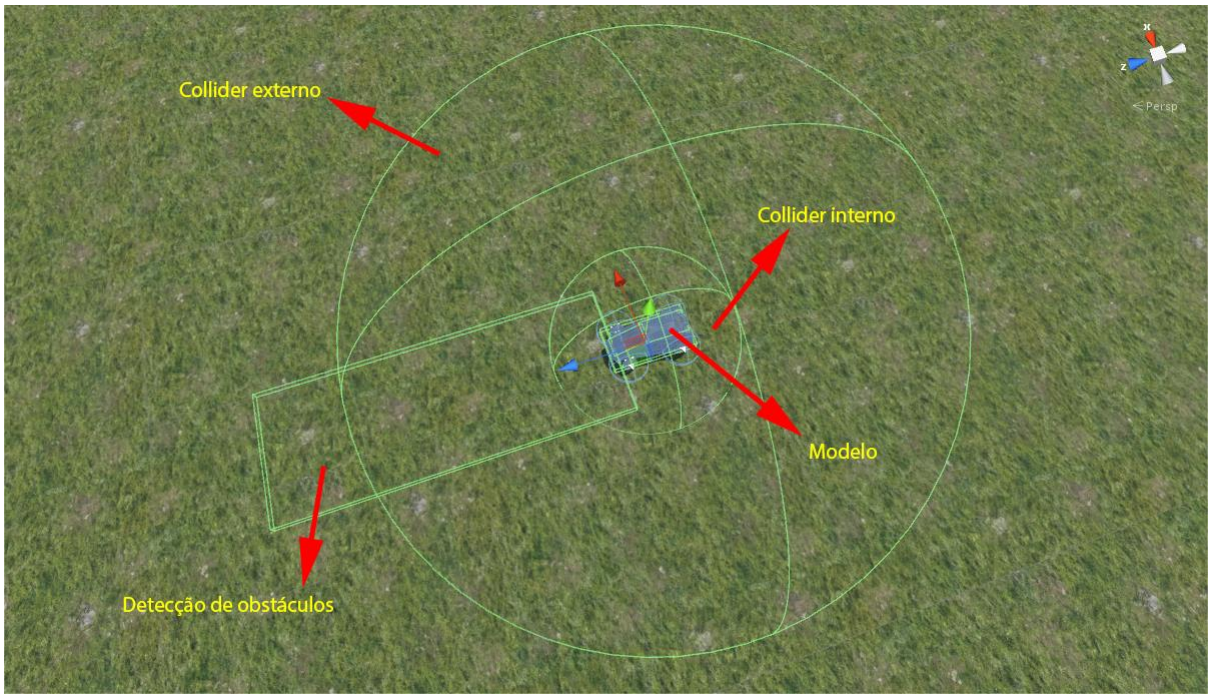
Para a detecção dos veículos próximos, foi adicionado *Triggers Zones* esféricas ao veículo e, para a detecção de obstáculos, *Trigger Zones* paralelepipédicas, a frente para quando o veículo anda para frente e atrás para quando o veículo dá ré. No caso das *Trigger Zones* para a detecção de obstáculos, a *Trigger Zone* da frente e de trás são ativadas e desativadas conforme o veículo anda para frente ou para trás, respectivamente. Quando um objeto na Unity é desativado, seus componentes param de agir. *Scripts*, por exemplo, não são mais executados quando o objeto que os contém está desativado, no entanto, um objeto desativado ainda existe, de modo que o seu *Transform* ainda sofre as alterações refletidas das alterações dos *Transforms* pai. Desse modo, desativando o objeto que contém o *Collider* que não está sendo usado, é possível parar os *Scripts* de detecção, mas os *Colliders* estarão no lugar certo quando ativados novamente, pois acompanharam o veículo, seu objeto pai, durante o deslocamento.

Com essas *Trigger Zones* adicionadas, o veículo pode encontrar os objetos de interesse sem conhece-los previamente. É preciso, apenas, verificar os objetos que entrarem nas áreas de interesse: se objeto não é de interesse, o terreno, por exemplo, possui um *Collider* e é detectado pelas *Trigger Zones*, é ignorado; se é de interesse, como outro veículo para o caso da vizinhança e um obstáculo no caso da detecção de obstáculos, ele é adicionado à lista de objetos que devem ser considerados para o respectivo algoritmo. Da mesma maneira que um objeto deve ser adicionado à lista de elementos a serem considerados quando entra na *Trigger Zone*, um objeto deve ser removido dessa lista quando sai da *Trigger Zone*.

A Figura 14 mostra um modelo de veículo com esses *Colliders* para as detecções de objetos para os comportamentos. O *Collider* esférico mais externo é usado para a detecção de outros veículos na vizinhança para o comportamento de coesão, mas como o peso do vetor desse comportamento é zero, esse *Collider* pode ser excluído. O *Collider* esférico interno é usado para detecção de veículos próximos para o comportamento de separação. Ele é menor do que o *Collider* usado para o comportamento de coesão porque os veículos têm posições fixas na formação e essas posições já possuem uma distância entre si e essa distância é pequena. Desse modo, os veículos já estão próximos entre si e só devem buscar se distanciarem quando estiverem muito próximos, quase colidindo. O terceiro *Collider*, a frente do veículo, é o *Collider* para detecção de obstáculos em formato de paralelepípedo. O *Collider* possui uma largura um pouco maior que a do veículo para dar uma margem de segurança. Há, também, um *Collider* paralelepipédico atrás do veículo para a detecção de obstáculos quando o veículo está dando ré. No caso da figura, o veículo está andando para a frente de modo que o *Collider* está desativado e, desse modo, não aparece. Para separar os *scripts* de detecção e saber qual *Collider* detectou o que, cada um desses *Colliders* está em um objeto filho diferente do objeto principal do veículo, cada um desses objetos com o seu *script* específico.

Por fim, no centro de tudo, está o modelo do veículo. O modelo da imagem é um veículo disponibilizado gratuitamente pela Unity e foi usado para testes. Como o algoritmo para a movimentação de veículos autômatos descrito nesse capítulo é independente de modelo, ele pode ser aplicado em qualquer modelo, bastando, apenas, que o modelo possua os *Colliders* para as detecções, assim como os *Scripts* com a lógica do algoritmo.

Figura 14 – Modelo de veículo com os *colliders* para detecção de objetos para os comportamentos.



Fonte: Elaborada pelo autor.

6.2.2 Dispersão

Após a formação chegar ao destino, os veículos precisam se dispersar e se posicionar em baixo de árvores próximas para cobertura. Para tal, é preciso, em primeiro lugar, encontrar as árvores disponíveis próximas ao local de destino da formação. Para encontrar essas árvores, foi usado, novamente, uma *Trigger Zone*. A formação tem uma *Trigger Zone* que detecta as árvores que estão próximas o suficiente para serem usadas como coberturas. É o tamanho dessa *Trigger Zone* que define o tamanho da área de detecção e o quão distante uma árvore pode estar para poder ser considerada como uma cobertura viável.

Com as árvores para cobertura encontradas é preciso distribuir os veículos por essas árvores. O ideal seria fazer com que a soma das distâncias que cada veículo leva até a árvore que servirá de cobertura seja mínima. No entanto, esse é um problema de minimização que é muito custoso de resolver. Como todos os algoritmos de inteligência precisam dividir tempo de processamento com outros elementos da simulação, como renderização da parte gráfica e simulação de física, é importante que qualquer algoritmo implementado seja o mais rápido

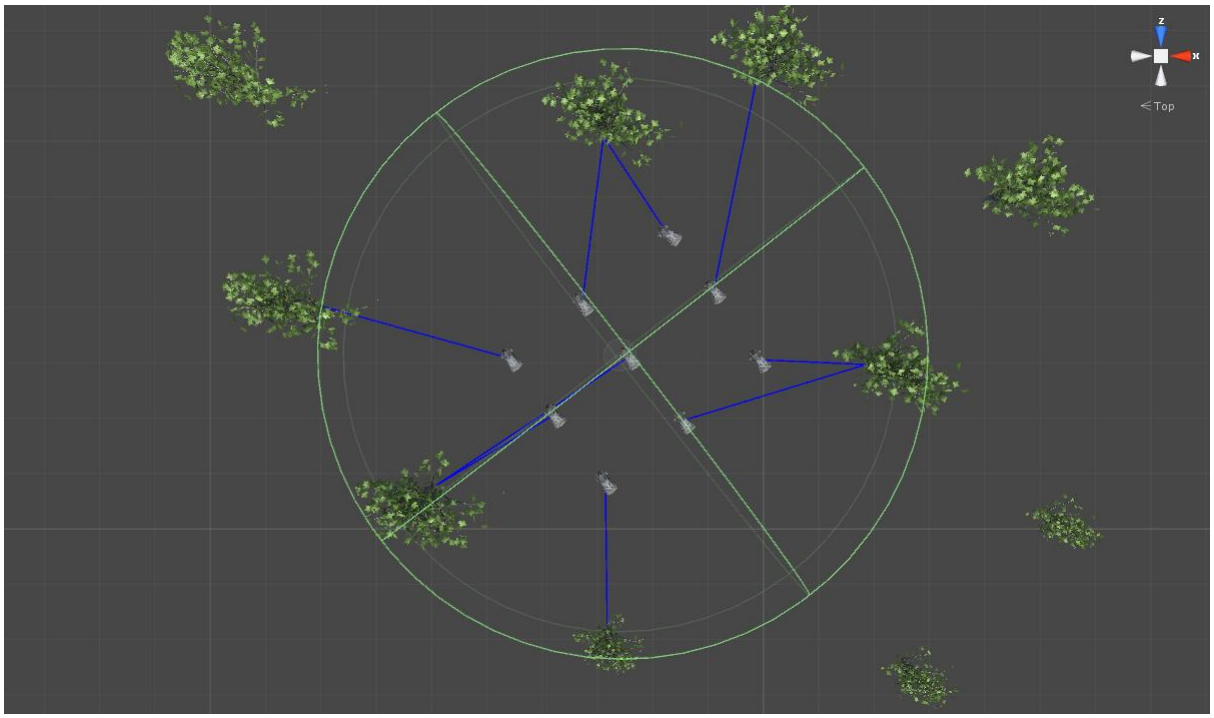
possível. Levando em consideração que o objetivo do simulador não é ser ótimo em algoritmos assim, mas simular um comportamento de modo que pareça real, foi usada uma solução não ótima, mas mais eficiente do que a solução ótima.

A solução implementada foi calcular a distância de cada veículo para cada árvore, colocando em uma lista e ordenando da menor distância para a maior. Com essa lista calculada, busca-se o elemento mais ao topo da lista cujo veículo ainda não foi alocado à nenhuma árvore e a árvore ainda possui espaço para veículos. Com esse elemento escolhido, envia-se esse carro para essa árvore, marcando o carro como alocado e marcando a árvore com um espaço a menos, ou com um espaço ocupado. Com esse método, não há garantia de que a distribuição seja ótima, mas a distância que os veículos percorrem até sua cobertura designada é menor do que uma distribuição aleatória.

A Figura 15 mostra a dispersão obtida com esse método. Essa figura é uma visão da formação vista de cima e o terreno foi removido para facilitar a visualização dos elementos importantes. Nessa figura, o círculo verde é a área na qual a formação busca árvores para enviar os veículos e as linhas azuis liga cada veículo à árvore que foi designado. A distribuição na figura mostra que os veículos foram distribuídos em árvores que não estão longe, evitando, por exemplo, enviar um veículo que está à esquerda até uma árvore que está à direita ou um veículo que está na frente até uma árvore que está atrás; casos que aconteceram na versão inicial da distribuição sem critério. Também é possível observar que nenhum veículo foi enviado para árvores que estão distantes, fora da área de busca da formação.

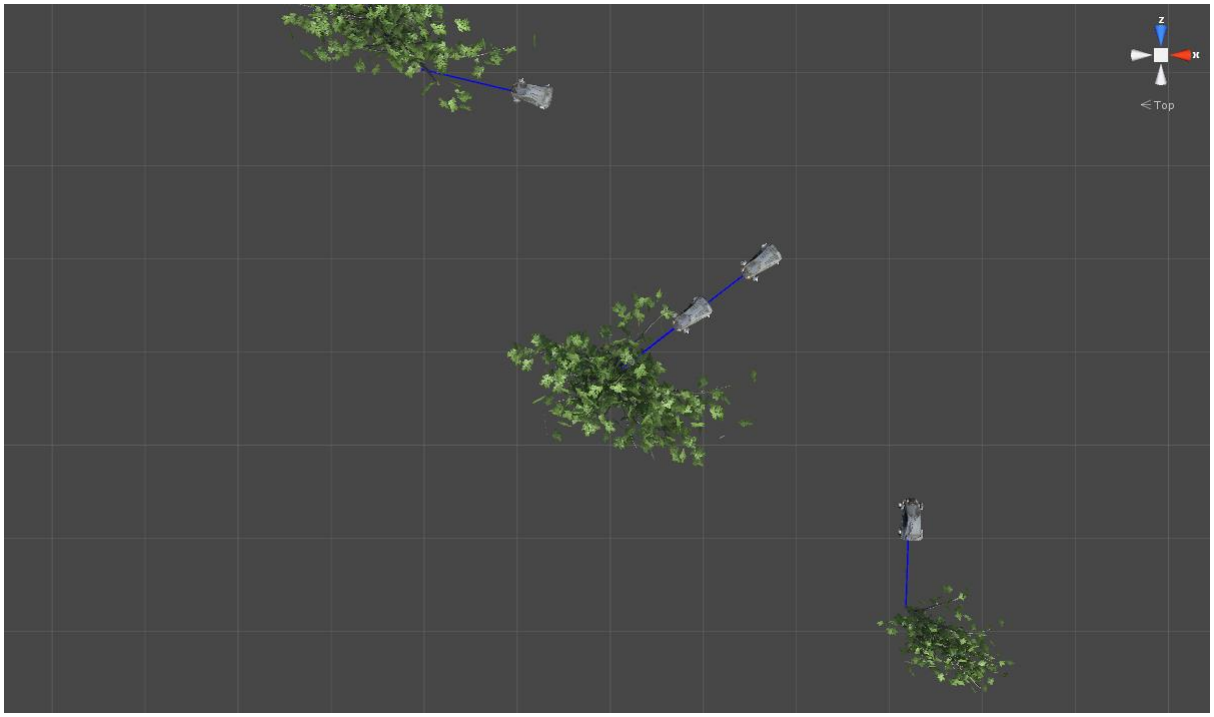
Outro problema que teve que ser tratado durante a etapa de dispersão dos veículos foi o alinhamento de veículos. Quando dois veículos são designados para a mesma árvore e acabam se alinhando (os vetores de direção de ambos os veículos são paralelos entre si), o veículo que está atrás acaba entrando em conflito com o veículo da frente, pois ele tenta se mover em direção à árvore, mas, quando se aproxima do outro veículo da frente, o veículo de trás tenta se afastar e, com os veículos alinhados, essas duas forças possuem a mesma direção, mas sentidos opostos. Desse modo, o veículo de trás fica parado, se a força do comportamento de separação for mais forte, ou colide com o veículo da frente, caso a força do comportamento de alinhamento adaptado seja mais forte. Esse caso pode ser observado na Figura 16 onde há dois veículos se dirigindo para uma mesma árvore e suas direções acabam se alinhando. Para garantir que o veículo de trás contorne o veículo à frente, o algoritmo de desvio de obstáculos passa a considerar outros veículos como obstáculos durante essa etapa de dispersão, de modo que o veículo de trás contorna o veículo da frente como se fosse um obstáculo.

Figura 15 – Distribuição dos veículos para a dispersão



Fonte: Elaborada pelo autor.

Figura 16 – Caso de alinhamento de veículos durante a dispersão

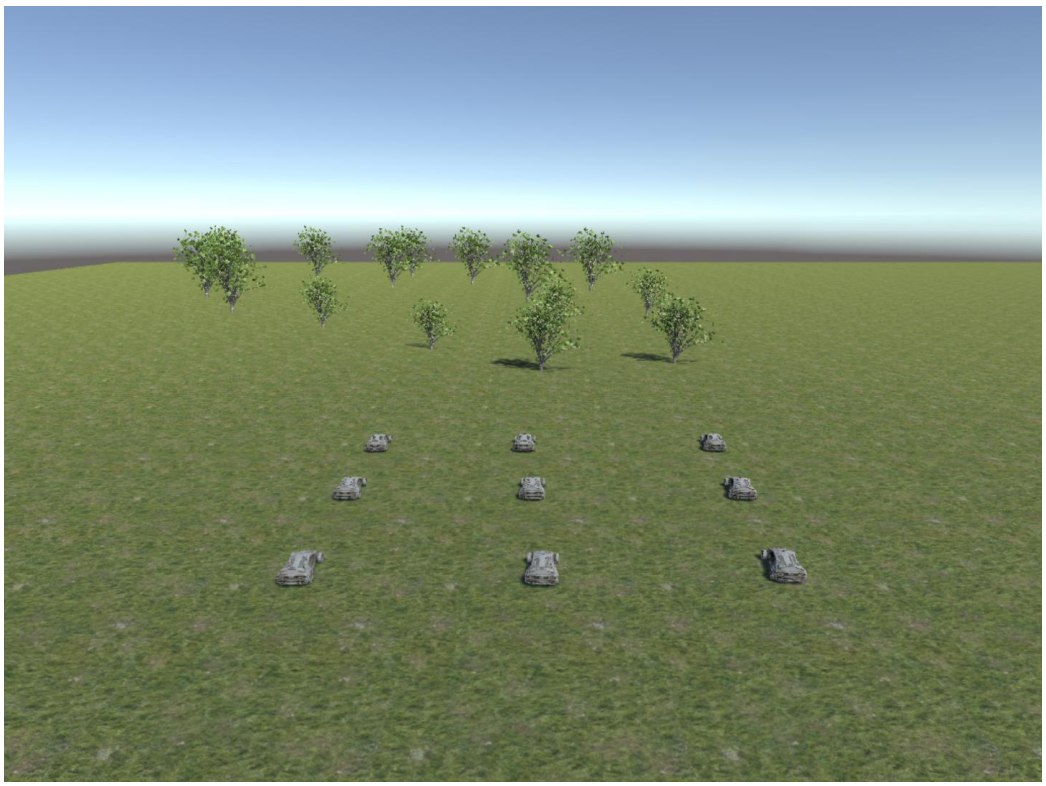


Fonte: elaborada pelo autor.

7 RESULTADOS

As figuras a seguir demonstram os o resultado dos principais pontos da simulação implementada. As figuras Figura 17, Figura 18 e Figura 19 mostram os veículos posicionados em três exemplos de formação diferentes logo antes de partir. As formações dessas figuras são respectivamente: uma formação quadrada 3x3, com três colunas de veículos lado a lado e três veículos em cada coluna; uma formação triangular, com o triângulo contendo três veículos de lado e totalizando 6 veículos na formação; uma formação em coluna simples, contendo um total de quatro veículos nessa coluna única. Os pontos das formações, sendo *Game Objects* vazios, não são visíveis para o usuário, mas os veículos conhecem o ponto que corresponde a sua posição na formação para seguir.

Figura 17 – Veículos posicionados em uma formação quadrada 3x3

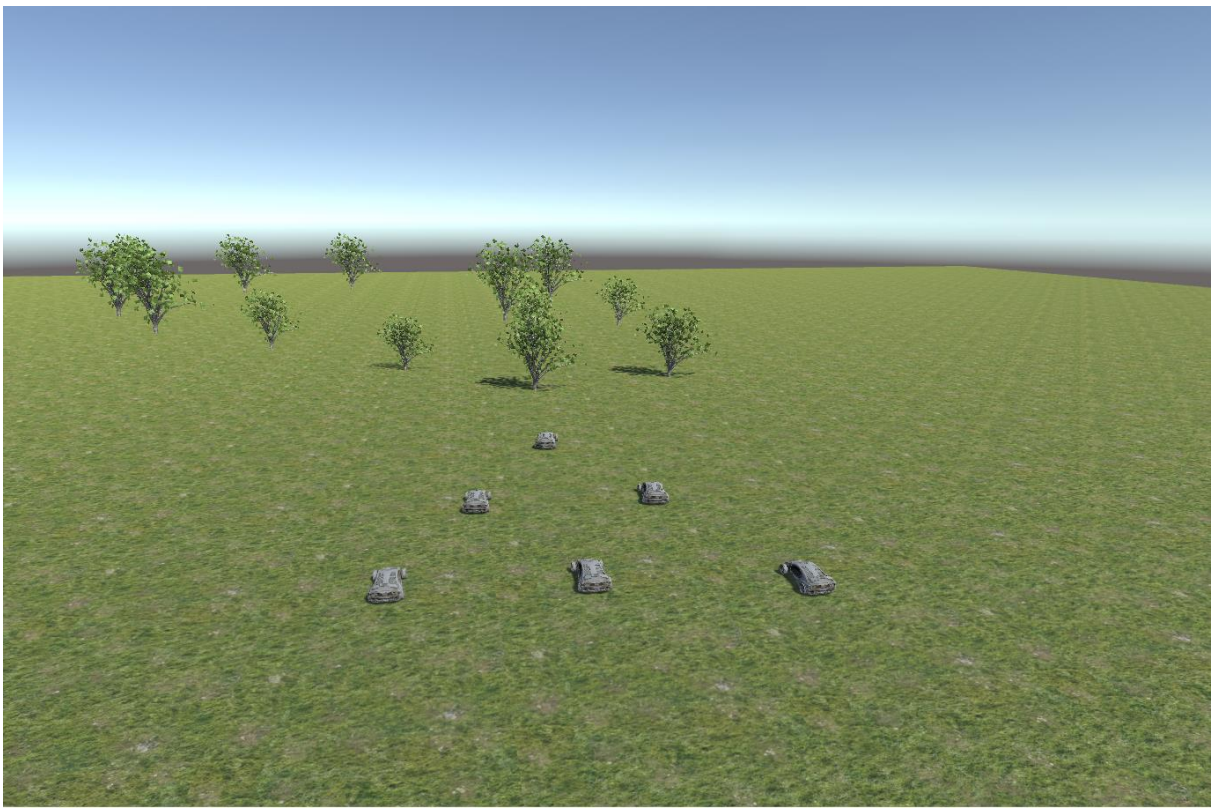


Fonte: elaborada pelo autor.

A Figura 20 mostra os veículos em deslocamento onde os veículos da frente das fileiras do meio e da direita começam a fazer uma curva com o objetivo de desviar da árvore, o obstáculo, à frente. Os demais veículos estão mais distantes das árvores a frente, de modo que

ainda não detectaram nenhum obstáculo como ameaça, sendo assim, eles continuam em linha reta. Esse é o comportamento esperado que foi descrito na área de detecção de obstáculos onde os veículos buscam fazer curvas para desviar de obstáculo que estão à frente e dentro da área livre, mas ignoram obstáculos fora dessa área livre, não considerando esses obstáculos como ameaça.

Figura 18 – Veículos posicionados em uma formação triangular



Fonte: Elaborada pelo autor.

A Figura 21 demonstra o estado final do que foi desenvolvido. Essa figura mostra os veículos posicionados em baixo das árvores para usá-las de cobertura. No caso da imagem há mais veículos do que árvores disponíveis, desse modo, há árvores em que mais de um veículo foi alocado para usá-la como cobertura. Nesses casos, também é possível observar que os veículos conseguem se organizar de modo que fiquem um do lado do outro. Também é possível observar as outras árvores ao fundo que aparecem sem veículos em baixo, essas são árvores que estão mais distantes. Essas árvores estão além da distância limite definida para a área de busca

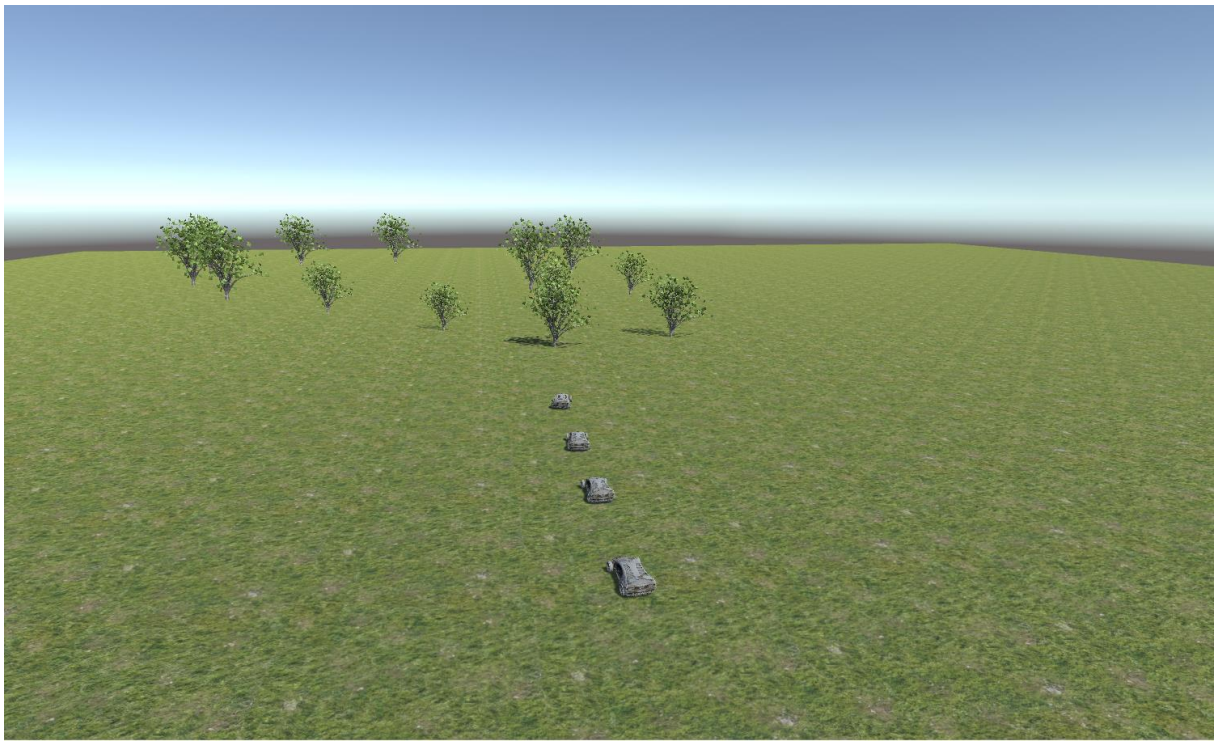
de coberturas na simulação que gerou essas imagens, e, por essa razão, não foram escolhidas para serem usadas como cobertura.

A Figura 15 mostra uma visão de cima da distribuição logo antes do estado final da Figura 21. O caso dessa figura é quando há mais árvores do que veículos e certos veículos precisam dividir a mesma árvore. As figuras Figura 22 e Figura 23 mostram os outros casos mais simples, com mais árvores para cobertura do que veículos e com o mesmo número de árvores e veículos, respectivamente. Nesses casos, não há nenhuma árvore com mais de um veículo, uma vez que o algoritmo foi feito para distribuir os veículos o mais disperso possível e como há árvore suficiente para que cada veículo se dirija a uma árvore diferente, foi isso o que aconteceu. A distribuição também acaba sendo a mesma em ambas as figuras, pois as árvores em destaque que sobraram e nenhum veículo foi enviado na Figura 22 não existem na Figura 23, não sendo usadas em nenhum dos casos.

Outro caso interessante de observar é o caso que ocorre na Figura 24. Pode ser visto nessa figura que o modelo (que é visível para o usuário final) das árvores em destaque não chega a estar dentro da região delimitada para busca de coberturas. Em um caso assim, um usuário poderia pensar que elas estão fora da área de busca. No entanto é possível ver que o *Collider* dessas árvores (os círculos verdes em volta das árvores para aproximar o formato do obstáculo) está em contato com a região de busca e, para a *Trigger Zone*, basta o contato entre os *Colliders* para que haja detecção, não precisando um *Collider* estar totalmente dentro de outro. Como são esses *Colliders* que o algoritmo enxerga, e não que o usuário final vê, mesmo os modelos estando fora da área de busca, é esperado que essas árvores sejam consideradas para a distribuição e veículos sejam enviadas para elas.

Esse detalhe dos *Colliders* na sua utilização para detecção é algo que deve ser observado na hora de definir o tamanho dos *Colliders*. Um *Collider* muito grande para o obstáculo pode fazer com que o veículo passe longe demais desse obstáculo, assim como um *Collider* muito pequeno pode fazer com que o veículo passe perto demais. No entanto, o tamanho do *Collider* também pode ser utilizado para obter efeitos interessantes. Um exemplo seria o caso de uma árvore possuir galhos muito largos, distante do tronco, mas não havendo problema de o veículo encostar na ponta desses galhos. Desse modo, o *Collider* pode ser um pouco menor do que o modelo da árvore, fazendo com que o veículo desvie um pouco perto, batendo na ponta dos galhos.

Figura 19 – Veículos posicionados em uma formação em coluna

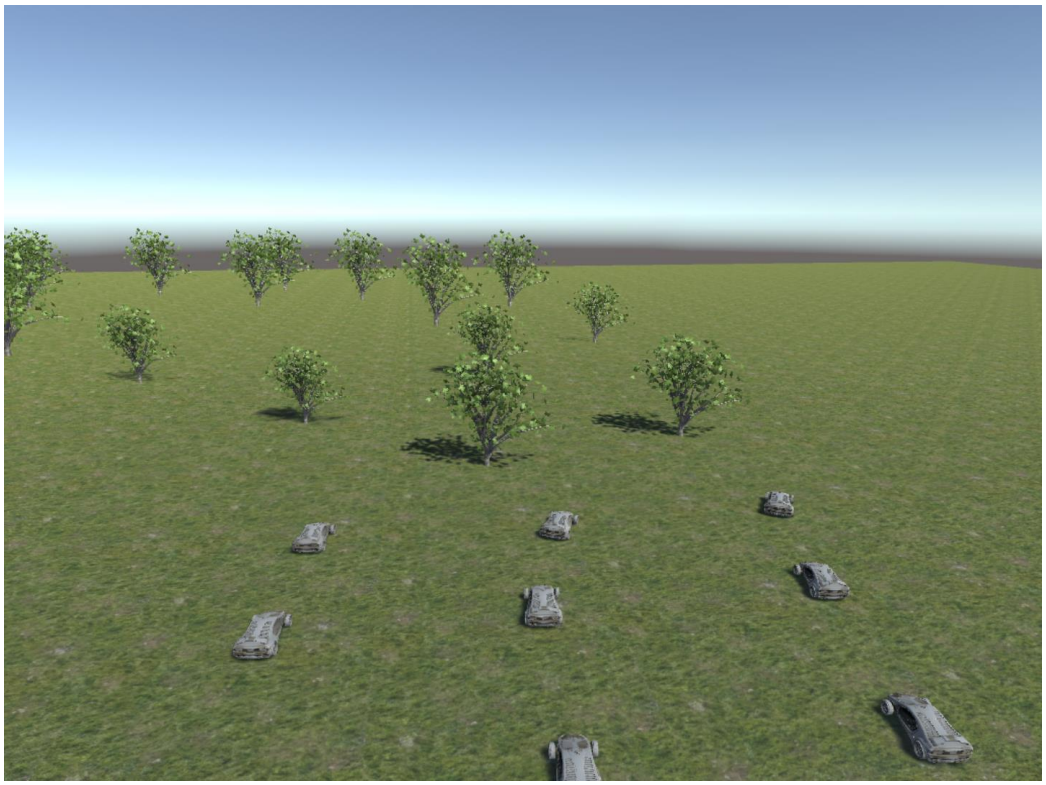


Fonte: Elaborada pelo autor.

Vídeos com as simulações completas de deslocamento e dispersão podem ser vistos nos endereços https://youtu.be/EAT3C4pCTZM, para demonstração com uma formação quadrada 3x3, https://youtu.be/3d_p93EM_TM, para demonstração com uma formação triangular e https://youtu.be/-Fro8fguBVA, para demonstração com uma formação em coluna. Em cada uma dessas três demonstrações, os algoritmos usados são exatamente os mesmos, a única diferença é na formação, mostrando que o algoritmo não funciona com apenas uma única formação. Para fazer essa mudança na formação, basta mudar a posição dos pontos que representam as posições na formação, todos filhos de um mesmo *Game Object*, para mudar o formato da formação.

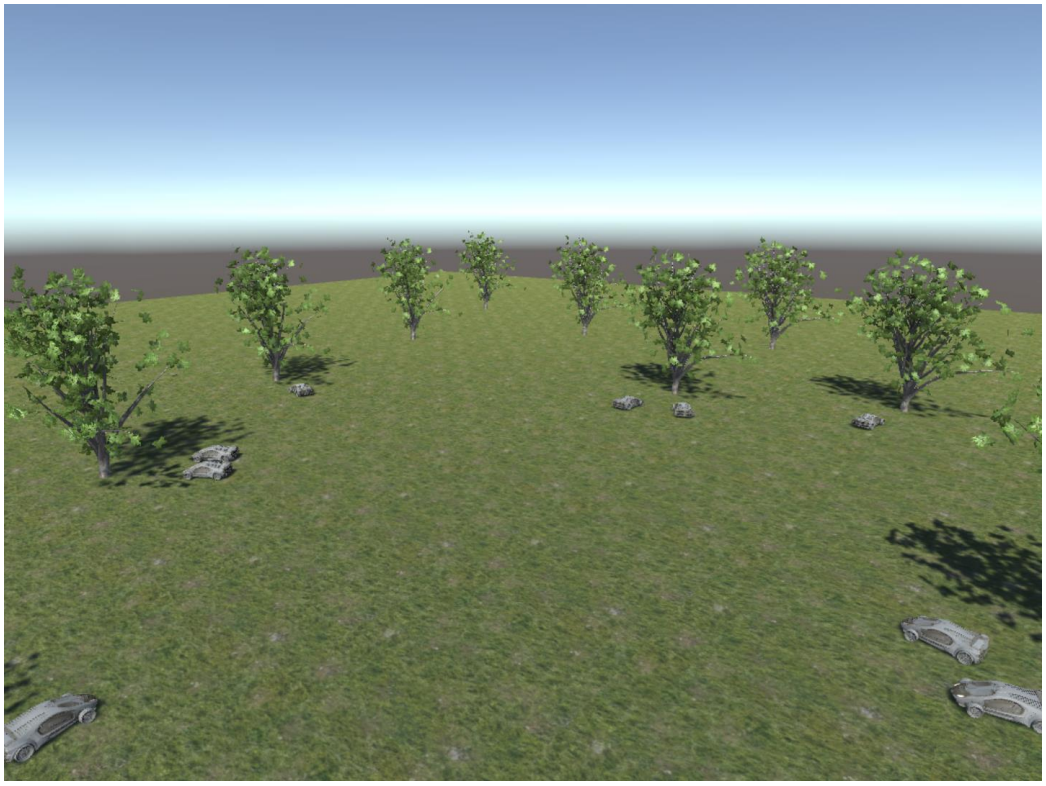
O algoritmo final desenvolvido que é mostrado nessas demonstrações obteve um bom desempenho com um tempo de execução entre 4µs e 10µs para cada veículo. Apesar desse tempo de execução baixo, o algoritmo ainda tem algumas limitações em casos extremos como obstáculos muito grandes (como uma montanha) e não foi tratado caso haja a necessidade de dividir a formação (quando o caminho a ser seguido se divide em dois caminhos separados).

Figura 20 – Veículos iniciando curva para desviar de obstáculos



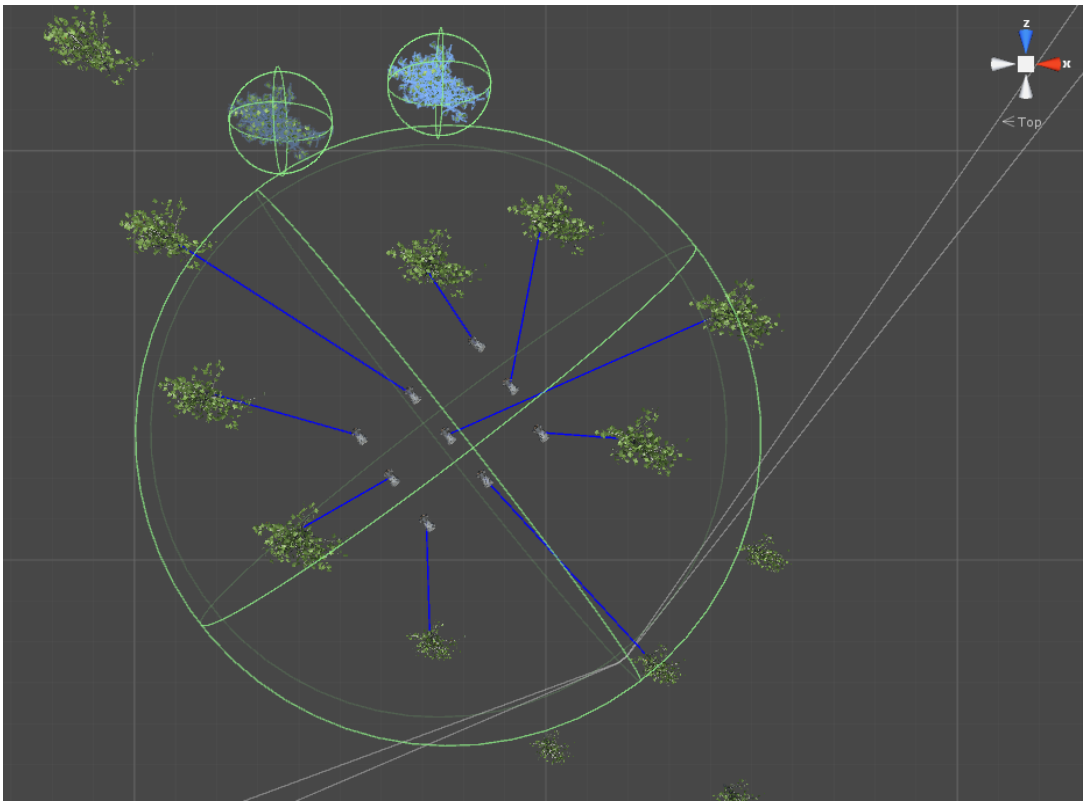
Fonte: Elaborada pelo autor.

Figura 21 – Veículos posicionados em baixo das árvores



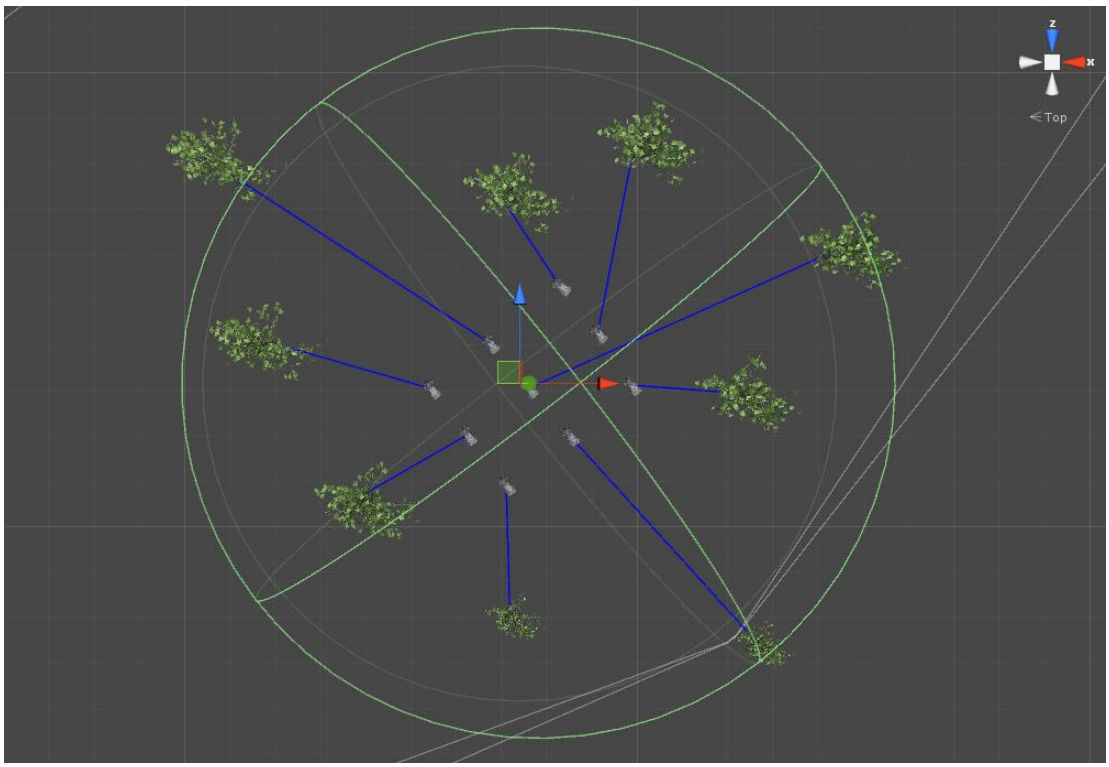
Fonte: Elaborada pelo autor

Figura 22 – Distribuição de veículos para a dispersão com mais árvores do que veículos



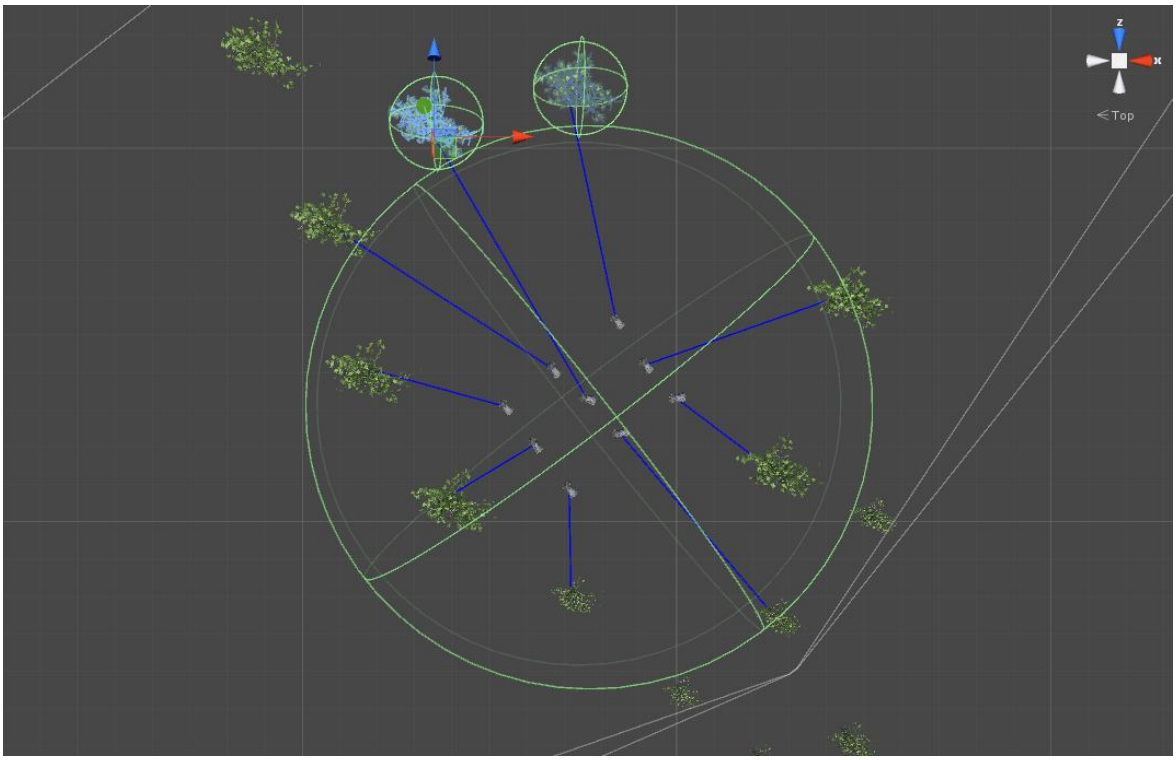
Fonte: Elaborada pelo autor.

Figura 23 – Distribuição de veículos para a dispersão com o mesmo número de árvores e veículos



Fonte: Elaborada pelo autor.

Figura 24 – Dispersão de veículos com árvores na borda da zona de busca



Fonte: Elaborada pelo autor.

8 CONCLUSÃO E TRABALHOS FUTUROS

Nesse trabalho foram estudadas técnicas para a simulação de movimentação e algumas ferramentas da plataforma Unity. O modelo computacional criado por Craig W. Reynolds se mostrou simples de implementação e apresenta uma boa simulação de movimento de bandos. Com as adaptações feitas para o simulador e em conjunto com a formação definida, foi possível obter uma boa inteligência para a movimentação do comboio de veículos que se mostrou simples e conseguiu o objetivo de mover o comboio junto de forma ordenada.

O algoritmo para desvio de obstáculos e a técnica de *steering* também se mostraram eficazes. O fato de todas as técnicas usarem vetores como produto final facilitou a combinação dessas técnicas para a obtenção do produto final. Outro ponto importante foi a simplicidade das técnicas implementadas. É importante, em uma aplicação gráfica, que os quadros sejam gerados rápidos para que a taxa de quadros por segundo fique alta o bastante para não prejudicar a experiência do usuário. Desse modo, é essencial que os algoritmos de inteligência ocupem o menor tempo de processamento possível.

Outro ponto estudado foi a plataforma Unity. A plataforma mostrou possuir ferramentas que facilitam o desenvolvimento dos algoritmos e muito do que a Unity possui não foi explorado, o que indica o potencial da plataforma.

O foco desse trabalho foi no desenvolvimento da inteligência artificial para a movimentação dos veículos. O próximo passo para esse trabalho seria melhorar o realismo do movimento dos veículos com a adição de elementos sonoros, como o som do motor, e animações mais elaboradas, como o movimento das rodas nas curvas.

Para melhorar ainda mais o realismo do movimento, é possível, futuramente, estudar o *Wheel Collider* da plataforma Unity que serve para a simulação de rodas de veículos. Esse *Collider* pode vir a ser combinado com as técnicas desenvolvidas nesse trabalho para dar mais realismo ao movimento do carro assim como sua colisão com terrenos irregulares.

Outro ponto a ser adicionado ao trabalho é um algoritmo para buscar o caminho da formação. Como o foco do trabalho foi na inteligência para o deslocamento dos veículos, o caminho que a formação percorre durante os testes é um caminho fixo criado manualmente. Ao adicionar essa inteligência ao simulador final do projeto ASTROS 2020, será preciso combiná-

la com algum algoritmo que encontre um caminho no mapa para a formação seguir, como, por exemplo, o algoritmo A* que é um dos mais usados.

Outra melhoria a ser adicionada ao trabalho é dar uma maior flexibilidade a formação. Atualmente, o algoritmo usa uma formação pré-definida durante toda a execução. Futuramente é possível adicionar uma interface para a criação dessa formação assim como criar mais de uma formação que possam ser alteradas durante a execução.

REFERÊNCIAS

BALADEZ, F. O passado, o presente e o futuro dos simuladores. FaSCi-Tech, [S.l.], v.1, n.1, 2012.

BEVILACQUA, F. Understanding Steering Behaviors. **Envato Tuts+**, 2012. Disponivel em: <http://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-seek--gamedev-849>. Acesso em: 03 Maio 2016.

CIRIACO, D. O que é Inteligência Artificial? **TecMundo**, 25 Novembro 2008. Disponivel em: <http://www.tecmundo.com.br/intel/1039-o-que-e-inteligencia-artificial-.htm>. Acesso em: 3 Janeiro 2012.

CRUNCHBASE. Sobre a empresa Unity Tecnologies. **Site da CrunchBase**. Disponivel em: <https://www.crunchbase.com/organization/unity-technologies>. Acesso em: 19 Maio 2016.

INSTITUTE FOR SIMULATION & TRAINING. Just what is "simulation" anyway? **University of Central Florida**, 2014. Disponivel em: <http://www.ist.ucf.edu/background.htm>. Acesso em: 03 jun. 2016.

MILLINGTON, I. Artificial Inteligence For Games. In: MILLINGTON, I. **Artificial Inteligence For Games**. 1ª. ed. [S.l.]: Morgan Kaufmann Publishers, 2006. Cap. 3, p. 151-180.

REYNOLDS, C. W. Flocks, Herds and Schools: a distributed behavioral model. SIGGRAPH Comput. Graph.,New York, NY, USA, v.21, n.4, p.25 – 34, August 1987.

REYNOLDS, C. W. Steering behaviors for autonomous characters. Game developers conference, [S.l.], v.1999, p.763 – 782, 1999.

REYNOLDS, C. W. Reynolds Engineering & Design. **Boids**. Disponivel em: <http://www.red3d.com/cwr/boids/>. Acesso em: 26 abr. 2016.

SILVEIRA, R. et al. Motion in Games: third international conference, mig 2010, utrecht, the netherlands, november 14-16, 2010. proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p.410 – 421.

UNITY TECHNOLOGIES. Colliders. **Unity Manual**. Disponivel em: <http://docs.unity3d.com/Manual/CollidersOverview.html>. Acesso em: 08 Maio 2016.

UNITY TECHNOLOGIES. Colliders as Triggers. **Unity Tutorials**. Disponivel em: <https://unity3d.com/pt/learn/tutorials/modules/beginner/physics/colliders-as-triggers>. Acesso em: 08 Abril 2016.

UNITY TECHNOLOGIES. Fatos Rápidos. **Site da Unity Technologies**. Disponivel em: <http://unity3d.com/pt/public-relations>. Acesso em: 19 Maio 2016.

UNITY TECHNOLOGIES. Unity Manual. **Game Objects**. Disponivel em: <http://docs.unity3d.com/Manual/GameObjects.html>. Acesso em: 25 Abril 2016.

HENRY, J.; SHUM, H. P. H.; KOMURA, T. Environment-aware Real-time Crowd Control. In: ACM SIGGRAPH / EUROGRAPHICS CONFERENCE ON COMPUTER ANIMATION, 11., Aire-la-Ville, Switzerland, Switzerland. **Proceedings. . .** Eurographics Association, 2012. p.193 – 200. (EUROSCA'12).

KARAMOUZAS, I.; GERAERTS, R.; OVERMARS, M. Indicative Routes for Path Planning and Crowd Simulation. In: INTERNATIONAL CONFERENCE ON FOUNDATIONS OF DIGITAL GAMES, 4., New York, NY, USA. **Proceedings. . .** ACM, 2009. p.113 – 120. (FDG '09).

AOYAGI, M.; NAMATAME, A. Massive Multi-Agent Flocking Simulation in 3D. In: CONFERENCE ON

BEHAVIOR REPRESENTATION IN MODELING AND SIMULATION. Anais. . . [S.l.: s.n.], 2005.