Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Bonsai: um editor de programação gráfica para o motor de jogos Unity 3D

Anderson Campos Cardoso

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Carla Denise Castanho

Coorientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenadora: Prof.ª Dr.ª Maristela Terto de Holanda

Banca examinadora composta por:

Prof.ª Dr.ª Carla Denise Castanho (Orientadora) — CIC/UnB
Prof. Dr. Rodrigo Bonifácio de Almeida — CIC/UnB
Prof. Dr. Guilherme Novaes Ramos — CIC/UnB
Prof.ª Dr.ª Fernanda Lima — CIC/UnB

CIP — Catalogação Internacional na Publicação

Cardoso, Anderson Campos.

Bonsai: um editor de programação gráfica para o motor de jogos Unity 3D / Anderson Campos Cardoso. Brasília : UnB, 2013.
127 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. jogos , 2. desenvolvimento, 3. programação gráfica, 4. *Unity 3D*

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Bonsai: um editor de programação gráfica para o motor de jogos Unity 3D

Anderson Campos Cardoso

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Carla Denise Castanho (Orientadora)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Guilherme Novaes Ramos
CIC/UnB

Prof.ª Dr.ª Fernanda Lima
CIC/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 09 de março de 2013

Agradecimentos

Em primeiro lugar, agradeço aos meus orientadores, professores Carla Denise Castanho e Rodrigo Bonifácio de Almeida, que tantas vezes mostraram seu apoio e auxiliaram na superação de desafios. Também agradeço aos professores Guilherme Novaes Ramos e Fernanda Lima pelos valiosos comentários e sugestões, e por terem aceito o convite para participarem da banca de avaliação deste trabalho.

Agradeço aos meus pais, Lisérgio e Sthela, por sempre terem acreditado em mim, e propiciado um ambiente onde eu pudesse alcançar meus objetivos. Agradeço ao meu avô materno, Gerisvander, pelo apoio durante toda esta jornada e por ter possibilitado que este momento se concretizasse.

Quero também agradecer à minha esposa Viviane, por me apoiar de inúmeras formas e estar sempre presente quando preciso. Agradeço ao meu filho Derick, pelos maravilhosos momentos de lazer e por me lembrar, todos os dias, o porquê de estar aqui.

Agradeço aos meus amigos de curso por estarem juntos durante toda a graduação, sempre ajudando-se uns aos outros, e por compartilhar momentos inesquecíveis de estudo, desespero e, claro, diversão.

Abstract

As pessoas estão cada vez mais imersas no mundo virtual dos jogos eletrônicos. Celulares, *tablets*, computadores, televisores são alguns exemplos de dispositivos que comportam esta mídia. Lojas de distribuição digital (como *Apple Store*, *Google Play*, *Steam*, etc) permitem que jogos eletrônicos possam ser facilmente comercializados em todo mundo. Além disto, existem inúmeras ferramentas gratuitas, ou de baixo custo, para desenvolvimento de *games*. Este cenário fomenta um crescente número de novas empresas, e pequenos grupos, na indústria de jogos eletrônicos. Entretanto, o desenvolvimento de jogos eletrônicos não é uma tarefa trivial, exige-se um forte conhecimento técnico na área de computação.

Neste contexto, esta monografia apresenta o *software* Bonsai, um *plugin* de programação gráfica para o motor de jogos eletrônicos *Unity 3D* que dispensa a necessidade de conhecimento de programação. O objetivo é possibilitar a rápida produção de jogos, principalmente por grupos que não possuem uma equipe de programação, de uma maneira simples e intuitiva, através da criação e determinação de comportamentos de objetos via interface gráfica. O Bonsai fornece uma abstração do código específico do jogo em forma de árvore, que permite configuração em tempo de execução, visualização dos passos de execução, criação de novas estruturas via programação e reutilização de comportamentos em outros objetos e projetos.

Palavras-chave: jogos , desenvolvimento, programação gráfica, *Unity 3D*

Abstract

People are increasingly immersed in the virtual world of video games. Cell phones, tablets, computers, televisions are examples of devices that enable this media. Digital distribution stores (like Apple Store, Google Play, Steam, etc) allow electronic games to be easily marketed worldwide. Moreover, there are numerous free, or low cost, tools for game development. This setting fosters a growing number of new companies and small groups in the game industry. However, game development is not a trivial task, it requires a strong expertise in the field of computing.

In this context, this work presents the Bonsai tool, a visual programming plugin for the Unity 3D game engine which does not require programming skills. The main goal is to enable rapid production of games, especially by groups that do not have a programming team, in a simple and intuitive way, by creating and determining object's behaviour with a graphical user interface. Bonsai provides an abstraction of the game's specific code as a tree structure, which enables run-time configuration, visualization of execution's steps, programmatically creating new structures and reuse of behaviors in others objects and projects.

Keywords: games, development, visual programming, Unity 3D

Sumário

1 Introdução **1**
- 1.1 Problema . . . 2
- 1.2 Hipótese . . . 2
- 1.3 Objetivo . . . 2
- 1.4 Metodologia . . . 3
- 1.5 Organização do Documento . . . 3

2 Trabalhos Correlatos **4**
- 2.1 Antares Universe . . . 4
- 2.2 UScript . . . 6
- 2.3 Playmaker . . . 9
- 2.4 *NoCode* e *NoCode-FSM* . . . 12

3 Revisão Teórica **15**
- 3.1 Agentes Racionais . . . 15
  - 3.1.1 Agente Reativo Simples . . . 15
  - 3.1.2 Agente Reativo Baseado em Modelo de Ambiente . . . 16
  - 3.1.3 Agentes Reativos e Jogos Eletrônicos . . . 16
- 3.2 Principais Metodologias para Tomada de Decisão . . . 17
  - 3.2.1 Máquina de Estados . . . 17
  - 3.2.2 Árvore de Comportamento . . . 18
- 3.3 *Unity 3D* . . . 23
  - 3.3.1 Objetos de Jogo e Componentes . . . 23
  - 3.3.2 Interface Principal . . . 23
  - 3.3.3 *Prefab* . . . 24
  - 3.3.4 Construção de um Jogo na *Unity 3D* . . . 24

4 Bonsai **26**
- 4.1 Visão Geral . . . 26
- 4.2 Componente Bonsai . . . 27
- 4.3 *Behaviours* ou Nós . . . 27
  - 4.3.1 *FunctionBehaviour* ou Funções . . . 28
  - 4.3.2 *BoxBehaviour* ou Variáveis . . . 28
  - 4.3.3 *PrefabTask* . . . 29
- 4.4 *Global Bonsai* . . . 29
- 4.5 Interface . . . 29
  - 4.5.1 Menu Principal . . . 29

4.5.2 Janela Bonsai . . . 30
4.5.3 Janela de Adição de Nós . . . 32
4.5.4 *Inspector View* . . . 33
4.5.5 Janela *Create Behaviour* . . . 34
4.5.6 Janela de Preferências . . . 35
4.6 Relação com Linguagem de Programação . . . 35
4.6.1 Condição . . . 36
4.6.2 Composição . . . 36
4.6.3 Atribuição . . . 38
4.6.4 Não-determinismo . . . 38
4.6.5 Iteração . . . 38
4.7 Implementação . . . 39
4.7.1 *Editor* . . . 40
4.7.2 *Runtime* . . . 40

5 Aplicações do Bonsai **44**
5.1 Controlador em Terceira Pessoa . . . 44
5.1.1 Construindo a Cena . . . 44
5.1.2 Árvore de Comportamento . . . 44
5.2 Eventos de Colisão (*Triggers*) . . . 47
5.2.1 Construindo a Cena . . . 47
5.2.2 Árvore do Comportamento dos Eventos de Colisão . . . 48
5.3 Controlador de Inimigo . . . 49
5.3.1 Construindo a Cena . . . 49
5.3.2 Árvore de Comportamento do Inimigo . . . 49

6 Conclusão **52**
6.1 Trabalhos Futuros . . . 52

Referências **54**

Lista de Figuras

2.1 Bloco lógico *Get Component* do *Antares Universe*. 5
2.2 Editor da ferramenta *Antares Universe* com um grafo para rotacionar um objeto a cada atualização de quadro. 5
2.3 Encaixes do nó *Spawn Prefab* do editor *uScript*. 7
2.4 Exemplos dos três tipos de nós (evento, ação e variável) existentes nos grafos do editor do *uScript*. 8
2.5 Interface do *uScript* com um grafo para rotacionar um objeto a cada atualização de quadro. 9
2.6 Máquina de estados no editor *Playmaker* que modela o comportamento de um interruptor. 10
2.7 Interface do *Playmaker* com uma máquina de estados para rotacionar um objeto a cada atualização de quadro. 10
2.8 Ferramenta *NoCode* com três instruções em sua janela. 13
2.9 Um componente *NCLogic* com três variáveis na *Inspector View*. 13
2.10 Ferramenta *NoCode-FSM* com três instruções em sua janela. 14

3.1 Exemplo de uma máquina de estados finitos de um agente robô que coleta recursos. 18
3.2 Exemplo de uma máquina de estados finitos de um agente robô que coleta recursos e precisa recarregar sua bateria para manter o funcionamento. . . 19
3.3 Tarefa de combinação do tipo sequência de uma árvore de comportamento. 20
3.4 Tarefa de combinação do tipo seleção de uma árvore de comportamento. . 21
3.5 Exemplo de uma árvore de comportamento. 21
3.6 Exemplo de uma árvore de comportamento com um nó decorador. 22
3.7 Instrução *if*. 22
3.8 Janela principal do motor de jogos *Unity 3D* 24

4.1 Nós *Update*, *OnTriggerEnter* e *Restart*, do tipo *FunctionBehaviour* 28
4.2 Menu principal do *plugin* Bonsai 30
4.3 Janela principal da ferramenta Bonsai 31
4.4 Janela principal da ferramenta Bonsai durante a execução. 31
4.5 Janela para adição de nós na árvore 32
4.6 A *Inspector View* é utilizada para mostrar detalhes sobre o *behaviour* selecionado 33
4.7 Janela *Create Behaviour* com a propriedade *isPlaying* da classe *Animation* selecionada. 34
4.8 Janela de Preferências do *plugin* Bonsai. 35

4.9 *Hierarcchy View* e Janela Principal modificadas pelas opções de preferências 35
4.10 Comportamento *if-else*. 36
4.11 Comportamento *if-elseif*. 37
4.12 Nó *Folder*, composição sequencial de árvores de comportamento 37
4.13 Operação de atribuição no *plugin* Bonsai 38
4.14 Exemplo de não-determinismo no *plugin* Bonsai 39
4.15 Repetição com condição no Bonsai. 39
4.16 Diagrama de classes do módulo de *Runtime* e de parte do motor *Unity 3D*. 41
4.17 Diagrama de classes contendo a hierarquia de nós da ferramenta Bonsai. . 42

5.1 Configuração inicial da cena com um plano, uma cápsula, uma câmera e uma luz direcional. 45
5.2 Árvore de comportamento de um controlador em terceira pessoa que não altera a velocidade de movimento do personagem. 46
5.3 Árvore de comportamento de um controlador em terceira pessoa. 47
5.4 Comportamento para rotacionar a câmera em direção à personagem. 47
5.5 Cena composta por um plano, a personagem uma luz, uma câmera e um alarme. 48
5.6 Comportamento para ativar e desativar o objeto alarme. 49
5.7 Árvore de comportamento com o comportamento do inimigo com parte da estrutura condicional. 50
5.8 Comportamento de seguir o objeto *Player* e mostrar uma mensagem de ataque. 51

Capítulo 1

Introdução

Em 1958 William Alfred Higinbotham desenvolveu um programa chamado *Tennis for Two* para um computador analógico [22]. *Tennis for Two* possibilitava que duas pessoas se confrontassem em um jogo de tênis em duas dimensões. O curioso programa que misturava interação com entretenimento deu início a uma das mídias mais populares atualmente, os jogos eletrônicos. Existem diversos jogos eletrônicos disponíveis para celulares, computadores e consoles, com as mais diversas finalidade, tais como, entretenimento, educação, treinamento e reabilitação [15, 16, 24].

A indústria de jogos eletrônicos sofreu um enorme crescimento nas últimas décadas, e desde 2007, seu faturamento anual ultrapassou o da indústria cinematográfica [27].

Antes da década de 90 apenas um pequeno grupo de programadores, ou um único programador, participavam de todo o processo de desenvolvimento de um jogo eletrônico. Estes programadores desenvolviam, geralmente a partir do zero, todo o código e a arte do jogo [1, 23]. Fatores como a competição no setor, a pressão por produtividade e a contínua expansão da indústria de jogos, aumentaram a demanda tanto por profissionais especializados como por novas técnicas de desenvolvimento e produção de jogos [23]. Nos anos 90, motores de jogos (ou *Engines*), como o *Voxel*, *Ultima Underworld* e *Doom Engine* [18], tornaram-se populares no contexto de desenvolvimento de jogos, pois criavam uma abstração da arquitetura de *hardware*. Isto facilitava a programação dos jogos, porque permitia que o mesmo motor fosse utilizado para a criação de diferentes jogos em diferentes plataformas.

A arquitetura da maioria dos jogos eletrônicos atuais pode ser dividida em três partes: motor de jogo (ou *engine*), lógica do jogo e arte do jogo. O motor de jogo é responsável por fornecer um ambiente onde a lógica do jogo irá funcionar. Ele possui funções básicas para renderização, áudio, matemática, mecânica, etc. A lógica do jogo, também denominado de código específico, é responsável por definir as regras e comportamentos de seus objetos, e corresponde a *scripts*, *bytecodes* ou *DLLs* (*Dynamic-Linked Library*). A arte do jogo são arquivos de texturas, mapas, modelos 3D e áudio, etc [17].

O motor de jogo é um sistema de *software* que fornece um conjunto de programas e ferramentas para simplificar e abstrair o processo de desenvolvimento de um jogo. Dentre os motores de jogos atualmente disponíveis no mercado, destacam-se: a *CryEngine 3* [6] pela qualidade gráfica, a *Unreal Development Kit* [31], também conhecida como *UDK*, por possuir um editor completo que não necessita de outras ferramentas para a confecção

de *games* e a *Unity 3D* [30] pela sua grande popularidade no desenvolvimento de jogos eletrônicos.

Apesar de um motor de jogo fornecer funções básicas para agilizar o desenvolvimento de jogos, toda a lógica do jogo ainda deve ser programada. Geralmente este código específico do jogo é desenvolvido através de uma linguagem de *script*. A tarefa de se programar um jogo eletrônico, mesmo utilizando um motor de jogo, é complexa e desafiadora. Exige-se uma base sólida de programação e envolve diferentes áreas da ciência da computação como: inteligência artificial, análise de algoritmos e etc [23].

Alguns motores de jogos facilitam o desenvolvimento da lógica do jogo permitindo que esta seja programada através de uma ferramenta visual. A *CryEngine 3* e a *Unreal Development Kit* possuem, respectivamente, os editores *Flowchart* e *Kismet* para desenvolvimento do código específico do jogo de forma visual. A *Unity 3D* não possui um editor nativo para tal facilidade, mas existem *plugins* (módulos que estendem a funcionalidade padrão do editor) proprietários, tais como, *Antares Universe* [2], *uScript* [28] e *Playmaker* [8], que estendem o editor e fornecem tal funcionalidade [2, 6, 8, 28, 31]

Uma categoria diferente de motores de jogos são os voltados para o público não programador. Estas ferramentas se distinguem das anteriores mencionadas pois desde sua concepção levam em consideração o suporte à construção da lógica do jogo de forma visual. Este é o caso das ferramentas *Game Maker* [9], *Multimedia Fusion* [29], *Kodu* [20] e *Gameka* [5]. A maioria suporta somente a construção da lógica do jogo de forma visual, ou seja, não possuem suporte a programação.

1.1 Problema

Observou-se nos estudos que dentre as metodologias para modelagem de comportamento presentes nas ferramentas de programação gráfica disponíveis para a *engine Unity 3D* (como a *Antares Universe*, *uScript*, *Playmaker*), não existe uma que mantenha uma visualização de fácil compreensão a medida que se evolui a lógica. Também observou-se, que dentre as ferramentas estudadas não existe uma que possibilite a edição total da lógica em tempo de execução, seja de fácil manutenção para lógicas complexas e tenha um fluxo de trabalho similar as demais tarefas do motor *Unity 3D*.

1.2 Hipótese

A lógica específica de um jogo eletrônico é melhor visualizada, confeccionada e evoluída, por não programadores, através de um modelo, com interface gráfica, que forneça uma abstração para linguagens de programação de alto nível. Isto é, o modelo deve abstrair detalhes específicos da implementação e da linguagem de programação utilizada, ao mesmo tempo que fornece um conjunto de simples regras que possibilite a construção de semânticas similares às presentes em tais linguagens.

1.3 Objetivo

Neste contexto, o objetivo deste trabalho é o desenvolvimento de uma abstração visual para linguagens de programação, denominado Bonsai, no âmbito de desenvolvimento de

jogos eletrônicos. Mais precisamente o Bonsai é um *plugin* para o motor de jogos *Unity 3D* que se encaixa na categoria de editor de programação gráfico. O Bonsai não requer conhecimentos sobre codificação de programas e fornece uma visualização gráfica da lógica do código específico de um jogo eletrônico. Seu fluxo de trabalho é similar às demais tarefas presentes no motor *Unity 3D*, e se assemelha a organizar uma estrutura de diretórios em sistemas operacionais modernos.

1.4 Metodologia

A metodologia proposta para trabalho é dividida em uma sequência de etapas. Inicialmente é feito um levantamento bibliográfico de trabalhos correlatos, onde é verificado o estado da arte em termos de metodologias para modelagem de comportamento de agentes reativos. Então, com base nas informações coletadas, serão identificados eventuais problemas e deficiências destes trabalhos. A partir destes resultados serão desenvolvidos mecanismos e alterações, em uma solução proposta, que permitam mitigar estas deficiências. Também será desenvolvido um *software* que permite pôr em prática esta solução.

1.5 Organização do Documento

O restante desta monografia está organizado da seguinte forma. No Capítulo 2 são descritos alguns trabalhos relacionados com o objeto desta monografia. O Capítulo 3 contém conceitos e metodologias que serviram como base para as decisões de *design* no desenvolvimento do Bonsai. As características, funcionalidades e implementação do *software* desenvolvido são descritos no Capítulo 4. No Capítulo 5, o *plugin* Bonsai é utilizado para desenvolver a lógica de alguns cenários presentes no desenvolvimento de jogos eletrônicos. Por fim, no Capítulo 6, são relatados os resultados do trabalho e expostas algumas conclusões e propostas de trabalhos futuros.

Capítulo 2

Trabalhos Correlatos

Este capítulo tem por objetivo oferecer ao leitor a oportunidade de conhecer outros projetos relacionados a este trabalho. As Seções 2.1, 2.2 e 2.3 apresentam as ferramentas *Antares Universe*, *uScript* e *PlayMaker* respectivamente. Estas ferramentas são *plugins* de programação visual para a ferramenta *Unity 3D*.

2.1 Antares Universe

Antares Universe [2] é um editor visual para programação de jogos na *Unity 3D*. Neste editor, a lógica do jogo é construída em forma de grafos onde os nós representam ações, eventos e variáveis, enquanto as arestas definem a ordem de execução dos nós.

Neste editor os nós são chamados de blocos lógicos, cada bloco pode possuir gatilhos e variáveis de entrada ou saída. Os gatilhos são utilizados para definir a ordem de execução dos blocos lógicos. Ao terminar sua execução o bloco envia um sinal para que todos os blocos lógicos conectados ao seu gatilho de saída iniciem sua execução. Os gatilhos são representados por setas que apontam para a direita. As setas à esquerda do bloco representam gatilhos de entrada e as setas à direita gatilhos de saída. Alguns blocos lógicos não possuem gatilhos de entrada, estes iniciam sua execução quando um determinado evento ocorre.

As variáveis são geralmente representadas por círculos, que são cinzas quando um valor já foi atribuído à variável e vermelhos caso contrário. As variáveis de entrada se localizam na parte de cima do bloco lógico e as variáveis de saída abaixo. Na Figura 2.1 tem-se o bloco *Get Component*, que possui um gatilho de entrada, três de saída, uma variável de entrada e três variáveis de saída, como exemplo.

Na Figura 2.2 é apresentado o editor da ferramenta *Antares Universe* com um grafo para rotacionar um objeto a cada atualização de quadro. O grafo possui três nós e duas arestas. O nó mais acima é o *Transform* que representa uma variável, neste caso o componente *Transform* do objeto que irá rotacionar. O nó mais abaixo é o *Rotate*, que tem como função atualizar o componente *Transform* do objeto de forma que este seja rotacionado. Por último temos o nó *Update* que está ligado ao nó *Rotate* e é responsável por ativá-lo a cada atualização de quadro.

A janela de edição do *Antares Universe* pode ser dividida em quatro partes:

1. **Área de trabalho**: região onde o grafo é confeccionado.

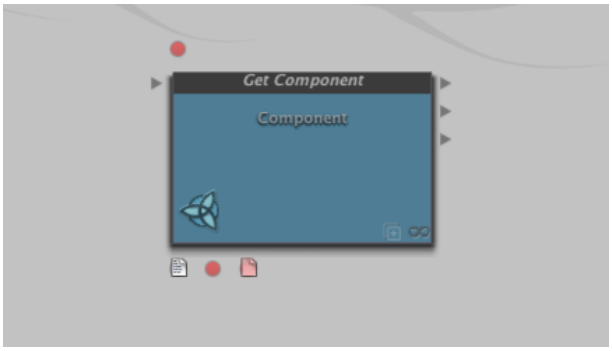


Figura 2.1: Bloco lógico *Get Component* do *Antares Universe*.

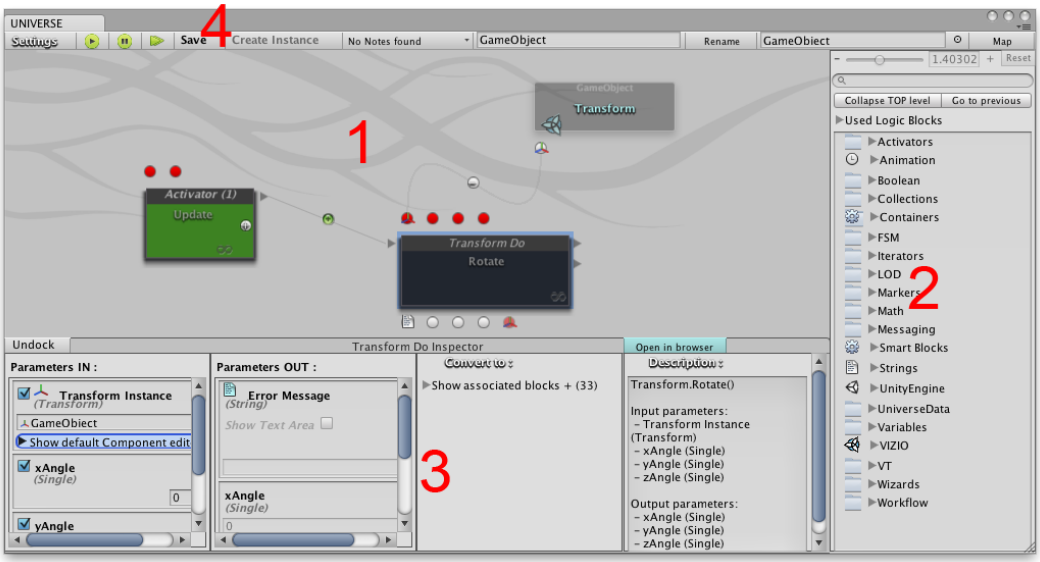


Figura 2.2: Editor da ferramenta *Antares Universe* com um grafo para rotacionar um objeto a cada atualização de quadro.

2. **Caixa de ferramentas**: nesta parte da interface encontram-se os blocos lógicos disponíveis para desenvolvimento do grafo. Os blocos lógicos estão organizados por funcionalidade, ao se clicar no botão do bloco este ficará disponível para uso na área de trabalho. Também é possível pesquisar por um bloco através do nome.

3. **Inspecionador**: informa detalhes sobre o bloco selecionado na área de trabalho. Esta região da interface possui quatro partes:

   (a) parâmetros de entrada;

   (b) parâmetros de saída;

   (c) área de conversão de bloco;

   (d) descrição;

4. **Painel de Instrumentos**: possui menus para interação com o grafo, através dos quais é possível salvar, carregar ou renomear um grafo, além de ajustar configurações globais da janela.

```
using UnityEngine;
using Antares.Vizio.Runtime;

[VisualLogicBlock("Sum of Floats Block", "Smart Blocks")]
public class MySinBlock : LogicBlock
{
  [Parameter(VariableType.In, typeof(float))]
  public Variable a;

  [Parameter(VariableType.In, typeof(float))]
  public Variable b;

  [Parameter(VariableType.Out, typeof(float))]
  public Variable aPlusB;

  public override void OnInitializeDefaultData()
  {
    RegisterOutputTrigger("Exit");
  }

  [EntryTrigger]
  public void In()
  {
    aPlusB.Value = (float)a.Value + (float)b.Value;
    ActivateTrigger("Exit");
  }
}
```

Código Fonte 2.1: Ação personalizada do editor *Antares Universe*.

O *Antares Universe* possibilita, via programação, a criação de novos blocos personalizados. Para isto, estende-se a classe *LogicBloc* e implementam-se as variáveis e gatilhos. O Código Fonte 2.1 é um bloco que calcula a soma de dois números. Este bloco possui duas variáveis de entrada (a e b) do tipo *float* e uma variável de saída ($aPlusB$), também do tipo *float*. O método *OnInitializeDefaultData* é chamado quando o bloco é instanciado e registra um gatilho de saída, neste caso, *Exit*. O método *In* é um gatilho de entrada pois possui o atributo *EntryTrigger*, este método é executado sempre que o gatilho de entrada é ativado. O método *In* calcula a soma das duas variáveis de entrada e armazena o resultado na variável *aPlusB*, logo após ativa o gatilho de saída *Exit*.

2.2 UScript

O *uScript* [28] é uma ferramenta de programação visual para a *Unity 3D* inspirada no *Kismet* da *UDK* [31]. No *uScript*, a lógica do jogo é representada por grafos e assim como no *Antares*, os nós dos grafos representam eventos, ações e variáveis, e as arestas representam a ordem de execução dos nós ou atribuem referências a variáveis. Ao se salvar um grafo a ferramenta gera três arquivos: um binário que contém informações sobre a

apresentação visual do grafo e dois outros *scripts* na linguagem C#. Um dos *scripts* é um componente e o outro contém toda a lógica presente no grafo. O componente deve ser anexado a algum objeto de jogo na cena, sua função é criar uma instância do *script* que contém a lógica do grafo.

Neste editor os nós podem se ligar através de encaixes, que podem ser de entrada/saída ou de variáveis de entrada/saída. Os encaixes de entrada/saída são representados por círculos de cor preta, os de entrada ficam à esquerda do nó, e os de saída à direita. Esses encaixes definem a ordem de execução dos nós. Ao terminar a sua execução um nó envia um sinal para que todos os nós conectados em seu encaixe de saída iniciem sua execução. Os encaixes de variáveis de entrada são representados por círculos e os de variáveis de saída por setas, a cor do encaixe indica o tipo da variável.

Na Figura 2.3 tem-se o nó *Spawn Prefab* que instancia um objeto na cena. Este nó possui um encaixe de entrada, dois encaixes de saída, três encaixes de variáveis de entrada e um encaixe de variável de saída.

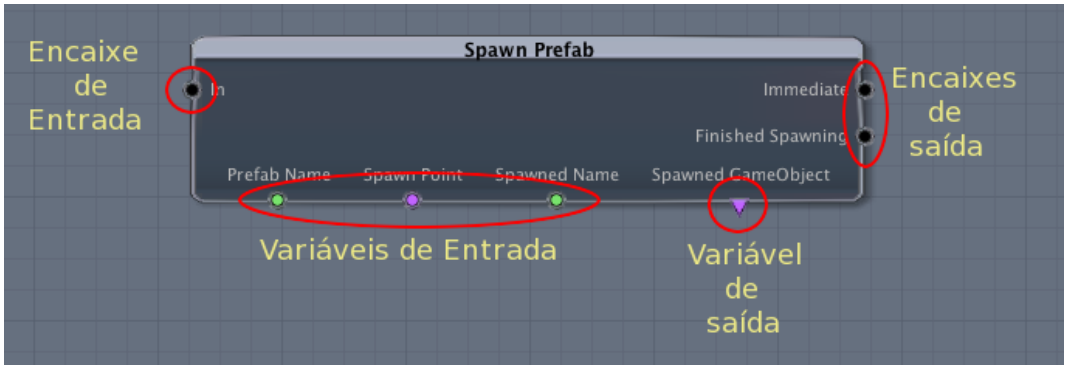


Figura 2.3: Encaixes do nó *Spawn Prefab* do editor *uScript*.

A Figura 2.4 apresenta um exemplo dos três tipos de nós existentes no *uscript*, da esquerda para a direita tem-se:

- Evento: possui a borda na cor alaranjada, não possui encaixe de entrada e é executado quando um determinado evento ocorre.
- Ação: possui a borda na cor cinza e pelo menos um encaixe de entrada.
- Variável: não possi encaixes e pode ser ligado a encaixes de variáveis de entrada ou saída do mesmo tipo.

O grafo presente na Figura 2.5 rotaciona um objeto a cada atualização de quadro. O nó *Global Update* aciona a execução do nó *Rotate* a cada atualização de quadro. O *Rotate* é responsável por rotacionar o *Game Object*, referenciado pelo nó mais abaixo, cada vez que entra em execução.

A interface do *uScript* pode ser dividida em cinco partes, conforme indicado pela numeração em vermelho na Figura 2.5 [28]:

1. **Painel de paletas de nós**: nesta parte encontra-se uma lista de nós a serem utilizados no painel de tela. Os nós aparecem neste painel como botões, clicando-se

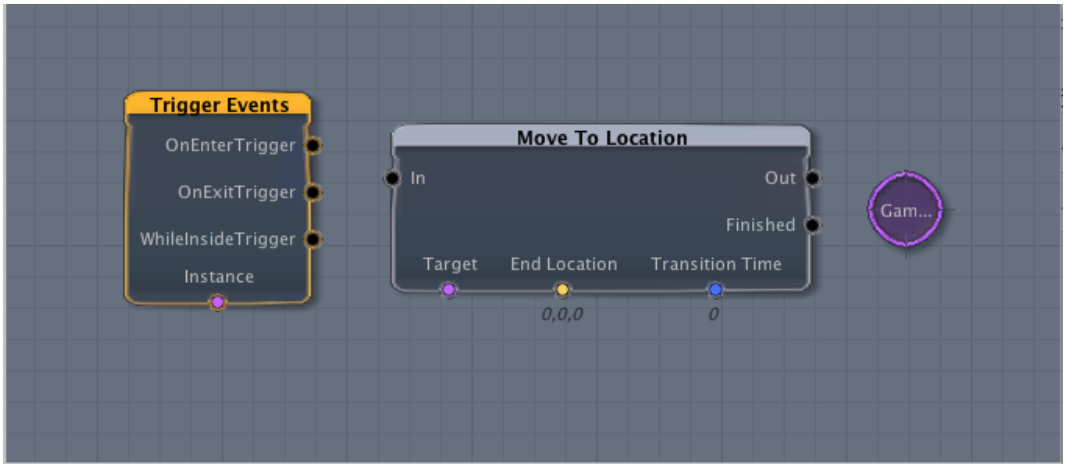


Figura 2.4: Exemplos dos três tipos de nós (evento, ação e variável) existentes nos grafos do editor do *uScript*.

em um destes botões cria-se um nó, correspondente a este botão, no centro do painel tela.

2. **Painel de tela**: a tela é a região onde os nós são inseridos e conectados para se formar um grafo lógico.
3. **Painel de propriedades**: este painel mostra todas as propriedades do nó selecionado na tela. É possível editar os valores de uma propriedade ou esconder/mostrar um encaixe de variável.
4. **Painel de referência**: contém informações textuais sobre o nó selecionado na tela. Também possui botões para a documentação *on-line* e o arquivo correspondente ao código fonte do nó.
5. **Painel *uScript***: este painel mostra todos os grafos criados no projeto. Também possui informações sobre os grafos, como por exemplo, qual grafo está carregado, qual não está salvo, etc.

Estendendo-se a classe *uScriptLogic* ou a classe *uScriptEvent* é possível criar novos nós de ações ou eventos respectivamente. O Código Fonte 2.2 mostra um nó de ação personalizado que possui duas variáveis de entrada (a e b) e uma de saída ($aPlusB$), todas do tipo *float*. O método *In* é apresentado na tela como um encaixe de entrada, e o método *Out* é representado como um encaixe de saída. Ao receber o sinal de entrada, o método *In* é executado e realiza a soma dos números.

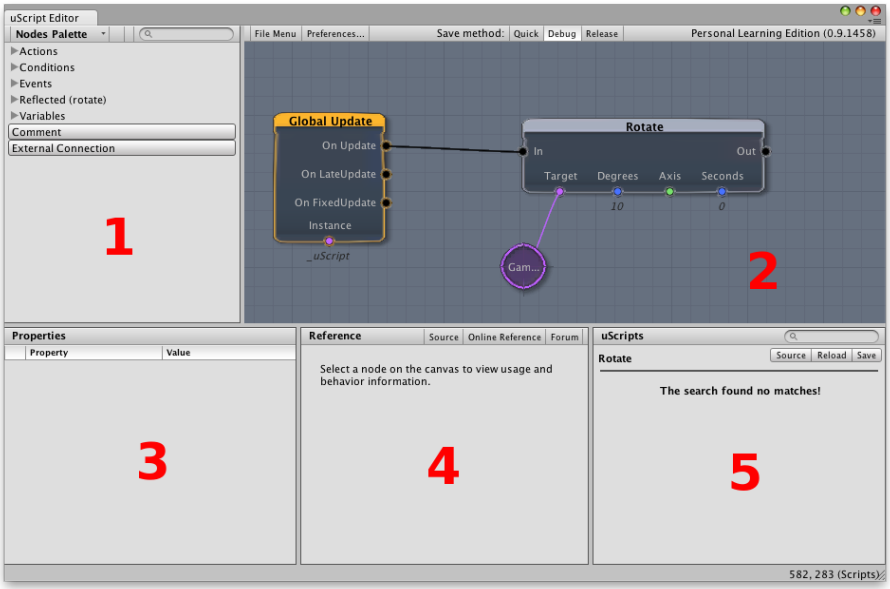


Figura 2.5: Interface do *uScript* com um grafo para rotacionar um objeto a cada atualização de quadro.

```
using UnityEngine;

[NodePath("Actions/Math")]
public class MySumAction : uScriptLogic
{
  public bool Out { get { return true; } }

  public void In( float a, float b, out float aPlusB )
  {
    aPlusB = a + b;
  }
}
```

Código Fonte 2.2: Código de uma ação personalizida no editor *uScript*.

2.3 Playmaker

O *Playmaker* [8] é uma extensão ao editor da *Unity3D* que possibilita o desenvolvimento de máquinas de estados finitos de forma visual. Através deste editor é possível criar estados, definir ações para estes estados, e determinar as transições entre os estados. Mudanças de estados são desencadeadas por eventos. Existem alguns eventos pré-definidos que estão relacionados com eventos do motor *Unity 3D* como, por exemplo, os eventos de clique de mouse. O *Playmaker* também possibilita a criação de eventos personalizados que podem ser disparados através de ações.

A Figura 2.6 ilustra um exemplo de uma máquina de estados no editor *Playmaker*. Esta máquina de estados possui dois estados, *On* e *Off*, e três transições desencadeadas pelos eventos *Start*, *Mouse Down* e *Mouse Up*. Quando a máquina de estados se torna

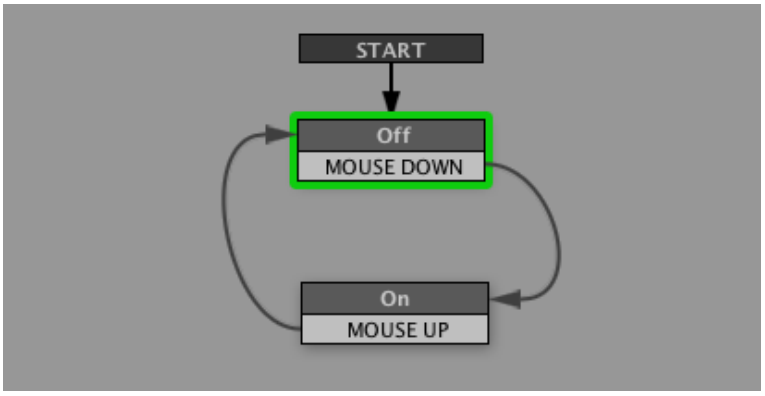


Figura 2.6: Máquina de estados no editor *Playmaker* que modela o comportamento de um interruptor.

ativa o evento *Start* é disparado e torna o estado *Off* ativo. Ao se tornar ativo um estado executa todas as suas ações, na ordem em que foram definidas. O evento *Mouse Down* e *Mouse Up* são disparados, respectivamente, pelo sistema quando o usuário pressiona algum botão do mouse e quando solta algum botão do mouse que esteja pressionado. Assim a transição do estado *Off* para o estado *On* ocorre quando o usuário pressiona algum botão do mouse, e do estado *On* para o *Off* quando o usuário solta algum botão do mouse.

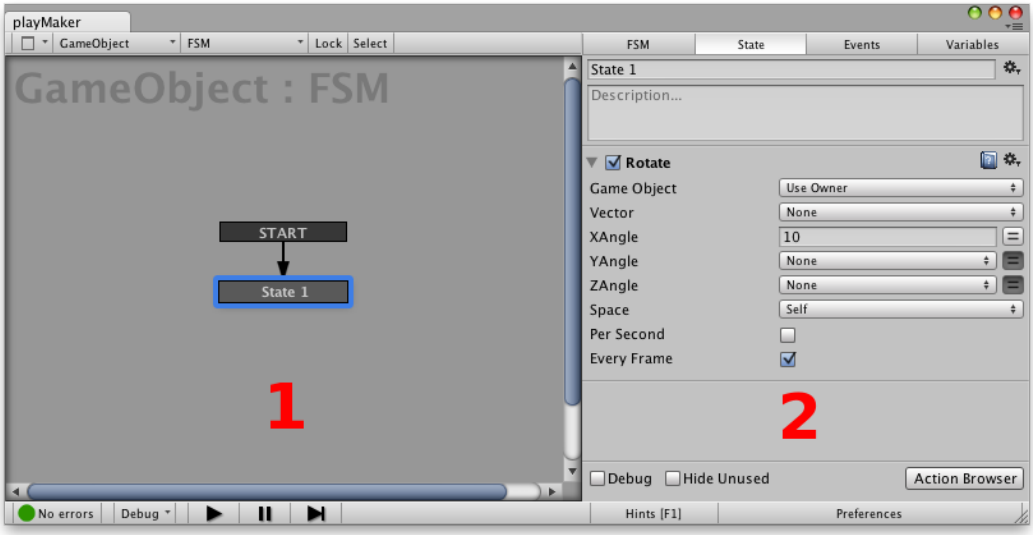


Figura 2.7: Interface do *Playmaker* com uma máquina de estados para rotacionar um objeto a cada atualização de quadro.

A Figura 2.7 é uma máquina de estados com um evento *Start* e um único estado chamado de *State 1*. O estado *State 1* possui uma única ação (*Rotate*) responsável por rotacionar o objeto. O evento *Start* torna o estado *State 1* ativo e inicia a execução da ação *Rotate*.

A janela principal do *Playmaker* pode ser dividida em duas partes, conforme indica as numeração em vermelho na Figura 2.7 e a respectiva descrição abaixo:

1. **Editor de grafo**: nesta região é possível adicionar novos estados, determinar as transições, entre os estados, e os eventos que as desencadeiam.
2. **Inspecionador**: através deste painel é possível editar propriedades da máquina de estados finito e também de seus estados. Este painel possui as abas:
   (a) ***FSM***: nesta aba é possível editar propriedades da máquina de estados finitos.
   (b) ***State***: contém as propriedades das ações do estado selecionado no editor de grafo, possibilita a alteração, inclusão, remoção e mudança de ordem das ações.
   (c) ***Event***: aba para criação de eventos personalizados.
   (d) ***Variables***: possibilita a criação de variáveis que podem ser utilizadas para troca de informações entre as ações dos estados.

O *Playmaker* possibilita a programação de novos tipos de ações, que são desenvolvidas estendendo-se a classe *FsmStateAction* e implementando-se os métodos *Reset*, *OnEnter* e/ou *OnUpdate*. O método *Reset* é utilizado para inicializar as variáveis da ação, o método *OnEnter* é acionado quando a ação deve ser executada e o método *OnUpdate* é chamado a cada atualização de quadro, enquanto o estado estiver ativo.

O Código Fonte 2.3 mostra uma ação personalizada que soma dois números do tipo *float*. A ação possui duas variáveis de entrada do tipo *FsmFloat*, este tipo é utilizado pelo *Playmaker* para armazenar variáveis do tipo *float*. O método *Reset* é utilizado para inicializar as variáveis do objeto e o método *OnEnter* realiza a soma dos números.

```
using UnityEngine;
using System.Collections;

namespace HutongGames.PlayMaker.Actions
{
  [ActionCategory(ActionCategory.Math)]
  [Tooltip("Sums one Float by another.")]
  public class MySumAction : FsmStateAction {
    [UIHint(UIHint.Variable)]
    public FsmFloat floatVariable;
    public FsmFloat sumBy;

    public override void Reset()
    {
      floatVariable = null;
      sumBy = null;
    }

    public override void OnEnter()
    {
      floatVariable.Value += sumBy.Value;
      Finish();
    }
  }
}
```

Código Fonte 2.3: Ação personalizida do *plugin Playmaker*.

2.4 *NoCode* e *NoCode-FSM*

Dois outros *plugins* de programação gráfica para a *Unity 3D* precederam o desenvolvimento do Bonsai: *NoCode* e *NoCode-FSM*. O *NoCode*, Figura 2.8, possui inspiração na ferramenta *Kodu* [20]. A lógica é construída através de instruções que possuem um conjunto de condições e ações. Durante a execução, cada instrução avalia seu conjunto de condições para decidir se executa, ou não, seu conjunto de ações. A execução de uma instrução é condicionada a um evento, que é indicado no primeiro atributo da instrução. Por exemplo, a primeira instrução da Figura 2.8 ocorre no evento *Start*, padrão do motor *Unity 3D* e correspondente à eventos como início de jogo, atualização de quadro, atualização da física, etc.

As instruções são armazenadas em um componente chamado de *NCLogic*, que possui toda a informação necessária para a execução da lógica específica. Além das instruções, o componente *NCLogic* também armazena variáveis, que são utilizadas pelas condições e ações. A Figura 2.9 corresponde à visualização de um componente *NCLogic* na janela de componentes, ou *Inspector View*, da *Unity 3D*. Este componente possui três variáveis: *speed*, *target* e *direction*, que armazenam valores do tipo *float*, *GameObject* e *Vector3* respectivamente. Nesta mesma janela é possível incluir novas variáveis e editar atributos (como nome, cor , símbolo, etc) das variáveis já criadas.

NoCode-FSM, Figura 2.10, é uma evolução da *NoCode*, que se baseia em máquinas de estados finitos, em que cada estado corresponde a um conjunto de instruções como os da

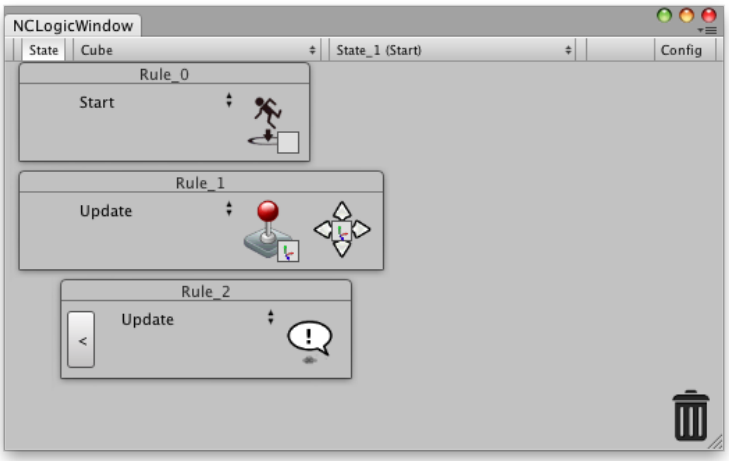


Figura 2.8: Ferramenta *NoCode* com três instruções em sua janela.

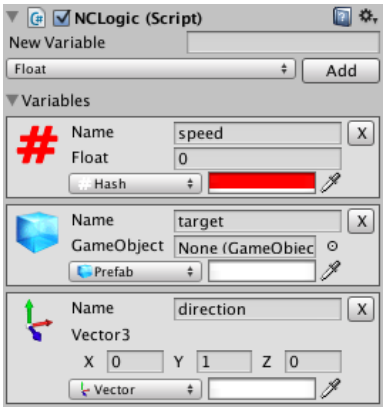


Figura 2.9: Um componente *NCLogic* com três variáveis na *Inspector View*.

ferramenta *NoCode*. O componente *NCLogic* deste *plugin* armazena, além das instruções e variáveis, uma máquina de estados finitos (com seus estados e transições).

Os dois *plugins*, *NoCode* e *NoCode-FSM*, permitem que o conjunto de ações e condições possam ser estendidos via programação. Os nós de ação possuem como classe base a classe *NCActionBaseNode*, os nós do tipo condição herdam da classe *NCConditionBaseNode*. A classe *NCActionBaseNode* possui o método virtual *Do*, que é invocado quando a ação é executada. A classe *NCConditionBaseNode* possui o método virtual *Test*, que retorna um valor booleano para validação da condição.

O Código Fonte 2.4 mostra uma ação personalizada, denominada *FloatAdd*, dos *plugins* *NoCode* e *NoCode-FSM*. Esta ação possui três variáveis, que armazenam valores do tipo *float*, como atributo. Em sua execução, método *Do*, a variável *aPlusB* recebe a soma das variáveis *a* e *b*.

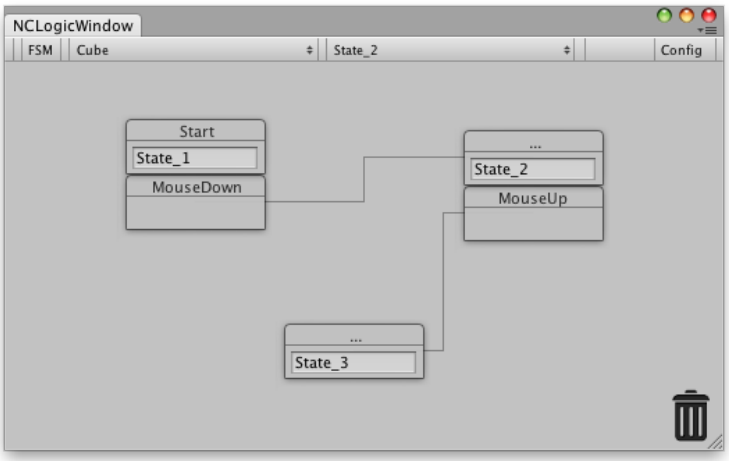


Figura 2.10: Ferramenta *NoCode-FSM* com três instruções em sua janela.

```
using UnityEngine;
using NoCode.Nodes;

[Info("FloatAdd",
  iconPath="Assets/NoCode/Editor/Images/EditFloat.png",
  description = "store = a + b")]
public class FloatAdd : NCActionBaseNode {

  public VariableFloat a;
  public VariableFloat b;

  [RequireVariable]
  [NCVariableInfo(SlotPosition.Corner4)]
  public VariableFloat aPlusB;

  public override void Do () {
    aPlusB.Value = a.Value + b.Value;
  }
}
```

Código Fonte 2.4: Código de uma ação personalizida dos *plugins NoCode* e *NoCode-FSM*.

Apesar de simples, a ferramenta *NoCode* possui pouca expressividade, não possui estruturas de repetição, blocos de instrução e outras estruturas. Embora seja fácil representar a lógica em forma de máquina de estados, a ferramenta *NoCode-FSM*, possui alguns problemas, além dos inerentes à máquina de estados finitos: representar estruturas condicionais complexas (*if-elseif* ou *switch*) ou estruturas de repetição é uma tarefa difícil.

Capítulo 3

Revisão Teórica

Nesse capítulo é feita uma revisão teórica sobre os conceitos envolvidos na construção do *plugin* Bonsai. A Seção 3.1 contém informações sobre os principais agentes inteligentes utilizados em jogos eletrônicos. Durante a Seção 3.2 é apresentada uma comparação entre as metodologias de tomada de decisão para comportamento de agentes reativos. Esta comparação é fundamental para compreender as decisões de projeto presentes no desenvolvimento da Bonsai. Na Seção 3.3 tem-se uma breve introdução ao motor de jogos eletrônicos *Unity 3D*, seu fluxo de trabalho e os conceitos envolvidos no processo de desenvolvimento de um jogo eletrônico nesta *engine*.

3.1 Agentes Racionais

Russel e Norvig [25] descrevem um agente como tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e agir sobre este ambiente através de atuadores. Uma definição mais recente é apresentada em [7]: "Um agente é uma entidade autônoma capaz de interagir com o ambiente e outros agentes". Assim, um ser humano no ambiente a sua volta pode ser considerado um agente, uma vez que é uma entidade que toma decisões por conta própria, possui sensores (ouvidos, olhos e outros órgãos) e atuadores (braço, corpo, etc) que agem sobre o mesmo ambiente ou outros seres humanos (agentes).

Não existe um consenso quanto a definição de agente [3], porém as idéias de autonomia e interação com o ambiente norteiam este conceito. No contexto deste trabalho entende-se por agente algo que possui autonomia e interage com o ambiente.

Um agente é considerado racional quando é capaz de tomar a melhor decisão possível com as informações que possui. Para isto deve-se estabelecer uma medida de desempenho. Esta medida deve classificar a ação escolhida pelo agente em contrapartida com o estímulo recebido. Pode-se entender um agente racional como aquele que escolhe a ação que tenta maximizar o desempenho a partir de seu conhecimento [25].

3.1.1 Agente Reativo Simples

O tipo mais simples de agente é o reativo simples. Este agente escolhe qual ação executar baseando-se apenas nas informações disponibilizadas por seus sensores naquele

momento. Esses agentes possuem um modelo simples porém limitado. Tais agentes decidem levando em consideração apenas a percepção atual [25].

Em jogos eletrônicos diversas entidades podem ser modeladas como agentes reativos simples e gerar agentes racionais. Considere a seguinte situação em um jogo: o personagem do jogador deve pressionar um botão para abrir uma porta e prosseguir no jogo. A ação (abrir uma porta) do botão depende unicamente de um estímulo do personagem.

O comportamento do botão descrito anteriormente pode ser generalizado para o seguinte: dada uma condição execute uma ação. Este tipo de comportamento é comumente encontrado em diversas entidades de jogos eletrônicos.

3.1.2 Agente Reativo Baseado em Modelo de Ambiente

Os agentes reativos baseados em modelo possuem a capacidade de manter um estado interno parcial do ambiente. Este estado interno deve influenciar na tomada de decisão do agente. Conforme o agente baseado em modelo recebe estímulos do ambiente seu estado interno é atualizado. Deste forma, ao longo do tempo o estado interno do agente é modificado.

Este mecanismo faz com que o agente decida levando em consideração o seu histórico de estímulos, diferente do agente reativo simples que considera apenas a situação atual. Entretanto, para ter um bom resultado o agente precisa ter algum conhecimento de como é o seu ambiente, saber como seu ambiente funciona e com base na memória estimar a situação atual. Este conhecimento se encontra codificado no sistema que atualiza os estados internos dos agentes. A idéia é que este tipo de agente possa manter uma representação, mesmo que simbólica, de como é o seu ambiente. Esta representação é conhecida como modelo do mundo [25].

Agora, considere a seguinte situação em um jogo eletrônico: o personagem do jogador deve pressionar um botão duas vezes para abrir uma porta e prosseguir no jogo. Nesta situação, a ação (abrir a porta) do botão depende de dois estímulos que ocorrem em ocasiões diferentes. Logo, o comportamento deste botão não pode levar em consideração apenas os estímulos em um dado instante. A informação de que o botão foi pressionado na primeira vez deve ser armazenada no tempo, para se ter conhecimento quando o botão for pressionado novamente. Assim, o botão precisa atualizar o seu estado interno no tempo para poder decidir se ao ser pressionado deve, ou não, abrir a porta.

3.1.3 Agentes Reativos e Jogos Eletrônicos

Existem diversos outros tipos de agentes, dentre eles os agentes baseados em objetivos, agentes baseados na utilidade e com aprendizagem [25]. Alguns agentes possuem a capacidade de armazenar informações internamente, ou todos estímulos recebidos, e utilizar tais dados para decidir sobre a ação a ser tomada.

Uma importante diferença entre a robótica e os jogos eletrônicos é que neste último o ambiente do agente é o próprio mundo do jogo [10]. Isto implica que toda a informação do ambiente pode ser disponibilizada para os agentes. Outro importante fato é que os sensores dos agentes em jogos eletrônicos baseiam-se em símbolos e não em sensores reais (que podem conter erro) [10]. Por estes fatores os agentes reativos são frequentemente utilizados em jogos eletrônicos.

3.2 Principais Metodologias para Tomada de Decisão

O comportamento de tomada de decisão pode ser modelado através de diversas representações, entre elas temos as máquinas de estados finitos, máquinas de estados hierárquicas, árvore de comportamento, etc. Basicamente estas representações tentam responder à mesma pergunta, ou seja, como mapear um conjunto de entradas para um conjunto de ações?

Nem sempre esta lógica é trivial, existem comportamentos complexos com centenas de entradas, condições e ações. O modelo deve se responsabilizar por tratar todas estas informações e ainda possuir escalabilidade ao mesmo tempo que se mantém inteligível para os seres humanos. Durante esta seção são apresentados diversas representações para modelar o comportamento de tomada de decisão, suas vantagens e desvantagens.

3.2.1 Máquina de Estados

Máquinas de estados são comumente utilizadas para descrever comportamentos. A vantagem de uma máquina de estados está na simplicidade. Uma máquina de estados é composta por apenas dois componentes [21]:

- **Estado:** Caracteriza-se por uma sequência de ações.
- **Transição:** Corresponde a um caminho de um estado para outro. Uma transição relaciona um evento com dois estados, informa qual será a mudança de estado caso uma determinada condição seja verdadeira.

Uma máquina de estado nada mais é do que um conjunto de estados e um conjunto de transições entre estes estados. Em um dado instante somente um estado pode estar em execução, a mudança de estado ocorre quando uma determinado condição é satisfeita.

Máquina de Estados Finitos

Máquinas de estados finitos (FSM - *Finite State Machine*) são máquinas de estados que possuem um conjunto finito de estados. FSM é a técnica mais comum para modelagem de comportamento de tomada de decisão em jogos eletrônicos [21]. Apesar do conceito simples, estas máquinas podem ter milhares de estados, o que prejudica sua escalabilidade e visualização gráfica.

A Figura 3.1 mostra uma máquina de estados que modela o comportamento de um agente robô que tem a função de coletar recursos e trazê-los à base. A máquina de estados contém os seguintes estados: *Procurando*, *Extraindo* e *Entregando*. As transições entre os estados são desencadeadas pelos eventos: *Coleta Recursos*, *Entrega Recursos* e *Encontra Recursos*.

Em comportamentos complexos, frequentemente têm-se a necessidade de duplicação de estados [21]. O aumento do número de estados ocorre quando se deseja representar agentes orientados a objetivos. Para representar tais agentes define-se um objetivo como um conjunto de estados. Porém, quando um outro objetivo possui alguns estados semelhantes, é necessário duplicar estes estados no novo objetivo. Isto é necessário para que em uma transição de estados o objetivo atual seja mantido.

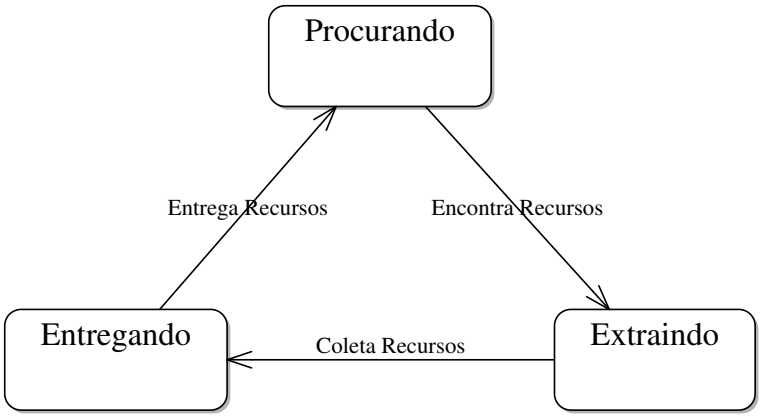


Figura 3.1: Exemplo de uma máquina de estados finitos de um agente robô que coleta recursos.

No exemplo do agente robô que coleta recursos, caso este também necessite recarregar sua bateria quando esta estiver fraca, é necessário criar um novo estado, denominado Recarregando, na *FSM*. Este estado deve ser prioridade para o agente, uma vez que sem bateria o agente não pode exercer a função de coletar recursos. Desta forma os três outros estados (Procurando, Entregando e Extraindo) devem conter transições para um estado onde a bateria deve ser recarregada. Porém, ao se carregar a bateria o agente deve retornar para o estado anterior ao estado Recarregando, para que a função de coletar recursos continue de um estado válido. Nesta situação é preciso criar um estado Recarregando para cada estado relacionado com a função de coletar recursos. A Figura 3.2 representa a máquina de estados finitas deste agente.

Máquina de Estados Finitos Hierárquica

Ao invés de modelar todo o comportamento em uma única máquina de estados, em uma máquina de estados finitos hierárquica (HFSM - *Hierarchical Finite State Machine*), o comportamento é dividido em vários sub-comportamentos. Cada um destes sub-comportamentos corresponde a uma FSM, e estas máquinas também são ligadas entre si por transições. Em um dado instante somente uma destas máquinas pode estar em execução.

A vantagem da HFSM em relação à uma FSM é uma melhor modularidade. Isto aumenta o reaproveitamento de partes de uma máquina em outra, e facilita a representação de comportamentos mais complexos [21]. Porém, as HFSMs sofrem do mesmo problema que as FSMs, o aumento da complexidade do comportamento impacta em um aumento no número de transições, estados e FSMs; dificultando a sua manutenção, atualização e visualização [10]. Representar objetivos ainda é uma tarefa difícil para estas máquinas.

3.2.2 Árvore de Comportamento

Árvore de comportamento (BT - *Behaviour Tree*) é uma metodologia para projetar comportamentos de sistemas autônomos [13]. Esta metodologia reúne diversas técnicas de inteligência artificial: HFSM, *scheduling*, execução de ações e planejamento [21].

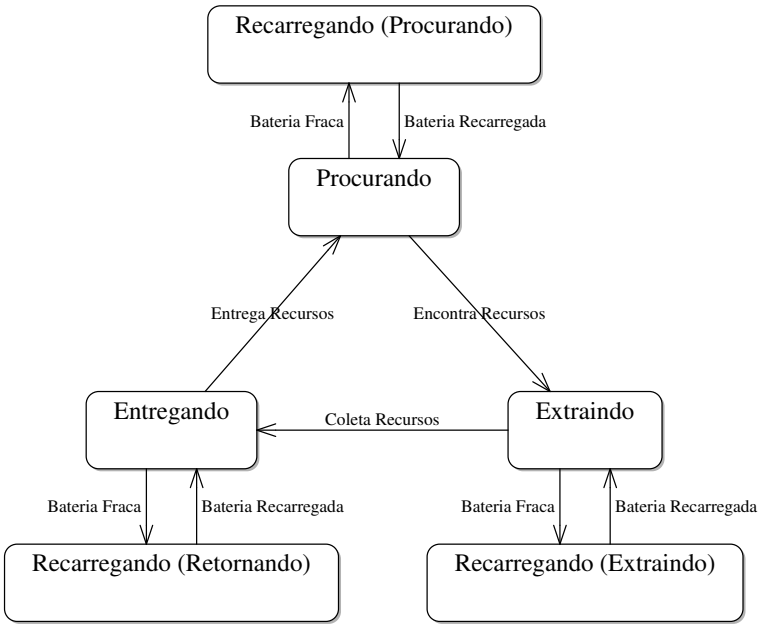


Figura 3.2: Exemplo de uma máquina de estados finitos de um agente robô que coleta recursos e precisa recarregar sua bateria para manter o funcionamento.

A proposta da BT é representar o comportamento em uma estrutura de árvore, de modo que cada sub-árvore corresponda a um sub-comportamento. Estas sub-árvores são chamados de tarefas.

A BT se assemelha à uma HFSM, entretanto, ao contrário de um estado, o bloco principal é uma tarefa [21]. Uma importante diferença entre uma HFSM e uma árvore de comportamento, é que nesta última a própria estrutura define a prioridade de execução das tarefas (seu bloco principal). Facilitando a modelagem de comportamentos que possuem objetivo, um dos problemas presentes nas HFSMs.

Execução

A execução da árvore de comportamento ocorre com busca em profundidade iniciando-se pela raiz. A primeira folha encontrada será executada, o valor retornado de sua execução será repassado para seu pai. O pai processa esta informação e, dependendo de sua lógica de tomada de decisão, ou seus demais filhos são processados ou a execução nesta sub-árvore é interrompida e a árvore continua a ser processada com busca em profundidade na próxima sub-àrvore.

Ações ou Condições (Folhas)

As folhas de uma BT podem ser de dois tipos: ação ou condição. As folhas representam a interface de comunicação com o ambiente. As condições lêem informações sobre o estado do ambiente e executam validações, as ações modificam o ambiente [4]. As ações ou condições podem ser complexas ou simples, isto é, uma condição pode corresponder a identificar se um inimigo está no campo de visão ou pode ser simplesmente uma comparação entre os valores de duas variáveis.

Tarefas de Combinação (Ramos)

Enquanto as folhas possuem condições e ações, os ramos (denominados de tarefa de combinação) contém a lógica de tomada de decisão. Os ramos controlam quais filhos (condições, ações e outros ramos) serão executados.

No término da execução tanto os ramos quanto as folhas retornam um valor ao seu pai, este valor indica se o nó ou seus sub-nós, no caso dos ramos, foram executados. Diversas implementações de árvores também retornam outras informações sobre sua execução, como por exemplo se houve algum erro. A escolha dos ramos é realizada levando-se em conta o valor retornado por seus filhos.

Existem vários tipos de ramos, e cada um possui uma lógica diferente para tomada de decisão. Os tipos mais comuns de tarefa de combinação são a seleção e a sequência.

A "sequência" retorna o valor *verdadeiro* quando todos os seus filhos também retornam o valor *verdadeiro*. Esta tarefa de combinação executa os seus filhos em ordem, e caso um deles retorne o valor *falso* a "sequência" interrompe a execução e retorna o valor *falso*.

A árvore na Figura 3.3 representa o comportamento de luta de uma entidade em um jogo eletrônico. As folhas de cor alaranjada são condições e as de cor cinza ações. O ramo "Lutar" é do tipo sequência e portanto só retornará o valor *verdadeiro* se todas as suas folhas ("Encontrou Inimigo?","Inimigo na mira" e "Atacar inimigo") também retornarem o valor *verdadeiro*.

Após a raiz, a condição "Encontrou Inimigo?" é executada na árvore da Figura 3.3. Se esta condição for verdadeira o próximo nó a ser executado é a condição "Inimigo na mira?". Se a condição "Inimigo na mira?" for falsa a execução dos filhos de "Lutar" é interrompida e o valor *falso* é retornado por este último nó. A ação "Atacar" é executada se o retorno de "Inimigo na mira?" também for verdadeiro.

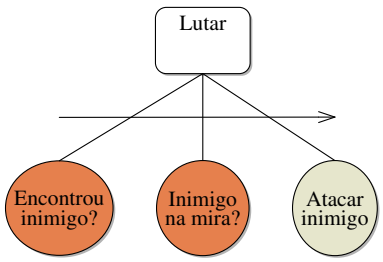


Figura 3.3: Tarefa de combinação do tipo sequência de uma árvore de comportamento.

A "seleção" retorna o valor *verdadeiro* se pelo menos um de seus filhos também retornar o valor *verdadeiro*. Caso um de seus filhos retorne o valor *verdadeiro* a execução é interrompida e o valor *verdadeiro* é retornado para seu o pai. Se a execução chegar ao final e nenhum valor *verdadeiro* for retornado pelos seus filhos, a "seleção" retorna o valor *falso*.

A árvore na Figura 3.4 representa o comportamento de descanso de um agente. Este comportamento possui uma tarefa de combinação do tipo seleção ("Descansar") e três ações ("Deitar", "Sentar", "Se escorar"). O primeiro nó a ser executado na árvore 3.4 é a ação "Deitar". Caso esta ação retorne *verdadeiro* a execução dos filhos da tarefa de seleção de "Descansar" é interrompida e o valor *verdadeiro* é retornado ao pai deste último nó. Caso o valor retornado por "Deitar" seja *falso*, o próximo filho ("Sentar") será executado.

Se todos os filhos de “Descansar” retornarem o valor *falso* então este último também retornará o valor *falso*.

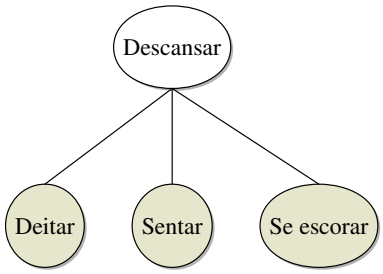


Figura 3.4: Tarefa de combinação do tipo seleção de uma árvore de comportamento.

A árvore na Figura 3.5 possui mais tarefas que as anteriores e representa o comportamento de patrulhar de um agente. A raiz desta árvore é um nó de seleção, assim, ou o agente executa a tarefa “Atacar” ou a tarefa “Procurar inimigo”. A tarefa "Atacar"corresponde a mesma árvore apresentada na Figura 3.3. A possibilidade de reutilizar tarefas em outras árvores é uma das vantagens da árvore de comportamento [13].

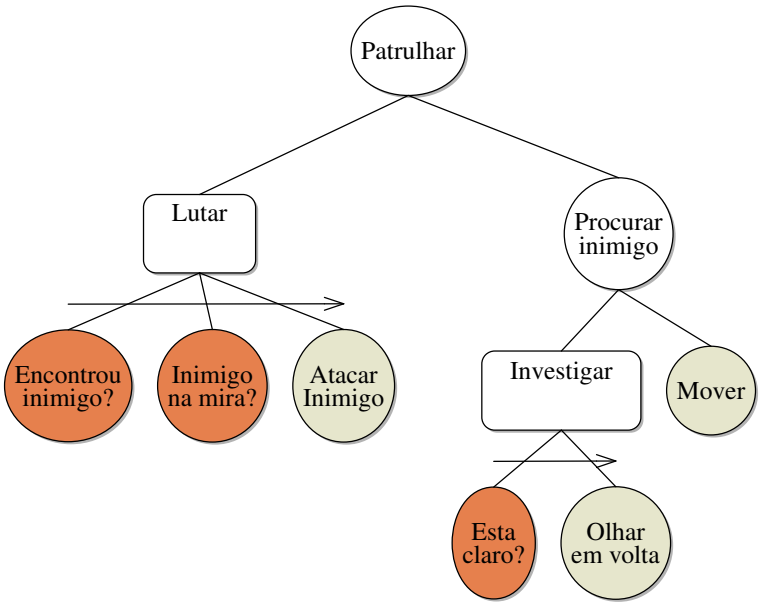


Figura 3.5: Exemplo de uma árvore de comportamento.

Decoradores (Ramos com um único filho)

O decorador é um tipo especial de tarefa de combinação que possui apenas um único filho. A função desta tarefa é adicionar um comportamento diferente a seu filho, modificando a execução desta sub-árvore. Geralmente estes decoradores modificam o retorno de um nó ou definem, utilizando algum parâmetro dinâmico, quando executar seus filhos. Tais tarefas são inspiradas no padrão de projeto de mesmo nome [4, 21].

Em um exemplo mais complexo, a árvore na Figura 3.6 também representa o comportamento de descansar de um agente. Porém, um nó decorador, denominado "Apenas 2 Vezes", é inserido para adicionar um limite ao número de vezes que a ação "Deitar" pode ser executada.

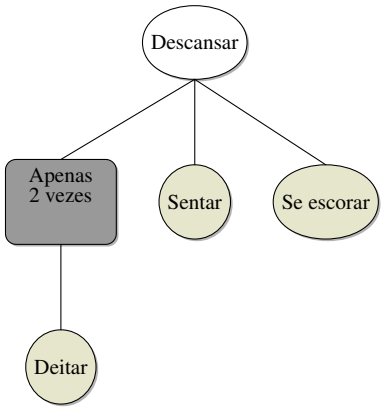


Figura 3.6: Exemplo de uma árvore de comportamento com um nó decorador.

Relação com Linguagem de Programação

As árvores de comportamento possuem uma estreita relação com as linguagens de programação [21, 25]. Por exemplo, na Figura 3.7 é mostrado como construir uma condição, similar à instrução *if* de linguagens de programação, em uma árvore de comportamento. Utiliza-se uma tarefa de combinação do tipo seleção e duas sub-árvores P e b. A sub-árvore P será executada somente se a sub-árvore b retornar o valor verdadeiro.

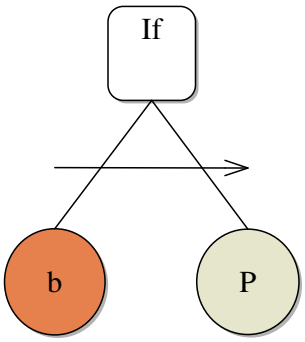


Figura 3.7: Instrução *if*.

Árvore de Comportamento em Jogos Eletrônicos

Halo 2 [14], lançado em 2004, foi um dos primeiros jogos a utilizar árvores de comportamento para modelar o comportamento dos agentes. Este modelo foi implementado no jogo *Halo 2* para satisfazer alguns requisitos de projeto, como: escalabilidade, trânsparência, coerência e facilidade de trabalho [14]. A árvore de comportamento possibilitou que

os *designers*, sem conhecimentos sobre programação, pudessem desenvolver a lógica de tomada de decisão dos inimigos e *NPC's*[1]. Desde então, as BT's tornaram-se um modelo popular para criação de inteligência artificial de personagens em jogos eletrônicos. Em 2011 o jogo *Spore* utilizou uma versão modificada das BTs presente em *Halo* para criar um sistema de inteligência artificial para milhares de criaturas virtuais [11].

3.3 *Unity 3D*

A *Unity 3D* é um motor para jogos eletrônicos com foco na usabilidade e desenvolvimento para múltiplas plataformas [30]. Seu principal público alvo são pequenas empresas e desenvolvedores independentes de jogos eletrônicos. Apesar de estar disponível desde 2005 a *Unity 3D* só começou a se tornar popular ao disponibilizar uma versão gratuita e um editor para o sistema operacional *Windows*. Segundo a pesquisa da revista *Game Developer*, realizada entre desenvolvedores de jogos para plataformas móveis e sociais, 53.1% deles utilizam o motor *Unity 3D* [19].

3.3.1 Objetos de Jogo e Componentes

Neste motor os jogos são desenvolvidos em uma arquitetura baseada em entidade componente, em que todo o jogo é formado por um único tipo de entidade chamada de objeto de jogo. Os objetos de jogo são containers para componentes. Um componente adiciona uma determinada funcionalidade a um objeto de jogo. Por exemplo, o componente *Rigidbody* adiciona características de corpo rígido (massa, gravidade, etc) ao objeto, o componente *MeshRendering* desenha o objeto na cena. Desta forma, o que diferencia um personagem de um veículo ou de uma parede é o conjunto de componentes adicionado neste objeto de jogo.

3.3.2 Interface Principal

A Figura 3.8 apresenta a janela principal da ferramenta *Unity 3D*. Esta janela se subdivide em regiões:

- *Project View*: contém todos os recursos (também chamados de *assets*) a serem utilizados na construção do jogo. Exemplos de *assets* são: arquivos de áudio, texturas, *scripts*, *prefabs*, etc. A *project view* corresponde a arquivos e pastas do projeto no sistema do usuário.
- *Hierarchy View*: contém todos os objetos de jogo utilizados na cena.
- *Scene View*: está intimamente relacionada com a *hierarchy view*. Os objetos de jogo da *hierarchy view* são selecionados e manipulados na *scene* de modo a transformarem-se no cenário do jogo.
- *Game View*: representa como será o seu jogo após publicado. É desenhada a partir das câmeras de seu jogo.

[1] *NPC*, ou Non Player Character, é um tipo de personagem em jogos eletrônicos que não pode ser controlado pelo jogador.

- *Inspector View*: mostra informações detalhadas sobre um objeto de jogo e seus componentes.
- Barra de Ferramentas: contém botões e comandos para se manipular/criar objetos de jogo e adicionar ou remover componentes.

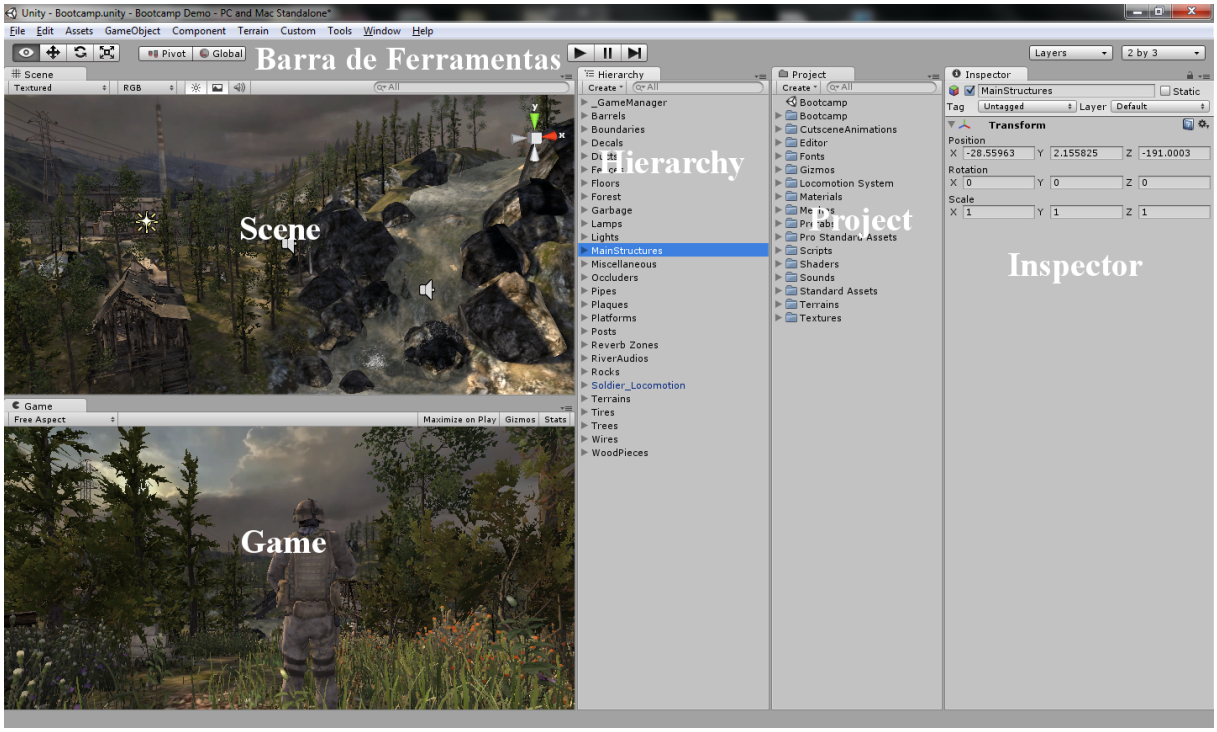


Figura 3.8: Janela principal do motor de jogos *Unity 3D*

3.3.3 *Prefab*

O *prefab* é um *asset* modelo para criação de objetos de jogo. Isto é, corresponde a um objeto de jogo pré-configurado pelo usuário (com componentes, etc) a ser utilizado na confecção da cena. Os *prefabs* são construídos pelo usuário conforme sua necessidade e se encontram na *project view* como os demais recursos de jogo.

3.3.4 Construção de um Jogo na *Unity 3D*

No motor *Unity 3D*, um jogo corresponde a um conjunto de cenas. Uma cena pode corresponder à uma fase do jogo ou uma tela de menu, por exemplo. Somente uma cena pode ser construída por vez. A cena é um *asset* composto por um conjunto de objetos de jogo, os objetos de jogo da cena em edição no momento são apresentados nas regiões *hierarchy* e *scene view*.

A construção de um jogo na *Unity 3D* consiste em criar uma cena, acrescentar os objetos de jogo desejados e adicionar comportamento/função aos objetos através de componentes. Os comportamentos ou funções adicionados correspondem a lógica específica

do jogo. A *engine* já contém diversos componentes que podem ser utilizados, mas não é o suficiente para se construir qualquer tipo de jogo. A lógica específica do jogo é basicamente desenvolvida através da construção de novos componentes, que consistem em *scripts*, nas linguagens *Boo*, *UnityScript* ou C#. Estes *scripts*, obrigatoriamente devem herdar da classe abstrata *MonoBehaviour*.

O fluxo de trabalho na *Unity 3D* consiste em operações de arrastar e soltar com o *mouse*. Por exemplo, para adicionar uma textura (na *project view*) em um objeto de jogo na cena, basta arrastar a textura para cima do objeto de jogo na *hierachy* ou *scene view*. O mesmo mecanismo pode ser utilizado para adicionar um componente criado através de um *script* ou para adicionar um objeto de jogo baseado em um *prefab* na cena.

Capítulo 4

Bonsai

Os motores de jogos conquistaram um importante espaço no processo de desenvolvimento de jogos eletrônicos de modo que, atualmente com uma mesma *engine* são construídos diversos jogos diferentes em um menor intervalo de tempo. Analisando superficialmente, o que diferencia um jogo de outro são seus modelos, texturas, arquivos de áudio, *scripts* e como o motor é configurado para trabalhar com estes objetos.

Apesar dos motores de jogos agilizarem o processo de criação de jogos eletrônicos, todos os recursos (ou *assets*), incluindo o código específico do jogo, ainda devem ser desenvolvidos. O desenvolvimento do código específico requer conhecimentos sólidos em codificação de programas, e muitas vezes abrangem áreas da ciência da computação como inteligência artificial, análise de algoritmos, etc. O foco deste trabalho é o desenvolvimento de um *plugin*, denominado Bonsai, que tem como objetivo facilitar e agilizar o processo de desenvolvimento da lógica específica de jogos eletrônicos no motor *Unity 3D*.

Na Seção 4.1 tem-se uma visão geral sobre o *plugin*, as Seções 4.2, 4.3 e 4.4 possuem detalhes de projeto da Bonsai. A interface do *plugin* é apresentada na Seção 4.5. A Seção 4.6 mostra a correlação semântica que a Bonsai possui com as linguagens de programação de alto nível. Por fim, a Seção 4.7 apresenta a implementação do *plugin*.

4.1 Visão Geral

A Bonsai é um *plugin* para o motor de jogos *Unity 3D* que se enquadra na categoria de *visual scripting*, isto é, possibilita o desenvolvimento da lógica do jogo de forma visual, sem necessidade de codificação de programas. Existem diversas outras ferramentas para este fim, porém observa-se que a visualização e semântica destas ferramentas são prejudicadas dependendo do tipo de comportamento e/ou da metodologia de tomada de decisão adotada.

Dentre as metodologias presentes na Seção 3.2, a árvore de comportamento se destaca por sua relação com linguagens de programação e modularização de comportamentos em sub-árvores. A relação com linguagens de programação é interessante do ponto de vista semântico, uma vez que possibilita a construção de estruturas com mesma semântica que instruções presentes em linguagens de programação. A modularização do comportamento em sub-árvores, por sua vez, facilita a visualização de comportamentos complexos.

O desenvolvimento da Bonsai é inspirado na relação semântica entre as árvores de comportamento e as linguagens de programação. O conceito base do *plugin* é o desen-

volvimento de uma abstração de linguagens de programação. A lógica é confeccionada sem necessidade de codificar programas, permitindo que não-programadores (como *game designers*, por exemplo) possam utilizá-la, sem que se perca a expressividade presente nas linguagens de programação.

A Bonsai fornece uma abstração visual para linguagens de programação de alto nível. O fluxo de trabalho, do ponto de vista do usuário, é similar à tarefa de organizar uma estrutura de diretórios em sistemas operacionais modernos. A lógica do jogo é construída com simples ações como navegar por menus e arrastar objetos.

4.2 Componente Bonsai

No *plugin* Bonsai a lógica de um objeto de jogo é representada por uma árvore de comportamento. As árvores de comportamento são associadas a objetos de jogo através do componente Bonsai, cuja principal função é armazenar e executar uma árvore de comportamento.

O componente Bonsai corresponde a um *script* desenvolvido na linguagem C# que herda diretamente do componente *MonoBehaviour*. Os principais atributos do componente Bonsai são:

- *Name*: nome do componente.
- *Root*: referência para a raiz da árvore de comportamento.
- *CurrentFunction*: indica qual o tipo de evento originou a execução da árvore.

Um componente Bonsai só pode armarzenar uma única árvore de comportamento por vez. Entretanto, um mesmo objeto de jogo pode possuir mais de um componente Bonsai e, consequentemente, mais de uma árvore de comportamento.

4.3 *Behaviours* ou Nós

Cada árvore adiciona um comportamento ou função a um objeto de jogo. Uma árvore de comportamento é formada por nós, também chamados de *behaviours*. Cada nó é independente e possui uma função específica. Estes nós são combinados através de uma relação de pai e filho para formar uma árvore.

Um *behaviour* corresponde a um *script* que herda da classe *BonsaiBehaviour*. Existem diversos tipos de *behaviours*, sendo os principais: *CompositeBehaviour*, *DecoratorBehaviour*, *FunctionBehaviour*, *BoxBehaviour* e *PrefabTask*. As classes *CompositeBehaviour* e *DecoratorBehaviour* não possuem um comportamento definido, estes tipos apenas adicionam, respectivamente, as funcionalidades de nós do tipo tarefa de combinação e decoradores.

Apesar das árvores de comportamento possuírem estruturas com mesma semântica que estruturas de condição e repetição presentes em linguagens de programação, elas não possuem um mecanismo que se assemelhe, semanticamente, as funções ou variáveis de linguagens de programação. Os nós do tipo *FunctionBehaviour* e *BoxBehaviour* são extensões à árvore de comportamento que adicionam funcionalidades semelhantes às funções e variáveis, respectivamente, de linguagens de programação.

4.3.1 *FunctionBehaviour* ou Funções

Na Bonsai, a execução da árvore é associada a um evento. Este evento pode corresponder a um evento nativo do motor (atualização de quadro, detecção de colisão, etc) ou iniciado por um outro nó da árvore. Um nó do tipo decorador identifica este evento em execução para decidir se continua ou não a execução de seus filhos. Os nós que possuem tal tarefa pertencem a uma classe especial chamada de *FunctionBehaviour*. Estes nós agrupam um conjunto de nós em uma sub-árvore. Este comportamento é similar aos de funções em linguagens de programação, estas por sua vez agrupam um conjunto de instruções em blocos.

A Figura 4.1 mostra três nós do tipo *FunctionBehaviour*: *Update*, *OnTriggerEnter* e *Restart*. Os nós *Update* e *OnTriggerEnter* estão relacionados com eventos do componente MonoBehaviour. O nó Restart é uma função customizável relacionada a um evento disparado por um outro nó.

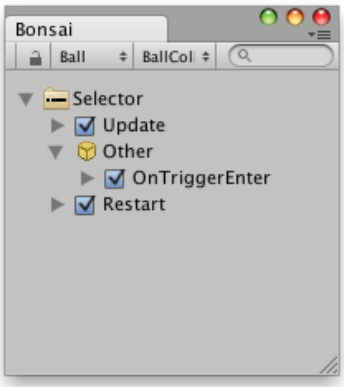


Figura 4.1: Nós *Update*, *OnTriggerEnter* e *Restart*, do tipo *FunctionBehaviour*

4.3.2 *BoxBehaviour* ou Variáveis

As árvores de comportamento não possuem um mecanismo explícito de comunicação entre seus nós, ou seja, isto geralmente é feito através de relações diretas, gerando acoplamento e dificultando o reuso de nós. Uma solução para este problema é o uso de uma abstração do tipo *blackboard* [21]. Entretanto, apesar desta abstração centralizar as variáveis e não gerar acoplamento entre os nós, dificulta o desenvolvimento de um mecanismo para definir o escopo de uma variável.

A Bonsai possui uma classe de nós decoradores do tipo *BoxBehaviour*, também denominados de *box*. Estes nós não possuem código de execução, sua função é armazenar uma determinada informação. Somente os filhos destes nós podem ter acesso (ler ou escrever) aos seus dados. Possibilitando um mecanismo que define um escopo para estes nós.

A Bonsai contém nove tipos básicos de *BoxBehaviour*: *BoolBox*, *FloatBox*, *GameObjectBox*, *IntBox*, *ObjectBox*, *QuaternionBox*, *RectBox*, *StringBox* e *Vector3Box*. Estes tipos básicos armazenam respectivamente os valores: booleanos, ponto flutuante, referências a objetos de jogo, inteiros, objetos (texturas, arquivos de música, etc), *quaternion*,

retângulo (estrutura com os valores de ponto flutuante x, y, largura e altura), *string* e vetor (estrutura com os valores de ponto flutuante x, y e z). A quantidade de tipos básicos é extensível via programação. Para cada um dos nove tipos básicos existem diversos outros nós que executam ações sobre seus valores (somar *IntBox*, destruir objeto de jogo em um nó *GameObjectBox*, etc).

Existem outros tipos de nós que são derivados dos tipos básicos de *BoxBehaviour*, isto é, herdam de um dos noves tipos básicos. Alguns destes nós fazem referência direta a um atributo de um componente. Por exemplo, o nó *PositionBox* armazena a posição (como um vetor) de um objeto de jogo definido por referência. Isto facilita o desenvolvimento, uma vez que qualquer nó que realize uma operação sobre um nó do tipo *Vector3Box* também pode executar esta operação em um *PositionBox*.

4.3.3 *PrefabTask*

É possível armazenar uma determinada sub-árvore como um *asset*, que pode ser utilizado em outras árvores ou projetos. A associação deste arquivo com a árvore é realizada através de um tipo especial de *behaviour* chamado de *PreafabTask*. Este nó possui uma referência para tais arquivos e cria uma cópia do *asset* em tempo de execução. A vantagem deste mecanismo é possibilitar o aproveitamento de partes da lógica de uma árvore em outra.

Os componentes Bonsai de *prefabs*, do motor *Unity 3D*, só armazenam/referenciam árvores no formato de *assets*.

4.4 *Global Bonsai*

O *Global Bonsai* é um componente que estende o componente Bonsai, e sua árvore é formada somente por nós do tipo *BoxBehaviour*. Uma *box* que esteja no *Global Bonsai* possui escopo global, isto é, qualquer nó em qualquer árvore na cena pode acessar suas informações. Este componente fornece um mecanismo para facilitar a comunicação entre árvores distintas.

4.5 Interface

O *plugin* Bonsai possui poucas janelas, todas as interações destas interfaces foram desenvolvidas baseadas nas interações presentes nas janelas do editor da *Unity 3D*. Assim, evita-se que o usuário necessite aprender um novo fluxo de trabalho para utilizar o *plugin*. As principais interfaces da Bonsai são: janela principal, de adição de nós, de preferências, de criação de nós (ou *create behaviour*), e sobre a Bonsai.

4.5.1 Menu Principal

O *plugin* Bonsai adiciona um novo menu à barra de ferramentas da *Unity 3D* (Figura 4.2). Neste menu encontram-se opções para as principais ações e janelas relacionadas do *plugin*. As opções presentes no menu principal são:

- *Bonsai Window*: inicializa a janela principal.

- *Add Behaviour Window*: abre a janela de adicionar nós na árvore.
- *Create PrefabTask*: cria um *asset* correspondente a uma sub-árvore. Esta sub-árvore pode ser utilizada em qualquer árvore de comportamento de um componente Bonsai.
- *Create Bonsai Behaviour*: abre a janela de criação de novos *behaviours*.
- *Create Global Bonsai*: cria um objeto de jogo na cena com um componente *Global Bonsai*.
- *Preferences*: inicializa a janela de preferências do *plugin*.
- *About Bonsai*: mostra a janela de informações sobre o *plugin*.

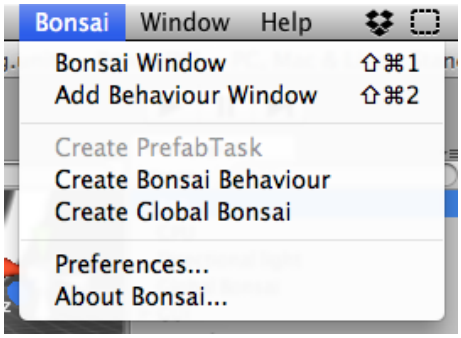


Figura 4.2: Menu principal do *plugin* Bonsai

4.5.2 Janela Bonsai

No *plugin* Bonsai a lógica é construída em forma de árvore, similar a uma árvore de diretórios. Esta forma de visualização possibilita a modularização do comportamentos em sub-comportamentos ou sub-árvores. Isto facilita a visualização e desenvolvimento de comportamentos complexos.

A janela principal, ou Bonsai, (Figura 4.3) exibe a árvore de comportamento de um componente Bonsai. Somente uma árvore é apresentada por vez. No canto superior direito há um campo para filtrar os nós da árvore. Na parte superior central existem dois botões de menu, um para selecionar algum objeto de jogo que contenha um componente Bonsai e outro para selecionar um componente Bonsai do objeto selecionado.

Sempre que um objeto é selecionado na *Hierarchy* ou *Scene View* a janela principal é atualizada com as informações deste objeto. O botão no canto superior esquerdo, em forma de cadeado, pode ser utilizado para focar em uma árvore específica e evitar esta atualização automática.

Cada nó da árvore é representado em uma linha por um ícone seguido de seu nome. Os nós do tipo *FunctionBehaviour* possuem uma caixa de seleção que é utilizada para habilitar/desabilitar a sub-árvore. Os ícones dos ramos (tarefas de combinação ou decoradores) são precedidos por uma seta, que indica se seus filhos estão visíveis ou não. O

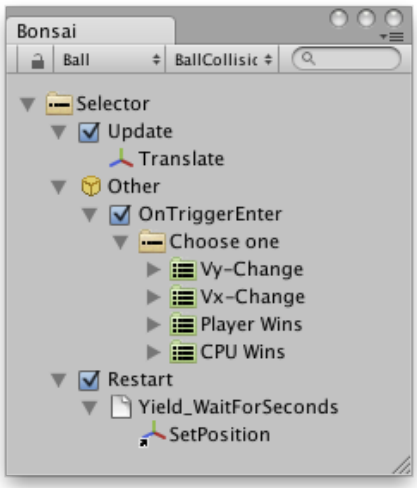


Figura 4.3: Janela principal da ferramenta Bonsai

ícone identifica a classe do nó, os nós do tipo sequência são representados por pastas de cor verde, os nós do tipo seleção por pastas de cor amarela, as *boxes* por uma caixa, etc.

Na sub-árvore *Choose one* (Figura 4.3) existem quatro comportamentos distintos: *Vy-change*, *Vx-change*, *Player Wins* e *CPU Wins*. O nó pai destas quatro sub-árvores é do tipo seleção, e somente uma das sub-árvores será executada. Estas quatro sub-árvores são independentes entre si, assim como qualquer outra sub-árvore na Bonsai que não possua nós em comum.

Durante a execução da árvore de comportamento, na janela principal da Bonsai, é possível identificar o valor de execução de cada nó, este valor é indicado pela cor de um retângulo (Figura 4.4). O retângulo de cor verde indica que o nó foi executado com sucesso, o de cor amarela indica um valor de não execução, ou falso, o retângulo vermelho indica um erro na execução do nó e o de cor roxa indica uma pausa na execução da árvore.

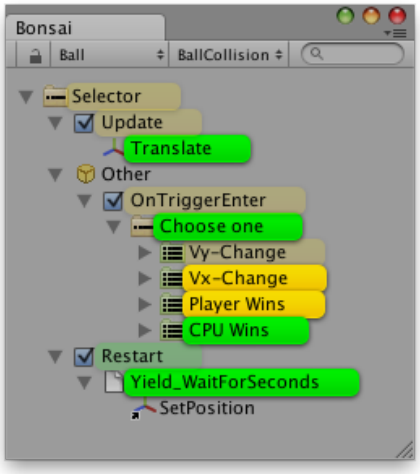


Figura 4.4: Janela principal da ferramenta Bonsai durante a execução.

Na janela principal da Bonsai os nós são dispostos de forma similar aos objetos de jogo da *Hierarchy View* e os *assets* da *Project View*. Estabelece-se relações entre pais e filhos de nós com ações de arrastar e soltar, iguais as ações de mover objetos destas duas *Views* mencionadas.

4.5.3 Janela de Adição de Nós

Utiliza-se a janela de adição de nós (Figura 4.5), também denominada de *Add Bonsai Behaviour*, para inserir novos nós na árvore. Esta janela contém todos os *scripts* de *behaviours* presentes na *Project View*. A parte superior desta janela contém, de cima para baixo: o ícone e nome do nó que será pai (nó selecionado na janela principal), um campo para pesquisa de possíveis nós a serem adicionados e o caminho para o *behaviour* selecionado. A parte central da janela contém os nós que podem ser adicionados.

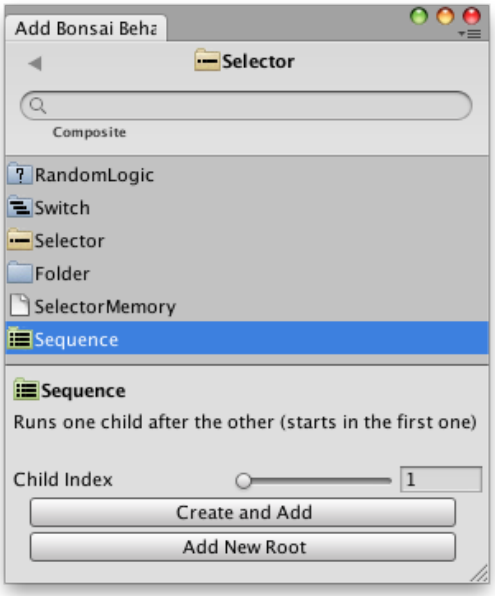


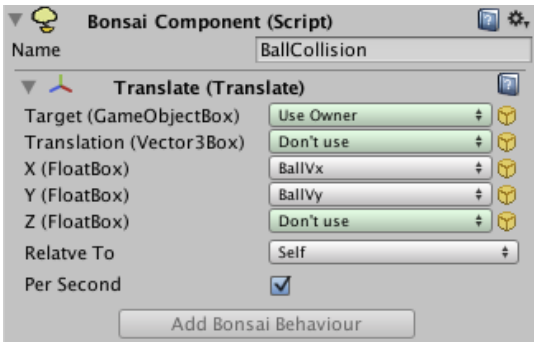
Figura 4.5: Janela para adição de nós na árvore

A região inferior da janela contém informações detalhadas sobre o nó a ser adicionado. Esta região possui o ícone, o nome, uma descrição, o índice desejado para o novo *behaviour* e dois botões que executam a ação de adicionar o nó. O botão "*Create and Add*" insere o novo nó como filho do nó selecionado na janela principal. E o botão "*Add New Root*" adiciona o novo nó como raiz da árvore. Este último botão só está disponível para nós do tipo tarefas de combinação ou decoradores.

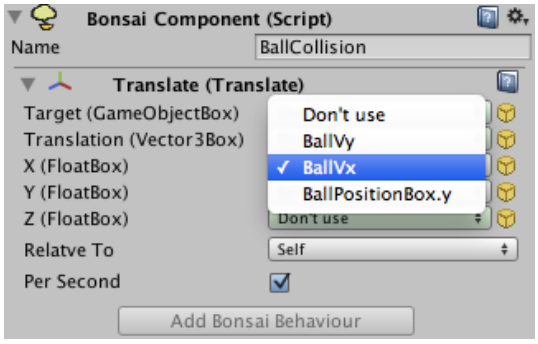
Existem duas outras formas de adicionar novos nós à árvore. Através das opções "*Add New Behaviour*" ou "*Add New Root*" do menu de *mouse* da janela Bonsai. Além disto, arrastando-se o *script*, na *Project View*, correspondente ao *behaviour* desejado para a janela principal, tem-se a ação similar a de adicionar componentes em objetos de jogo na ferramenta *Unity 3D*.

4.5.4 *Inspector View*

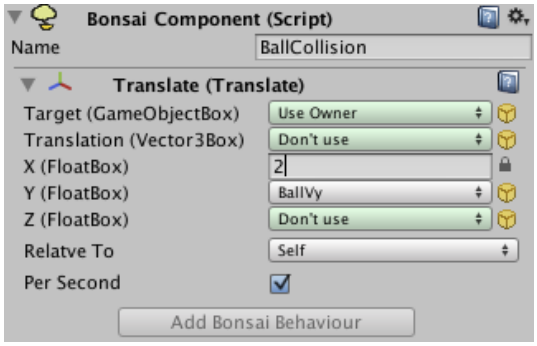
A *Inspector View* é utilizada pela *Unity 3D* para mostrar informações sobre os componentes de um objeto de jogo. O *plugin* Bonsai adiciona nesta janela informações sobre o nó selecionado na janela Bonsai (Figura 4.6). Nesta tela é possível determinar o nome do componente Bonsai e valores para as propriedades dos nós, sendo que estes valores podem ser constantes ou relacionados com uma *box*. Atributos com valores padrão possuem a coloração verde, enquanto que a coloração vermelha indica que o atributo não pode possuir tal valor. O botão "*Add Bonsai Behaviour*", na parte inferior da janela, incializa a janela de adição de nós.



(a) Atributos expostos na *Inspector View*.



(b) Menu de opções de *boxes*, ou variáveis, para o atributo x.



(c) Campo para se definir uma valor constante ao atributo x.

Figura 4.6: A *Inspector View* é utilizada para mostrar detalhes sobre o *behaviour* selecionado

As propriedades que podem ser relacionadas com *boxes* possuem o tipo, entre parênteses, ao lado do nome. Estas propriedades possuem um pequeno ícone, de uma caixa ou cadeado, ao lado direito. O ícone de caixa indica que a propriedade está relacionada com alguma *box*, enquanto o ícone de cadeado indica que a propriedade possui um valor constante. Este ícone pode ser pressionado para determinar se a propriedade deve se relacionar com alguma *box* ou possuir um valor constante. Quando este ícone possui o valor de caixa, é possível selecionar uma *box* (Figura 4.6(b)) com o botão localizado entre o nome da propriedade e o ícone. Este botão de menu é substituído por um campo,

para se definir um valor para o atributo, quando o ícone possui o símbolo de um cadeado (Figura 4.6(c)).

4.5.5 Janela *Create Behaviour*

O *plugin* Bonsai não possui *boxes* para todos os atributos de todos os componentes da *Unity 3D*. Entretanto, possui um mecanismo que auxilia a criação, conforme a necessidade do usuário, de novos *behaviours* deste tipo.

A janela *Create Behaviour* (Figura 4.7) é utilizada para criar *scripts*, via geração de código na linguagem C#, de novos *behaviours*. Atualmente só é possível criar *boxes* que referenciem algum atributo (campo ou propriedade) de um objeto que herde da classe *UnityEngine.Object*. Isto inclui quase todos os objetos relacionados com o motor *Unity 3D*, como: objetos de jogo, componentes, materiais, texturas, etc.

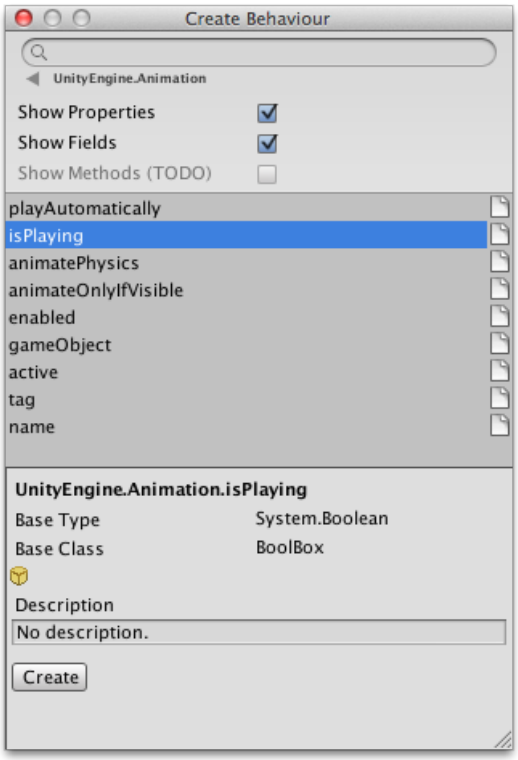


Figura 4.7: Janela *Create Behaviour* com a propriedade *isPlaying* da classe *Animation* selecionada.

A janela *Create Behaviour* contém os atributos das classes presentes nas *DLLs UnityEngine* e *BonsaiRuntime*, e de qualquer outra *DLL* ou *script* que esteja no projeto atual da *Unity 3D*. Na parte superior há um campo para pesquisa de classes, propriedades e *DLLs*. Nesta região também encontram-se opções para selecionar campos e/ou propriedades. A parte central é utilizada para navegar entre as *DLLs* e classes, e para selecionar um atributo. Os detalhes sobre o atributo selecionado são apresentados na parte inferior da janela, onde encontra-se o botão "*Create*", que dispara a ação para gerar o *script* desejado.

4.5.6 Janela de Preferências

A janela de preferências (Figura 4.8) é dividida em duas partes, uma de configurações gerais (Figura 4.9(a)), e outra para modificar a coloração dos *behaviours* na janela bonsai (Figura 4.9(b)). As configurações gerais possuem opções para mostrar a quantidade de filhos do nó, mostrar o índice do nó, além da alternativa para mostrar um ícone que identifica quais objetos do jogo possuem um componente bonsai na *Hierarchy View*.

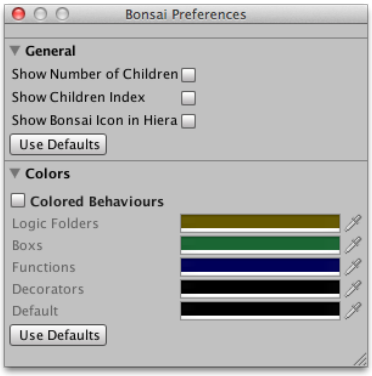
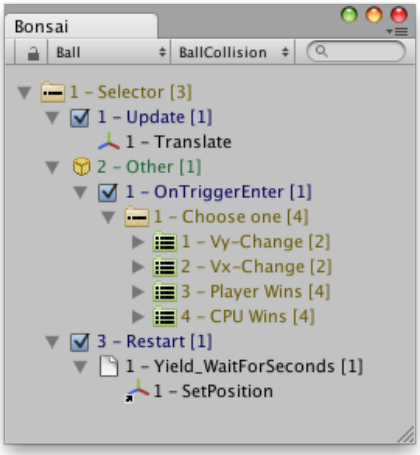


Figura 4.8: Janela de Preferências do *plugin* Bonsai.

(a) *Hierarchy View* com ícones que identificam os objetos com componentes Bonsai.



(b) Janela Bonsai com todas as opções da janela de preferências marcadas.

Figura 4.9: *Hierarcchy View* e Janela Principal modificadas pelas opções de preferências

4.6 Relação com Linguagem de Programação

As árvores de comportamento possuem estruturas que se assemelham, semanticamente, às instruções condicionais de linguagens de programação. O *plugin* Bonsai explora

esta característica, e estreita a relação entre árvores de comportamento e linguagens de programação através de novas estruturas e nós. Nesta Seção são apresentadas estrturas no *plugin* Bonsai com mesma semântica que algumas instruções em linguagens de programação de alto nível, esta comparação visa demonstrar a expressividade que o *plugin* possui.

4.6.1 Condição

Qualquer programa de computador não-trivial requer um mecanismo que possibilite a seleção entre duas ações, P ou Q dependendo de uma condição b [12]. Caso a condição b seja verdadeira apenas a ação P é executada, caso contrário somente Q é executada. A Notação 4.1 é utilizada para representar esta condição.

$$P \triangleleft b \triangleright Q \tag{4.1}$$

A Figura 4.10 representa o operador condicional (*if-else*) em uma árvore de comportamento. A construção do operador é obtida utilizando-se as tarefas de combinação do tipo sequência e seleção, e três sub-árvores P, Q e b.

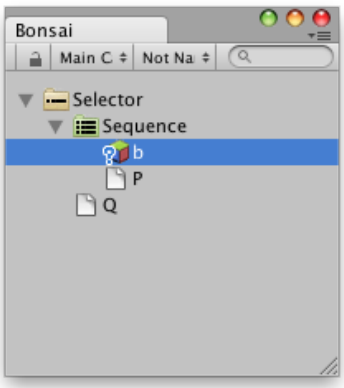


Figura 4.10: Comportamento *if-else*.

A estrutura da Figura 4.10 pode ser utilizada várias vezes para formar estruturas de condição mais complexas como a da Figura 4.11. Nesta imagem tem-se uma estrutura similar à instrução *if-elseif* de linguagens de programação, formada por um nó do tipo seleção na raiz, e um conjunto de nós do tipo sequência como filhos. Os nós do tipo sequência possuem uma condição $(b1, b2 \ldots bn)$ e uma ação $(P1, P2 \ldots Pn)$ como filhos.

4.6.2 Composição

O caractere ";" é comumente utilizado, em linguagens de programação de alto nível, para separar instruções, esta separação define uma composição sequencial de partes do programa [12]. A Notação 4.2 é utilizada para composição sequencial, descrevendo que, na execução do programa, a ação P deve preceder a ação Q.

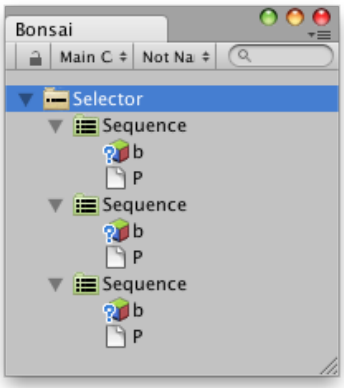


Figura 4.11: Comportamento *if-elseif*.

$$P; Q \tag{4.2}$$

Como apresentado na Seção 3.2.2, as tarefas de combinação são utilizadas para definir a lógica de execução de seus filhos em uma árvore de comportamento. A sequência pode ser utilizada para estabelecer uma composição sequencial de execução, uma vez que seus filhos são executados um após o outro.

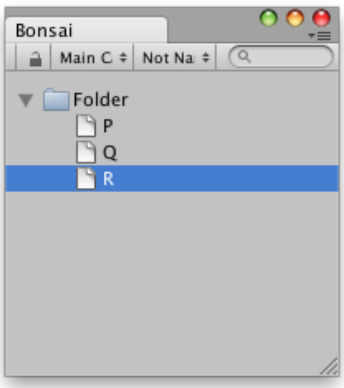
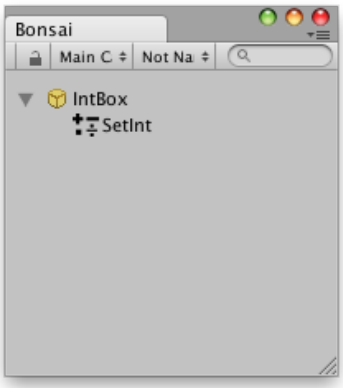


Figura 4.12: Nó *Folder*, composição sequencial de árvores de comportamento

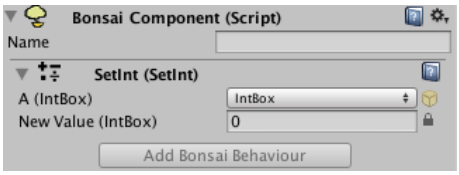
A Figura 3.3 contém um exemplo de um nó do tipo "sequência", que é utilizado para se definir uma ordem de execução sequencial para seus filhos. Entretanto, caso uma das duas condições falhe, o nó "Atacar inimigo" não é executado. Para evitar este comportamento um novo tipo de tarefa de combinação, denoninada *folder*, foi desenvolvida (Figura 4.12). Este nó executa seus filhos em sequência, e sua execução somente é interrompida se um de seus filhos retornar algum valor de erro.

4.6.3 Atribuição

Em linguagens de programação a atribuição corresponde à ação de atribuir um determinado valor a uma variável, cada nó do tipo *BoxBehaviour* possui uma ação para este fim (geralmente esta ação é chamada de *Set*). O nó *SetInt*, Figura 4.13(a), representa o operador atribuição para nós do tipo *IntBox*. A *Inspector View* (Figura 4.13(b)) contém informações sobre o nó *SetInt*, o campo *a* indica qual *box* terá seu valor alterado, o campo *newValue* contém o novo valor da *box*.



(a) Janela Principal com um nó do tipo *SetInt*.



(b) Detalhes do nó *SetInt* na *Inspector View*.

Figura 4.13: Operação de atribuição no *plugin* Bonsai

4.6.4 Não-determinismo

A Notação 4.3 indica que uma das duas ações (P ou Q) será executada mas não qual. Para ter este comportamento na *Bonsai*, cria-se um novo tipo de tarefa de combinação que seleciona seus filhos de forma não-determinística. Por exemplo, o nó *RandomLogic* (Figura 4.14) executa aleatoriamente um de seus filhos. Todos os filhos do nó *RandomLogic* possuem igual probabilidade de serem executados.

$$P \sqcap Q \tag{4.3}$$

4.6.5 Iteração

Um tipo simples de recursão é a iteração [12]. Na Notação 4.4 a ação P será executada enquanto a condição b for verdadeira, comportamento similar à instrução *while* de linguagens de programação.

$$b * P \tag{4.4}$$

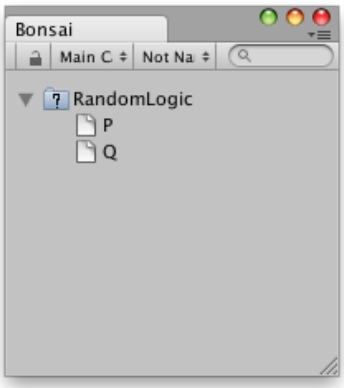


Figura 4.14: Exemplo de não-determinismo no *plugin* Bonsai

A Figura 4.15 representa uma iteração com condição no *plugin* Bonsai. Esta estrutura é construída utilizando uma condição b, uma ação P, um nó do tipo sequência e um decorador do tipo *While*. O decorador *While* executa seu filho enquanto este último retornar o valor verdadeiro. Este comportamento não é comum no uso da Bonsai, pois existem diversos *FunctionBehaviours* que são executados em iterações como o *Update* e o *FixedUpdate*.

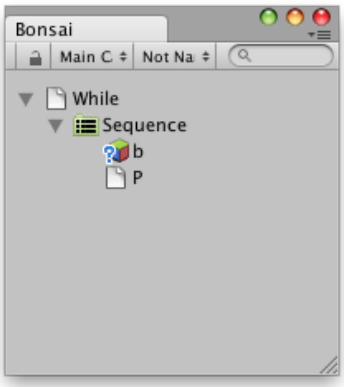


Figura 4.15: Repetição com condição no Bonsai.

4.7 Implementação

A implementação do *plugin Bonsai* está dividida em duas partes principais: o *Editor*, responsável por gerar a interface de configuração das árvores de comportamento, e o *Runtime*, responsável por executar a lógica presente nas árvores de comportamento. Os detalhes de implementação do *Editor* e do *Runtime* estão a seguir.

4.7.1 *Editor*

O *Editor* consiste em uma *DLL* (*Dynamic-link library*) localizada no diretório *Assets/Standard Assets/Editor/Bonsai/BonsaiEditor.dll*. A biblioteca *BonsaiEditor.dll* é utilizada apenas pelo editor do motor *Unity 3D* e não é incluída no executável final da aplicação.

A arquitetura do *Editor* é estruturada em três camadas [26]:

1. Apresentação: classes *MainEditorWindow*, *AboutBonsaWindow*, *AddBehaviourWindow*, *CreateBehaviourWindow*, *PrefWindow*, *VisualDebugging*, *BonsaiComponentInspector* e *BoxDrawer*.
2. Negócio: classes *BonsaiCodeGeneration*, *BonsaiEditorUtility* e *BonsaiSelection*.
3. Persistência: classes *UnityEditor.EditorUtility* e *UnityEditor.AssetDatabase*.

A camada de apresentação é responsável por fornecer uma interface gráfica para o usuário. As classes *MainEditorWindow*, *AboutBonsaWindow*, *AddBehaviourWindow*, *CreateBehaviourWindow* e *PrefWindow* correspondem à implementação das janelas descritas na Seção 4.5. Estas classes implementam a funcionalidade de janelas do editor *Unity 3D* através de sua super classe *UnityEditor.Editor* [30]. *BoxDrawer* é uma subclasse da classe *PropertyDrawer*. Esta classe informa à *Inspector View* como esta deve desenhar os atributos do tipo *BoxBehaviour*. A classe *VisualDebugging* indica qual o status dos nós em execução e é utilizada pela janela *MainEditorWindow* para alterar a cor de fundo do nó durante a execução.

O objetivo da camada de negócios é processar e prover dados para a camada de apresentação. A classe *BonsaiEditorUtility* é a principal classe desta camada e possui apenas métodos estáticos. Quase todas as classes presentes na camada de apresentação utilizam algum serviço desta classe. A classe *BonsaiCodeGeneration* faz parte do módulo de geração de código, e possibilita a criação de novos objetos do tipo *BoxBehaviour*, que referenciam algum atributo de uma classe do tipo *UnityEngine.Object*.

A camada de negócios também utiliza a camada de persitência para armazenar informações não voláteis. As classes *UnityEditor.AssetDatabase* e *UnityEditor.EditorUtility* possuem os métodos necessários para gerenciar objetos persistentes no motor *Unity 3D*. A classe *BonsaiCodeGeneration* é responsável por gerar o código de *BoxBehaviours*, esta classe utiliza a *System.IO* para armazenar os arquivos de *scripts*. Grande parte da camada de persistência é tratada pelo próprio motor através das classes: *MonoBehaviour*, classe base para os *scripts* de componentes, e *ScriptableObject*, classe base de todos os nós (Figura 4.16). A ferramenta *Unity 3D* serializa automaticamente todos os campos públicos de objetos do tipo *ScriptableObject* e *MonoBehaviour*.

4.7.2 *Runtime*

O módulo *Runtime* é responsável por garantir que as árvores de comportamento sejam executadas corretamente. O *plugin* Bonsai fornece uma abstração para o código específico do jogo. Assim, todo o código do módulo de *Runtime* corresponde a parte do código específico do jogo. Este módulo corresponde à *DLL BonsaiRuntime.dll* e todos os arquivos de *scripts* dos nós. Estes últimos estão localizados por padrão no diretório

Assets/Standard Assets/Bonsai/Behaviours, mas podem estar em qualquer pasta dentro do diretório *Assets*, a *DLL* do *runtime* está localizada no diretório *Assets/Standard Assets/Bonsai/BonsaiRuntime.dll*.

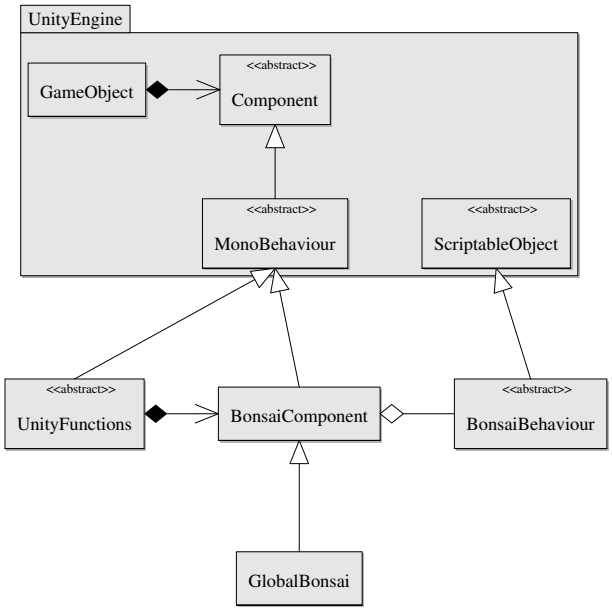


Figura 4.16: Diagrama de classes do módulo de *Runtime* e de parte do motor *Unity 3D*.

A Figura 4.16 representa a arquitetura do módulo *Runtime*. Esta arquitetura é simples e está relacionada com a arquitetura do motor *Unity 3D*, mas pode ser facilmente implementada em outras *engines*, uma vez que possui poucos elementos e estes estão bem definidos.

Ao iniciar o jogo a ferramenta *Unity 3D* procura por qualquer método chamado *Awake* nas classes *MonoBehaviours* [30]. Este método é o ponto de partida da lógica específica do jogo. O componente *BonsaiComponent* implementa este método para construir a hierarquia de objetos do tipo *BehaviourTick*. O *BehaviourTick* é um *delegate*, objeto similares a um ponteiro de função, que não possui parâmetro e retorna um *enum* do tipo *Status*. O *enum* do tipo *Status* é utilizado para indicar qual o status de execução da sub-árvore; este *enum* possui os valores de *Failure*, *Success*, *Error* e *Break*. Por questões de performance, durante a execução os nós fazem chamadas aos objetos *BehaviourTick* ao invés de referenciar um outro nó para invocar um método neste. O motivo reside no fato da hierarquia de classes de nós ser profunda (Figura 4.17) e consequentemente chamadas a métodos virtuais em uma estrutura com esta característica podem impactar no desempenho.

A construção dos *delegates* ocorre no método *OnBuildTick* dos nós (Figura 4.17), que é responsável por construir e retornar um objeto do tipo *BehaviourTick*. Internamente este método procura por métodos com o nome de *Tick* e que possuam a mesma assinatura do *delegate BehaviourTick*. Caso encontrem um método neste formato um objeto do tipo *BehaviourTick* é criado e retornado, caso contrário será retornado um *delegate* para um método que imprime um erro no console.

Logo após a construção da hierarquia de *BehaviourTicks*, o método *OnBuild* é executado em cada nó. A função deste método é possibilitar que o nó possa realizar uma inicialização local antes de alguma execução na árvore. Somente após o evento *OnBuild*, em cada nó, que a árvore pode ser executada.

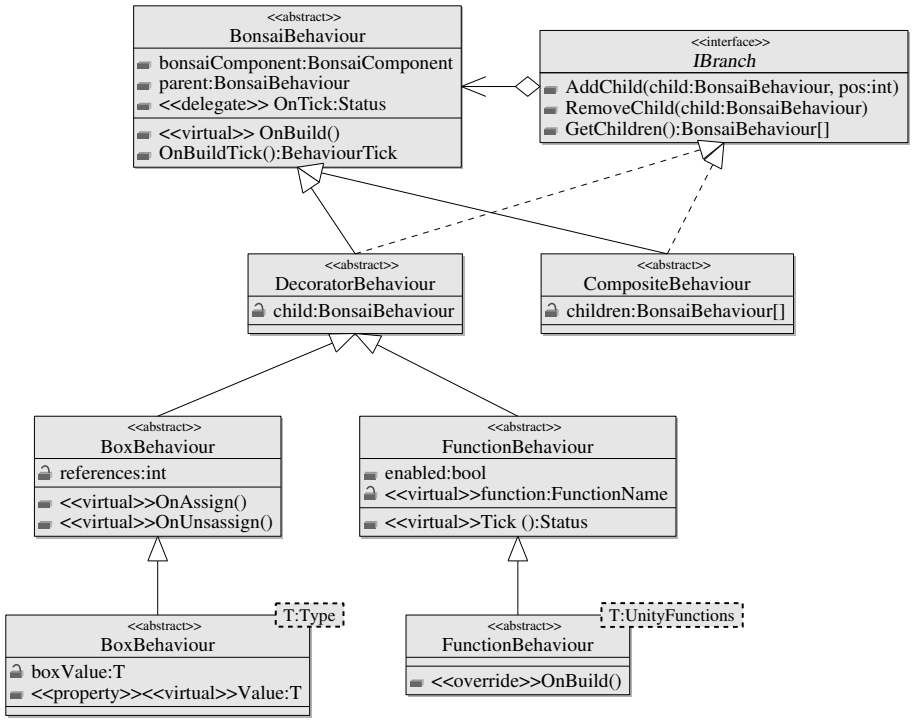


Figura 4.17: Diagrama de classes contendo a hierarquia de nós da ferramenta Bonsai.

O componente *BonsaiComponent* possui os métodos públicos *Tick* para iniciar a execução da árvore (Figura 4.17). Estes métodos são utilizados pelos componentes do tipo *UnityFunctions*. Estes últimos componentes possuem um método que corresponde a um *callback*, da classe *MonoBehaviour* do motor *Unity 3D*, que invoca o método *Tick* no componente Bonsai. Por exemplo, o componente *UpdateComponent* possui o método *Update*, invocado pela *engine* a cada atualização de quadro, que aciona o método *Tick* no componente Bonsai. Os componentes do tipo *UnityFunctions* são transparentes aos usuários, estes componentes são criados e inseridos no objeto pelos nós *templates*, ou genéricos, *FunctionBehaviour* (Figura 4.17).

Classes Concretas dos Nós

A classe *BonsaiBehaviour* corresponde a classe base dos nós. Esta classe e suas subclasses são abstratas, ou seja, não podem ser instanciadas (Figura 4.17). Estas classes são apenas uma estrutura modelo para a criação de classes concretas que serão utilizadas na confecção das árvores de comportamento. A seguir, tem-se alguns exemplos de como extender estas classes e criar novos nós para se construir a lógica específica de um jogo.

O nó *FunctionBehaviour* é uma classe *template*. No código deste nó é necessário indicar com qual componente do tipo *UnityFunctions* o nó está relacionado. Este componente

será criado durante o evento *OnBuild*. O Código Fonte 4.1 contém a implementação do nó *Update*, a declaração desta classe está relacionada com o componente *UpdateComponent*.

```
using FiraSoft.Bonsai;

public class Update : FunctionBehaviour<UpdateComponent> {
  protected override FunctionName function {
    get {return FunctionName.Update;}
  }
}
```

Código Fonte 4.1: Implementação da classe FunctionBehaviour.

A classe *BoxBehaviour* necessita de um tipo em sua declaração, esta classe facilita a criação de nós com a funcionalidade de armazenar um valor. Por exemplo, o Código Fonte 4.2 corresponde a classe *FloatBox* que armazena um valor do tipo ponto flutuante, o tipo *float* é indicado na declaração desta classe.

```
using FiraSoft.Bonsai;

public class FloatBox : BoxBehaviour<float> {}
```

Código Fonte 4.2: Implementação da classe FloatBox.

Os nós do tipo *CompositeBehaviour* podem ter mais de um filho, nós deste tipo geralmente são utilizados para se definir o fluxo de execução da árvore. O Código Fonte 4.3 contém a implementação da classe *Sequence*, esta classe executa uma vez cada um de seus filhos enquanto estes retornarem o valor *Success*. Neste exemplo a classe concreta implementa o método *Tick* que possui mesma assinatura do *delegate BehaviourTick*.

```
using FiraSoft.Bonsai;

public class Sequence : CompositeBehaviour {
  Status Tick() {

    var childStatus = Status.Failure;

    for (int i = 0; i < children.Count; i++) {
      childStatus = children[i].OnTick();
      if (childStatus != Status.Success)
        return childStatus;
    }

    return childStatus;
  }
}
```

Código Fonte 4.3: Implementação da classe Sequence.

Capítulo 5

Aplicações do Bonsai

Este capítulo tem por objetivo apresentar o fluxo de trabalho para criação de alguns comportamentos e lógicas utilizando a ferramenta Bonsai. A Seção 5.1 contém a lógica para se mover um personagem em um ambiente em três dimensões, enquanto a Seção 5.2 possui o comportamento de um alarme que é acionado quando o personagem está em uma determinada região. Por fim, a Seção 5.3 apresenta um comportamento simples, em um jogo eletrônico, de um inimigo que persegue o personagem na cena e o ataca quando está a uma determinada distância.

5.1 Controlador em Terceira Pessoa

O controlador em terceira pessoa consiste em uma lógica para movimentar um personagem, visível pela câmera, em um ambiente de três dimensões. O personagem é movimentado através das setas do teclado, e através da tecla *shift* é possível aumentar sua velocidade. Com a Bonsai esta tarefa pode ser facilmente realizada utilizando-se apenas um componente bonsai e os *assets* padrões da *Unity 3D*.

5.1.1 Construindo a Cena

Primeiramente, é necessário criar os objetos que compõem a cena do controlador em terceira pessoa. Para isto, cria-se um novo projeto (*File* → *New Project*) e adicionam-se três objetos à cena: um plano, uma cápsula e uma luz direcional. Estes objetos encontram-se no menu *Game Object* → *Create Order* da barra de ferramentas com os respectivos nomes de *Plane*,*Capsule* e *Directional Light*. O plano deve estar na origem e a cápsula um pouco acima do plano. Adiciona-se o componente *Character Controller* na cápsula. A Figura 5.1 mostra a cena com os três objetos. Renomeia-se a cápsula para *Player*, este objeto é o personagem a ser movimentado na cena e o plano representa a superfície por onde o personagem pode caminhar.

5.1.2 Árvore de Comportamento

Ao se executar a cena, clicando-se no botão *Play*, nada acontece porque ainda não existe algum tipo de interação. Isto é esperado, uma vez que os objetos de jogo não possuem algum componente que adicione uma lógica ou comportamento aos mesmos.

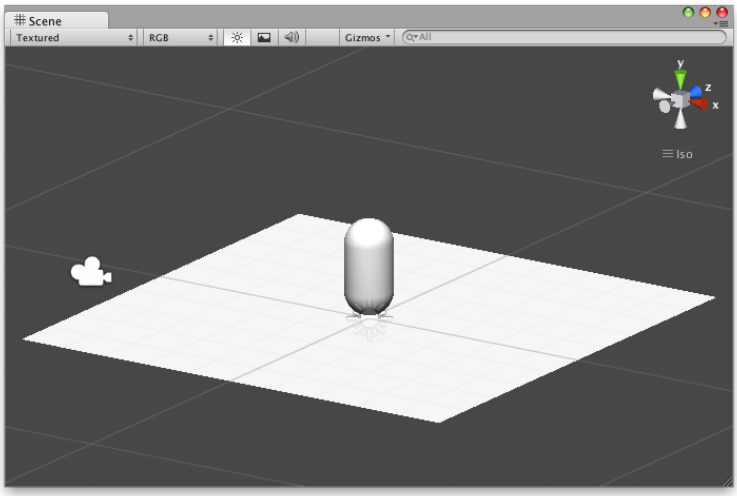


Figura 5.1: Configuração inicial da cena com um plano, uma cápsula, uma câmera e uma luz direcional.

Toda a lógica presente na cena será desenvolvida através do *plugin Bonsai*. Seleciona-se o objeto cápsula e adiciona-se um componente bonsai clicando na janela principal do *plugin*. Por padrão um componente bonsai possui uma árvores com três nós: *Selector*, *Start* e *Update*. Para o controlador em terceira pessoa é necessário apenas o nó *Update*, os outros dois podem ser removidos utilizando-se a tecla *delete* no *Mac OS X* ou a tecla *Backspace* no *Windows*.

O nó *Update* é a raiz da árvore de comportamento. A lógica do controlador em terceira pessoa é construída através de duas subárvores, uma com a função de coletar os dados de entrada e a outra com a de movimentar o objeto. Para executar estas funções em ordem utiliza-se um nó do tipo *Sequence* como filho do nó *Update*.

Dados de Entrada

O passo seguinte é capturar as entradas do teclado, para isto utiliza-se um nó do tipo *GetAxisVector3* e um outro do tipo *Vector3Box*. O nó *GetAxisVector3* interpreta as setas do teclado como um vetor e armazena esta informação no nó *Vector3Box*. Adiciona-se um nó do tipo *Vector3Box*, renomeia-se para *direction*, entre os nós *Update* e *Sequence*. O nó do tipo *GetAxisVector3* deve ser inserido como filho do nó *Sequence*. Em seguida, seleciona-se o nó *GetAxisVector3* na janela principal da Bonsai, na *Inspector View* referencia-se o objeto de jogo *Main Camera* no atributo *Relative To*, no atributo *Store Input* o nó *direction* deve ser selecionado.

Movimentação do Personagem

Através de um nó do tipo *Move* movimenta-se o personagem, insere-se um nó deste tipo como filho do nó *Sequence*. Na *Inspector View* seleciona-se o nó *direction* no atributo *Direction* e atribui-se o valor "2" no atributo *Velocity*.

Ao iniciar o jogo, clicando-se no botão *Play*, é possível mover o personagem na cena. Porém, a direção do personagem não acompanha a direção do movimento, e portanto, é

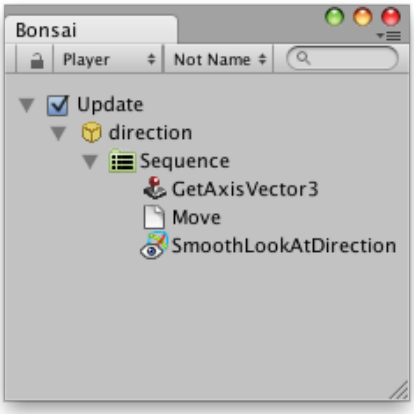


Figura 5.2: Árvore de comportamento de um controlador em terceira pessoa que não altera a velocidade de movimento do personagem.

necessário adicionar um um nó do tipo *SmoothLookAtDirection* como filho do nó *Sequence*. Até o momento a árvore de comportamento deve estar semelhante a da Figura 5.2.

A árvore ainda não possui uma lógica para alterar a velocidade do personagem quando a tecla *shift* está pressionada. Para atribuir este comportamento compõem-se uma estrutura similar ao um *if-else*, ou seja, caso a tecla esteja pressionada a velocidade de movimento será maior, caso contrário será menor. Um nó do tipo *FloatBox* armazena a velocidade de movimento do personagem. Uma estrutura, semelhante à mostrada na Figura 4.10 define um valor para a velocidade (o nó do tipo *FloatBox*).

Em seguida, insere-se um nó do tipo *FloatBox*, com o nome de *velocity*, como filho do nó *direction*. A estrutura de condição dever estar entre os nós *GetAxisVector3* e *Move*. Esta estrutura possui um nó do tipo *Selector* na raiz e um filho do tipo *Sequence*, o primeiro filho da *Sequence* deve ser a condição, neste caso o nó *GetKey* (com o valor *LeftShift* no atributo *Key Code*). O segundo filho da *Sequence* é a ação a ser executada quando a condição é verdadeira e o segundo filho do *Selector* a ação a ser executada quando a condição é falsa. Nestas duas posições deve ser inserido um nó do tipo *SetFloat* e selecionado o nó *velocity* no atributo *Box*. Coloca-se o valor "4" para o atributo *New Value* do nó *SetFloat* filho da *Sequence* e o valor "2" para o mesmo atributo do outro nó.

A Figura 5.3 contém um exemplo da árvore de comportamento de um controlador em terceira pessoa. Ao iniciar o jogo é possível movimentar o personagem pela cena e aumentar sua velocidade de deslocamento ao manter a tecla *shift* pressionada.

Câmera

Para que a câmera acompanhe a posição do personagem necessita-se adicionar um componente bonsai na câmera. A árvore de comportamento deste componente, Figura 5.4(a), consiste em um nó do tipo *Update* na raiz e um nó do tipo *SmoothLookAt* como seu filho. Atribui-se o objeto de jogo *Player* ao campo *Object To Be Looked* e um valor de "180" ao campo *Speed* do nó *SmoothLookAt* (Figura 5.4(b)).

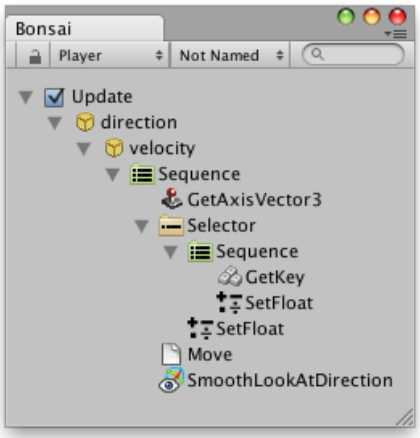
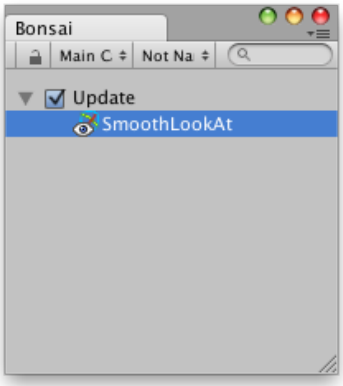
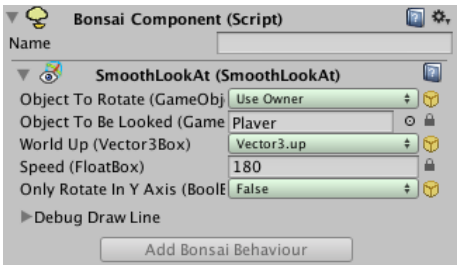


Figura 5.3: Árvore de comportamento de um controlador em terceira pessoa.



(a) Árvore de comportamento da câmera.



(b) Configuração do nó *SmoothLookAt* da árvore de comportamento de movimentação da câmera.

Figura 5.4: Comportamento para rotacionar a câmera em direção à personagem.

5.2 Eventos de Colisão (*Triggers*)

Eventos de colisão, recorrentes em jogos eletrônicos, ocorrem na interação entre um objeto de jogo e outro. Neste exemplo o evento de colisão ocorre entre um alarme e o *Player* quando este último está dentro da área do alarme. Esta área é delimitada por duas plataformas, no evento de colisão uma luz vermelha é acionada, e quando o jogador se afasta desta área esta luz é apagada. Para esta cena é necessário adicionar novos objetos e confeccionar novas árvores.

5.2.1 Construindo a Cena

Cria-se um objeto de jogo vazio (*Game Object* → *Empty Game Object*) na posição $(0, 1, 3)$ e com o nome de *Alarm*. Adiciona-se um componente *Box Collider* ao objeto

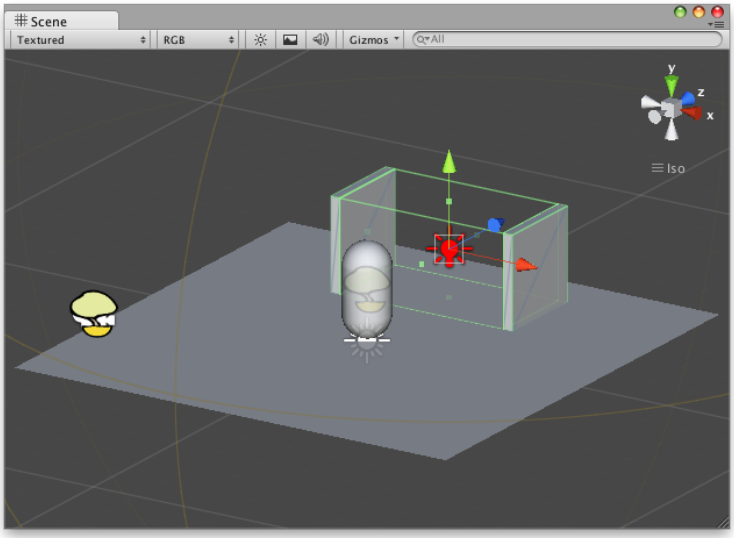


Figura 5.5: Cena composta por um plano, a personagem uma luz, uma câmera e um alarme.

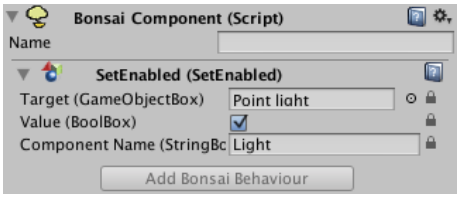
Alarm e modifica-se o atributo *Size* para $(3.8, 2, 2)$. . Além disso, marca-se a opção *Is Trigger* do colisor. Este colisor representa a área do alarme, e caso o jogador esteja dentro desta área o alarme irá disparar. Para as laterais do alarme cria-se um cubo (*Game Object* $\rightarrow$ *Create Order* $\rightarrow$ *Cube*) como filho do objeto *Alarm* na posição $(2, 0, 0)$ e com escala $(0.2, 2, 2)$. Duplica-se este cubo (botão direito do *mouse* e *Duplicate*) com posição em $(-0.2, 2, 2)$. Agora, cria-se uma objeto de jogo do tipo *Point Light* (*Game Object* $\rightarrow$ *Create Order* $\rightarrow$ *Point Light*) filho do *Alarm* nas coordenadas $(0, 0, 0)$. Desativa-se o componente *Light*, desmarcando a caixa de seleção ao lado do nome do componente, muda-se o atributo *Color* para uma cor vermelha e o valor do atributo *Intensity* para "5". Seleciona-se o objeto *Directional Light* e muda-se o valor do atributo *Intensity*, do componente *Light*, para "0.2". A Figura 5.5 apresenta a configuração desta cena.

5.2.2 Árvore do Comportamento dos Eventos de Colisão

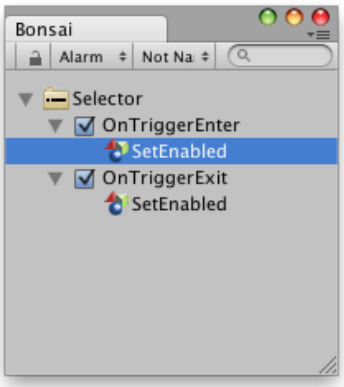
Para o comportamento de alarme deve-se ativar a luz vermelha quando o personagem estiver entre as plataformas. Este lógica necessita de uma árvore de comportamento para os eventos de quando o personagem entra e sai da área do alarme. Adiciona-se um componente bonsai no objeto *Alarm*, remove-se os nós *Update* e *Start* e insere-se um nó do tipo *OnTriggerEnter* e outro do tipo *OnTriggerExit* como filhos da raiz (*Selector*). Este dois últimos nós correspondem aos eventos de início e fim de colisão.

Altera-se o estado do alarme ativando ou desativando o componente *Light* presente na hierarquia do objeto *Alarm*. Insere-se um nó do tipo *SetEnabled*, como filho do nó *OnTriggerEnter*, para esta função. No nó *SetEnabled* seleciona-se uma constante (clicando no ícone de caixa ao lado esquerdo) para o atributo *Target* como valor do objeto *Point Light*. Também seleciona-se uma constante para o atributo *Value* e marca-se a caixa de seleção, e no atributo *Component Name* escreve-se *Light* (Figura 5.6(a)). Este é o comportamento para habilitar o componente *Light* do objeto *Point Light* quando algo

entrar na região de colisão. Copie o último nó criado, cole como filho do nó *OnTriggerEnter* e desmarque a caixa de seleção do atributo *Value*. A Figura 5.6(b) mostra a árvore de comportamento do alarme.



(a) Nó, filho do *behaviour OnTriggerEnter*, que ativa o componente *Light*.



(b) Árvore de comportamento do objeto *Alarm*.

Figura 5.6: Comportamento para ativar e desativar o objeto alarme.

5.3 Controlador de Inimigo

Em diversos jogos eletrônicos o comportamento do inimigo possui um papel fundamental na jogabilidade. Nesta seção é construído um comportamento simples para um inimigo, que consiste em seguir o jogador e atacá-lo a uma certa distância. Este comportamento consiste em seguir o jogador e ataca-lo a uma certa distância. O inimigo se move em direção ao personagem e para o movimento a uma certa distância para atacar. O comportamento de atacar consiste em simplesmente disparar uma mensagem no console. Esta ação pode ser posteriormente substituída por uma ação de invocar um evento em uma árvore de comportamento do personagem.

5.3.1 Construindo a Cena

Para esta cena remove-se o objeto *Alarm*, uma vez que este não será necessário. Seleciona-se o objeto *Directional Light* e modifica-se o valor do atributo *Intensity*, do componente *Light*, para "0.5". Em seguida, deve ser Adicionado um cube (*Game Object* $\rightarrow$ *Create Order* $\rightarrow$ *Cube*) à cena, na posição $(0.2, 0.5, 3)$, e renomeado para "*Enemy*" .

5.3.2 Árvore de Comportamento do Inimigo

Para construir o comportamento do inimigo adiciona-se um componente bonsai no objeto *Enemy*. Remove-se os nós *Start* e *Selector* e coloca-se um nó do tipo *Sequence* como filho do nó *Update*, que é utilizado para executar as funções de "olhar" e "mover" em direção ao personagem nesta sequência. Para que o inimigo fique na mesma direção que o

personagem cria-se um nó do tipo *SmoothLookAt* como filho do nó *Sequence* e seleciona-se o objeto *Player* no atributo *Target Object* deste nó. Por enquanto o objeto *Enemy* apenas "olha" em direção ao *Player*.

É necessário criar uma condição para parar o movimento e iniciar a mensagem de ataque. Para isto, usa-se novamente uma estrutura de condição semelhante à presente na Figura 4.10. Adiciona-se um nó do tipo *Selector* como filho do nó *Sequence* e cria-se um novo nó do tipo *Sequence*, este último, filho do nó *Selector*. A condição e a mensagem serão filhos do nó *Sequence* recém-criado, o comportamento de mover será filho do nó *Selector* (Figura 5.7). Assim, se a condição for verdadeira a mensagem de atacar é apresentada, caso contrário o inimigo se move.

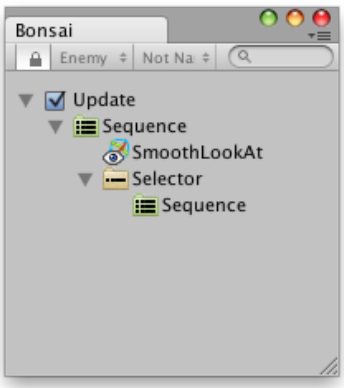
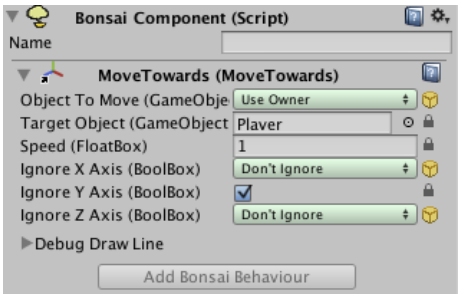


Figura 5.7: Árvore de comportamento com o comportamento do inimigo com parte da estrutura condicional.

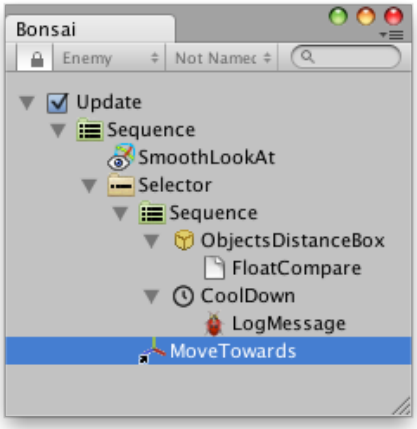
A condição para mover ou atacar consiste em comparar a distância entre os objetos *Enemy* e *Player*, ou seja, caso esta distância seja menor que um valor o inimigo ataca, caso contrário ele se move. Constrói-se esta condição com dois nós: um do tipo *ObjectsDistanceBox*, que armazena a distância entre os objetos, e um outro do tipo *FloatCompare* que compara o valor do nó *ObjectsDistanceBox* com uma constante. Cria-se o nó do tipo *ObjectsDistanceBox* como filho do último nó *Sequence*, os atributos *A* e *B*, do nó *ObjectsDistanceBox*, correspondem aos objetos *Player* e *Enemy*. Adiciona-se o nó *FloatComprare* como filho do nó *ObjectsDistanceBox*, para o nó *FloatComprare* seleciona-se os valores *ObjectsDistanceBox*, *LessOrEqual* e o valor constante "1, 5" para os atributos *A*, *Compare Method* e *B* respectivamente.

O comportamento de atacar é formado por uma hierarquia de dois nós, o nó *LogMessage* imprime a mensagem na janela de console e seu nó pai, *CoolDown*, garantindo que a mensagem seja mostrada em um intervalo constante de tempo. Coloca-se o nó *CoolDown* filho do último nó *Sequence* e marca-se a opção *Always Return Success*. Insere-se o nó *LogMessage* como filho do nó *CoolDown* com o valor de "*Attack*" no atributo *Message*.

Para seguir o personagem utiliza-se um nó do tipo *MoveTowards* com o objeto *Player* referenciado pelo atributo *Target Object* e um valor constante de "1" para o atributo *Speed*, marca-se a opção *Ignore Y Axis* (Figura 5.8(a)). Insere-se este nó como filho do nó *Selector*. A árvore de comportamento do objeto *Enemy* está na Figura 5.8(b).



(a) Configuração dos atributos do nó *MoveTowards*.



(b) Árvore de comportamento do objeto *Enemy*.

Figura 5.8: Comportamento de seguir o objeto *Player* e mostrar uma mensagem de ataque.

Capítulo 6

Conclusão

Devido a dificuldade inerente ao desenvolvimento de jogos eletrônicos, de início, esta prática ficou restrita a equipes e/ou pessoas experientes. Os motores de jogos surgiram para auxiliar e acelerar o processo de desenvolvimento de *games*. Recentemente, a *engine Unity 3D* tem-se tornado uma popular ferramenta entre os desenvolvedores independentes e os estúdios de desenvolvimento de jogos eletrônicos. Porém, esta ferramenta não possui um mecanismo nativo para programação visual da lógica específica do jogo. Observou-se que as soluções existentes para este problema são *plugins* que se baseiam em ferramentas de outras *engines*. Por este motivo, estas soluções acabam se distanciando do fluxo de trabalho da ferramenta *Unity 3D*.

Este trabalho consistiu no desenvolvimento do Bonsai, um *plugin* para o motor de jogos *Unity 3D* que possibilita o desenvolvimento da lógica específica sem necessidade de codificação de programas. A Bonsai fornece uma abstração visual para linguagens de programação em forma de árvore, seu fluxo de trabalho é semelhante às demais tarefas presentes na *engine*. O *plugin* é extensível, isto é, possibilita a criação de novos nós via programação. Além disto, a Bonsai fornece um gerador de código para nós do tipo *BoxBehaviour*, que podem ser utilizados para referenciar atributos de outros objetos.

Com a ferramenta Bonsai é possível desenvolver diversos comportamentos frequentemente utilizados no desenvolvimento de jogos eletrônicos. O *plugin* também permite visualizar a execução da árvore e os estados dos nós durante o funcionamento do jogo. Também é possível alterar a lógica, e qualquer parâmetro dos nós, enquanto o jogo está em execução. Todas estas ações são realizadas em uma interface visual, isto permite uma melhor depuração e facilita a prototipação de jogos por pessoas que não possuem conhecimentos de programação, como *game designers*, por exemplo.

6.1 Trabalhos Futuros

A seguir estão algumas sugestões de trabalhos futuros e melhorias para o *plugin* Bonsai:

- Construir jogos completos utilizando somente o *plugin* para a confecção da lógica específica do jogo.
- Criar novos nós que possuam a funcionalidade de tarefas comumente utilizadas em jogos eletrônicos. Isto é, nós mais abrangentes que englobem a tarefa de mais de um nó.

- Possibilitar que o módulo de geração de código também crie nós que invoquem métodos em outros objetos.
- Desenvolver estudos de usabilidade da ferramenta com usuários finais.
- Possibilitar que a ferramenta gere um *script* contendo todo o código da árvore. Esta funcionalidade possui dois principais benefícios: o primeiro é uma melhoria na performance durante a execução da árvore e o outro é que possibilitaria que o jogo final pudesse usufruir dos comportamentos criados, com os ajustes necessários no código gerado, durante a fase de prototipação.

Referências

[1] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh McLoughlin. The case for research in game engine architecture. In *Conference on Future Play: Research, Play, Share*, Future Play '08, pages 228–231, New York, NY, USA, 2008. ACM.

[2] Antares. Antares Universe. http://antares-universe.com/, 2011. Acessado em dezembro/2011.

[3] I.S. Brasil, F.M.M. Neto, J.F.S. Chagas, R.M. de Lima, D.F.L. Souza, M.F. Bonates, and A. Dantas. An inteligent agent-based virtual game for oil drilling operators training. In *Virtual Reality (SVR), 2011 XIII Symposium on*, pages 9 –17, may 2011.

[4] Alex J. Champandard. Behavior Trees for Next-Gen Game AI. http://aigamedev.com/insider/presentations/behavior-trees/, 2008. Acessado em maio/2012.

[5] Igor Augusto de Faria Costa. Gameka: Uma ferramenta de desenvolvimento de jogos para não programadores. In *Simpósio Brasileiro de Games e Entretenimento Digital*, pages 1–4, Brasil, 2011. Sociedade Brasileira de Computação (SBC).

[6] Crytek. CryEngine 3. http://www.crytek.com/cryengine, 2011. Acessado em dezembro/2011.

[7] B.N. Di Stefano and A.T. Lawniczak. Cognitive agents: Functionality; performance requirements and a proposed software architecture. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*, pages 509 – 514, sept. 2009.

[8] Hutong Games. Playmaker. http://hutonggames.com/playmaker.html, 2011. Acessado em dezembro/2011.

[9] Yoyo Games. Game maker 9. http://www.yoyogames.com/gamemaker/, 2011. Acessado em dezembro/2011.

[10] F.W.P. Heckel, G.M. Youngblood, and N.S. Ketkar. Representational complexity of reactive agents. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 257 –264, aug. 2010.

[11] Chris Hecker. My Liner Notes for Spore. http://chrishecker.com/My_Liner_Notes_for_Spore, 2011. Acessado em abril/2012.

[12] C.A.R. Hoare and He Jifeng. *Unifying theories of programming*. Prentice Hall, London New York, 1998.

[13] B. Iske and U. Ruckert. A methodology for behaviour design of autonomous systems. In *Intelligent Robots and Systems, 2001 IEEE/RSJ International Conference on*, volume 1, pages 539 –544 vol.1, 2001.

[14] D. Isla. Handling Complexity in the Halo 2 AI. http://naimadgames.com/publications.html, 2005. Acessado em abril/2012.

[15] V. Janarthanan. Serious video games: Games for education and health. In *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, pages 875 –878, april 2012.

[16] Younbo Jung, Koay Jing Li, Ng Sihui Janissa, Wong Li Chieh Gladys, and Kwan Min Lee. Games for a better life: effects of playing wii games on the well-being of seniors in a long-term care facility. In *Sixth Australasian Conference on Interactive Entertainment*, pages 1–6, New York, NY, USA, 2009. ACM.

[17] Blazej Kot, Burkhard Wuensche, John Grundy, and John Hosking. Information visualisation utilising 3d computer game engines case study: a source code comprehension tool. In *6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: making CHI natural*, CHINZ '05, pages 53–60, New York, NY, USA, 2005. ACM.

[18] Paul Lilly. Doom to Dunia: A Visual History of 3D Game Engines. http://www.maximumpc.com/article/features/doom_dunia_visual_history_3d_game_engines, 2009. Acessado em fevereiro/2013.

[19] Game Developer's Magazine. Game Developer's mobile and social technology survey. http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity, 2012. Acessado em janeiro/2013.

[20] Microsft. Kodu Game Lab. http://fuse.microsoft.com/page/kodu, 2011. Acessado em dezembro/2011.

[21] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[22] K.J. Nyitray. William alfred higinbotham: Scientist, activist, and computer game pioneer. *Annals of the History of Computing, IEEE*, 33(2):96 –101, feb. 2011.

[23] Steve Rabin. *Introduction To Game Development (Game Development)*. Charles River Media, Inc., Rockland, MA, USA, 2005.

[24] J.C.K.H. Riedel and J.B. Hauge. State of the art of serious games for business and industry. In *Concurrent Enterprising (ICE), 2011 17th International Conference on*, pages 1 –8, june 2011.

[25] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[26] I. Sommerville. *Engenharia de software*. Pearson Addison-Wesley, 2008.

[27] Jochen Strube, Sven Schade, Patrick Schmidt, and Peter Buxmann. Simulating indirect network effects in the video game market. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, page 160b, jan. 2007.

[28] Detox Studios. uScript. http://www.detoxstudios.com/products/uscript/, 2011. Acessado em dezembro/2011.

[29] Click Team. Multimedia fusion 2: Ultimate game creation and powerful applications. http://www.clickteam.com/eng/mmf2.php, 2011. Acessado em dezembro/2011.

[30] Unity Technologies. Unity 3D. http://unity3d.com/, 2011. Acessado em dezembro/2011.

[31] Unreal. Unreal Engine 3. http://www.unrealengine.com/, 2011. Acessado em dezembro/2011.