INSTITUTO POLITÉCNICO DO CÁVADO E AVE

TRABALHO PRÁTICO – FASE 3

RELATÓRIO

JORGE MIGUEL AREZES NORO | 15705

ENGENHARIA E DESENVOLVIMENTO DE JOGOS DIGITAIS

INTRODUÇÃO À PROGRAMAÇÃO 3D

DOCENTE: JOSÉ HENRIQUE BRITO

2º ANO - 1º SEMESTRE

OBJETIVOS

Este relatório serve de apoio ao código e programa entregue. Os objetivos definidos pelo professor para a terceira e última fase do trabalho prático da unidade curricular são adicionar as múltiplas camaras pretendidas, *SurfaceFollowCamera*, *FreeCamera* e *ThirdPersonCamera*. Será necessário implementar as colisões entre os tanques e adicionar projéteis disparados, pelo menos pelo tanque controlado pelo utilizador. É necessário ainda implementar sistemas de partículas para gerar vários efeitos visuais, como por exemplo a ideia de que o tanque gera pó ao movimentar-se. Por último, é necessário implementar movimento autónomo aos tanques que não são controlados pelo jogador. Neste documento serão apelidados de *Bots*. Estes objetivos podem ser observados no programa entregue, dentro da pasta "programa". Todo o código fonte pode ser consultado na pasta "codigo".

Durante a elaboração deste trabalho foi tida em consideração a portabilidade e modularidade do código, de forma a poder ser reaproveitado ao máximo para outros projetos e facilitar futuras consultas.

Para realizar este trabalho, foi utilizado o software de edição e compilação de código Microsoft Visual Studio[1], a *framework* Monogame[2] 3.6 e GitHub[3] para *source control*.

[1] https://visualstudio.microsoft.com
[2] http://www.monogame.net
[3] https://github.com

CÂMARAS

Na primeira fase, foram implementadas várias câmaras e descritas no relatório que acompanhava o código e o programa. Nesta fase, foi implementada uma câmara que não constava nessa primeira fase, uma câmara em terceira pessoa, que se orienta de acordo a uma posição (*Target*), ao terreno (que define qual o *Pitch* mínimo que é permitido à câmara em determinada posição) e ao cursor do utilizador, que gere livremente o *Pitch* e *Yaw* em torno do *Target*.

A implementação desta câmara tirou partido de código já existente das outras câmaras, como por exemplo a forma como processa o *input* do utilizador. No entanto, o método *Update*() é bastante distinto do resto das outras. O objetivo desta câmara é rodar sobre um *Target* fornecido, no caso desta aplicação, é o tanque controlado pelo utilizador. De forma a ser possível restringir o movimento da câmara de acordo com o terreno, este é necessário ser fornecido ao método para podermos realizar verificações e cálculos em sua função.

O algoritmo da lógica da câmara passa por primeiramente processar o input do utilizador. Com este input, é atualizada a posição provável desta câmara. Esse cálculo é feito com base num *Vector3* com o *offset* em Z pretendido. Este *offset* é depois transformado por duas matrizes de rotação, uma matriz de rotação em X, com o parâmetro *Pitch* e outra matriz de rotação em Y, com o *Yaw* como argumento. Após transformar o *offset*, a Posição será igual ao resultado da transformação somado com o *Target*.

O facto desta posição ser provável e não final, é devido à necessidade de restringir o movimento de acordo a pelo menos duas regras. A primeira restrição são os limites do terreno. A câmara não deverá ultrapassar estes limites e a posição é atualizada conforme. A segunda restrição, é a altura em Y. Esta altura não deve ultrapassar o terreno de modo a que seja possível visualizar “de baixo para cima”. Esta restrição foi difícil de implementar, pois há vários aspetos a ter em consideração quando restringimos a posição em Y. Não podemos simplesmente dizer que a posição em Y será igual à coordenada Y da posição do terreno que acabamos de intercetar, pois aí estaremos a fazer com que o movimento da câmara não pareça natural. O que é correto fazer, é calcular qual o *Pitch* que a câmara deve ter, para obter um Y igual ao terreno.

Este tipo de câmara, move-se em coordenadas polares, portanto este *approach* é o mais adequado. Para obter o novo *Pitch*, é utilizada trigonometria para calcular exatamente qual o ângulo pretendido. Na figura 1 é possível observar o movimento pretendido da câmara. A forma como o novo *Pitch* foi calculado, foi calcular o produto interno entre um vetor horizontal e um vetor com a coordenada Y pretendida. Depois foi calculado o produto externo entre este resultado e um vetor com uma direção absoluta. Este produto externo serve-nos para saber qual o sinal do ângulo. Finalmente, o novo ângulo será o produto externo calculado multiplicado pelo sinal.

Com o novo *Pitch*, é novamente calculada a posição final, pelo mesmo método que a posição provável foi.

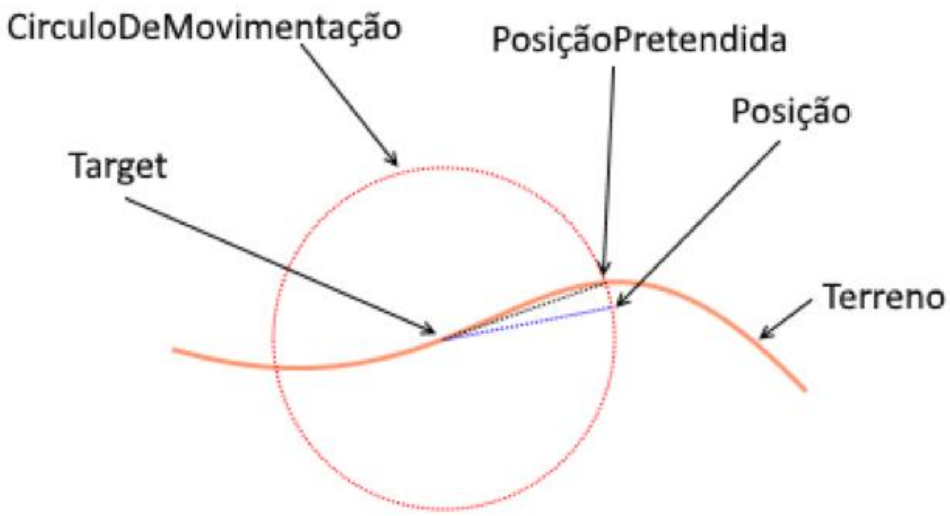

Figura 1- Esquema da rotação da câmara

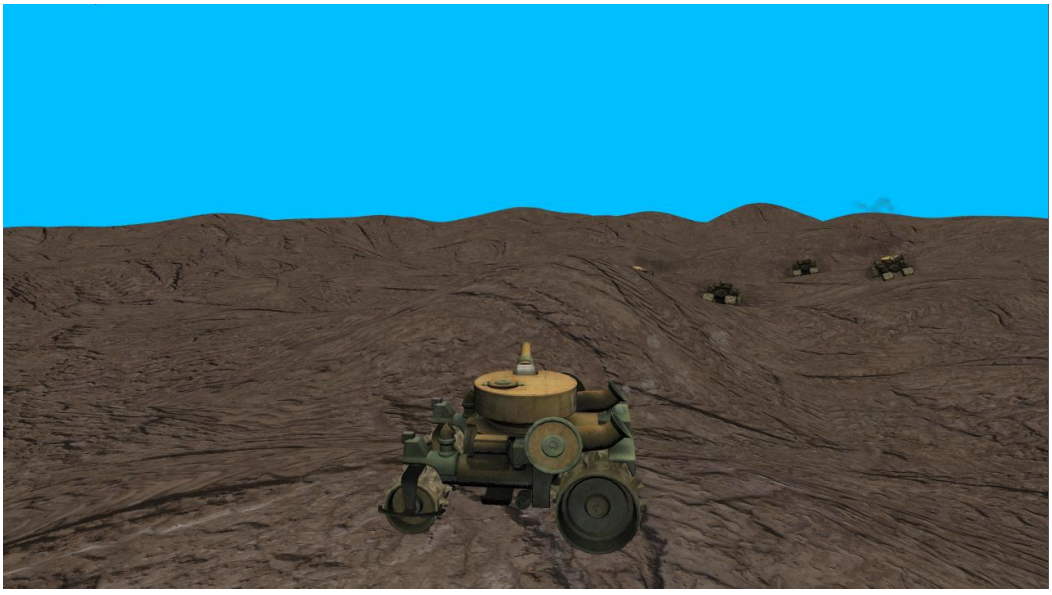

Figura 2-Cena visualizada através da câmara

FÍSICA

Simular comportamentos de corpos rígidos nunca foi tarefa de tomar de leve ânimo. A computação das fórmulas físicas deve ser simplificada de forma a ser viável conseguir executar vários cálculos sobre uma multitude de corpos. No caso dos jogos, um sistema de física deve privilegiar a otimização dos algoritmos e não a perfeição da interseção e resposta a colisões, por exemplo.

No caso desta simulação, o algoritmo que foi escolhido para implementar, foi o *Separating Axis Theorem*[4]. Na base da justificação desta escolha está não só a eficiência do algoritmo, mas também a sua simplicidade. O SAT é usado para calcular a interseção entre volumes não alinhados pelo sistema de eixos global. A forma como executa esta verificação é projetando os volumes em todos os possíveis eixos em que os volumes se possam intercetar. A implementação do SAT nesta aplicação tem apenas em conta volumes paralelepípedos, definidos daqui em diante por *BoundingBox*.

Antes do algoritmo ser implementado, foi desenhada uma estrutura e relação que os objetos em jogo deveriam ter. Todos os objetos a ser alvo da simulação de física, são compostos pelo diagrama na figura 3 (simplificado).

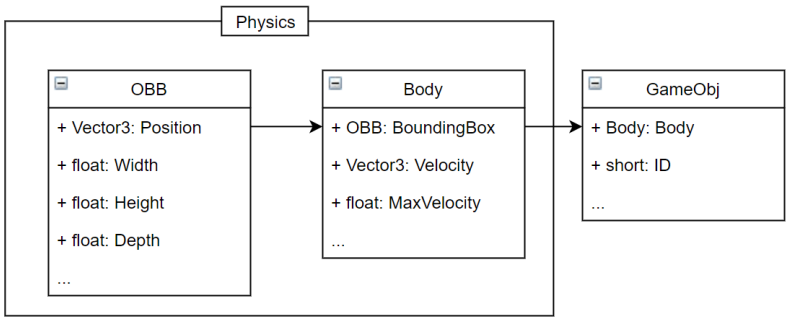

Figura 3-Relação dos objetos para física

Com esta estrutura, são depois calculados os movimentos de cada corpo e posteriormente o estado das interceções. Se algum dos corpos intercetar outro, é aplicada uma resposta.

Para calcular interceções é então utilizado o SAT. A implementação do algoritmo foi feita com base no documento escrito por Johnny Huynh[5].O método passa por obter todos os eixos onde seja possível uma *BoundingBox* intercetar outra. Estes eixos são as normais de cada face, que no caso das *BoundingBoxs*, podemos apenas utilizar os eixos XYZ locais. O cálculo das interceções podia ser só realizado com base nestes eixos se estes estivessem alinhados pelo sistema de eixos global, ou se fosse o caso de uma representação 2D, mas existem mais eixos que é necessário verificar no mundo 3D. Estes outros eixos são as arestas de cada objeto. No total, são verificados 15 eixos.

[4] https://en.wikipedia.org/wiki/Hyperplane_separation_theorem
[5] http://www.jkh.me/files/tutorials/Separating Axis Theorem for Oriented Bounding Boxes.pdf

Após obter todos os eixos necessários, é projetado o volume sobre ele. Em teoria, esta seria a aplicação, mas existe um método mais optimizado. A fórmula consiste em verificar se a soma das projeções de metade de cada *BoundingBox,* é maior que a projeção da distância entre os seus centros. Não é intenção deste relatório explicar todos os detalhes da implementação, por isso o leitor é encorajado a analisar a implementação no código fonte, no ficheiro *Physics.cs*.

O SAT diz-nos que dois objetos não se interceptam se, e só se, nenhuma das projeções se sobrepõe, mas basta uma sobreposição para termos a certeza que os corpos se intercetam. Para realizar a separação dos corpos, basta afastar os objetos, subtraindo à posição o eixo em que foi detetada a interceção multiplicado pelo tamanho do *overlap*.

Figura 4- Visualização das BoundingBoxs de cada tanque em colisão.

Durante o desenvolvimento do sistema de física implementado, foi sentida alguma dificuldade em perceber como o algoritmo funcionava com objetos de três dimensões. Após serem investigadas várias fontes, a implementação foi algo simples. No entanto, é necessário ter em conta que a implementação é algo simplista, por isso é verificado na aplicação que existem alguns erros na resolução das colisões. Estes erros devem-se à partida pela forma que os objetos são orientados em Y. O eixo Y é rigidamente imposto pelas normais do terreno, pelo que se supõe que esta seja uma possível causa.

PROJÉTEIS

A implementação de projéteis tem como base um comportamento balístico semi-real. O objeto *Projectile* está ligado ao sistema de física pelo uso de um corpo rígido.

O tanque controlado pelo utilizador é capaz de disparar projéteis que colidem com os tanques adversários. No momento de criação do tanque, é gerada uma lista com um determinado número de projéteis. Estes serão reutilizados, de forma a não alocar memória durante o *runtime* da aplicação.

A deslocação do corpo no espaço, é realizada através do corpo rígido associado e dos seus parâmetros de aceleração e velocidade. A aceleração inicial será sempre nula, visto que a física dos projéteis[6] diz-nos que a única força que modifica a sua velocidade é a força da gravidade.
A velocidade inicial, é dada pelo ângulo *Pitch* e *Yaw* da torre do tanque, sendo então a equação da velocidade inicial a seguinte:

$$Velocity.X = Cos(Yaw) * Power;$$
$$Velocity.Y = Sin(Pitch) * Power;$$
$$Velocity.Z = Sin(Yaw) * Power;$$

Power, é uma variável que pertence ao tanque e define a força com que o projétil é disparado. Durante o ciclo de *update* do projétil, é aplicada a força da gravidade em Y. A gravidade é uma constante definida como estática à aplicação. Esta propriedade pertence à classe *Physics*.

O ângulo do projétil é definido pelo delta do seu movimento. A classe *Body* disponibiliza em cada *frame*, a posição em que o corpo se encontrava no *frame* anterior. Com esta informação podemos então dizer que a orientação do projétil é a posição atual, menos a posição anterior, normalizado. Este vetor será a frente do nosso projétil, e os restantes vetores são calculados com essa base. A posição inicial do projétil é também calculada com base nos ângulos da torre do tanque, um offset e trigonometria. Outro método igualmente válido, poderia ser utilizar as transformações dos *bones* do tanque.

Sendo o projétil um corpo rígido, podemos facilmente testar a interceção do mesmo com os *Bots* que seguem o utilizador. Para isso damos uso à função discutida anteriormente no capítulo de física. Quando detetada uma interceção, é ativada uma resposta e o projétil é removido da cena e devolvido à lista para reutilização.

O tanque é também responsável por gerir o intervalo com que dispara, sendo uma variável ajustada à situação. Podemos ver na figura 5 uma demonstração dos projéteis com intervalo de disparo de 200ms, uma força de 2.8 e gravidade aplicada de -2.1.

[6] https://courses.lumenlearning.com/boundless-physics/chapter/projectile-motion/

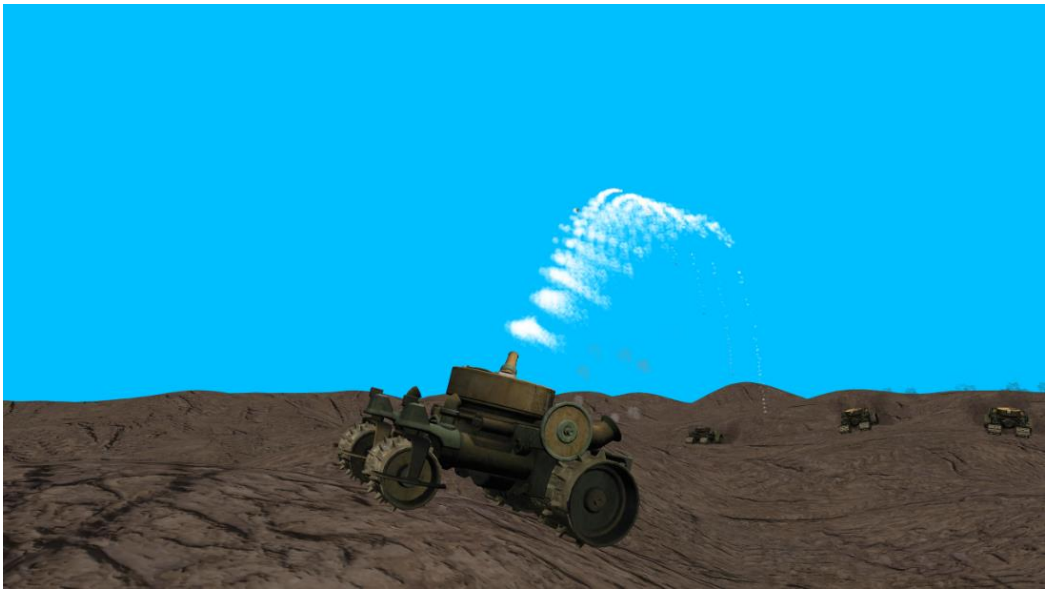

Figura 5- Demonstração de disparo de projéteis

Sistema de Partículas

Um sistema de partículas é um sistema que emite e simula o conceito de partículas. As partículas podem ser definidas com várias características e o sistema é responsável pela sua gestão. No entanto, estas características são propriedades do sistema que as gere. Um sistema de partículas deve ser altamente configurável para ser possível a simulação de vários efeitos. A vantagem de utilizar um destes sistemas é a forma dinâmica como as partículas podem ser utilizadas.

O sistema implementado na aplicação partiu com base em experiências anteriores em aplicações 2D. O sistema foi reescrito para suportar três dimensões e para ser renderizado através de *buffers* na GPU. A geometria das partículas é simplesmente um *quad* construído em *runtime*, em que é aplicada uma posição, rotação, escala, textura, valor e *alpha* e *tint*. Estas propriedades são alimentadas a um *shader* escrito em HLSL. As propriedades *alpha* e *tint* servem para especificar qual a transparência e a cor a tingir na textura. Esta implementação oferece enumeras possibilidades.

No caso da aplicação entregue, as partículas desvanecem quando chegam ao final de vida, e mudam de cor quando são feitas certas ações. Por exemplo, as partículas de fumo do motor, tornam-se pretas quando o utilizador acelera. Isto é atingido ao utilizar a propriedade *tint*.

A chave da implementação de sistemas de partículas, é a sua performance. Infelizmente, não foi possível dispensar mais tempo na otimização dos mesmos. Existe um *bottle neck* ao enviar informação desde o CPU para o GPU. A solução passaria por utilizar *Instances* e *Batching*. Estas soluções permitem reduzir o número de dados necessários a serem transferidos para a GPU de cada vez que se pretende renderizar as partículas. Neste momento, a forma como está implementado requer uma *draw call* por cada partícula, o que no mundo real, é simplesmente impraticável. Fica então aqui, uma oportunidade de melhora e estudo.

No entanto, houve outros problemas que foram solucionados, como é o caso da transparência. Nos primeiros testes à implementação com partículas que requeriam o uso de transparência, verificou-se que existia uma anomalia, como pode ser visualizado na figura 6. O que podemos observar é o *quad* que é renderizado. Isto acontece porque a GPU simplesmente substitui a cor que já está no buffer, pela cor que está a processar. Se o Z-Buffer for superior, essa cor irá prevalecer. A solução simples, sem requerer a manipulação de *BlendStates*, é renderizar as partículas que estão mais próximas da câmara, em primeiro lugar.

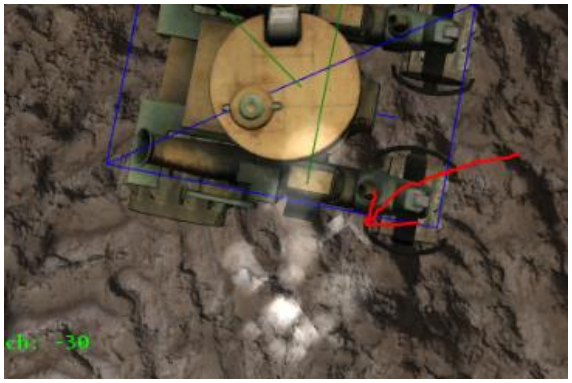

Figura 6- Anomalia ao renderizar transparência

As partículas são também orientadas sempre em direção da câmara, para que sejam sempre visíveis. A forma simples com que a orientação foi atingida sem requerer *billboarding*, foi calcular o vetor entre a posição da partícula e a posição da câmara. Este vetor resultante, será o vetor *forward* da partícula.

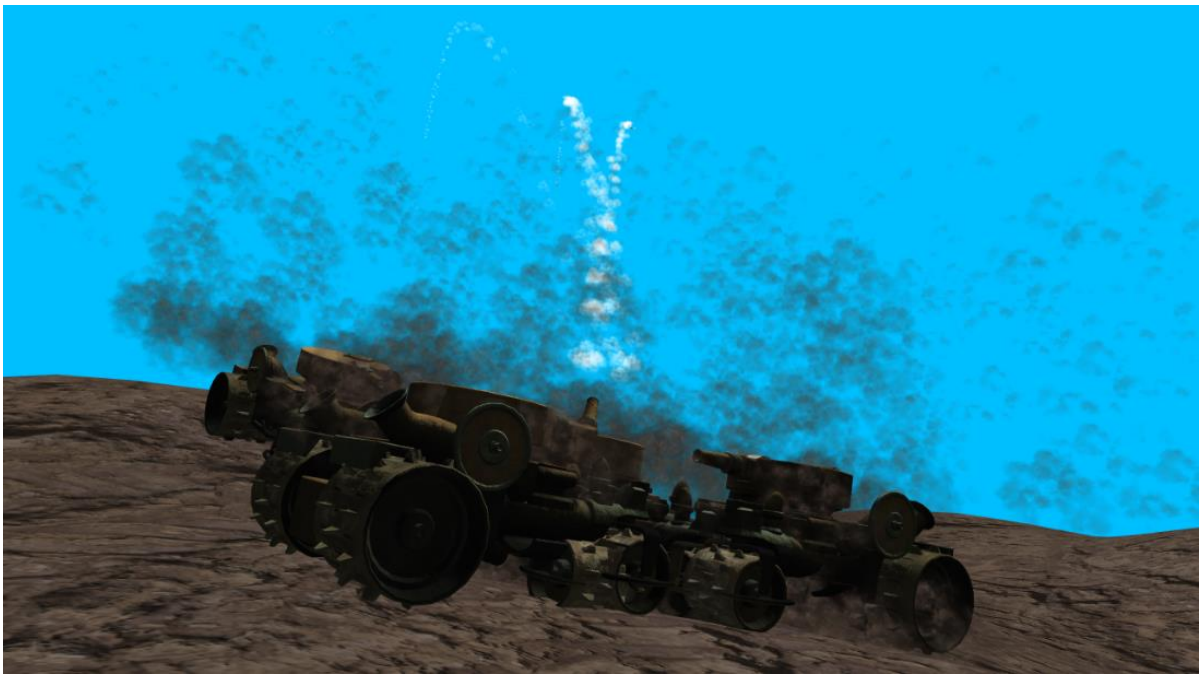

Figura 7- Pó e fumo gerado através de sistema de partículas.

MOVIMENTO AUTÓNOMO DOS *BOTS*

O movimento autónomo dos tanques não controlados pelo utilizador, foi implementado sobre o modelo de *boids* apresentado por Craig W. Reynolds[7]. Os comportamentos implementados foram o *seek*, *flee*, *pursuit* e *evade*. Cada tanque dispõe de todos estes comportamentos, podendo este comportamento ser alterado durante o *runtime* da aplicação. Este parâmetro encontra-se dentro da classe tanque, e é possível atribuir um comportamento a um *Bot* da seguinte forma:

```
Tank t = new Tank(this);
t.BotBehaviour = Global.BotBehaviour.Pursuit;
t.TargetBody = player.Body;
```

Code Snippet 1- Criação de um tanque e atribuição de comportamento.

É também necessário atribuir um corpo para que o comportamento possa ser computado. Embora para alguns comportamentos a apenas a posição serviria, no caso de comportamentos como *pursuit* e *evade*, precisamos de aceder à velocidade do corpo para prever a posição em que ele se encontra no futuro. Desta forma garantimos um comportamento mais fidedigno.

Após aplicados os movimentos autónomos, verificou-se que os *Bots* tentavam circular todos pelo mesmo caminho, acabando por coliderar e gerar alguma confusão. Para resolver este problema, foi implementado o modelo de *separation*, que também consta no modelo de *boids* referido. Com esta implementação, os *Bots* movimentam-se de uma forma muito mais fluida, simulando um esquadrão que tenta capturar o jogador. Este comportamento pode ser verificado na figura 8.

As propriedades dos *Bots* no que diz respeito a movimento são distintas do jogador, como por exemplo, têm uma velocidade máxima menor e maior massa. Toda a implementação pode ser analisada na função *UpdateAutonomousMovement()*, da classe *Tank.cs*.

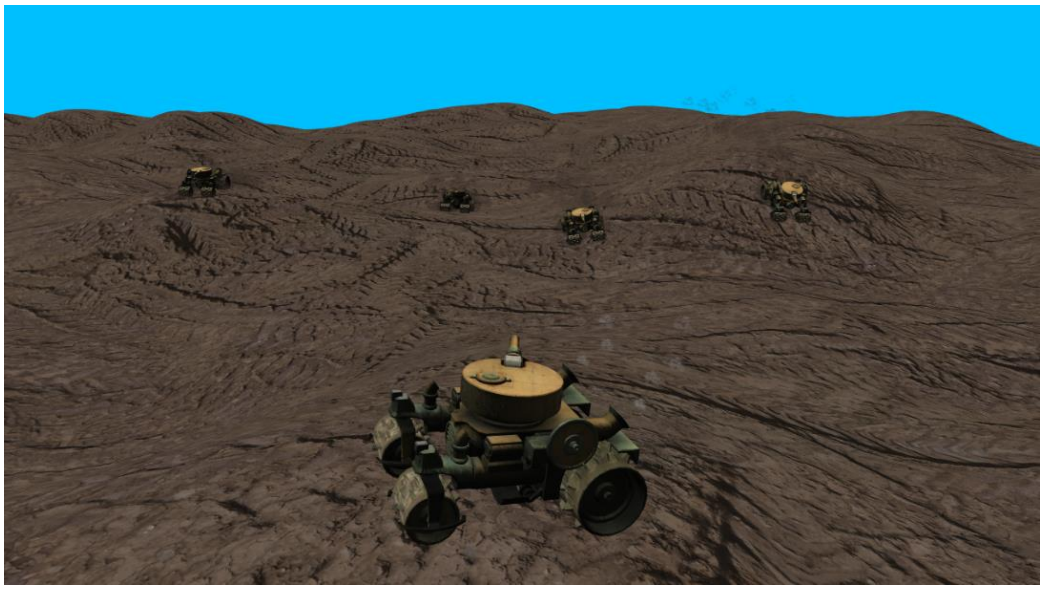

Figura 8-Bots perseguem o jogador em pursuit, com separation

[7] http://www.red3d.com/cwr/steer/gdc99/

Conclusões

Desde a segunda fase, foram feitas atualizações a vários componentes do jogo, nomeadamente ao terreno e ao tanque. O terreno passou de utilizar luz básica, para utilizar um material com luz *ambient*, *diffuse*, *specular* e também, relevo através de um *normal map*. Esta implementação foi proposta como um desafio pessoal, que foi atingido com sucesso. O tanque usa também o mesmo *shader*, e a rugosidade é dada através de um *nomal map* de ferro com marcas de ferrugem. A textura do tanque fornecida foi alterada de forma a fazer *unbake* da luz das rodas. Foram adicionadas animações às rodas to tanque, que rodam de acordo à velocidade entre frames. Este método permite que quando um tanque é arrastado por outro, as rodas rodem. O tanque também roda as rodas dianteiras na direção pretendida a seguir.

Na conclusão do projeto, resta fazer uma retrospetiva e avaliar o que se aprendeu e o que faria de forma diferente. Ficam muitos temas em aberto sobre performance e otimização, que certamente serão estudados no futuro. Este projeto foi sem dúvida um dos mais laboriosos que o curso ofereceu. A unidade curricular é profunda nos conteúdos e embora seja preciso muito trabalho para dominar os conceitos, o trabalho compensa.