Análise e Implementação de uma Árvore de Comportamentos para Unity3D

Guilherme Derner Targa

Trabalho de Conclusão de Curso

-

Universidade Federal de Santa Catarina

Curso: Sistemas de Informação

Orientador: Prof. Mauro Roisenberg

Florianópolis, Outubro de 2010

Análise e Implementação de uma Árvore de Comportamentos para Unity3D

Resumo

Uma das técnicas mais utilizadas pelos desenvolvedores de jogos para modelar o comportamento das entidades é a utilização de Máquinas de Estados Finitos. Apesar de simples de serem implementadas, nem sempre é simples gerenciar suas transições a medida em que ela se torna maior e mais complexa. Desenvolvedores de jogos começaram então a pesquisar outras técnicas para a criação de comportamentos, sendo uma delas as Árvores de Comportamentos. Esta técnica tem sido utilizada recentemente com sucesso na criação de jogos recentes, pois faz com que os comportamentos sejam modulares e fáceis de serem reutilizados. Um dos fatores limitantes para a utilização mais abrangente desta técnica na indústria de jogos é a falta de ferramentas para a sua criação. Este trabalho demonstra a utilização de árvores de comportamentos por intermédio da implementação de um *plugin* para a *game-engine* Unity3D e, fornecer detalhes sobre sua implementação e como projetar uma árvore. Para facilitar a sua utilização, o *plugin* permite que as árvores sejam criadas de forma visual dentro do próprio ambiente de desenvolvimento da *game-engine*. Ao final da implementação do *plugin*, é criada uma cena simples, demonstrando a sua utilização.

Palavras-chave: Árvore de comportamento, Unity3D, Máquina de Estado, Plugin.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As técnicas de inteligência artificial para jogos vêm se tornando cada vez mais complexas, tendo de gerenciar um maior número de entidades, física mais realista e animações avançadas. Um dos desafios destas técnicas é fornecer um método simples e escalável para se editar o comportamento das entidades que estão presentes na cena do jogo.
Máquinas de estados finitas(MEF) foram umas das primeiras técnicas utilizadas na indústria de jogos para se projetar comportamentos. As MEF possuem a vantagem de ser extremamente simples de implementar, porém, é difícil utilizá-las para a construção de sistemas mais complexos (David M. Bourg, 2004). Como uma solução para este problema, desenvolvedores começaram a utilizar máquinas de estado finitas hierárquicas como descritas por David Harel (Harel, 1987).As MEF Hierárquicas proveem transições entre estados que podem ser reutilizáveis, e acabaram se tornando uma técnica bastante popular na área de jogos. No entanto, ainda existem problemas na sua utilização, sendo um dos exemplos, a dificuldade de projetar a máquina de estados em um ambiente dinâmico (Champandard, 2008a).

Jogos mais modernos vêm buscando aperfeiçoar a capacidade de se criar comportamentos mais complexos, e ao mesmo tempo, mais simples de se projetar. Uma das propostas para a solução deste problema é a utilização de árvore de comportamento.(Champandard, 2008a) Entre suas vantagens pode-se citar a simplicidade de projetar e implementar comportamentos de forma modular, auxiliando na sua reusabilidade.(Chong-U Lim e Colton, 2010)

Árvores de comportamento foram recentemente utilizadas em jogos comerciais dos mais variados gêneros, como Halo3, desenvolvido pela Bungie/Microsoft e Spore, desenvolvido pela Maxis/EA.(C. Hecker e Dyckho, 2007)

Muitos jogos utilizam ferramentas chamadas *game-engine* no seu desenvolvimento. Game-engine é uma suíte de ferramentas que auxiliam o processo de desenvolvimento de um jogo, muitas vezes diminuindo o custo de sua produção. Uma *game-engine* completa consiste em componentes para gerenciar os gráficos, som, rede e física do jogo. Muitas delas também possuem ferramentas extras para gerenciar inteligência artificial, interfaces gráficas, entre outros.

Em sua versão grátis, a Unity3D é uma *game-engine* que permite o desenvolvimento para jogos de PC, Mac e navegadores web. Um dos problemas desta *game-engine* é a falta de uma ferramenta para a edição de comportamentos de entidades do jogo. Esse tipo de ferramenta normalmente é encontrado em *game-engines* pagas e, muitas vezes com custos proibitivos para empresas menores.

1.1 Objetivo Geral

Implementar um plugin para a *game-engine* Unity3D que permita a criação de árvores de comportamento de forma visual.

1.2 Objetivos Específicos

Para que o objetivo geral deste trabalho possa ser atingido é necessário que os seguintes objetivos específicos sejam alcançados:

- Analisar as técnicas de implementação de comportamentos mais utilizadas na indústria de jogos;
- Detalhar o funcionamento de uma árvore de comportamentos;
- Definir dos requisitos da ferramenta que será implementada;
- Implementar a ferramenta para a *game-engine* Unity3D;
- Validar a utilização da ferramenta desenvolvida, com um exemplo demonstrando o seu funcionamento;

1.3 Estruturação do Texto

Este trabalho está estruturado em cinco capítulos, sendo este, o primeiro. Este capítulo consiste na introdução do objetivo que este trabalho pretende alcançar, bem como os objetivos específicos para que ele possa ser realizado.

O segundo capítulo fará uma revisão bibliográfica com o propósito de apresentar os conceitos que serão necessários para o entendimento da implementação e utilização da ferramenta a ser desenvolvida. Neste capítulo será apresentado o sistema de agregação de componentes utilizado pela *game-engine* Unity3D, bem como aspectos de funcionamento do mesmo. Alem disso, serão também apresentadas algumas das técnicas de implementação de comportamentos utilizadas na indústria de jogos, bem como o detalhamento da utilização de árvores de comportamento.

O terceiro capítulo engloba a definição dos requisitos da ferramenta a ser desenvolvida e o detalhamento dos aspectos da sua implementação.

O quarto capítulo mostra os resultados obtidos na implementação do *plugin*, bem como exemplos de sua utilização

O último capítulo contém uma revisão dos resultados obtidos e sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os conceitos necessários para a compreensão do processo de implementação do *plugin*. É necessário o entendimento do funcionamento geral de uma *game-engine*, assim como o sistema específico de entidades utilizado pela Unity3D. Este capítulo também contém algumas técnicas utilizadas para a criação de comportamentos de entidades, bem como o detalhamento do funcionamento de uma árvore de comportamentos.

2.1 Game Objects - Hierarquia Versus Agregação

Toda *game-engine* possui o conceito de *game-object*, também chamado de *scene-object* ou simplesmente de entidade (Wilson, 2010). Uma entidade pode ser um personagem animado ou um objeto invisível com um *script* que é ativado quando o personagem o atravessa. Como exemplo de entidade, pode-se citar:

- Carro
- Pedra
- Granada
- Personagem

Entidades podem realizar várias ações no jogo como:

- Rodar um *script*
- Se mover
- Seguir um caminho
- Animação

De modo geral, a entidade será uma classe que expõe uma ou múltiplas interfaces para os sistemas que gerenciam colisão, *rendering*, som, entre outros (Gregory, 2009).

Uma das formas de se representar entidades é utilizar uma hierarquia de classes com herança (sistema baseado em hierarquia), no entanto, esta situação está mudando para métodos de composição de entidades a partir da agregação de componentes(sistema baseado em agregação) (Wilson, 2010).

Sistema de Game Objects Baseado em Hierarquia

Em um sistema baseado em hierarquia, pode-se dividir entidades da seguinte forma:

- Objeto
  - Moveable
    - Veículo
    - Carro
    - Moto
    - Jet-Pack
  - Humanoide
    - Pedestre
    - Jogador
    - Alien
  - Rigid
    - Pedra
    - Granada
    - Arma

É importante enfatizar que em um jogo completo, esta hierarquia teria um número muito maior de entidades. Esta estrutura precisará ser frequentemente modificada ao decorrer do desenvolvimento do jogo, principalmente, se a *game-engine* for reutilizada para um jogo diferente (Wilson, 2010).

Além disso, será necessário adicionar vários componentes a uma entidade. Estes componentes, podem estar encapsulados na própria entidade ou serem herdados de uma entidade de nível superior que o implemente.

Este tipo de arquitetura apresenta um problema comum, no qual, uma funcionalidade é adicionada a uma classe próxima a raiz da hierarquia, como a entidade "Objeto" do exemplo. Isto dá o beneficio de que a funcionalidade esteja disponível para todos os objetos que derivam desta classe, mas, como desvantagem, pode gerar classes com um número enorme de funcionalidades que não são realmente necessárias. Este tipo de *"anti-pattern"* normalmente acontece próximo a raiz da árvore, podendo ocorrer também em suas folhas. A entidade chamada Jogador, mostrado no exemplo de hierarquia acima, seria um dos candidatos a possuir este tipo de problema, já que o jogo estaria centrado nele, possuiria um grande número de funcionalidades (Gregory, 2009).

Concentrar a funcionalidade nas folhas da hierarquia também acarreta problemas, fazendo com que muitas vezes haja duplicação de código e, fazendo com que mais tarde, seja necessário a refatoração e re-estruturação da hierarquia (Gregory, 2009).

Sistema baseado em agregação

Neste tipo de sistema, uma entidade é simplesmente a soma de suas partes, ou seja, ele nada mais é que uma coleção de componentes. Tomando como exemplo a *game-engine*

Unity3D, uma entidade vazia sempre possui um componente chamado Transform, que possui a posição da entidade na cena. Outros componentes podem ser agregados a entidade: scripts, *rigidbody*, *render*, entre outros.

Este tipo de arquitetura faz com que o código se torne mais limpo, facilitando a criação de novas entidade no decorrer do desenvolvimento do jogo, uma vez que a entidade não é nada mais do que uma coleção de componentes previamente programados. Caso uma entidade precise de um componente que ainda não foi implementado, basta implementá-lo e agregá-lo a entidade que o necessite Wilson, 2010.

É importante levar em consideração este sistema de entidades na implementação do *plugin*, de forma que o deixe integrado com o restante da *game-engine* e, fazendo com que seja possível que uma árvore de comportamentos funcione exatamente como qualquer outro componente da Unity3D. A figura 2.1 mostra um exemplo de funcionamento do sistema de entidades utilizado pela Unity3D, onde cada entidade possui os componentes necessários para a sua execução.

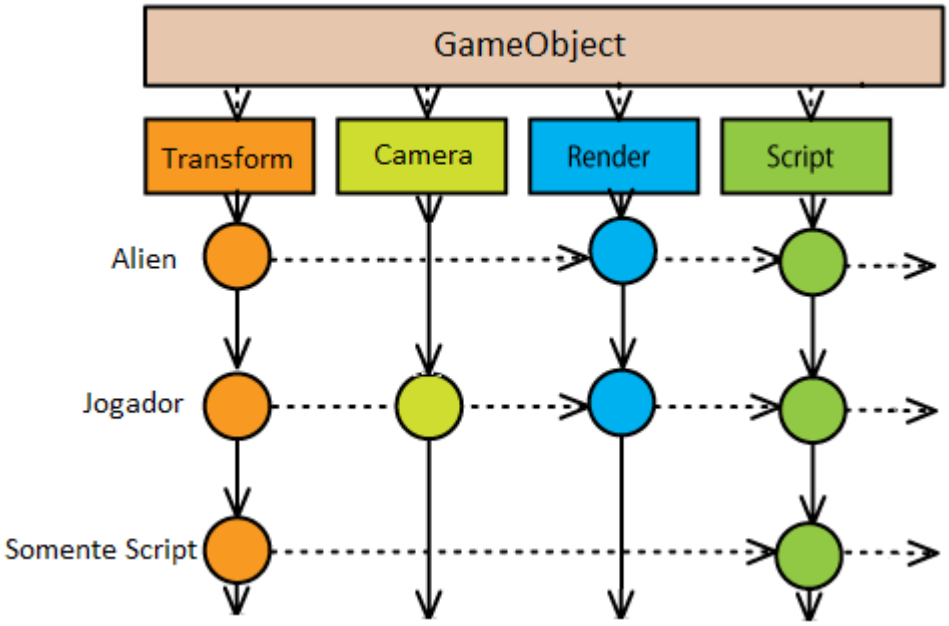

Figura 2.1: *Exemplo de funcionamento de GameObjects na Unity3D*

2.2 Unity3D

A Unity3D é uma *game-engine* para a criação e desenvolvimento de vídeo-games. Também pode ser chamado de game *middleware*, uma vez que seu principal objetivo é minimizar o tempo e o custo do desenvolvimento de um jogo, provendo ferramentas que todos os tipos de jogos normalmente necessitam. Entre as funcionalidades que uma *game-engine* provê pode-se destacar um *renderer* para gráficos 2D ou 3D, uma *engine* de física, som, animação, inteligência artificial, rede, gerenciamento de memória, edição de cenas, entre outros. Essas ferramentas, são normalmente integradas em único ambiente de desenvolvimento, facilitando e simplificando o processo de criação de um jogo (Goldstone, 2009). A seguir, está descrito de forma sucinta, alguns conceitos que a Unity3D utiliza em seu ambiente de desenvolvimento

(Goldstone, 2009):

- *Assets*: são todos os arquivos que compõem um projeto. Arquivos de imagem, texturas, materiais, scripts, modelos 3D, entre outros fazem parte desta categoria.

- Cenas: cada cena na Unity3D pode ser um mapa ou área individual do jogo. Um jogo normalmente possui várias cenas, de forma que o carregamento de assets seja distribuído e que se possa testar diferentes partes do jogo de forma individual.

- *Game-Objects* e *Components*: Quando um *Asset* é utilizado dentro de uma cena, ele se transforma em um *Game-Object*. Este Game-Object automaticamente recebe um componente chamado *Transform*, que, utilizando coordenadas X,Y,Z, contém a posição e rotação do objeto dentro da cena. Também é possível mudar a escala do objeto por este componente. Ele pode ser acessado via programação para que sejam efetuadas transformações em seus atributos. Outros componentes podem ser adicionados ao *Game-Object*, servindo para criar comportamentos, definir aparência e influenciar outros aspectos do mesmo. A Unity3D proporciona componentes que são frequentemente utilizados em jogos, como *rigidbody*, câmeras, emissores de partículas. Também é possível programar novos componentes para adicionar comportamentos específicos a um *Game-Object*.

- *Prefabs*: muitas vezes é necessário criar *Game-Objects* complexos com vários componentes e atributos, e estes deverão ser utilizados mais de uma vez. Um *Prefab* é um *template* de *Game-Object*. Quando ele é inserido na cena, já possui todos os componentes e atributos que foram estipulados a ele.

A figura 2.2(Goldstone, 2009) mostra o ambiente de desenvolvimento da Unity3D:

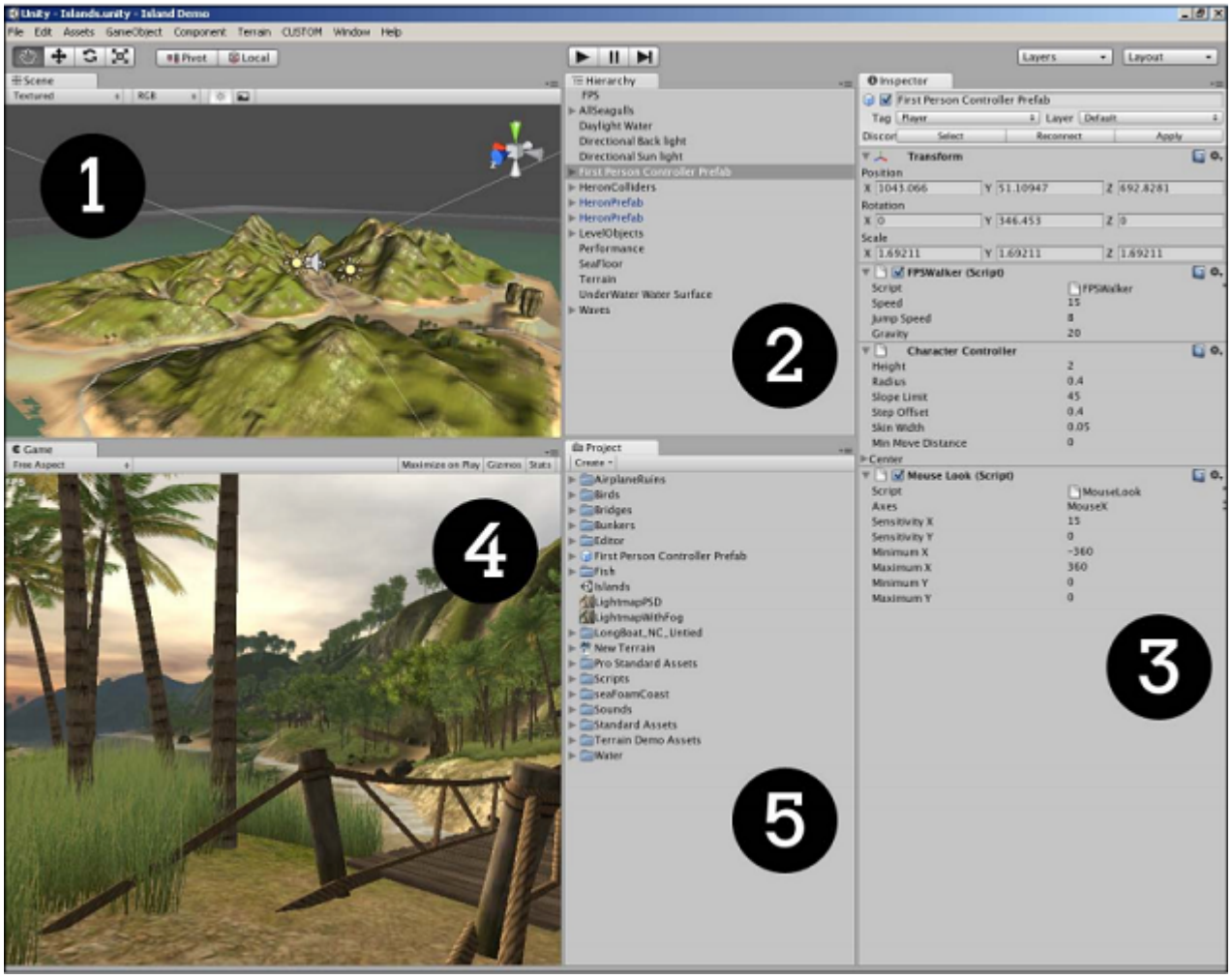

Figura 2.2: *Ambiente Unity3D. 1 – Editor de Cena; 2 - Game objects que estão na cena; 3 - Componentes que pertencem ao game object selecionado; 4 - Janela de preview; 5 – Assets do projeto que está aberto no momento; (Goldstone, 2009)*

2.3 Técnicas Para Criação de Comportamentos

2.3.1 Máquina de Estado Finita (MEF)

"Máquinas de Estados são estruturas lógicas compostas por um conjunto de estados e um conjunto de regras de transição entre os estados. São ferramentas úteis em qualquer aplicação que envolva o controle de processos, na qual seja possível descrever cada uma das situações discretas em que os processos podem se encontrar a cada momento. Essas situações definem os estados da Máquina, e as regras de transição representam as condições necessárias para que o processo atinja uma nova situação a partir de um determinado estado corrente. Quando o número de estados da Máquina é finito, tem-se uma Máquina de Estados Finitos (ou simplesmente FSM, do inglês Finite State Machine)." (dos Santos, 2004)

Máquinas de estado finitas vêm sendo utilizadas para a criação de comportamentos de jogos.Por exemplo, o jogo Pac Man, produzido em 1980 utilizava máquinas de estado para definir o comportamento dos fantasmas do jogo. Eles podiam andar livremente pelo cenário, perseguir ou fugir do jogador. Em cada estado eles se comportavam de maneiras diferentes,

e a transição de um estado para outro dependia das ações do jogador. Por exemplo, quando o jogador pegava um item que o tornava capaz de destruir os fantasmas, eles mudavam do estado de perseguir para o estado de fugir.

Apesar das máquinas de estado finitas serem demasiadamente antigas, ainda são largamente utilizadas na construção de jogos. A facilidade de compreensão, implementação e testes, contribuem para seu frequente uso (David M. Bourg, 2004).

A figura 2.3 (David M. Bourg, 2004) mostra o exemplo de uma máquina de estados finita simples. Cada estado é representado por um retângulo, onde "Em Guarda" é o estado inicial. As flechas mostram as transições para que a máquina passe de um estado para o outro, ou seja, a máquina fica no estado "Em Guarda" até que ocorra a transição "Ver inimigo pequeno" ou "Ver inimigo grande".

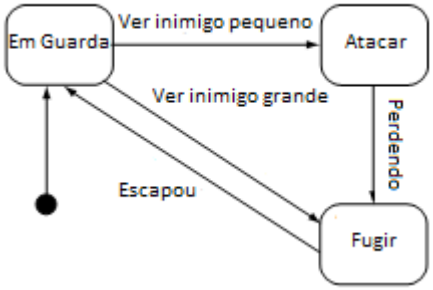

Figura 2.3: *Máquina de Estados Finita (David M. Bourg, 2004)*

Um dos desafios do desenvolvimento de inteligência artificial para jogos é o gerenciamento de sua complexidade. Uma máquina de estados finita possui problemas de escalabilidade, pois não é possível reutilizar estados em contextos diferentes (David M. Bourg, 2004). A única escolha é criar um novo estado com transições diferentes específicas para o novo contexto (David M. Bourg, 2004). Na figura 2.4 (David M. Bourg, 2004) há uma máquina de estados finita de um robô coletor de lixo, sempre que sua energia fica baixa, ele precisa ser recarregado. O exemplo mostra o problema descrito acima, sempre que o robô precisa ser recarregado, é preciso criar novas transições e estados para o contexto em que ele se encontra.

Figura 2.4: *Problemas de múltiplas transições em uma MEF (David M. Bourg, 2004)*

2.3.2 Máquina de Estado Finita Hierárquica (MEFH)

Uma MEF Hierárquica, também chamada de HFSM (do inglês: Hierarquical Finite State Machine), foi uma das soluções que os desenvolvedores de jogos passaram a utilizar para tentar resolver o problema de redundância e complexidade de uma MEF simples.(Harel, 1987) O processo de desenho de uma HFSM envolve:

- Construir a lógica estado por estado, conectando-os com transições de baixo para cima.
- Ao criar novos estados, é possível agrupá-los de forma que eles compartilhem transições.

Esta técnica foi originalmente concebida por David Harel, no qual, em seu artigo, ele utiliza o termo "super-estados" para indicar grupos de estados. Estes super-estados, teoricamente, previnem a redundância de transições, uma vez que é possível aplicar a transição apenas uma vez para o super-estado, ao invés de aplica-lá para cada estado específico. Harel chama essas transições de "transições generalizadas" (Harel, 1987).

A figura 2.5 (David M. Bourg, 2004) mostra a mesma máquina utilizada no exemplo da figura 2.4, porém, implementada de forma hierárquica. Quando um estado composto é chamado pela primeira vez, o estado H* indica qual sub-estado deve ser chamado. Caso o estado composto já tenha sido chamado anteriormente, o sub-estado anterior é retornado. Por essa razão, o nó H* é chamado de "history state". Como pode-se notar, a máquina de estados hierárquica fica menos complexa que a máquina de estados tradicional, e com um número menor de transições.

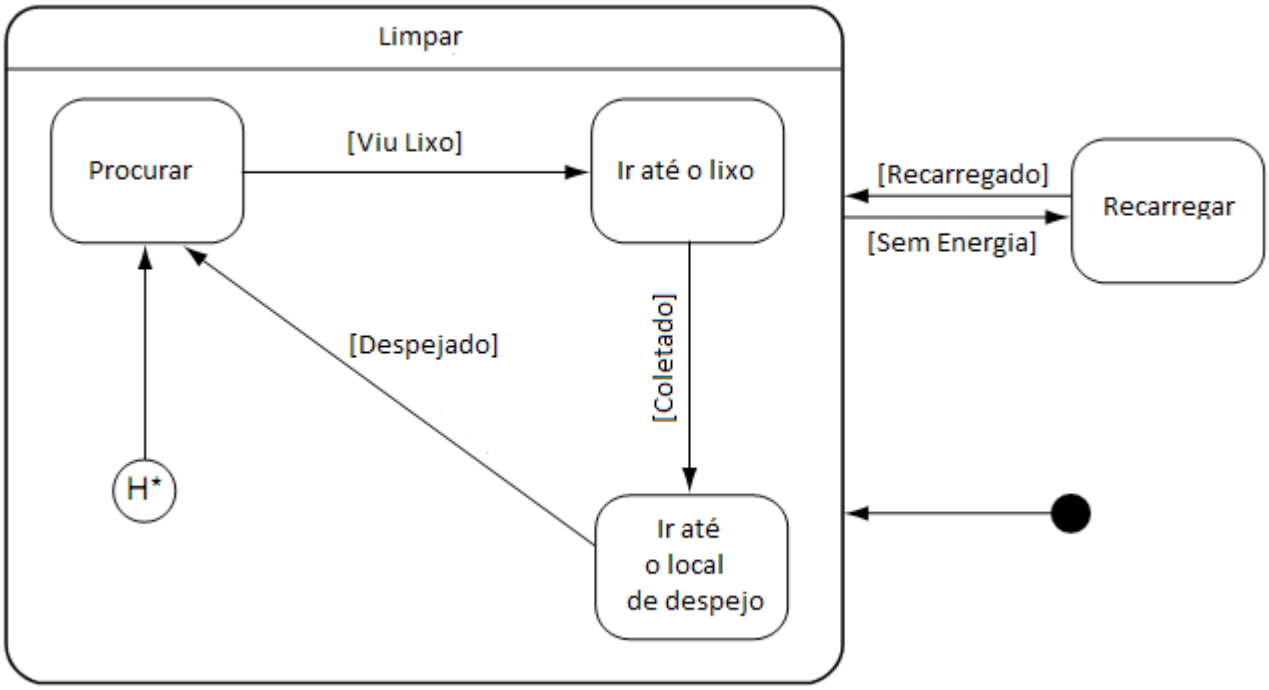

Figura 2.5: *Comparação MEF/MEFH (David M. Bourg, 2004)*

2.4 Árvore de comportamento

"Árvores de comportamento são uma síntese de várias técnicas que são utilizadas em inteligência artificial, por exemplo: Máquinas de estado finita hierárquica, *scheduling*, *planning* e execução de ações. A sua principal vantagem vêm da habilidade de englobar estes conceitos de uma forma que seja simples de entender e fácil de se criar. As árvores de comportamento possuem muito em comum com uma MEFH, mas ao invés de utilizar estados, o principal componente é chamado de tarefa"(Millington, 2006).

Tarefas são divididas recursivamente entre várias sub-tarefas até que se tenha uma árvore. Tomando como exemplo um personagem de algum jogo, ele poderia possuir o comportamento mostrado na figura 2.6.

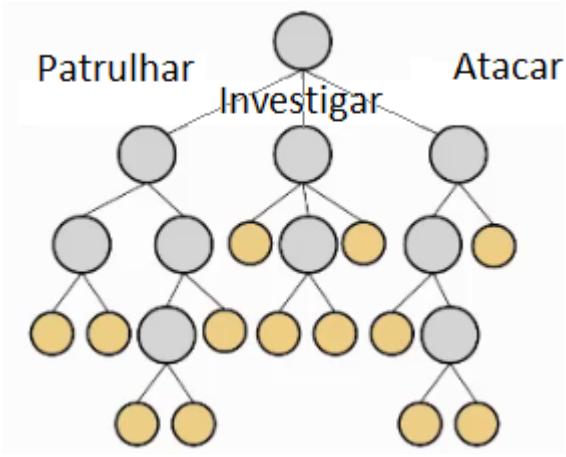

Figura 2.6: *Árvore de Comportamento*

Como pode ser observado, a tarefa principal é dividida em outras três tarefas: patrulhar, investigar e atacar. Cada uma dessas sub-tarefas possuem outras sub-tarefas de mais baixo nível. A figura 2.7 mostra tal situação.

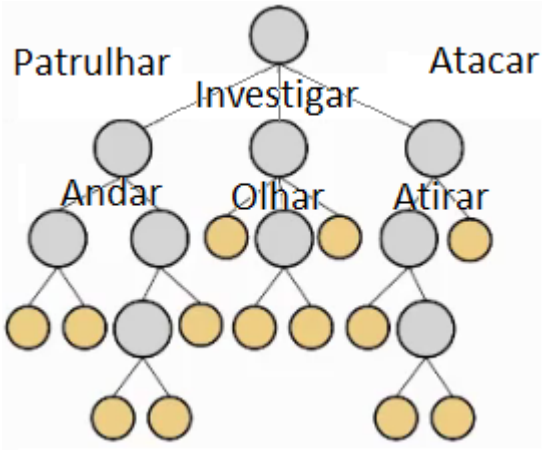

Figura 2.7: *Sub-tarefas*

Novamente, as tarefas vão sendo divididas em sub-tarefas até que se chegue nas folhas da árvore.

As folhas da árvore proporcionam a comunicação entre a lógica da inteligência artificial e a *game-engine* do jogo, sendo utilizadas para buscar informações do que está acontecendo no jogo e verificar se alguma condição é verdadeira ou falsa naquele momento. Alem disso, também são responsáveis por executar ações no jogo.(Millington, 2006)

2.4.1 Tarefas

Segundo Millington (2006), uma tarefa é uma função ou pedaço de código que é executado no jogo de forma latente, ou seja, ele é executado durante vários *frames* do jogo.

Existem dois tipos básicos de tarefas: condições e ações. Condições são utilizadas para se fazer checagens do que está acontecendo na cena do jogo, por exemplo, se o jogador está

com quantidade de vida baixa, se houve alguma colisão, se o jogador está na linha de visão de inimigo, entre outras condições. Elas são utilizadas apenas como leitura, ou seja, não realizam nenhuma mudança na cena.

Ações são utilizadas para realizar mudanças no mundo, por exemplo, tocar uma animação ou som, diminuir a vida do jogador quando um inimigo o acerta.

Quando em execução, uma condição é checada sobre múltiplos frames do jogo até que chegue ao seu término, retornando verdadeiro em caso de sucesso, e falso caso ela falhe. Uma ação funciona de forma similar. Por exemplo, se a ação é tocar uma animação, ela será executada sobre múltiplos *frames* até que a animação seja finalizada, e então, retornará verdadeiro ou falso. É esse retorno que permite a construção de uma lógica complexa na árvore. Essa complexidade é gerenciada pelos nós da árvore, que a partir do retorno gerado pelas tarefas alteram o fluxo de execução da árvore. Outra responsabilidade dos nós é de garantir que seus filhos sejam executados na ordem correta. Estes são também denominados de tarefas compostas.(Millington, 2006)

Existem dois tipos básicos de tarefas compostas, Sequências e Seletores. Sequências são responsáveis por executar suas sub-tarefas uma por uma, quando uma sub-tarefa obtém sucesso na sua execução, a sequência passa a executar a próxima sub-tarefa. Quando uma das sub-tarefas falha a sequência é terminada sem executar as sub-tarefas posteriores. A figura 2.8 mostra um exemplo de funcionamento de uma sequência.

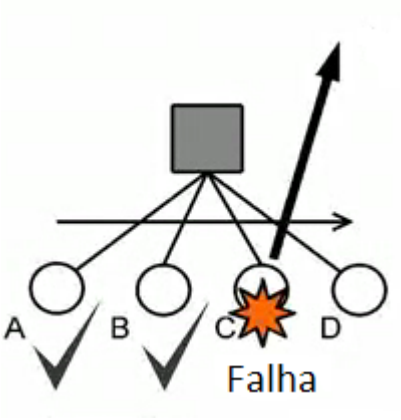

Figura 2.8: *Sequência*

Um seletor seleciona somente uma de suas sub-tarefas para ser executada, essa tarefa pode ser escolhida de várias maneiras dependendo de como o seletor foi implementado. Por exemplo, o seletor poderia escolher uma das tarefas de forma randômica ou sequencialmente. Caso a sub-tarefa falhe em sua execução, o seletor tenta executar outra sub-tarefa. Quando uma das tarefas for executada com sucesso, o seletor termina a sua execução. A figura 2.9 mostra o funcionamento da tarefa composta seletor.

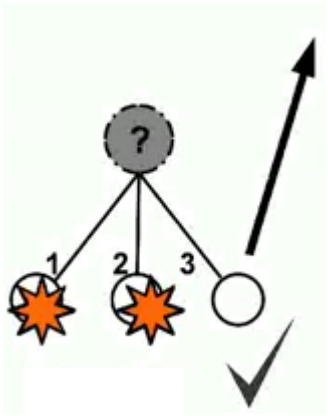

Figura 2.9: *Seletor*

Como pode ser notado, seletores e sequências são opostos no sentido em que uma sequência é terminada quando há uma falha, e um seletor é terminado ao haver um sucesso.(Millington, 2006)

A figura 2.10 mostra um exemplo do fluxo de execução de uma árvore de comportamentos:

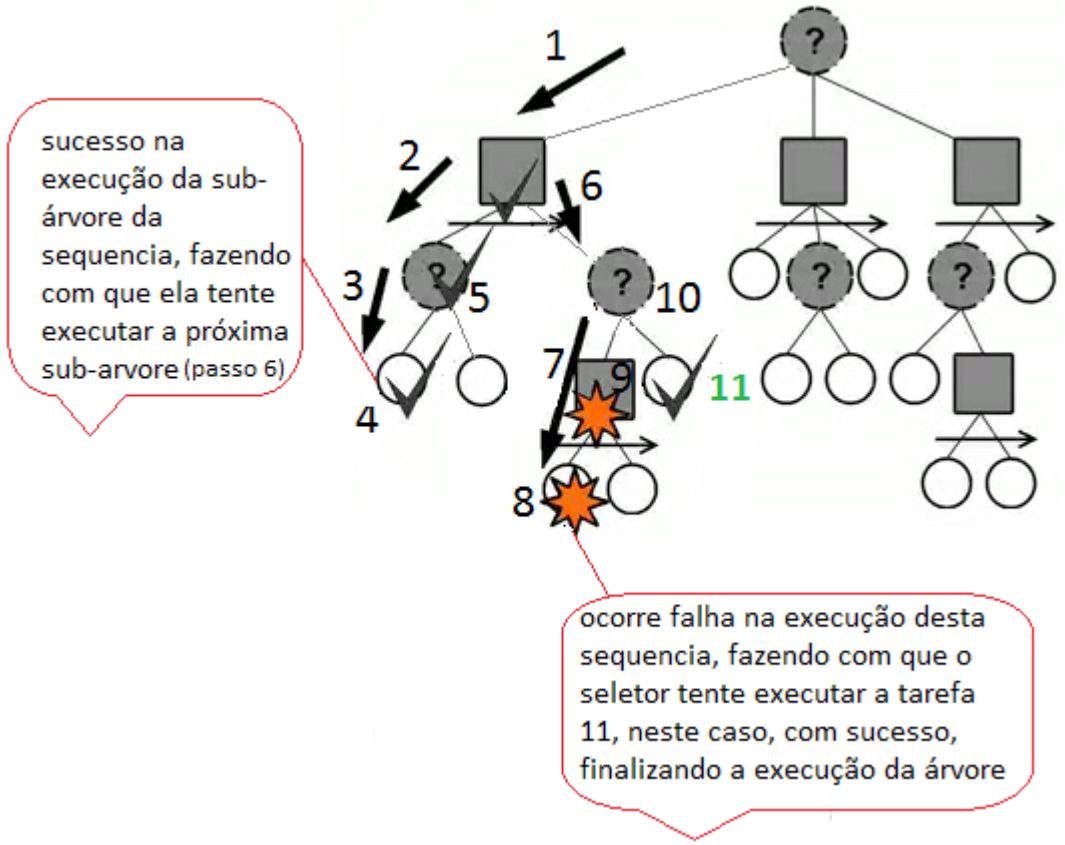

Figura 2.10: *Fluxo de execução de uma árvore de comportamento*

Uma das formas de aumentar a escalabilidade de uma árvore de comportamentos é a criação de tarefas de alto nível, chamadas *Decorators*.

2.4.2 Decorators

Em programação orientada a objetos, o *design pattern* chamado *decorator* tem como função permitir que sejam adicionados novos comportamentos ao método de um objeto sem haver a necessidade de modificar o código original do mesmo.Champandard, 2008a

No contexto de uma árvore de comportamentos, um *decorator* funciona de maneira similar. Ele encapsula a tarefa ou sub-árvore original, adicionando qualquer tipo de funcionalidade adicional que seja necessária. Um dos usos mais comuns dos *decorators* é a criação de filtros. Um filtro basicamente decide se a sub-árvore deve ser executada ou não baseado em alguma condição dinâmica. A figura 2.11 mostra um exemplo de utilização de *decorator*.

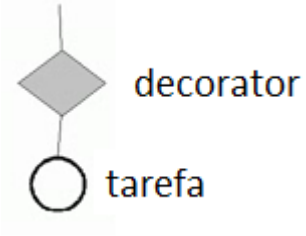

Figura 2.11: *Decorator*

Tomando como exemplo o seguinte comportamento mostrado na figura 2.12:

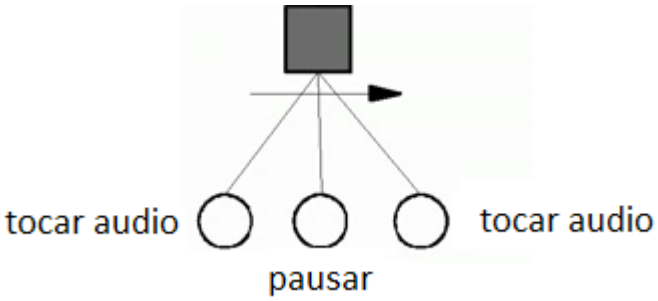

Figura 2.12: *Exemplo de utilização de Decorators*

Caso fosse necessário que este comportamento se repetisse mais de uma vez, poderia ser implementado um *decorator* chamado "loop" que receberia como parâmetro o número de vezes que a sub-árvore deve ser repetida. A figura 2.13 mostra um exemplo de utilização do *decorator loop*.

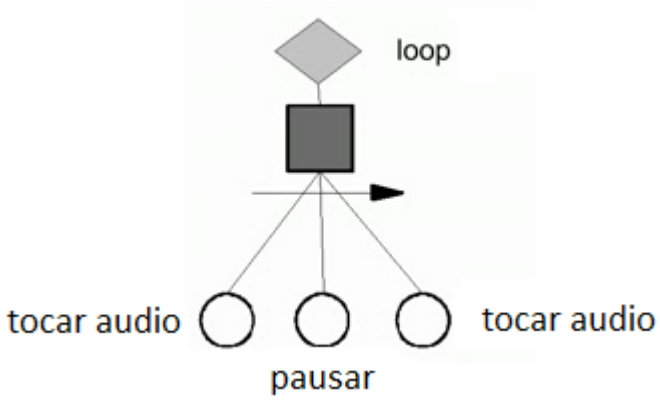

Figura 2.13: *Decorator de loop*

Poderia também ser decidido que seria necessário ignorar qualquer tipo de erro da camada de áudio ao se tentar executar as ações, para fazer isso, bastaria implementar um *decorator* com esta funcionalidade, chamado *wrapper*. Caso haja uma falha ao tocar o primeiro som, ela será ignorada e a sequência continuará a sua execução. A figura 2.14 mostra um exemplo de utilização do *decorator wrapper* aliado com a utilização do *decorator loop*.

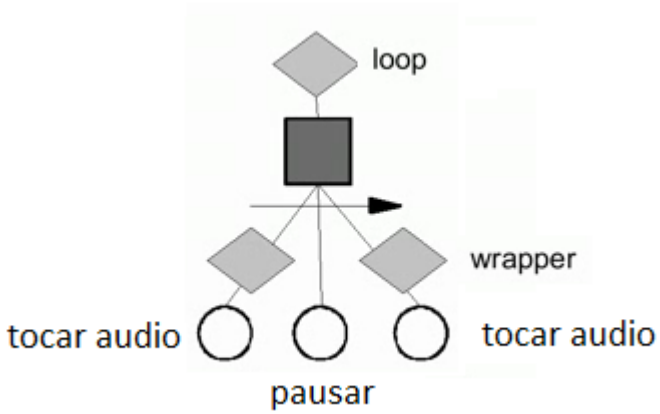

Figura 2.14: *Decorator wrapper*

Outros dois tipos comuns de *decorator* são de *timer* e contador. O *decorator* de *timer* é um filtro que faz com que a árvore não seja executada caso ela já tenha sido executada anteriormente em um período de tempo especificado.

Contadores fazem com que a árvore seja executada um certo número de vezes. Quando o contador chega à zero, a sub-árvore não é mais executada.

Decorators tornam a árvore de comportamentos muito flexível, por isso, é importante que eles sejam extensíveis, pois ao decorrer da criação de novas árvores, surgirá a necessidade de implementar novos tipos de decorators.(Millington, 2006)

2.4.3 Arquitetura dirigida a objetivos

Com o auxílio dos decorators, é possível criar várias árvores de comportamento de forma muito rápida. Para que o sistema seja eficaz, é necessário que as árvores criadas possuam um comportamento dirigido a certo propósito. (Champandard, 2008b)

Tomando como exemplo os comportamentos de um cachorro, seria interessante criar comportamentos do estilo: latir, morder, pular, sentar, comer osso, perseguir. Cada um destes comportamentos buscam atingir certo objetivo, e precisam ser finalizados em um espaço de tempo finito.

Ainda utilizando o exemplo dos comportamentos de um cachorro, se fosse necessário criar o comportamento "Comer Osso", uma das soluções para criar este comportamento é subdividi-lo em comportamentos menores, como "Encontrar Osso" e "Pegar Tigela" e também subdividir estes em comportamentos em comportamentos ainda menores, até que se tenham comportamentos muito simples. A figura 2.15 mostra um exemplo de comportamento utilizando esta técnica.

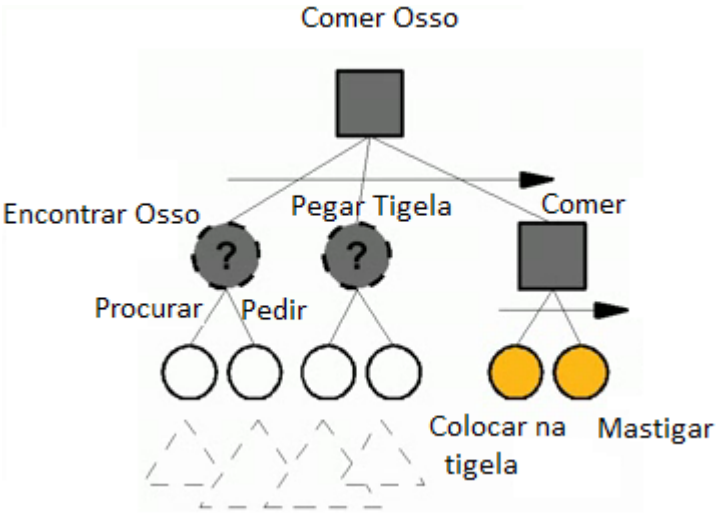

Figura 2.15: *Arquitetura dirigida a objetivos*

No exemplo acima, as sub-árvores "Encontrar Osso" e "Pegar Tigela" são seletores, pois caso a primeira alternativa (Procurar) falhe, o seletor tenta executar a segunda tarefa (Pedir) de forma a tentar finalizar com sucesso a execução. Caso haja sucesso nas duas primeiras tarefas compostas, a sequência "Comer" é chamada para que sejam tocadas duas animações.

O problema do exemplo acima é que a árvore vai acabar ficando muito grande e com várias dependências("Comer Osso" depende de "Encontrar Osso" que depende de "Procurar" e assim por diante), não dando nenhum tipo de modularidade ao sistema.

A solução adequada é separar o que fazer (os objetivos), de como fazer (os comportamentos) para que seja fácil combinar árvores. Uma das formas possíveis é utilizar uma tabela de *look-up*, como mostrado na figura 2.16. (Champandard, 2008b)

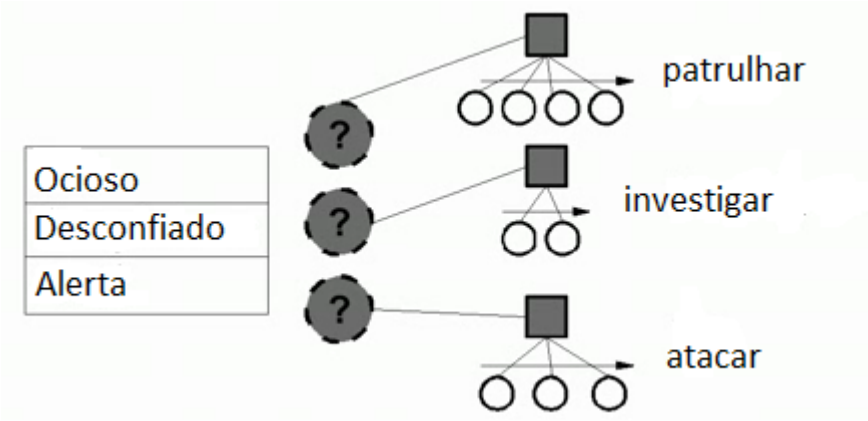

Figura 2.16: *Tabela de look-up*

A tabela possui os objetivos de alto nível e, ligados a cada um desses objetivos existe uma tarefa composta, mais especificamente um seletor. Os filhos do seletor ditarão as várias formas para se alcançar o objetivo a qual ele está ligado na tabela.

O mesmo exemplo mostrado na figura 2.15 utilizando a tabela de *look-up* ficaria da forma mostrada na figura 2.17

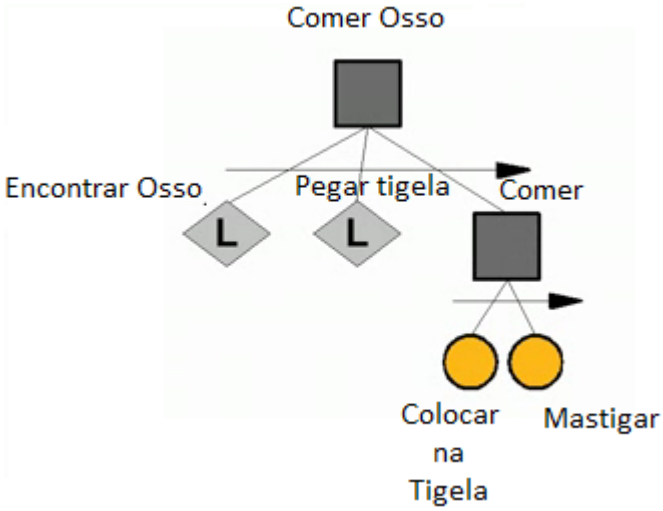

Figura 2.17: *Utilização do decorator de look-up*

Os seletores foram substituídos por um *decorator* de *look-up*. Este *decorator* tem como funcionalidade procurar pelo comportamento referenciado na tabela. Ao se construir a árvore desta forma, pode-se notar que os vários objetivos se tornam modulares, permitindo que eles possam ser utilizados em outras árvores ou contextos diferentes. (Champandard, 2008b)

2.4.4 Boas práticas para projetar árvore de comportamento

Assertions

Assertions são condições que devem ser satisfeitas antes que o restante da árvore seja executado, eles servem essencialmente para podar a árvore caso as condições iniciais falhem (Millington, 2006). Um exemplo do uso de *assertions* pode ser visto na figura 2.18

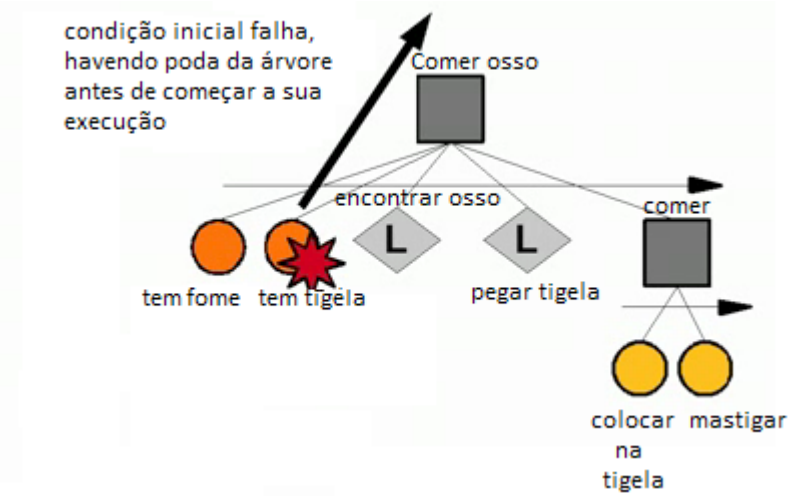

Figura 2.18: *Utilização de assertions em uma sequência*

Da mesma forma, ao se utilizar um seletor, pode-se fazer checagens de condições antes de executar o restante da sub-árvore. Caso haja sucesso na checagem da condição, haverá a poda da árvore, como mostrado na figura 2.19.

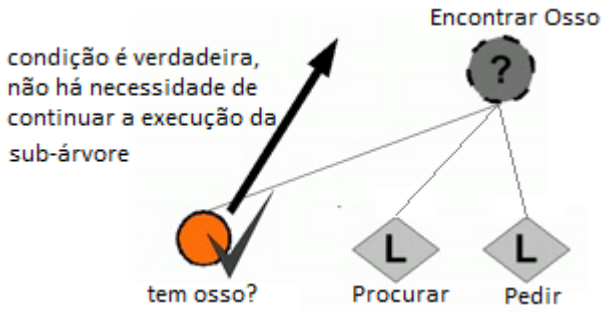

Figura 2.19: *Utilização de assertions em um seletor*

Ordenação de seletores

Outra boa prática na utilização de seletores é ordenar a sua execução, começando, de um plano simples para atingir o objetivo, a um plano mais complexo, ou seja, primeiramente haverá a checagem do *assertion*, caso a checagem falhe, o seletor irá passar a executar o plano mais simples para alcançar o objetivo, caso este também falhe, será executado o plano mais complexo. A figura 2.20 mostra um exemplo de utilização desta técnica.

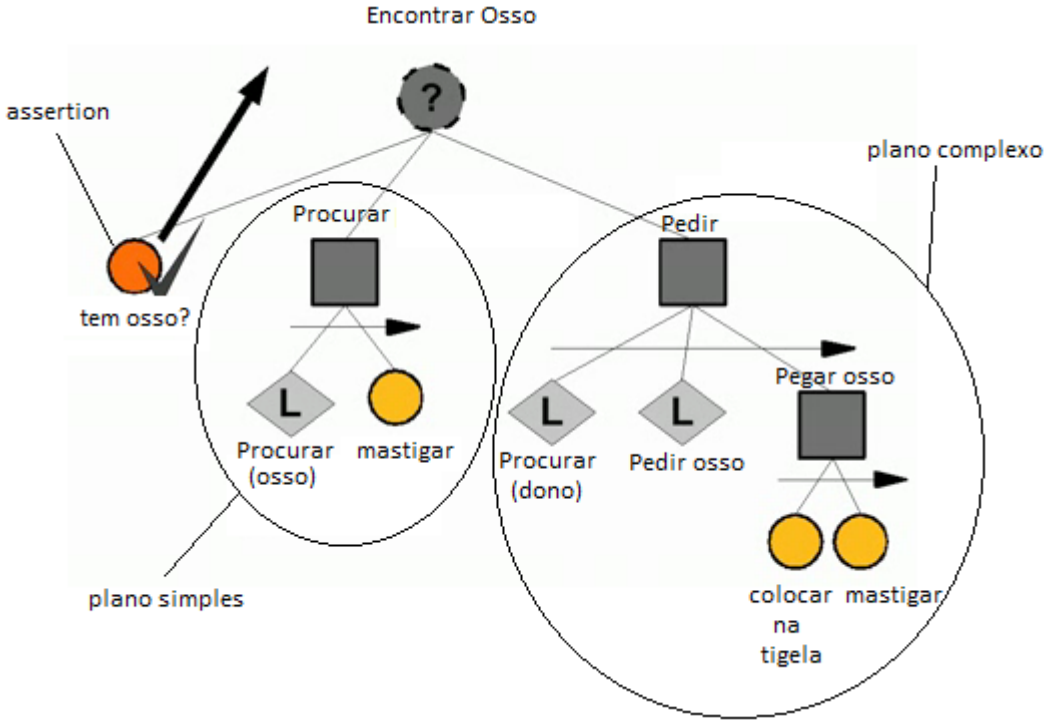

Figura 2.20: *Ordenação de seletores de um plano simples a um mais complexo*

2.4.5 Concorrência

Muitas técnicas para a criação de comportamentos como, por exemplo, uma máquina de estado finita, tem problemas para lidar com concorrência. Uma árvore de comportamentos consegue lidar com isso mais facilmente. (Champandard, 2008a)

Para lidar com concorrência, é necessário utilizar um novo tipo de tarefa composta chamada "Parallel". Um *parallel* funciona como qualquer outra tarefa composta, podendo possuir sub-tarefas e sub-árvores. A diferença de um *parallel* sobre os outros tipos de tarefas compostas, é que ele executa todos os seus filhos ao mesmo tempo. Ele pode ser implementado de várias maneiras. Por exemplo, uma das implementações poderia ser que haverá sucesso na execução do *parallel* quando todos os nós-filhos dele forem finalizados com sucesso. Outra opção poderia ser que vai haver sucesso quando qualquer um dos seus nós-filhos obtiver sucesso.

Concorrência Read-Only

Ainda tomando como exemplo o comportamento "Comer Osso"utilizado anteriormente na figura 2.16, é possível utilizar um *parallel* para constantemente checar por problemas enquanto está executando a ação de "Colocar na tigela"e "Mastigar". Neste caso, o *parallel* não fará nenhuma mudança no mundo, mas caso uma das condições que estão sendo checadas no *parallel* falhem,como por exemplo, um outro cachorro roube o osso, então, toda a execução da sub-árvore pode ser abortada, como mostrado na figura 2.21.

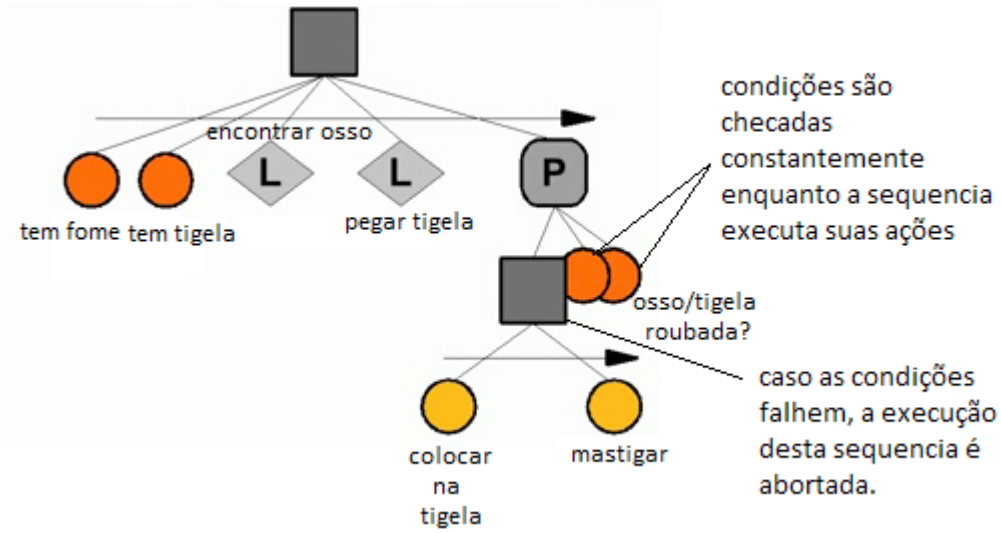

Figura 2.21: *Exemplo de concorrência read-only*

Concorrência de baixo nível

Seguindo o mesmo exemplo, se fosse necessário executar uma animação e tocar um som ao mesmo tempo,poderiam ser utilizados *parallels* em um nível mais baixo da árvore, fazendo com que as duas tarefas sejam executadas ao mesmo tempo conforme mostrado na figura 2.22.

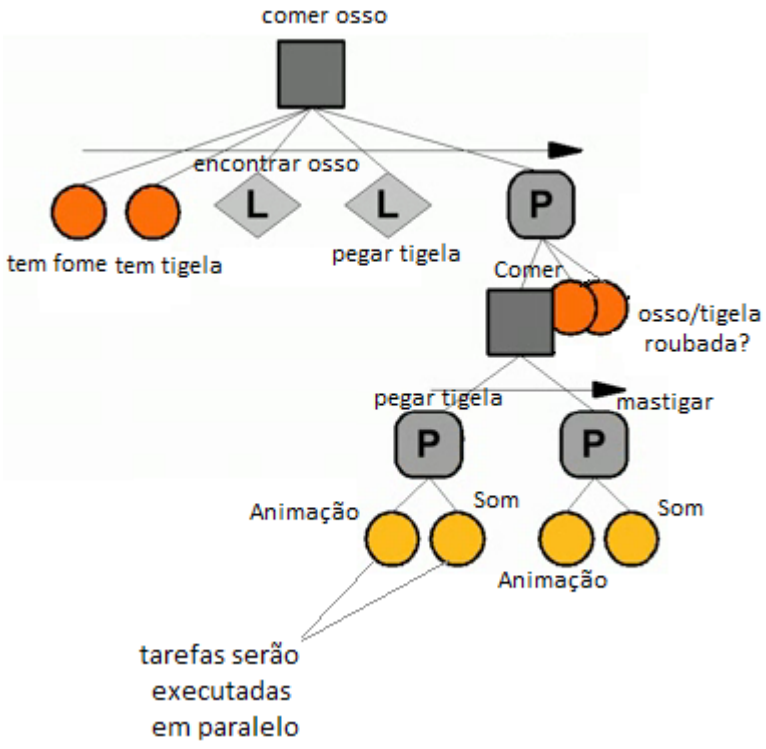

Figura 2.22: *Exemplo de concorrência baixo nível*

Tratando Conflitos e Gerenciando recursos

Ao utilizar concorrência em uma árvore de comportamento, têm-se problemas, os quais, os recursos podem estar sendo utilizados em uma árvore, porém outra também precisa utilizar o mesmo recurso. Por exemplo, caso uma árvore esteja executando uma animação no corpo do personagem, e outra árvore também precise executar uma animação, como saber qual animação deve ser executada? Não há garantia que as duas animações irão funcionar em conjunto. A solução para isso é garantir que não haja conflitos. Somente uma animação pode ser executada no corpo do personagem por vez, assim como somente um som pode ser executado por vez.(Millington, 2006)

Uma das soluções é implementar um outro tipo de decorator, que realize a função de um semáforo. Este é utilizado em programação para restringir acesso a um recurso compartilhado em um ambiente multiprogramado.

Na figura 2.23, os três decorators de semáforo possuem uma variável compartilhada, que diz se é possível continuar a execução da sub-árvore ou não. No exemplo, a árvore C checa se é possível continuar a execução, como a checagem é verdadeira, a execução continua normalmente e o recurso é bloqueado. Quando a árvore A tentar executar a sua sub-ávore, ela irá falhar, pois o recurso está bloqueado.

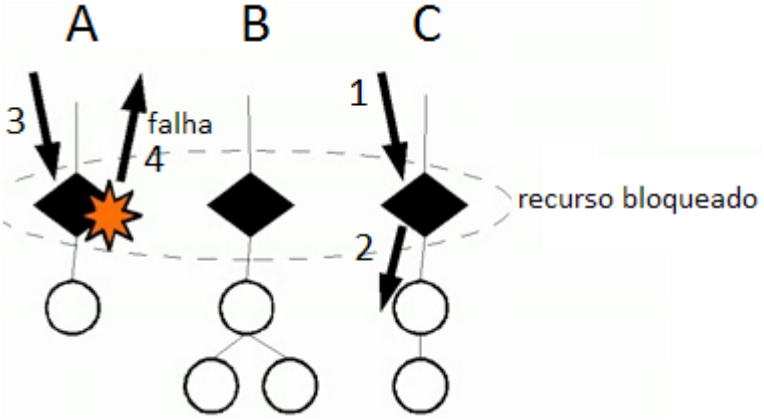

Figura 2.23: *Utilização de semáforos*

Uma das grandes vantagens na utilização de árvores de comportamento, é justamente a facilidade em se trabalhar com concorrência, mesmo quando houver uma falha devido a um recurso estar bloqueado, a árvore irá dinamicamente procurar por uma solução alternativa, ou seja, não haverá dead-lock na árvore. (Champandard, 2008a)

O decorator de semáforo não precisa ser somente utilizado para bloqueio no acesso de recursos, outros exemplos de utilização incluem: limitar o número de pessoas que possam se locomover em um grupo, restringir o número de inimigos que estão atirando ao mesmo tempo, entre outros.

2.5 Considerações

Neste capítulo foram apresentados os conceitos básicos que fundamentam este trabalho de conclusão de curso. Foi apresentado o sistema de entidades utilizado pela Unity3D, bem como

as técnicas de criação de comportamentos mais utilizadas na indústria de jogos. Também foi detalhado o funcionamento de uma árvore de comportamentos e as técnicas utilizadas na sua construção.

Capítulo 3

Implementação

Neste capítulo são apresentadas as etapas para o desenvolvimento da ferramenta. A implementação da ferramenta se dará de forma iterativa, sendo primeiro implementado um sistema simples, e a partir dele, adicionado novas funcionalidades.

Um dos aspectos importantes a se levar em conta ao longo do desenvolvimento da ferramenta é o sistema de agregação de componentes utilizado pela Unity3D como já explicado no capítulo 2. É necessário atentar-se a isso, pois vai garantir que a ferramenta fique mais integrada com o restante da *game-engine*.

3.1 Primeira etapa de desenvolvimento – Implementação simples

3.1.1 Requisitos

Na primeira etapa do desenvolvimento da ferramenta, os seguintes requisitos devem ser satisfeitos:

- Implementação da tarefa-composta Seletor;
- Implementação da tarefa-composta Sequência;
- Implementação de Ações e Condições;

Tarefas compostas, ações e condições deverão funcionar da mesma forma que foi descrita no capítulo 2.

Nesta etapa, não foi desenvolvida nenhuma parte da interface gráfica, e sim o núcleo de funcionamento do sistema. Ao alcançar os requisitos propostos acima, já será possível executar árvores de comportamentos no console da Unity3D.

3.1.2 Desenvolvimento

A *game-engine* Unity3D permite a programação nas linguagens C Sharp, Javascript e Boo. Foi decidido que o sistema será implementado em C Sharp por ser a linguagem com mais funcionalidades extras dentre as possíveis escolhas. Pode-se citar como funcionalidade extra o uso de *delegates*, métodos anônimos, entre outros.

Para que um componente possa ser agregado a uma entidade do jogo, é necessário que ele herde as funcionalidades da classe *MonoBehaviour*. Esta é a classe base da Unity3D, que expõe todas as funcionalidades do *game-engine*. Com esta classe herdada, é possível implementar os métodos *Start*, *Update* e *Stop*.

O método *Start* é executado assim que o jogo é iniciado ou assim que o componente é agregado a uma entidade.

O método *Update* é executado uma vez a cada frame do jogo até que o componente seja removido da entidade, e o método *Stop* é executado quando o componente é removido.

Classes

NA figura 3.1 se encontra o diagrama de classes do sistema. Primeiramente foi criada uma interface chamada *IBaseTask*. Toda tarefa(ação, condição) ou tarefa composta(sequência, seletor, *parallel*, *decorator*) deverá implementar esta interface pois ela possui os métodos, eventos e propriedades que serão necessárias para a sua execução.

A classe *CompositeTask* é a representação de uma tarefa composta. Além de implementar a interface *IBaseTask*, ela possui uma lista de filhos que serão executados.

TaskExample é uma tarefa de exemplo que implementa a interface *IBaseTask* e herda as funcionalidades da classe *MonoBehaviour*. Com isso, as funcionalidades da *game-engine* ficam expostas para que possam ser utilizadas dentro da tarefa. Para implementar novas tarefas, basta seguir o exemplo da tarefa implementada.

TreeExample é uma árvore de exemplo que possui uma *CompositeTask* que representa a raiz da árvore. Esta classe também herda as funcionalidades do *MonoBehaviour* pois ela poderá ser agregada a qualquer entidade presente na cena do jogo.

A classe *Scheduler* é responsável pela execução de todas as árvores que estão presentes na cena do jogo.

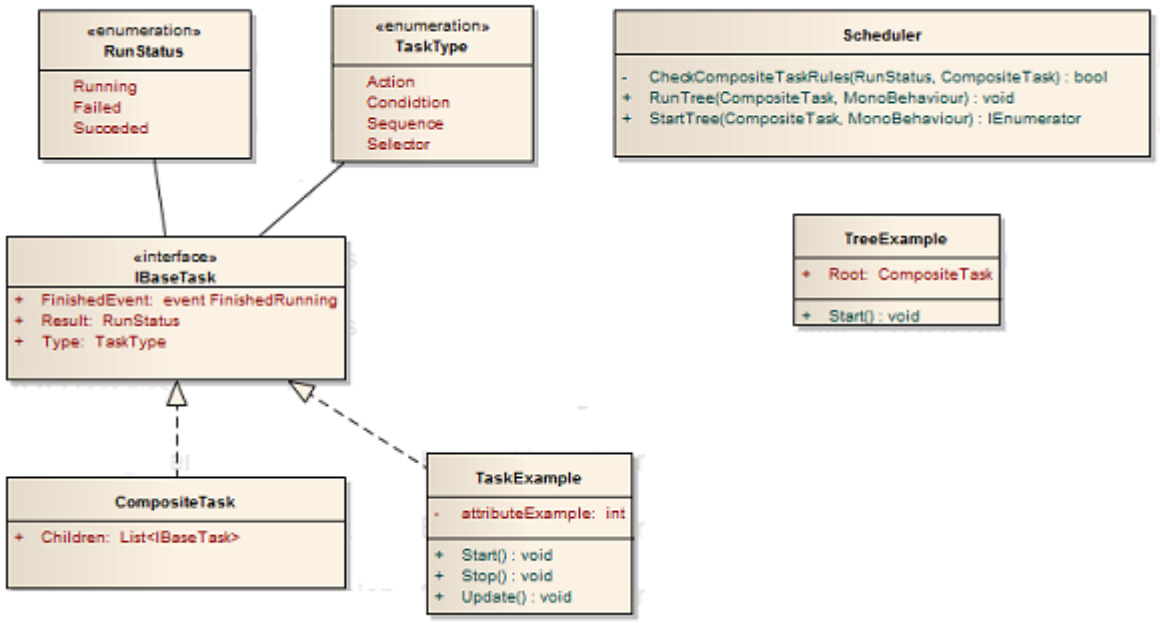

Figura 3.1: *Diagrama de classes do sistema simples*

Funcionamento do Scheduler

No início da cena do jogo, as árvores que estão agregadas as entidades da cena enviam um evento para a classe *Scheduler*, avisando que ela deve começar a ser executada. A classe *Scheduler* então passa a executar a árvore dentro de uma *Co-routine*, permitindo que várias árvores possam ser executadas ao mesmo tempo.

Tomando como exemplo a árvore mostrada na figura 3.2 :

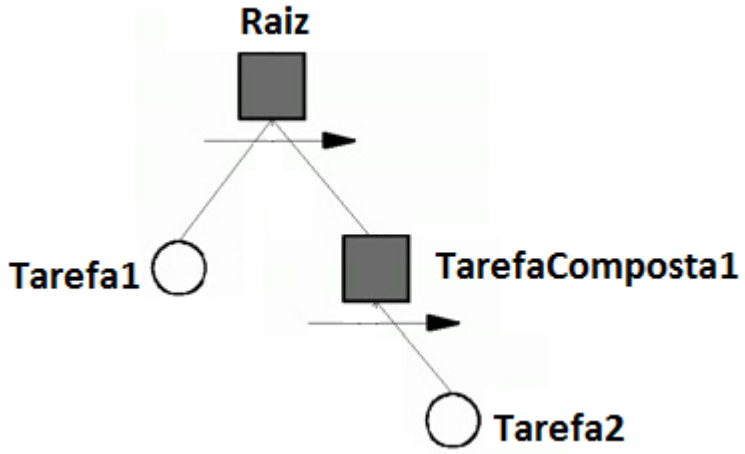

Figura 3.2: *Exemplo de árvore para demonstrar o funcionamento do Scheduler*

O fluxo de execução do *Scheduler* será o mostrado na figura 3.3:

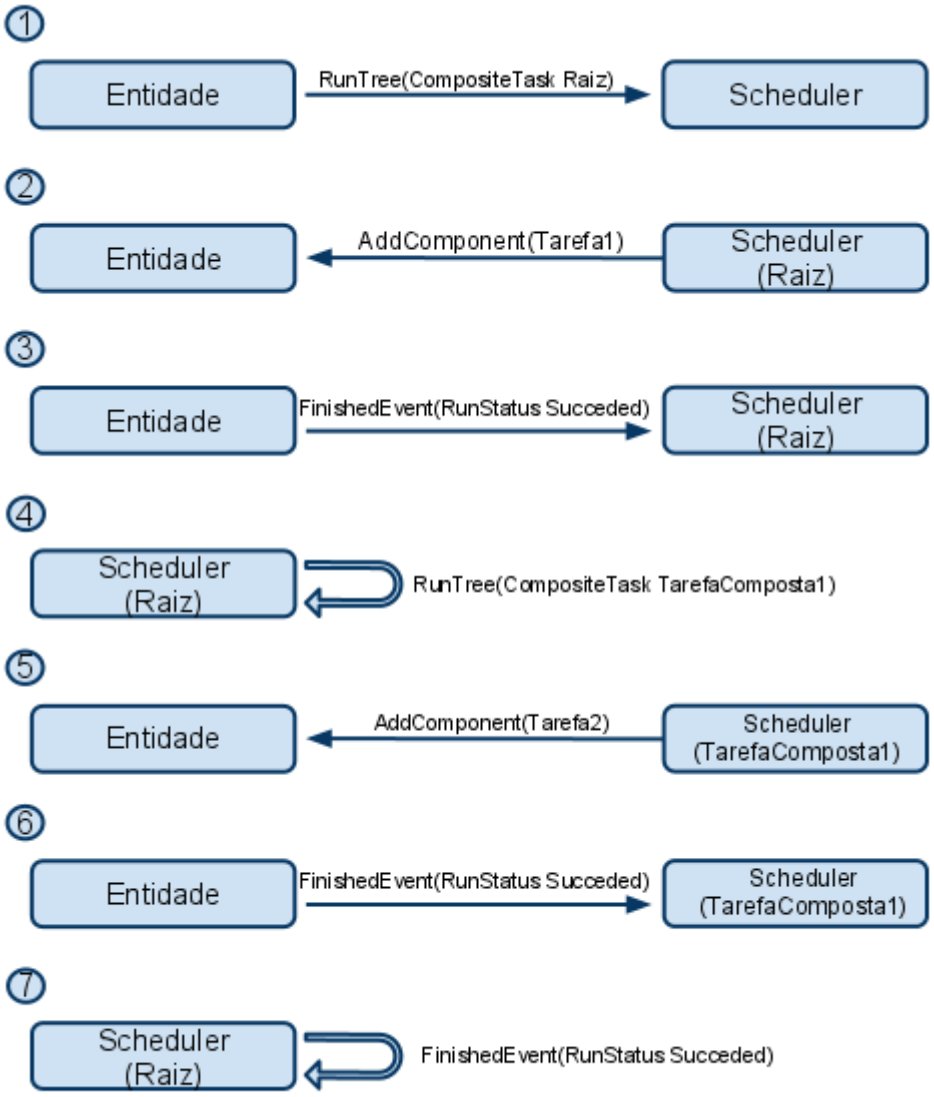

Figura 3.3: *Fluxo de execução do Scheduler*

1 - A entidade que possui uma árvore agregada como componente envia um evento para o *Scheduler*, passando como parâmetro a raiz da árvore.

2 - O *Scheduler* começa a executar os filhos da raiz da árvore. No exemplo, o primeiro filho é a Tarefa1. Quando uma ação ou condição é encontrada, ela é adicionada como um componente na entidade, sendo executado o método *Start* da tarefa, e começando a sua execução. Enquanto isso, o *Scheduler* espera o fim da execução da tarefa.

3 - A tarefa envia um evento avisando que terminou sua execução, neste caso, com sucesso.

4 - O *Scheduler* então passa a executar o próximo filho da raiz da árvore, a *TarefaComposta1*. Tarefas compostas são executadas de forma recursiva, chamando o próprio método *RunTree* e passando como parâmetro a tarefa composta. o *Scheduler* então espera o término da execução da tarefa composta.

5 - Os filhos da tarefa composta começam a ser executados, neste caso, a Tarefa2, que é adicionada como componente da entidade, como explicado anteriormente.

6 - A execução da *Tarefa*2 é finalizada com sucesso, sendo enviado um evento para o *Scheduler*.

7 - Como a TarefaComposta1 não tem mais filhos, então ela é finalizada com sucesso, enviando um evento para a raiz e finalizando a execução da árvore.

3.2 Segunda etapa de desenvolvimento – Parallels e Decorators

Ao final do desenvolvimento desta etapa é possível utilizar a tarefa composta *Parallel*, além do uso de *Decorators*. Um *Decorator* foi implementado como exemplo, outros *Decorators* podem ser implamentados seguindo este exemplo.

3.2.1 Parallels

Parallels devem ser implementados como descritos no capítulo 2, ou seja, todos os seus filhos devem ser executados ao mesmo tempo. Como o *Scheduler* executa apenas um filho de uma tarefa composta por vez foi necessário criar um novo método com esta função.

O fluxo de execução do *Scheduler* permanece o mesmo descrito anteriormente, com a diferença de que se for encontrado um Parallel na árvore, é chamado o método *RunAllChildsAtOnce* para executar todos os filhos ao mesmo tempo. O *Scheduler* então espera que o *Parallel* termine a execução da sua sub-árvore.

Da forma que o *Parallel* foi implementado, todos os seus filhos serão executados e, após a execução, caso algum tenha falhado, o *Parallel* retornará falso. Outras formas de execução do *Parellel* podem ser implementadas no futuro, fazendo com que ele fique mais flexível.

3.2.2 Decorators

Para a implementação dos *Decorators* foi criado uma uma nova interface chamada *IDecorator*. O *Decorator* deve implementar esta interface, além da interface *IBaseTask*.

Quando o *Scheduler* encontrar um *Decorator* que precisa ser executado na árvore, ele chama o método *RunDecorator*, passando como parâmetro a sub-árvore que deve ser executada. O *Decorator* então pode alterar o fluxo da execução da sub-árvore como desejar, e ao final, retorna o resultado da sua execução.

Como exemplo, foi implementado um *Decorator* de *loop*. Passando como parâmetro o número de vezes que a sub-árvore será executada, ele executará a sub-árvore o número de vezes estipulado independente do resultado da sua execução anterior. Outros tipos de *Decorators* podem ser implementados seguindo este exemplo.

3.3 Terceira etapa de desenvolvimento - Interface Gráfica

- A interface gráfica deve fornecer um método visual para editar as árvores de comportamento.
- Deve ser possível salvar e abrir árvores já criadas.
- Ao salvar uma árvore um arquivo de classe deve ser gerado, de forma que ele possa ser utilizado para agregar a árvore a qualquer entidade do jogo.
- A figura 3.4 mostra um projeto da interface que foi desenvolvida.

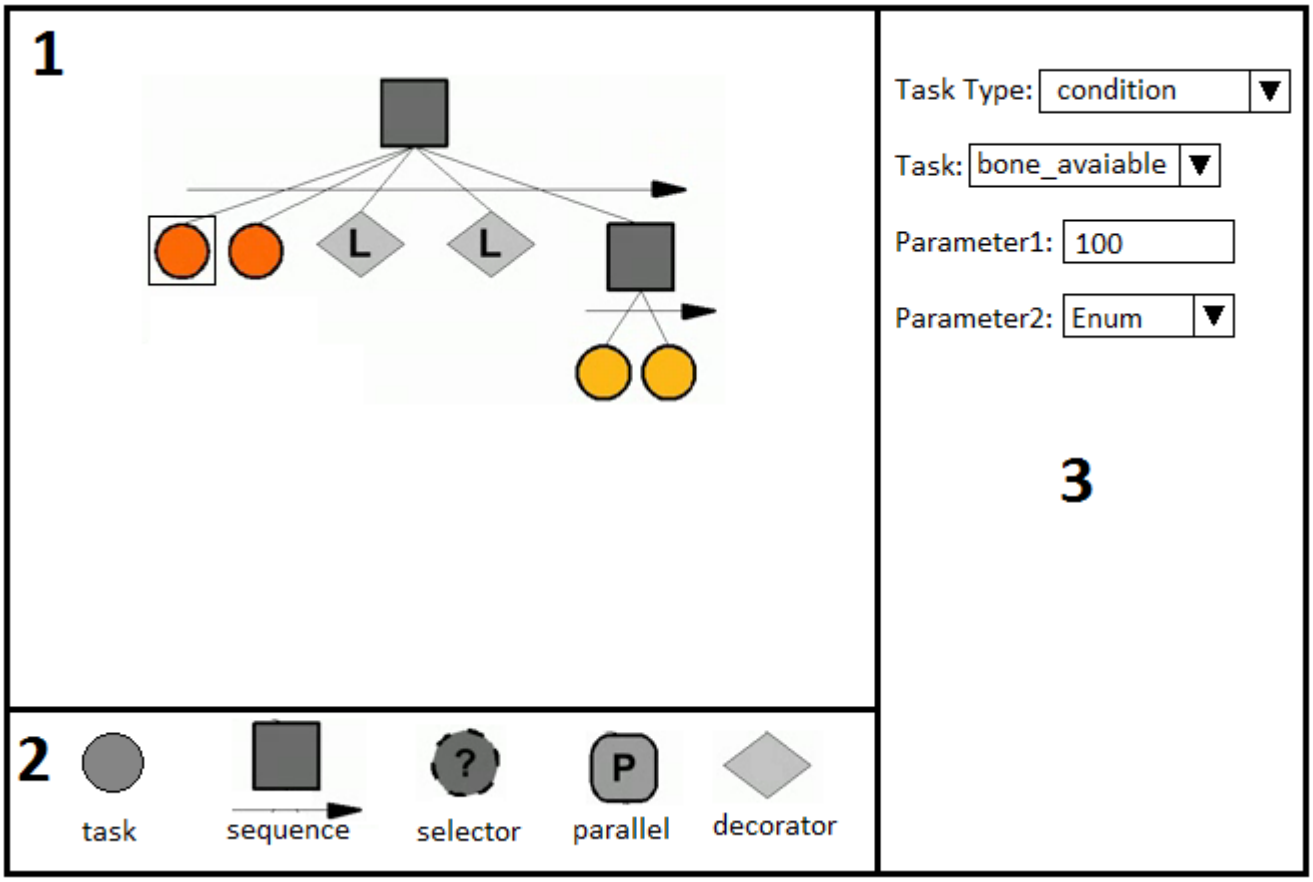

Figura 3.4: *Projeto da interface gráfica. 1 – Árvore 2 – tarefas/tarefas compostas/decorators 3 – Menu de contexto*

3.3.1 Desenvolvimento

A Unity3D permite que sua interface seja extendida bastando que a classe esteja dentro da pasta Editor e herdando as funcionalidades da classe *EditorWindow*. Com isso, os métodos de criação de interface ficam expostos a classe. Na figura abaixo pode ser observado o resultado final da implementação da interface. Cada nó da árvore é um botão que pode ser selecionado. Ao ser selecionado, caso ele possua alguma variável adicional, ela é mostrada no menu de contexto ao lado. Os nós vão são inseridos abaixo do nó que está selecionado até que se tenha a árvore completa.

Quando uma árvore é gerada, além do arquivo de classe que é utilizado para agregar a árvore a uma entidade, também é gerado um arquivo XML da árvore. Esse arquivo pode ser utilizado caso seja preciso abrir a árvore no editor novamente.

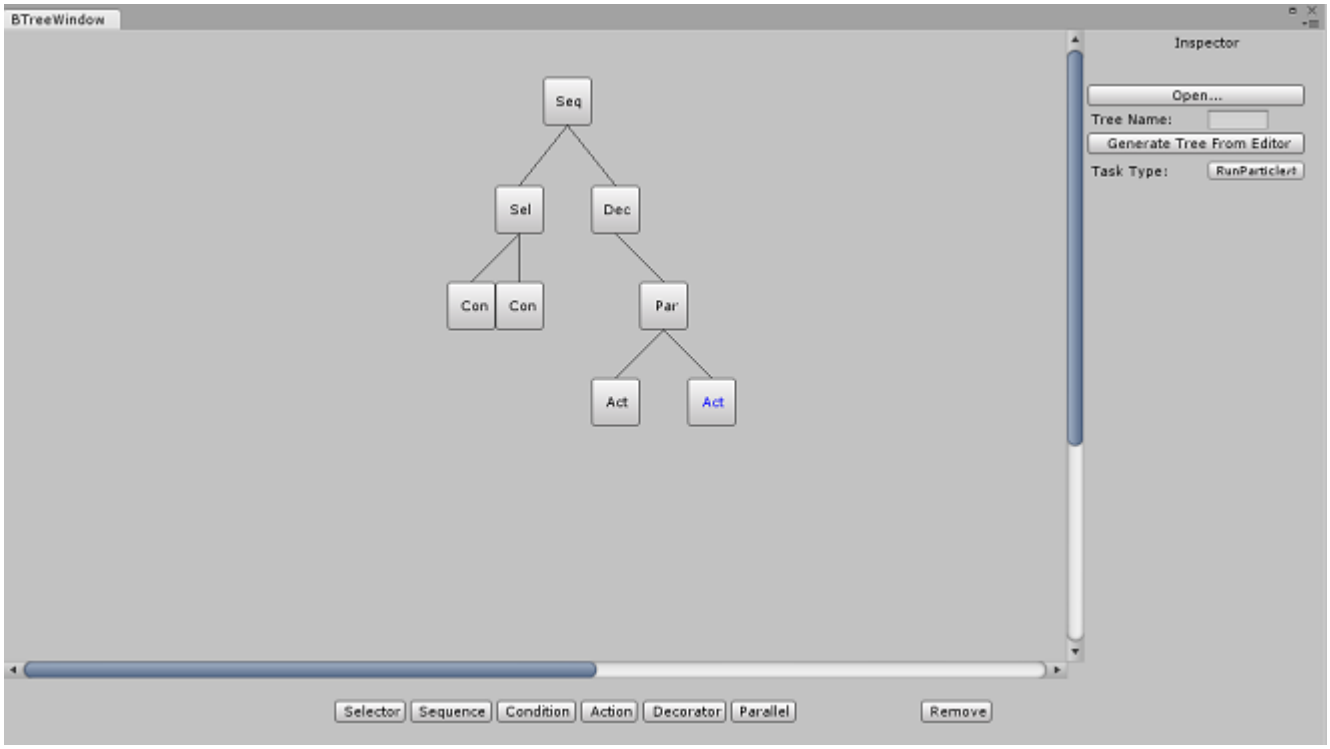

Figura 3.5: *Interface gráfica implementada*

3.4 Considerações

Neste capítulo foram apresentada as etapas de implementação da ferramenta. O sistema foi implementado de forma que um scheduler fique responsável por executar todas as árvores de comportamento do jogo. As árvores podem ser agregadas as entidades como qualquer outro componente utilizado na Unity3D, estando integrada com o resto do sistema de gerenciamento de entidades. A interface gráfica foi implementada em uma nova janela dentro do próprio ambiente da *game-engine*.

Capítulo 4

Experimentos e Resultados

Neste capítulo serão apresentados os resultados obtidos com a utilização da ferramenta implementada.

Para testar o funcionamento da ferramenta, foi criada uma cena de teste. Na cena existe uma entidade chamada Cubo e outra entidade chamada GUI. Existe um script agregado a entidade GUI que possui duas variáveis booleanas, "rotate" e "particle", que estão ligadas a dois botões na cena, conforme é mostrado na figura 4.1.

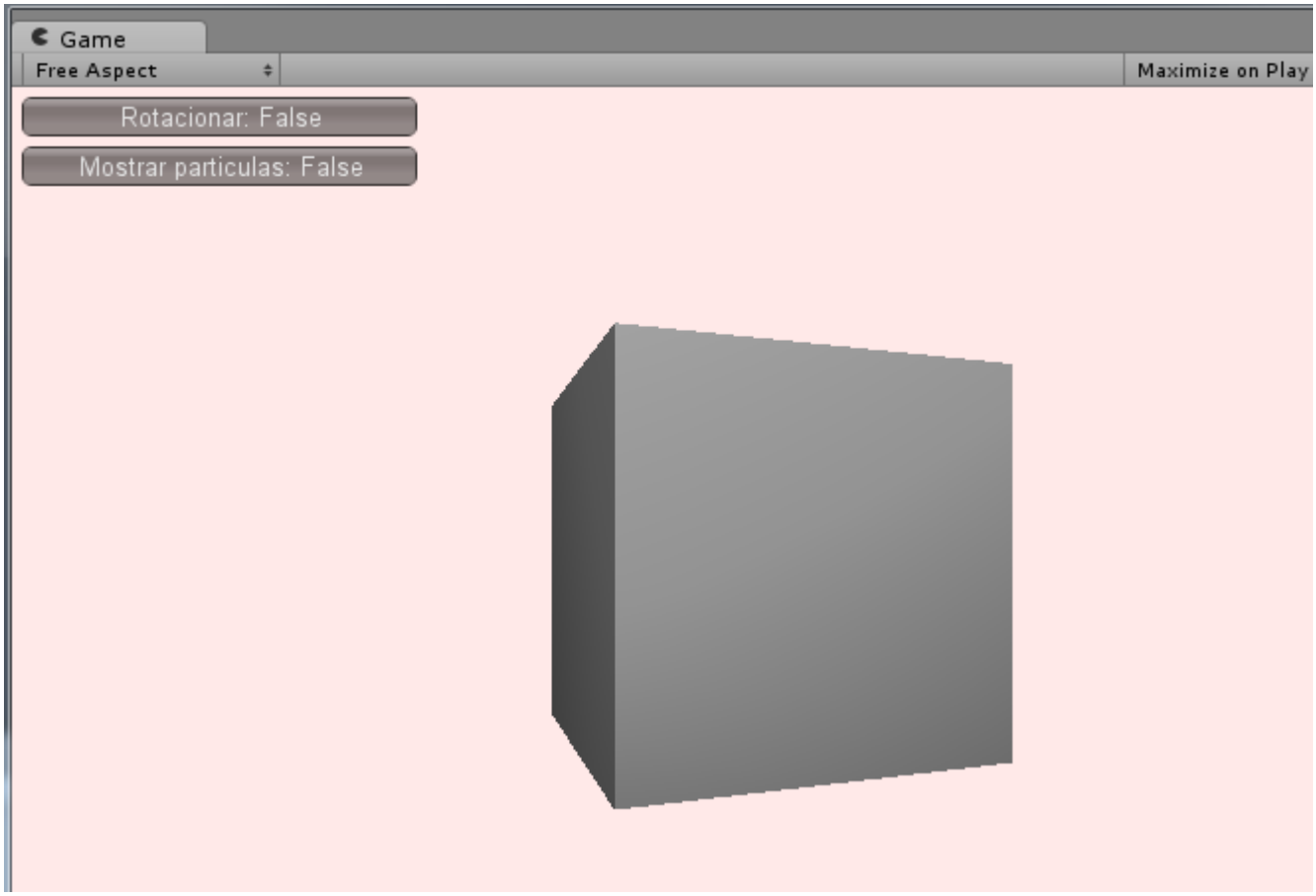

Figura 4.1: *Cena de teste da ferramenta*

Quando os botões são clicados, o valor da respectiva variável(true/false) muda. Foram criadas duas tarefas que executam ações quando o valor destas variáveis é verdadeiro. Por

exemplo, quando o valor da variável "rotate" é verdadeiro, a tarefa faz a entidade rotacionar por 10 segundos, terminando a sua execução com sucesso. Caso contrário a tarefa falha na sua execução. A tarefa "particle" funciona de forma similar, mas em vez de rotacionar a entidade são criadas partículas em volta dela.

Através do uso da interface gráfica da ferramenta desenvolvida foram criadas as árvores mostradas nas figuras 4.2 e 4.3.

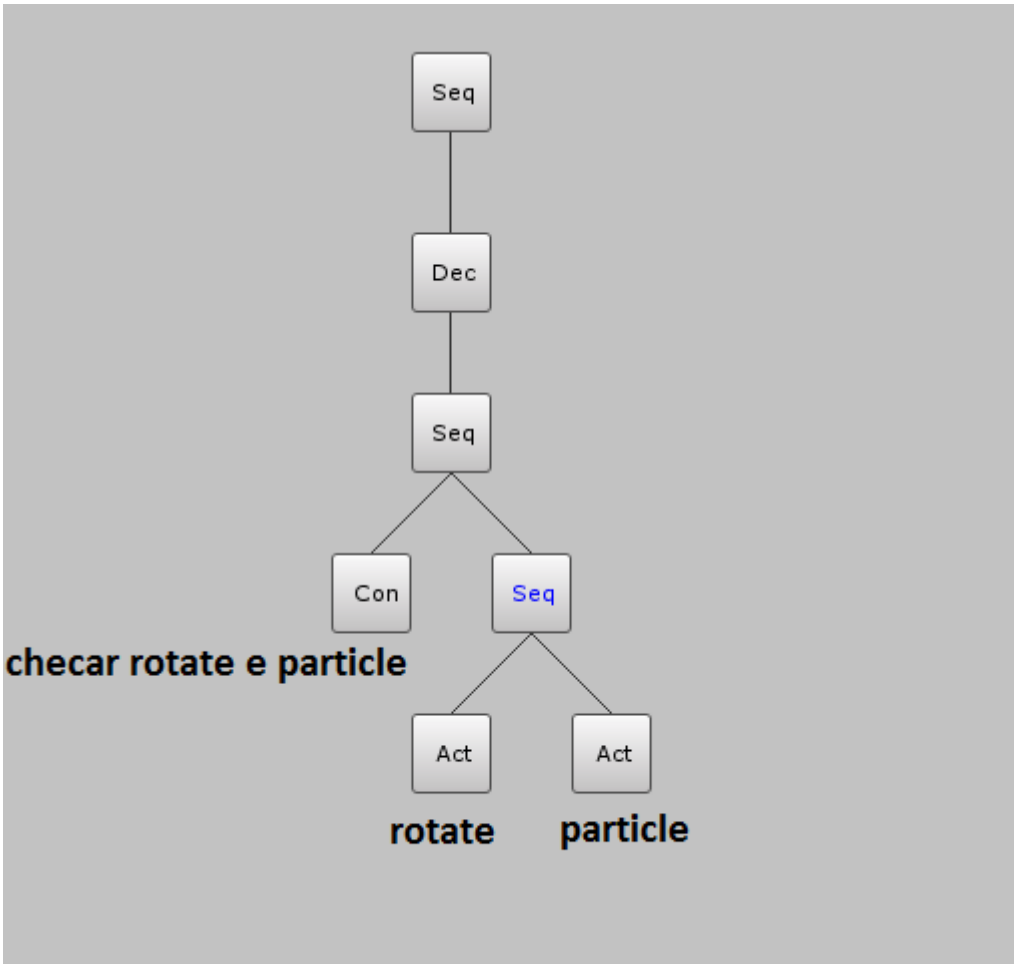

Figura 4.2: *Árvore Implementada de Exemplo - Funcionamento de Forma Síncrona*

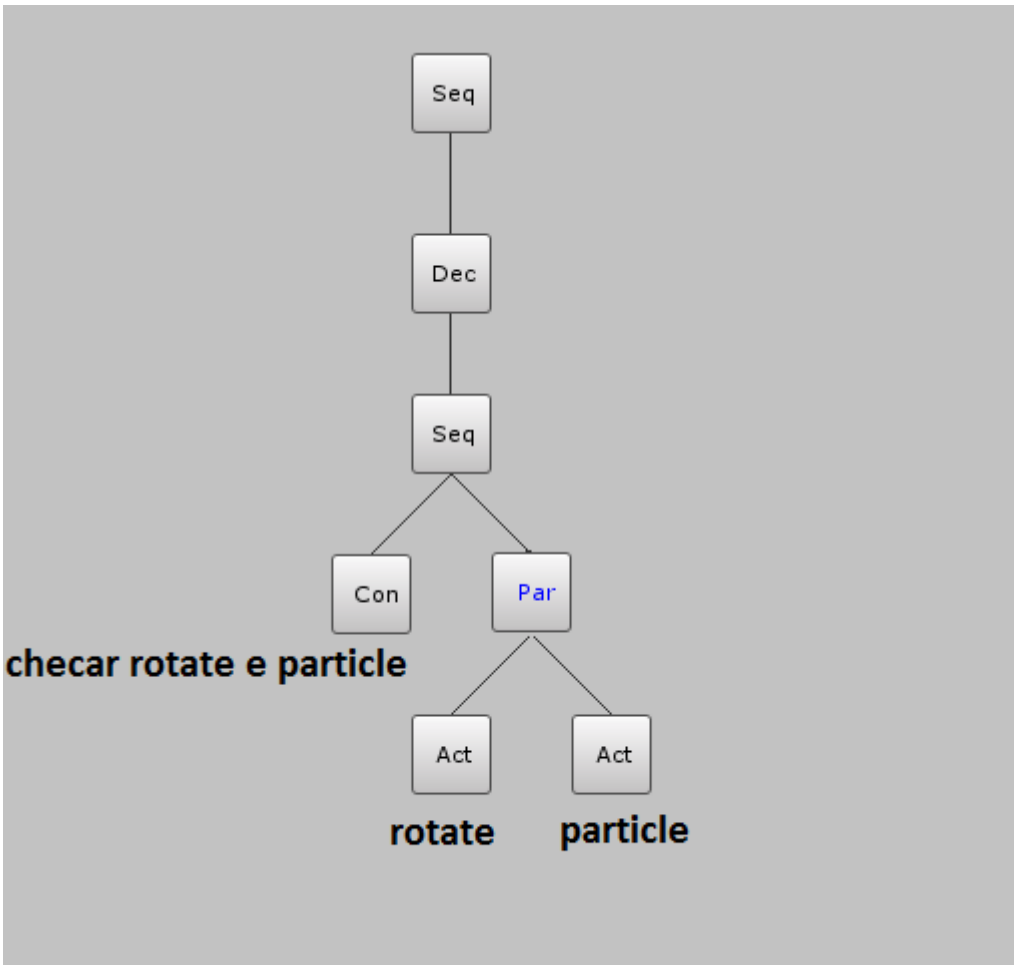

Figura 4.3: *Árvore Implementada de Exemplo - Utilizando a tarefa composta Parallel*

O *Decorator* de *loop* é utilizado logo no início das duas árvores, de forma que ela fique sendo executada por tempo indefinido. A primeira árvore é executada de forma síncrona, ou seja, a primeira tarefa a ser executada será a condição *"checar rotate e particle"*.Esta condição checa se o valor das variáveis *"rotate"* e *"particle"* é falso. Caso o valor de alguma das variáveis seja verdadeira, vai ser executada a sequência que contém as duas ações.A primeira tarefa a ser executada será a ação *"rotate"*, e em seguida a ação *"particle"*.

A diferença da segunda árvore é a utilização da tarefa composta *parallel* em vez de uma sequência para executar as ações. Desta forma, as duas ações serão executadas ao mesmo tempo.

Ao executar a cena de teste com as duas árvores, o Cubo se comportou da maneira esperada, validando a utilização da ferramenta.

4.1 Considerações

Pôde-se notar que o uso da ferramenta desenvolvida facilita a criação de novos comportamentos para as entidades, fazendo com que as tarefas implementadas possam ser reutilizadas na criação de novos comportamentos.

Caso a ferramenta não fosse utilizada no exemplo, seria necessário implementar um sistema parecido ou utilizar somente *scripts* agregados diretamente a entidade. O exemplo é muito simples, tornando viável o uso de *scripts*, mas caso o projeto fosse um jogo completo, o uso puramente de *scripts* comprometeria o desenvolvimento do jogo.

Capítulo 5

Conclusão

Neste trabalho de conclusão de curso foram apresentadas algumas técnicas que são utilizadas na indústria de jogos para a criação de comportamentos e foi feito um detalhamento do funcionamento das árvores de comportamento. Alem da apresentação destes conceitos, foi implementado uma ferramenta para a *game-engine* Unity3D que permite a construção de árvores de comportamento de forma visual.

Com a criação de uma cena para testar o funcionamento da ferramenta pôde-se concluir que a utilização de árvores de comportamento é uma solução viável para a criação de comportamentos para jogos. Aliadas com a utilização da interface gráfica desenvolvida, pôde-se notar a facilidade em se criar e projetar novos comportamentos.

Da forma como foi implementada a ferramenta, novos *Decorators* podem ser implementados com facilidade, deixando o sistema flexível para que se possa construir os mais variados comportamentos que um jogo de qualquer gênero pode exigir.

5.1 Trabalhos Futuros

- Podem ser implementadas novas ferramentas para a *game-engine* Unity3D, como por exemplo a implementação de uma ferramenta de *Pathfind*. Essas ferramentas podem ser integradas com a ferramenta desenvolvida neste trabalho, criando uma suíte de ferramentas.
- Melhorar a usabilidade da interface gráfica. Ela poderia ser melhorada com o uso de *drag-and-drop* por exemplo. Também poderiam ser implementados novos *Decorators* como semáforo e *look-up*.
- Segundo Millington (2006), uma das limitações das árvores de comportamento é responder a eventos externos. Um personagem que está executando a tarefa de patrulhar uma rota e subitamente precise acionar um alarme seria um dos exemplos. Não é impossível de se criar uma árvore de comportamentos com essa função, só é trabalhoso. Ele sugere como solução que as tarefas possam ser implementadas internamente como se fossem máquinas de estado, criando um sistema hibrido que é capaz de detectar os eventos importantes que acontecerem ao longo da execução da tarefa e terminar a sua execução caso seja necessário. Outra solução sugerida por Millington é fazer com que as entidades possuam múltiplas árvores de comportamento, e uma máquina de estados decida qual árvore deve ser executada.

Referências Bibliográficas

Bryson(2003) J. Bryson. The behavior-oriented design of modular agent intelligence. in proceedings of the agent technologies, infrastructures, tools and applications for e-services workshop, lncs 2592, springer, 2003. Citado na pág.

C. Hecker e Dyckho(2007) M. Argenton C. Hecker, L. McHugh e M. Dyckho. Three approaches to halo-style behavior tree ai. games developer conference, audio talk., 2007. Citado na pág. 1

Champandard(2008a) Alex J. Champandard. Gdc presentation: Behavior trees for next-gen ai. in proccedings od the game developer conference 2008, 2008a. Citado na pág. 1, 14, 19, 21

Champandard(2008b) Alex J. Champandard. Behavior tree design patterns: Goal-orientation. audio talk, 2008b. Citado na pág. 16, 17

Chong-U Lim e Colton(2010) Robin Baumgarten Chong-U Lim e Simon Colton. Evolving Behaviour Trees for the Commercial Game DEFCON. *Lecture Notes in Computer Science*, 6024/2010. Citado na pág. 1

David M. Bourg(2004) Gleen Seeman David M. Bourg. *Ai For Game Developers*. O'Reilly. ISBN 0-596-00555-5. Citado na pág. v, 1, 8, 9, 10

dos Santos(2004) Gilliard Lopes dos Santos. Máquinas de Estados Hierárquicas em Jogos Eletrônicos. Dissertação de Mestrado, PUC-Rio. Citado na pág. 7

Dyckhoff(2007) Max Dyckhoff. Evolving halo 3's behavior tree ai. in proceedings of the game developers conference., 2007. Citado na pág.

Goldstone(2009) Will Goldstone. *Unity Game Development Essentials*. Packt Publishing. ISBN 978-1847198181. Citado na pág. v, 5, 6, 7

Gregory(2009) Jason Gregory. *Game Engine Architecture*. A K Peters. ISBN 978-1568814131. Citado na pág. 3, 4

Harel(1987) David Harel. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming 8. Weizmann Institute of Science*. Citado na pág. 1, 9

Isla(2005) D. Isla. Managing complexity in the halo 2 ai system. in proceedings of the game developers conference, 2005. Citado na pág.

Millington(2006) Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann. ISBN 978-0124977822. Citado na pág. 10, 11, 12, 13, 15, 17, 21

Wilson(2010) Kyle Wilson. Game object structure: Inheritance vs. aggregation, 2010. URL http://www.gamearchitect.net/Articles/GameObjects1.html. Citado na pág. 3, 4, 5

Anexos

.1 Implementação do sistema base

.1.1 IBaseTask

```
1 //Interface that all nodes should implement
2 using System;
3   public interface IBaseTask
4   {
5     event FinishedRunning FinishedEvent;
6     CompositeType CompType{get;}
7     RunStatus Result{get;set;}
8     string TaskName{get;}
9   }
```

.1.2 CompositeTask

```
1 //Core class used to represent a CompositeTask
2 using UnityEngine;
3 using System.Collections;
4 using System.Collections.Generic;
5   public enum CompositeType{
6     Selector,
7     Sequence,
8     Parallel,
9     Action,
10    Condition,
11    Decorator
12  }
13  public delegate void FinishedRunning (RunStatus result);
14
15  public class CompositeTask : IBaseTask {
16
17  private RunStatus _result;
18  private CompositeType _compType;
19  private string _taskName;
20  public RunStatus Result {get {return _result;} set{_result = value; if(
        _result != RunStatus.Running && FinishedEvent != null){Debug.Log("
      COMPOSITE CHANGED"); FinishedEvent(_result); }}}
21  public CompositeType CompType { get {return _compType;} }
22  public string TaskName { get {return _taskName;} }
23  public List<IBaseTask> Children = new List<IBaseTask>();
24  public event FinishedRunning FinishedEvent;
25
26  public CompositeTask(CompositeType type, List<IBaseTask> children){
27
```

```
28      _compType = type;
29      _taskName = this.ToString();
30      Children = children;
31    }
32
33    public CompositeTask(CompositeType type){
34      _compType = type;
35      _taskName = this.ToString();
36    }
37
38    }
```

.1.3 RunStatus

```
1 //Node execution result
2 public enum RunStatus{
3    Running,
4    Failed,
5    Succeded
6 }
```

.1.4 IDecorator

```
1 //When a new decorator is created, it should implement this interface
2 using System;
3 using UnityEngine;
4    public interface IDecorator
5    {
6      DecoratorType DecType{get;}
7      void Run(MonoBehaviour treeObj);
8    }
```

.1.5 DecoratorType

```
1 //When a new decorator is created, a new enum should be added here
2    public enum DecoratorType{
3      Loop
4    }
```

.1.6 Scheduler

```
1 //This is the class that actually runs all the trees in the game, it
      receives an event with a reference to the
2 //Tree and the MonoBehaviour that called it.
3 using UnityEngine;
4 using System.Collections;
5
6 public class Scheduler : MonoBehaviour {
7
8    public delegate void RunTreeHandler (CompositeTask root,MonoBehaviour
       treeObj);
9
```

```
10  //event is received through this method
11  public void RunTree(CompositeTask root, MonoBehaviour treeObj){
12    StartCoroutine(StartTree(root,null,treeObj));
13  }
14
15
16  //this method starts the execution of a tree. it's called as a Coroutine
      .
17  public IEnumerator StartTree(CompositeTask compTask, FinishedRunning
      result,MonoBehaviour treeObj){
18    //set the composite task as running
19    compTask.Result = RunStatus.Running;
20
21    //if it's a parallel, run all childs at once
22    if(compTask.CompType == CompositeType.Parallel){
23      StartCoroutine(RunChildsAtOnce(compTask,treeObj));
24      yield break;
25    }
26
27    //not a parallel, run each child
28    foreach(IBaseTask t in compTask.Children){
29      Debug.Log("Running: "+compTask.CompType+" Child: "+t.CompType + "
          Count: "+ compTask.Children.Count);
30
31      //child is decorator
32      if(t.CompType == CompositeType.Decorator){
33        Debug.Log("Is Decorator!");
34        IDecorator decorator = (IDecorator)t;
35        RunDecorator(decorator,treeObj);
36        while(t.Result == RunStatus.Running){
37          yield return new WaitForEndOfFrame();
38        }
39        if(!CheckCompositeTaskRules(t.Result,compTask))
40          break;
41      }
42      else
43
44      //child is composite task
45      if(t.GetType() == typeof(CompositeTask)){
46
47        if(t.CompType == CompositeType.Parallel){
48          Debug.Log("PARALLEL FROM START TREE");
49          CompositeTask cpTask = (CompositeTask)t;
50          StartCoroutine(RunChildsAtOnce(cpTask,treeObj));
51          while(cpTask.Result == RunStatus.Running){
52            yield return new WaitForEndOfFrame();
53          }
54          if(!CheckCompositeTaskRules(t.Result,compTask)){
55            Debug.Log("PARALLEL BREAK");
56            break;
57          }
58        }
59        else{
60          CompositeTask cpTask = (CompositeTask)t;
61          Debug.Log("Composite Task found! "+cpTask.CompType);
62          StartCoroutine(StartTree(cpTask,null,treeObj));
63
64          while(cpTask.Result == RunStatus.Running){
65            yield return new WaitForEndOfFrame();
```

```
66            }
67
68            if(!CheckCompositeTaskRules(cpTask.Result,compTask)){
69              Debug.Log("Break!");
70              break;
71            }
72          }
73        }
74
75        //child is a regular task
76        else{
77          Debug.Log("Task Found! "+t.GetType());
78          IBaseTask component = treeObj.gameObject.AddComponent(t.TaskName)
                as IBaseTask;
79
80          while(component.Result == RunStatus.Running){
81            yield return new WaitForEndOfFrame();
82          }
83          if(!CheckCompositeTaskRules(component.Result,compTask))
84            break;
85        }
86
87      }
88
89      //finished composite task execution, set it's execution status
90      if(compTask.Result == RunStatus.Running){
91        if(compTask.CompType == CompositeType.Selector){
92          compTask.Result = RunStatus.Failed;
93          if(result != null)
94            result(RunStatus.Failed);
95        }
96        else if(compTask.CompType == CompositeType.Sequence){
97          compTask.Result = RunStatus.Succeded;
98          if(result != null)
99            result(RunStatus.Succeded);
100       }
101     }
102     else{
103       if(compTask.CompType == CompositeType.Selector){
104         if(result != null)
105           result(RunStatus.Succeded);
106
107       }
108       else{
109         if(result != null)
110           result(RunStatus.Failed);
111       }
112     }
113     yield break;
114   }
115
116
117   private bool CheckCompositeTaskRules(RunStatus childStatus,
          CompositeTask parentTask){
118     if(childStatus == RunStatus.Succeded && parentTask.CompType ==
          CompositeType.Selector){
119       parentTask.Result = RunStatus.Succeded;
120       return false;
121     }
```

```
122     if(childStatus == RunStatus.Failed && parentTask.CompType ==
            CompositeType.Sequence){
123       parentTask.Result = RunStatus.Failed;
124       return false;
125     }
126     return true;
127   }
128
129
130   //called when a parallel must be executes. Runs all it's hilds at once
131   private IEnumerator RunChildsAtOnce(CompositeTask parallelTask,
          MonoBehaviour treeObj){
132     int parallelRunningTasks = parallelTask.Children.Count;
133
134     Debug.Log("Running parallel! "+parallelRunningTasks);
135
136     RunStatus parallelResult = RunStatus.Succeded;
137
138     foreach(IBaseTask child in parallelTask.Children){
139       //parallel child is a regular task
140       if(child.CompType == CompositeType.Action || child.CompType ==
              CompositeType.Condition){
141         Debug.Log("Running parallel task!");
142         IBaseTask component = treeObj.gameObject.AddComponent(child.
                TaskName) as IBaseTask;
143         component.FinishedEvent += delegate(RunStatus result) {
144           if(result == RunStatus.Failed)
145             parallelResult = RunStatus.Failed;
146           parallelRunningTasks--;
147         };
148       }
149       //parallel child is a decorator
150       else if (child.CompType == CompositeType.Decorator){
151         IDecorator decorator = (IDecorator)child;
152         RunDecorator(decorator,treeObj);
153         child.FinishedEvent += delegate(RunStatus result) {
154          if(result == RunStatus.Failed)
155             parallelResult = RunStatus.Failed;
156          parallelRunningTasks--;
157         };
158       }
159       //parallel child is a composite task
160       else{
161         StartCoroutine(StartTree((CompositeTask)child,null,treeObj));
162         child.FinishedEvent += delegate(RunStatus result) {
163          if(result == RunStatus.Failed)
164           parallelResult = RunStatus.Failed;
165          parallelRunningTasks--;
166         };
167       }
168     }
169     while(parallelRunningTasks != 0){
170         yield return new WaitForEndOfFrame();
171     }
172
173     //returns parallel result
174     parallelTask.Result = parallelResult;
175
176   }
```

```
177
178   //Called when a decorator is found inside the tree
179   void RunDecorator(IDecorator decorator, MonoBehaviour treeObj){
180     decorator.Run(treeObj);
181   }
182
183
184
185 }
```

.1.7 Task Example - Rotate

```
1 //Implemented Task used as an example. It rotates the game object for 10
      seconds and then stops
2 using UnityEngine;
3 using System.Collections;
4
5 public class Rotate : MonoBehaviour, IBaseTask {
6   private CompositeType _compType;
7   private string _taskName;
8   public CompositeType CompType { get {return _compType;} }
9   public string TaskName { get {return _taskName;} }
10  public RunStatus Result {get {return _result;} set{_result = value; if(
        _result != RunStatus.Running && FinishedEvent != null) FinishedEvent(
        _result);}}
11  public bool SetupFinished;
12  private RunStatus _result;
13  public event FinishedRunning FinishedEvent;
14  private SomethingHappened something;
15  public Rotate(CompositeType type){
16    _compType = type;
17    _taskName = this.ToString();
18  }
19
20  public IEnumerator Start(){
21    //find the external variable that it's listening to.
22    something = GameObject.Find("Something").GetComponent(typeof(
        SomethingHappened)) as SomethingHappened;
23    //if the variable is false, the task failed.
24    if(!something.ShouldRotate){
25      Result = RunStatus.Failed;
26      Stop();
27    }
28    //variable is true! wait for 10 seconds while it's rotating
29    yield return new WaitForSeconds(10);
30    Result = RunStatus.Succeded;
31    Stop();
32  }
33
34  void Update(){
35    //rotates the object
36    if(something.ShouldRotate)
37      transform.Rotate(Vector3.down * Time.deltaTime * 20);
38    else
39    {
40      Result = RunStatus.Failed;
41      Stop();
42    }
```

```
43   }
44
45   public void Stop(){
46     Destroy(this);
47   }
48
49
50 }
```

.1.8 Decorator Example - Loop

```
1 //This is a loop decorator that was implemented as an example.
2 using UnityEngine;
3 using System.Collections;
4 using System.Collections.Generic;
5   public class Loop : IBaseTask, IDecorator {
6
7   public DecoratorType DecType {get {return DecoratorType.Loop;} }
8   private RunStatus _result;
9   private CompositeType _compType = CompositeType.Decorator;
10  private string _taskName;
11  public RunStatus Result {get {return _result;} set{_result = value;}}
12  public CompositeType CompType { get {return _compType;} }
13  public string TaskName { get {return _taskName;} }
14  private int _repeatTimes;
15  public List<IBaseTask> Children = new List<IBaseTask>();
16  public event FinishedRunning FinishedEvent;
17  private bool _ranOnce;
18  private Scheduler _scheduler;
19  private MonoBehaviour _gameObjRef;
20
21  public Loop(Scheduler scheduler,object[] repeatTimes){
22    _repeatTimes = (int)repeatTimes[0];
23    _scheduler = scheduler;
24    _taskName = this.ToString();
25  }
26
27  //This is called to actuallt run the decorator
28  public void Run(MonoBehaviour treeObj){
29      Debug.Log("Running loop "+_repeatTimes);
30      _gameObjRef = treeObj;
31      if(Children.Count == 0)
32        Debug.LogError("Decorator Loop doesn't have any childs!");
33
34      if(Children[0].CompType == CompositeType.Sequence || Children[0].
            CompType == CompositeType.Selector || Children[0].CompType ==
            CompositeType.Parallel){
35          _scheduler.StartCoroutine(_scheduler.StartTree((CompositeTask)
                Children[0],null,treeObj));
36      }
37
38      //preventing more events to be added
39      if(!_ranOnce)
40      Children[0].FinishedEvent += delegate(RunStatus result) {
41      Debug.Log("Tree Count: "+Children.Count);
42                                  if(0 < Children.Count-1){
43                                    RunSubtree(1,treeObj);
44                                  }else{
```

```
45                                          OnFinished(RunStatus.Succeded);
46                                        }
47                                      };
48
49    }
50
51    //called internally by Run()s
52    private void RunSubtree(int index, MonoBehaviour treeObj){
53
54      if(Children[index].CompType == CompositeType.Sequence || Children[
            index].CompType == CompositeType.Selector || Children[index].
            CompType == CompositeType.Parallel){
55        _scheduler.StartCoroutine(_scheduler.StartTree((CompositeTask)
              Children[index],null,treeObj));
56      }
57      //preventing more events to be added
58      if(!_ranOnce)
59      Children[index].FinishedEvent += delegate(RunStatus result) {
60                                         Debug.Log("FINISHED EVENT CALLBACK");
61                                         if(index < Children.Count-1){
62                                           RunSubtree(index+1,treeObj);
63                                         }else{
64                                           OnFinished(RunStatus.Succeded);
65                                         }
66                                       };
67    }
68
69    //called when an execution is finished. Checkes if it should run one
          more time.
70    void OnFinished(RunStatus result){
71      Debug.Log("Call OnFinished");
72        _repeatTimes--;
73        _ranOnce = true;
74        if(_repeatTimes > 0)
75          Run(_gameObjRef);
76        else{
77          Debug.Log("Decorator Succeded");
78          _result = RunStatus.Succeded;
79          if(FinishedEvent != null)
80            this.FinishedEvent(RunStatus.Succeded);
81        }
82      }
83
84
85    }
```

.2 Classes da Interface Gráfica

.2.1 EditorTask

```
1 //EDITOR TASK, CONTAINS THE DATA REQUIRED TO DRAW A TASK IN THE EDITOR
2 using UnityEngine;
3 using UnityEditor;
4 using System;
5 using System.Collections;
6 using System.Collections.Generic;
7
```

```
8 public struct Variable{
9   object value;
10  string type;
11 }
12 public class EditorTask {
13
14  public int depth = 0;
15
16  public Rect editorPosition;
17
18  public enum TaskType{
19    Selector,
20    Sequence,
21    Parallel,
22    Action,
23    Condition,
24    Decorator
25  }
26
27  public EditorTask parent;
28
29  public Dictionary<string,Variable> variables;
30
31  public List<EditorTask> children;
32
33  public Guid ID;
34
35  public TaskType taskType;
36
37  public EditorTask(TaskType type, EditorTask parentNode){
38    ID = System.Guid.NewGuid();
39    parent = parentNode;
40    taskType = type;
41    children = new List<EditorTask>();
42  }
43
44  public EditorTask(TaskType type){
45    taskType = type;
46    ID = System.Guid.NewGuid();
47    children = new List<EditorTask>();
48  }
49
50  public string selectedOption;
51  public int selectedOptionIndex = 0;
52 }
```

.2.2 BTreeWindow

```
1 //EDITOR MAIN WINDOW
2
3 using UnityEngine;
4 using UnityEditor;
5 using System.IO;
6 using System.Collections.Generic;
7 using System;
8 using System.Collections;
9 public class BTreeWindow : EditorWindow {
10
```

```
11  private List<string> _taskNames = new List<string>();
12  private List<string> _decoratorNames = new List<string>();
13  private EditorTask _nodes;
14  private EditorTask _selected;
15  private EditorTask _root;
16    private Vector2 _scrollPosition;
17  private bool _dataFilled;
18  private int _selectedTaskItem = 0;
19  private int _selectedDecoratorItem = 0;
20  private string _treeName = "";
21
22  //Create menu
23    [MenuItem ("Window/Behaviour Tree Editor")]
24     static void Init() {
25        // Get existing open window or if none, make a new one:
26        BTreeWindow window = (BTreeWindow)EditorWindow.GetWindow(typeof (
              BTreeWindow));
27    window.minSize = new Vector2(1100,600);
28    window.maxSize = new Vector2(1100,600);
29    }
30
31  //Main GUI method, is calles every frame to draw the window
32    void OnGUI() {
33    if(GUI.changed){
34      EditorUtility.SetDirty(this);
35    }
36
37    //Draw Tree
38    if(!_dataFilled){
39      GetAllTasks();
40      GetAllDecorators();
41      _dataFilled = true;
42    }
43        _scrollPosition = GUI.BeginScrollView(new Rect(0,0,900,540),
              _scrollPosition, new Rect(0, 0, 1600, 800));
44      if(_root != null){
45        DrawRoot();
46        DrawTree(_root.children);
47      }
48      GUI.EndScrollView();
49
50    //Draw Inspector
51    GUI.BeginGroup(new Rect(900,0,200,600));
52      GUILayout.BeginVertical();
53
54        GUILayout.BeginHorizontal();
55          GUILayout.Space(75);
56          GUILayout.Label("Inspector");
57        GUILayout.EndHorizontal();
58
59        GUILayout.BeginHorizontal();
60          GUILayout.BeginVertical();
61            GUILayout.Space(20);
62          GUILayout.EndVertical();
63        GUILayout.EndHorizontal();
64
65        GUILayout.BeginHorizontal();
66          if(GUILayout.Button("Open...",GUILayout.MaxWidth(180))){
```

```
67              string xmlPath = EditorUtility.OpenFilePanel("Select a
                    Behavior Tree (.Xml)", "/Editor/EditorTrees/", "xml");
68              _nodes = TreeGenerator.DeserializeTree(xmlPath);
69              _root = _nodes;
70            }
71
72          GUILayout.EndHorizontal();
73
74          GUILayout.BeginHorizontal();
75            _treeName = EditorGUILayout.TextField("Tree Name:", _treeName,
                  GUILayout.MaxWidth(150));
76          GUILayout.EndHorizontal();
77
78          GUILayout.BeginHorizontal();
79            if(GUILayout.Button("Generate Tree From Editor",GUILayout.
                  MaxWidth(180))){
80              if(_treeName == "")
81                EditorUtility.DisplayDialog("Error","Error: Tree Name is
                      empty!","Ok");
82              else
83                TreeGenerator.GenerateTree(_root,_treeName);
84            }
85
86          GUILayout.EndHorizontal();
87
88          GUILayout.BeginHorizontal(GUILayout.MaxWidth(180));
89            if(_selected != null && (_selected.taskType == EditorTask.
                  TaskType.Action || _selected.taskType == EditorTask.TaskType.
                  Condition)){
90              _selectedTaskItem = EditorGUILayout.Popup("Task Type:",
                      _selected.selectedOptionIndex,_taskNames.ToArray());
91              if(_selectedTaskItem != _selected.selectedOptionIndex){
92                _selected.selectedOptionIndex = _selectedTaskItem;
93                _selected.selectedOption = _taskNames[_selectedTaskItem];
94              }
95            }
96            if(_selected != null && (_selected.taskType == EditorTask.
                  TaskType.Decorator)){
97              _selectedDecoratorItem = EditorGUILayout.Popup("Decorator Type
                      :",_selected.selectedOptionIndex,_decoratorNames.ToArray()
                      );
98              if(_selectedDecoratorItem != _selected.selectedOptionIndex){
99                _selected.selectedOptionIndex = _selectedDecoratorItem;
100               _selected.selectedOption = _decoratorNames[
                      _selectedDecoratorItem];
101             }
102           }
103         GUILayout.EndHorizontal();
104
105       GUILayout.EndVertical();
106     GUI.EndGroup();
107
108     //Draw Toolbox
109     GUILayout.BeginVertical(GUILayout.ExpandHeight(true),GUILayout.
          ExpandWidth(true));
110     GUILayout.EndVertical();
111     GUILayout.BeginHorizontal(GUILayout.MinHeight(40));
112           GUILayout.FlexibleSpace();
113       if(GUILayout.Button("Selector")){
```

```
114         InsertTask(EditorTask.TaskType.Selector);
115       }
116       if(GUILayout.Button("Sequence")){
117         InsertTask(EditorTask.TaskType.Sequence);
118       }
119       if(GUILayout.Button("Condition")){
120         InsertTask(EditorTask.TaskType.Condition);
121       }
122       if(GUILayout.Button("Action")){
123         InsertTask(EditorTask.TaskType.Action);
124       }
125       if(GUILayout.Button("Decorator")){
126         InsertTask(EditorTask.TaskType.Decorator);
127       }
128       if(GUILayout.Button("Parallel")){
129         InsertTask(EditorTask.TaskType.Parallel);
130       }
131       GUILayout.Space(100);
132       if(GUILayout.Button("Remove")){
133
134       }
135       GUILayout.FlexibleSpace();
136     GUILayout.EndHorizontal();
137     }
138
139   //Gets all Tasks in the default task folder
140    void GetAllTasks(){
141     DirectoryInfo di = new DirectoryInfo(Application.dataPath+"/
          BehaviorTree/Tasks");
142
143     FileInfo[] rgFiles = di.GetFiles("*.cs");
144     foreach(FileInfo fi in rgFiles)
145     {
146         string name = Path.GetFileNameWithoutExtension(fi.FullName);
147       _taskNames.Add(name);
148     }
149   }
150
151   //Gets all Decorators in the default decorator folder
152   void GetAllDecorators(){
153     DirectoryInfo di = new DirectoryInfo(Application.dataPath+"/
          BehaviorTree/Decorators");
154
155     FileInfo[] rgFiles = di.GetFiles("*.cs");
156     foreach(FileInfo fi in rgFiles)
157     {
158         string name = Path.GetFileNameWithoutExtension(fi.FullName);
159       _decoratorNames.Add(name);
160     }
161   }
162
163   //Called when a new node is added to the tree
164   void InsertTask(EditorTask.TaskType type){
165     if(_selected == null && _root == null && (type == EditorTask.TaskType.
          Selector || type == EditorTask.TaskType.Sequence || type ==
          EditorTask.TaskType.Parallel))
166     {
167       EditorTask taskToBeAdded = new EditorTask(type);
168       taskToBeAdded.depth = 1;
```

```
169
170         _nodes = new EditorTask(type);
171         _root = taskToBeAdded;
172       }
173
174       if(_selected != null && _selected.taskType != EditorTask.TaskType.
              Action && _selected.taskType != EditorTask.TaskType.Condition){
175         EditorTask taskToBeAdded = new EditorTask(type,_selected);
176         taskToBeAdded.depth = taskToBeAdded.parent.depth+2;
177         if(taskToBeAdded.taskType == EditorTask.TaskType.Decorator)
178           taskToBeAdded.selectedOption = _decoratorNames[
                  _selectedDecoratorItem];
179         if(taskToBeAdded.taskType == EditorTask.TaskType.Action ||
              taskToBeAdded.taskType == EditorTask.TaskType.Condition)
180           taskToBeAdded.selectedOption = _taskNames[_selectedTaskItem];
181         _nodes.children.Add(taskToBeAdded);
182         _selected.children.Add(taskToBeAdded);
183       }
184     }
185
186     //Draw the tree root in the default location
187     void DrawRoot(){
188         _root.editorPosition = new Rect(450,40,40,40);
189         DrawNode(_root,new Rect(450,40,40,40));
190     }
191     //Draw the rest of the tree
192     void DrawTree(List<EditorTask> parentTask){
193       DrawNodes(parentTask);
194     }
195
196     //Draws all tree nodes by calculating the position where the node should
          be inserted in the tree.
197     void DrawNodes(List<EditorTask> tasks){
198       if(tasks.Count == 0){
199         return;
200       }
201       List<EditorTask> childTasks = new List<EditorTask>();
202         int i = 0;
203         int posX;
204         foreach(EditorTask task in tasks){
205           if(tasks.Count%2 == 0){
206             if(tasks.Count/2 == i)
207               i++;
208             posX = (int) task.parent.editorPosition.x + (-(tasks.Count * 40
                    / 2) + i * 40);
209           }else{
210             posX = (int) task.parent.editorPosition.x + (-((tasks.Count-1) *
                    40 / 2) + i * 40);
211
212           }
213           DrawNode(task,new Rect(posX,40*task.depth+10,40,40));
214           i++;
215           if(task.children != null || task.children.Count > 0){
216             foreach(EditorTask childTask in task.children){
217               childTasks.Add(childTask);
218             }
219           }
220         }
221       if(childTasks.Count > 0)
```

```
222         DrawNodes(childTasks);
223     }
224
225     //called internally by DrawNodes
226     void DrawNode(EditorTask node,Rect pos){
227       node.editorPosition = pos;
228       if(node != _root)
229         Handles.DrawLine(new Vector3(node.parent.editorPosition.x+20 ,node.
              parent.editorPosition.y+40),new Vector3(node.editorPosition.x+20,
              node.editorPosition.y));
230       Handles.color = Color.black;
231       if(node == _selected){
232         GUISkin skin = GUI.skin;
233         GUIStyle style = new GUIStyle(skin.button);
234         style.normal.textColor = Color.blue;
235         if(GUI.Button(pos, node.taskType.ToString().Substring(0,3),style)){
236           _selected = node;
237         }
238       }else{
239         if(GUI.Button(pos, node.taskType.ToString().Substring(0,3))){
240           _selected = node;
241         }
242       }
243     }
244
245
246
247
248 }
```

.2.3 TreeGenerator

```
1 //THIS CLASS IS USED TO GENERATE THE TREE CLASS AND SERIALIZE/DESERIALIZE
      THE TREE XML
2
3 using UnityEngine;
4 using UnityEditor;
5 using System.IO;
6 using System.Collections.Generic;
7 using System;
8 using System.Collections;
9 using System.Xml;
10
11 public class TreeGenerator {
12   static int taskNameAux = 1;
13   static int compositeNameAux = 1;
14
15   //Static method called to Generate the Tree Class
16   public static void GenerateTree(EditorTask root, string treeName){
17       GenerateTreeXml(root,treeName);
18       TextWriter tw = new StreamWriter(Application.dataPath+"/BehaviorTree
             /Trees/"+treeName+".cs");
19       GenerateDocumentHead(tw);
20       tw.WriteLine("public class "+treeName+" : MonoBehaviour{");
21         tw.WriteLine("void Start(){");
22           tw.WriteLine("Scheduler scheduler = GameObject.
                 FindGameObjectWithTag(\"AI Scheduler\").GetComponent(typeof(
                 Scheduler)) as Scheduler;");
```

```
23            tw.WriteLine("CompositeTask cpTask"+compositeNameAux+" = new
                  CompositeTask(CompositeType."+root.taskType.ToString()+");");
24            compositeNameAux++;
25            GenerateTasks(tw,root,compositeNameAux-1);
26            tw.WriteLine("scheduler.RunTree(cpTask1,this);");
27            tw.WriteLine("");
28            tw.WriteLine("");
29          tw.WriteLine("}");
30        tw.WriteLine("}");
31              // close the stream
32              tw.Close();
33    }
34
35    //Used internally by the GenerateTree method. Recursive
36    private static void GenerateTasks(TextWriter tw,EditorTask tasks,int
          parentName){
37      foreach (EditorTask task in tasks.children) {
38        if(task.taskType == EditorTask.TaskType.Sequence || task.taskType ==
              EditorTask.TaskType.Selector || task.taskType == EditorTask.
            TaskType.Parallel || task.taskType == EditorTask.TaskType.
            Decorator){
39          if(task.taskType == EditorTask.TaskType.Decorator)
40            tw.WriteLine(task.selectedOption+" cpTask"+compositeNameAux+" =
                  new "+task.selectedOption+"(scheduler,new object[] {5});");
41          else
42            tw.WriteLine("CompositeTask cpTask"+compositeNameAux+" = new
                  CompositeTask(CompositeType."+task.taskType.ToString()+");");
43          tw.WriteLine("cpTask"+parentName+".Children.Add(cpTask"+
                compositeNameAux+");");
44          compositeNameAux++;
45          GenerateTasks(tw,task,compositeNameAux-1);
46        }
47        else if(task.taskType == EditorTask.TaskType.Action || task.taskType
              == EditorTask.TaskType.Condition){
48          tw.WriteLine(task.selectedOption+" task"+taskNameAux+" = new "+
                task.selectedOption+"(CompositeType."+task.taskType.ToString()+
                ");");
49          tw.WriteLine("cpTask"+parentName+".Children.Add(task"+taskNameAux+
                ");");
50          taskNameAux++;
51        }
52      }
53    }
54
55    //Used internally by the GenerateTree method.
56    private static void GenerateDocumentHead(TextWriter tw){
57      tw.WriteLine("//THIS IS AN AUTO GENERATED FILE, DO NOT MODIFY");
58      tw.WriteLine("using UnityEngine;");
59      tw.WriteLine("using System.Collections.Generic;");
60      tw.WriteLine("using System;");
61      tw.WriteLine("using System.Collections;");
62    }
63
64
65    //Method that generates the Tree XML file.
66    private static void GenerateTreeXml(EditorTask root, string treeName){
67      XmlDocument xmlDoc = new XmlDocument();
68
69
```

```
70          // Write down the XML declaration
71          XmlDeclaration xmlDeclaration = xmlDoc.CreateXmlDeclaration("1.0",
                "utf-8",null);
72
73          // Create the root element
74          XmlElement rootNode  = xmlDoc.CreateElement("Tree");
75          xmlDoc.InsertBefore(xmlDeclaration, xmlDoc.DocumentElement);
76          xmlDoc.AppendChild(rootNode);
77
78          // Create a new element and add it to the root node
79          XmlElement parentNode  = xmlDoc.CreateElement("Composite");
80
81          // Set attribute values.
82          parentNode.SetAttribute("Type",root.taskType.ToString());
83      parentNode.SetAttribute("Rect",root.editorPosition.x+","+root.
            editorPosition.y+","+root.editorPosition.width+","+root.
            editorPosition.height);
84          parentNode.SetAttribute("Depth",root.depth.ToString());
85      xmlDoc.DocumentElement.PrependChild(parentNode);
86
87      // Serialize the children recursively
88      SerializeNodes(xmlDoc,root,parentNode);
89
90          // Save to the XML file
91          xmlDoc.Save(Application.dataPath+"/Editor/EditorTrees/"+treeName+"
                .xml");
92
93  }
94
95  //Called internally by GenerateTreeXML
96  static void SerializeNodes(XmlDocument xmlDoc, EditorTask parentTask,
        XmlNode parentNode){
97      foreach (EditorTask child in parentTask.children) {
98
99        //if composite task
100       if(child.taskType == EditorTask.TaskType.Parallel || child.taskType
              == EditorTask.TaskType.Sequence || child.taskType == EditorTask.
              TaskType.Selector)
101       {
102         XmlElement childNode = xmlDoc.CreateElement("Composite");
103         childNode.SetAttribute("Rect",child.editorPosition.x+","+child.
                editorPosition.y+","+child.editorPosition.width+","+child.
                editorPosition.height);
104         childNode.SetAttribute("Type",child.taskType.ToString());
105         childNode.SetAttribute("Depth",child.depth.ToString());
106         parentNode.AppendChild(childNode);
107         SerializeNodes(xmlDoc,child,childNode);
108       }
109
110       //if decorator
111       if(child.taskType == EditorTask.TaskType.Decorator)
112       {
113         XmlElement childNode = xmlDoc.CreateElement("Decorator");
114         childNode.SetAttribute("Rect",child.editorPosition.x+","+child.
                editorPosition.y+","+child.editorPosition.width+","+child.
                editorPosition.height);
115         childNode.SetAttribute("Type",child.taskType.ToString());
116         childNode.SetAttribute("SelectedOptionIndex",child.
                selectedOptionIndex.ToString());
```

```
117         childNode.SetAttribute("SelectedOption",child.selectedOption.
                ToString());
118         childNode.SetAttribute("Depth",child.depth.ToString());
119         parentNode.AppendChild(childNode);
120         SerializeNodes(xmlDoc,child,childNode);
121       }
122
123       //if regular task
124       if(child.taskType == EditorTask.TaskType.Action || child.taskType ==
              EditorTask.TaskType.Condition)
125       {
126         XmlElement childNode = xmlDoc.CreateElement("Task");
127         childNode.SetAttribute("Rect",child.editorPosition.x+","+child.
                editorPosition.y+","+child.editorPosition.width+","+child.
                editorPosition.height);
128         childNode.SetAttribute("Type",child.taskType.ToString());
129         childNode.SetAttribute("SelectedOptionIndex",child.
                selectedOptionIndex.ToString());
130         childNode.SetAttribute("SelectedOption",child.selectedOption.
                ToString());
131         childNode.SetAttribute("Depth",child.depth.ToString());
132         parentNode.AppendChild(childNode);
133       }
134     }
135   }
136
137   //method called to deserialize a Tree Xml file.
138   public static EditorTask DeserializeTree(string treeName){
139     //open document
140     XmlDocument xmlDoc = new XmlDocument();
141     xmlDoc.Load(treeName);
142
143     //get root
144     XmlNodeList root = xmlDoc.ChildNodes;
145     XmlNodeList compositeRoot = xmlDoc.GetElementsByTagName("Composite");
146
147     //set root values
148     EditorTask.TaskType parentType = (EditorTask.TaskType)Enum.Parse(
          typeof(EditorTask.TaskType),compositeRoot[0].Attributes["Type"].
          Value);
149     EditorTask parentEditorTask = new EditorTask(parentType);
150
151     string[] parentPosition = compositeRoot[0].Attributes["Rect"].Value.
          Split(',');
152     parentEditorTask.editorPosition = new Rect(float.Parse(parentPosition
          [0]),float.Parse(parentPosition[1]),float.Parse(parentPosition[2]),
          float.Parse(parentPosition[3]));
153
154     int parentDepth = int.Parse(compositeRoot[0].Attributes["Depth"].Value
          );
155     parentEditorTask.depth = parentDepth;
156
157     //deserialize rest of the tree recursively
158     EditorTask deserializedNode = DeserializeNode(compositeRoot[0],
          parentEditorTask);
159     return deserializedNode;
160
161
162   }
```

```
163
164   //called internally by DeserializeTree
165     static EditorTask DeserializeNode(XmlNode parentNode, EditorTask
          parentTask){
166       foreach (XmlNode childNode in parentNode.ChildNodes) {
167
168         //set parent values and create EditorTask
169         EditorTask.TaskType taskType = (EditorTask.TaskType)Enum.Parse(
              typeof(EditorTask.TaskType),childNode.Attributes["Type"].Value);
170         EditorTask editorTask = new EditorTask(taskType);
171
172         editorTask.parent = parentTask;
173
174         string[] position = childNode.Attributes["Rect"].Value.Split(',');
175         editorTask.editorPosition = new Rect(float.Parse(position[0]),float.
              Parse(position[1]),float.Parse(position[2]),float.Parse(position
              [3]));
176
177         int parentDepth = int.Parse(childNode.Attributes["Depth"].Value);
178         editorTask.depth = parentDepth;
179
180
181         //if composite task
182         if(editorTask.taskType == EditorTask.TaskType.Parallel || editorTask
              .taskType == EditorTask.TaskType.Selector || editorTask.taskType
              == EditorTask.TaskType.Sequence){
183           parentTask.children.Add(DeserializeNode(childNode,editorTask));
184         }
185
186         //if decorator
187         else if(editorTask.taskType == EditorTask.TaskType.Decorator){
188           string taskOptionName = childNode.Attributes["SelectedOption"].
                Value;
189           int taskOptionIndex = int.Parse(childNode.Attributes["
                SelectedOptionIndex"].Value);
190           editorTask.selectedOption = taskOptionName;
191           editorTask.selectedOptionIndex = taskOptionIndex;
192           parentTask.children.Add(DeserializeNode(childNode,editorTask));
193         }
194
195         //if regular task
196         else if(editorTask.taskType == EditorTask.TaskType.Condition ||
              editorTask.taskType == EditorTask.TaskType.Action){
197           string taskOptionName = childNode.Attributes["SelectedOption"].
                Value;
198           int taskOptionIndex = int.Parse(childNode.Attributes["
                SelectedOptionIndex"].Value);
199           editorTask.selectedOption = taskOptionName;
200           editorTask.selectedOptionIndex = taskOptionIndex;
201           parentTask.children.Add(editorTask);
202         }
203       }
204       //return complete tree
205       return parentTask;
206
207   }
208 }
```