Tutorial: Desenvolvimento de Jogos com Unity 3D

Erick Baptista Passos
epassos@ic.uff.br

José Ricardo da Silva Jr.
josericardo.jr@gmail.com

Fernando Emiliano Cardoso Ribeiro
one.fernando@gmail.com

Pedro Thiago Mourão
pedrothiago@hotmail.com

Apresentação

Esse tutorial é apresentado como uma breve introdução ao desenvolvimento de jogos com o motor Unity3D. Explicações muito detalhadas, mesmo que cobrindo apenas parte das funcionalidades existentes, ocupariam bem mais que as três dezenas de páginas disponíveis para este tutorial. Dessa forma, recomendamos o leitor interessado a buscar mais informações na abrangente e excelente documentação da ferramenta, disponível tanto localmente, ao se realizar a sua instalação, quanto online através do site http://www.unity3d.com/support.

Também por questões de espaço, não foi incluída uma introdução geral ao desenvolvimento de jogos. Espera-se que o leitor possua alguma familiaridade com os conceitos e ferramentas relacionados ao assunto tais como renderização em tempo-real, modelagem e animação 3D, texturas e *Shaders*. Também espera-se que o leitor possua noções de programação, preferencialmente com alguma linguagem orientada a objetos.

O conteúdo do texto está organizado em forma crescente de complexidade, de forma a facilitar a leitura. Entretanto, as seções apresentadas também podem ser lidas de forma independente.

A imagem da capa foi cedida do projeto França Antártica, de cuja equipe de desenvolvimento fazem parte alguns dos autores. O jogo França Antártica está sendo desenvolvido através da Unity3D e é um projeto financiado pela Secretaria de Cultura do Estado do Rio de Janeiro.

Por fim, informa-se que algumas das figuras e exemplos apresentados foram adaptados do manual da ferramenta, de tutoriais online (http://www.unity3d.com/resources), bem como de discussões disponíveis no forum comunitário de desenvolvedores Unity (http://forum.unity3d.com). Também recomendamos o leitor que explore essas referências para um maior aprofundamento.

Guia de leitura

1 - Introdução

Breve apresentação dos módulos da Unity3D e detalhamento da interface do editor de cenas.

2 - Criação e Manipulação de Game Objects

Explicação do modelo de objetos da Unity3D e as formas básicas de criação, composição e alteração dos mesmos.

3 - Materiais e Shaders

Breve introdução à linguagem de especificação de *Shaders* da Unity3D: *ShaderLab*. São demonstrados exemplos para *pipeline* de função fixa, bem como *Shaders* programáveis.

4 - Sistema de Física

Introdução aos componentes relacionados ao subsistema PhysX de simulação física.

5 - Scripting

Apresentação, através de exemplos, dos principais conceitos para a programação de scripts com a Unity3D.

6 - Conclusão

Considerações finais dos autores.

Bibliografia

Leituras recomendadas para aqueles interessados em um maior aprofundamento.

1 - Introdução

O desenvolvimento de jogos 3D é uma atividade ao mesmo tempo gratificante e desafiadora. Diversas habilidades, de diferentes áreas do conhecimento, são necessárias nesse processo. O uso de ferramentas para auxiliar nas tarefas repetitivas é fundamental nesse tipo de atividade, e ao longo do tempo, um tipo especial de ferramenta, conhecido como motor de jogos (*game engine*) foi evoluindo de maneira paralela aos próprios jogos, ao ponto que se tornaram produtos valiosos e de certa forma populares.

Alguns módulos e funcionalidades auxiliares são condições necessárias para que uma ferramenta seja considerada um motor de jogos completo. Em especial, um sistema de renderização 3D com suporte a *Shaders* programáveis e um sistema de simulação física são fundamentais. Uma boa arquitetura para a programação de *scripts*, um editor de cenas integrado, e a capacidade de se importar diretamente modelos 3d, imagens e efeitos de áudio produzidos em ferramentas externas, são as características existentes nos motores de jogos. Além disso, é desejável que os jogos desenvolvidos possam ser distribuídos em múltiplas plataformas como PC, consoles ou mesmo dispositivos móveis.

A Unity3D abstrai do desenvolvedor de jogos a necessidade de utilizar diretamente *DirectX* ou *OpenGL* (apesar de ainda ser possível, caso necessário), suportando a criação de *Shaders* complexos com a linguagem *Cg* da *NVidia*. Internamente, o subsistema de simulação física é o popular *PhysX*, também da *NVidia*. Para a execução de scripts, a Unity usa uma versão de alto desempenho da biblioteca *Mono*, uma implementação de código aberto do framework .Net da Microsoft.

Ainda que seja uma ferramenta que inclui o estado da arte no seu segmento, a Unity3D tem um preço acessível, o que é apenas mais uma das razões para sua crescente popularidade. Mesmo usando a versão mais barata da Unity3d, os jogos podem ser desenvolvidos para PC, Mac ou mesmo embutidos em uma página *Web*. Com a aquisição de licenças específicas, pode-se desenvolver e distribuir jogos para *iPhone*, através da loja online da *Apple*, ou mesmo para o console *Wii* da *Nintendo*.

1.1 - Interface

O motor de jogos Unity3D possui uma interface bastante simples e amigável que objetiva facilitar o desenvolvimento de jogos de diversos gêneros e outros sistemas de visualização. Sua área de trabalho é composta de várias janelas chamadas **views**, cada uma com um propósito específico. A figura abaixo é uma captura contendo uma representação esquemática e a identificação de cada uma dessas janelas no editor de cenas da Unity3D.

Figure 1 - Interface do editor de cena da Unity3D (UNITY TECHNOLOGIES 2009A)

Project view

A janela *Project* é a interface para manipulação e organização dos vários arquivos (*Assets*) que compõem um projeto tais como *scripts*, modelos, texturas, efeitos de áudio e *Prefabs*, os quais serão detalhados mais adiante na seção de *scripting*. A estrutura exibida na janela *Project* é correspondente à sub-pasta Assets dentro da pasta do projeto no sistema de arquivos do computador. Recomenda-se que a manipulação de sua estrutura e conteúdo seja efetuada somente dentro da Unity3D, a fim de manter a integridade dos metadados que são associados a estes elementos. Entretanto, certas mudanças, como atualização de uma textura por um editor de imagens por exemplo, ou mesmo a adição de novos *Assets,* pode ser feita de forma segura diretamente no sistema de arquivos.

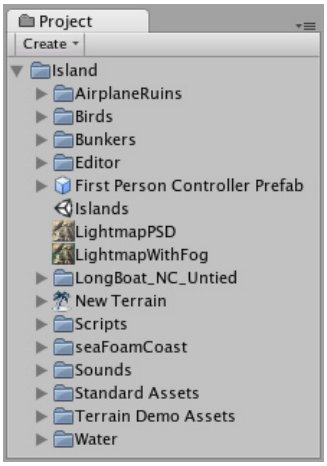


Figure 2 - Janela Project

Hierarchy view

A janela *Hierarchy* exibe todos os elementos da cena que encontram-se na cena que se está editando. Além disso, nessa janela podemos organizar e visualizar a hierarquia de de composição entre os vários objetos que compõem a cena (grafo de cena). O funcionamento desses objetos, bem como a hierarquia de transformação será explicado mais detalhadamente na próxima seção.

Figure 3 - Janela Hierarchy com dois objetos

Scene view

A janela *Scene* é a forma principal de manipulação dos elementos visuais no editor de cenas da Unity, possibilitando a orientação e posicionamento desses elementos com um *feedback* imediato do efeito das alterações efetuadas. Nesta janela, pode-se manipular graficamente os objetos através das opções de arrastar e soltar com o mouse. Essa manipulação é semelhante àquela de ferramentas de modelagem 3D e pode-se manipular objetos tais como câmeras, cenários, personagens e todos os elementos que compõem a cena.

Devido a sua grande importância durante o desenvolvimento de uma aplicação, várias formas de navegação são oferecidas a fim de aumentar ainda mais a produtividade do desenvolvedor. Além disso, as ferramentas básicas de manipulação dos elementos da cena, tais como *pan*, translação, rotação e escala também estão disponíveis para utilização nesta janela através de atalhos de teclado (teclas Q, W, E e R).

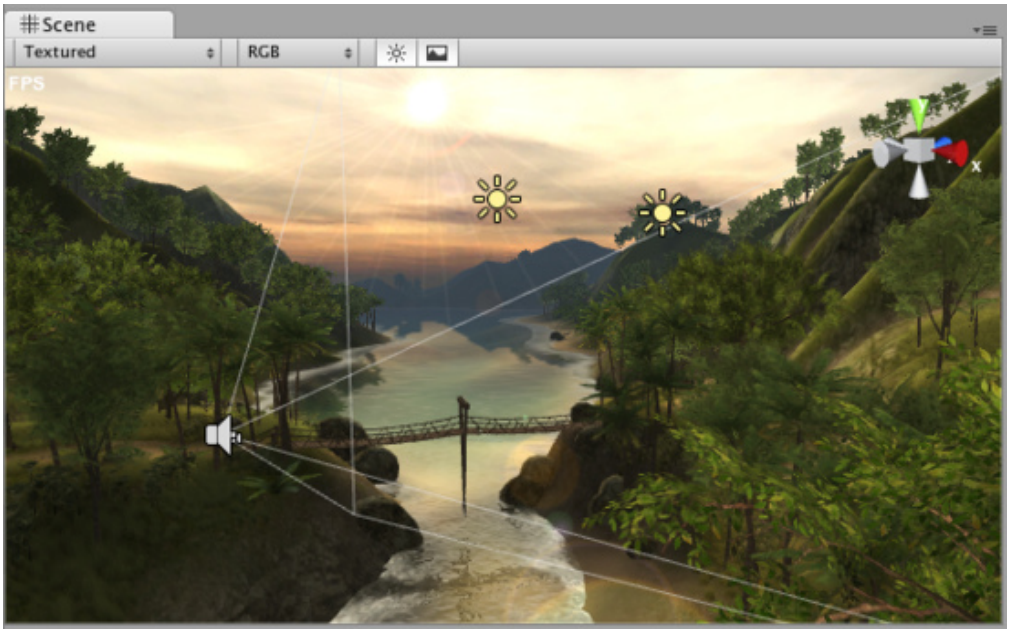


Figure 4 - Janela Scene

Game view

A janela *Game* é responsável pela visualização da aplicação em desenvolvimento da forma que ela será exibida quando finalizada. Nessa janela, pode-se rapidamente ter uma prévia de como os elementos estão se comportando dentro da aplicação. Além disso, a Unity fornece a opção de se paralisar (botão *pause*) a simulação enquanto ela estiver em depuração, de forma a possibilitar que os parâmetros dos vários elementos possam ser ajustados para experimentação. Lembramos que o ajuste desses parâmetros não necessitam que a simulação esteja paralisada, podendo ser alterados inclusive enquanto a simulação esteja em execução.

Nesta janela, também pode-se visualizar várias informações estatísticas (*stats*) sobre a simulação, tais como tempo de processamento e número de frames por segundo, número de triângulos e vértices renderizados, memória de textura utilizada, entre outras. Esta opção é importante para a depuração do desempenho da simulação para uma posterior otimização, caso seja necessário.

Inspector view

Na janela *Inspector,* tem-se acesso aos vários parâmetros de um objeto presente no cenário, bem como aos atributos de seus componentes (*Components*). Essa estrutura utilizada pela Unity para a composição de objetos será melhor explicada na próxima seção. Ainda na janela *Inspector*, pode-se ajustar os atributos públicos (parâmetros) de cada componente, inclusive durante a execução da aplicação.

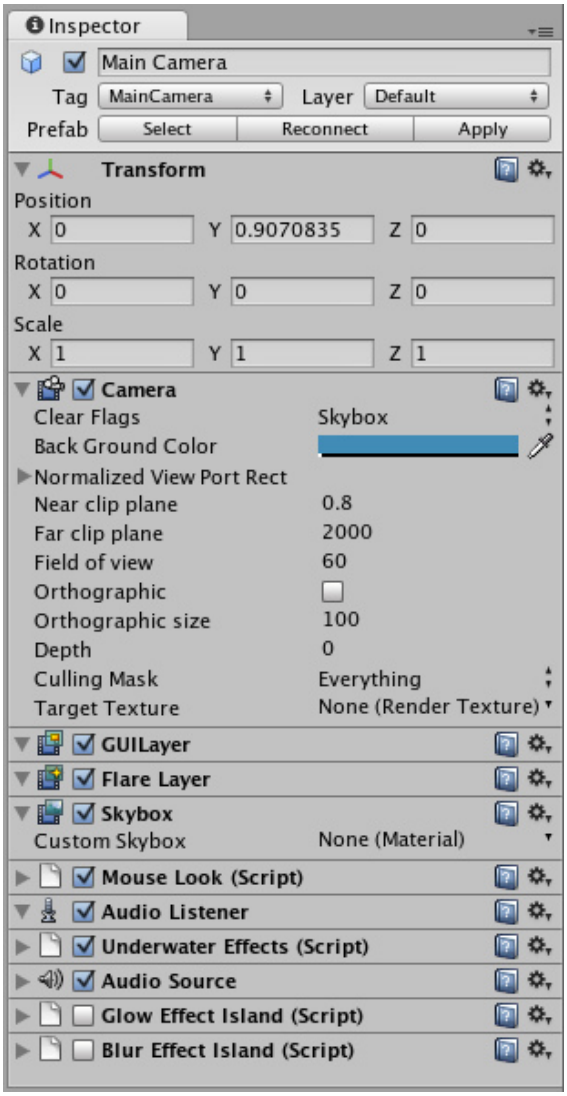


Figure 5 - Janela Inspector

2 - Criação e Manipulação de Game Objects

Muitos motores de jogos de geração anterior disponíveis no mercado são baseados no uso de especialização/herança para as classes que irão representar objetos de jogo. Nesta arquitetura, os elementos de um jogo herdam de uma classe básica (normalmente algo como *GameObject*) e as novas funcionalidades são acrescentadas. Assim como em outras categorias de desenvolvimento de software orientado a objetos, o uso demasiado de herança nesse tipo de situação tornou-se obsoleto, principalmente por sua pouca flexibilidade quando os objetos de jogo possuem múltiplos comportamentos, o que exigiria herança múltipla.

A Unity3D é baseada em um modelo mais moderno para a arquitetura de objetos de jogo baseado em composição [Bilas 2002, Stoy 2006, Passos et al. 2008]. Nesse modelo, um objeto de jogo é especificado através da composição de várias funcionalidades, que são agregadas (ou removidas). Cada funcionalidade é implementada por um componente (classe que herda de um componente básico). Esse container genérico ainda é denominado *Game Object* e funciona como um repositório de funcionalidades, ou mais especificamente, componentes.

Os componentes são então responsáveis por implementar os diversos comportamentos que um *Game Object* pode ter. Um componente pode ser desde um *script*, uma geometria de colisão, ou até uma textura de GUI. Ou seja, *Game Objects* podem representar qualquer coisa no cenário, sendo caracterizados como uma simples câmera ou um personagem apenas pelos diferentes componentes que agrega. Conforme observado no manual de usuário da Unity (tradução livre): "*Game Object é uma panela vazia e os componentes são os ingredientes que irão criar sua receita de jogabilidade*".

Abaixo apresentamos a estrutura de um *Game Object* padrão que representa uma câmera virtual. Nessa figura, a câmera é um *Game Object*, porém, só é definida como câmera pois possui um componente com essa funcionalidade. Além disso, possui os componentes auxiliares *GUILayer, Flare Layer* e *Audio Listener*.

Figure 6 - Composição de um Game Object (Camera)

Uma observação importante sobre os *Game Objects* é que todos eles já possuem pelo menos o componente *Transform*, responsável pelo seu posicionamento, orientação e escala no sistema referencial da cena. Além disso, esse componente é responsável pela definição da hierarquia de transformações, permitindo o efeito de transformação relativa de acordo com a estrutura de ascendência/descendência de cada *Game Object*.

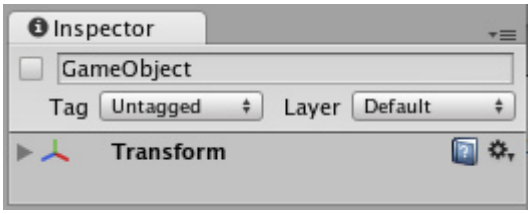


Figure 7 - Estrutura mínima de um objeto

Na Unity também temos o conceito de *Prefab*. Um *Prefab* é simplesmente um modelo de composição de *Game Object* já definido, ou mais precisamente, um *template* que define um elemento através da composição dos vários componentes. Podemos citar, por exemplo, a definição de um humanóide que necessita de um script de movimentação e um componente de colisão. Nesse caso, poderíamos criar um *Prefab* desse humanóide e criar várias cópias do mesmo, inclusive com parâmetros diferentes. Dessa forma, temos um ganho considerável de tempo pois isso evitaria que tivéssemos que recriar essa composição para cada instancia de humanóide presente no cenário. *Prefabs* serão mais detalhados na seção sobre *scripting*.

Importação de Assets

Geralmente, grande parte do desenvolvimento de um jogo está relacionado com a utilização e manuseio de *Assets* tais como texturas, modelos 3D, efeitos de som e *scripts*. Esses diferentes tipos de artefatos são desenvolvidos em ferramentas externas, especializados na construção de cada um dos tipos de *Assets*. Após sua criação ou edição, *Assets* precisam, de alguma forma, serem importados para dentro de editor de cenas do motor de jogos.

A Unity possui uma forma muito simples e robusta de importação de *Assets* para dentro de um projeto, bastando que os mesmos sejam "arrastados" para dentro de uma pasta da janela *Project*. Ao efetuar este procedimento, a importação é feita automaticamente para o projeto, sem nenhuma intervenção do usuário, ficando imediatamente disponível para ser utilizada dentro da aplicação. A Unity aceita formatos de distribuição populares para modelos 3D (.FBX), áudio (wav, mp3, etc) e texturas (jpg, png, bmp ou mesmo .PSD diretamente).

Além dessa simplicidade de importação dos *Assets*, a Unity também oferece a possibilidade da visualização em tempo real de qualquer alteração feita nos mesmos. Com isso, tem-se um ganho de produtividade, pois não precisa-se importar manualmente novas versões para dentro da Unity a cada vez que desejarmos efetuar uma alteração. A Unity verifica cada arquivo modificado e automaticamente atualiza o mesmo na cena.

3 - Materiais e Shaders

Jogos 3D normalmente possuem um grande apelo visual, onde a criatividade e capacidade dos artistas realiza um papel fundamental para o sucesso. Atualmente, os dispositivos gráficos permitem o uso de soluções sofisticadas para a exibição em tempo real das malhas 3D dos elementos de uma cena. Em especial, faz-se onipresente o uso de *Shaders* programáveis.

O motor de jogos Unity3D permite tanto a criação de *Shaders* em linguagens de programação como Cg ou GLSL, quanto o uso de função fixa, além de incluir uma versátil coleção desses

Shaders na instalação padrão. O vinculo entre um *Shader* e uma malha 3D se faz através de um *Material*, que funciona como um container para as propriedades visuais que cada objeto da cena possui. A figura a seguir mostra como a atribuição dessas propriedades pode ser feita na janela *Inspector* dentro do editor de cenas.

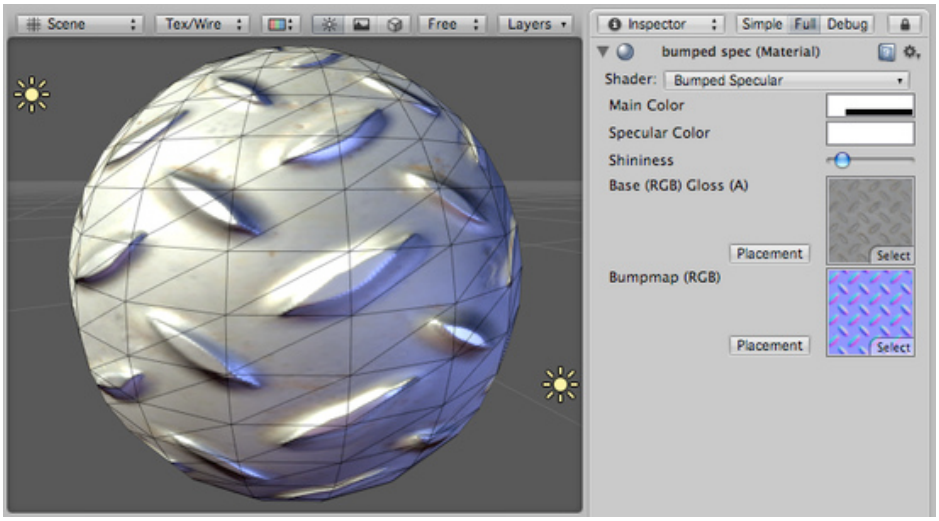


Figure 8 - Manipulação de um Material

A especificação de um *Shader* define quais são as propriedades que este necessita para realizar a exibição do objeto, enquanto o *Material* define os valores para essas propriedades. A próxima figura mostra uma representação esquemática dessa relação. Dois *Shaders* são usados: uma para o corpo do carro e um para as rodas. Para o corpo do carro, dois materiais são criados utilizando-se do mesmo *Shader*. Em um deles, o valor da propriedade *Color FX*, especificada pelo *Shader*, é atribuída com a cor vermelha, enquanto no outro é usada a cor azul. Dessa forma, pode-se aplicar esses diferentes materiais a objetos na cena, como ilustrado na figura.

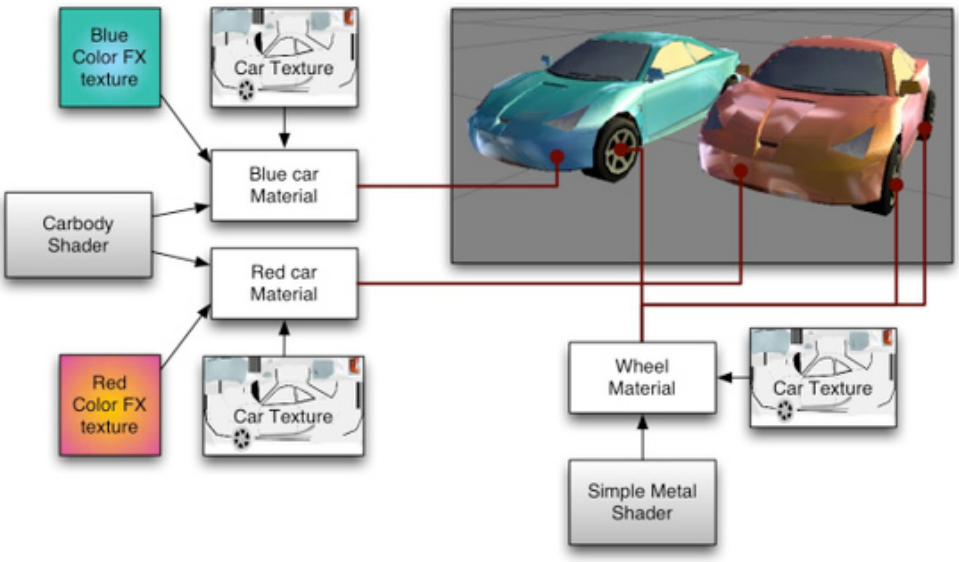


Figure 9 - Relaçnao entre shaders, materiais e objetos

De forma mais específica, um Shader define:

* O método de renderização de um objeto. Isso inclui o uso de diversas variações, dependendo do dispositivo gráfico do usuário;

* Todos os *Vertex* e *Pixel-Shaders* usados nessa renderização;

* As propriedades que serão atribuíveis nas especificações de materiais;

* Parâmetros numéricos ou de cor que também serão atribuíveis nos materiais;

Um *Material* define:

* Quais texturas usar para a renderização;

* Quais cores usar;

* Os valores de quaisquer outros *Assets* tais como *Cubmaps*, limiares de luminância, etc.

Para se criar um novo *Material*, usa-se *Assets->Create->Material* no menu principal. Uma vez criado o *Material*, pode-se aplicá-lo a um objeto e experimentar-se alterar os valores de suas propriedades. Para aplicar um *Material* a um objeto, basta de arrastar o mesmo da janela *Project* para qualquer objeto na cena ou na janela *Hierarchy.*

Para a especificação de *Shaders*, tanto de função fixa quanto programáveis, a Unity3D possui uma linguagem própria chamada *ShaderLab,* que possui alguma semelhança com os arquivos *.FX* definidos para o *Microsoft DirectX* ou a especificação *NVidia CgFX*. Essa linguagem declarativa possui uma sintaxe simples e inclui capacidades poderosas como reuso, múltiplos passos, criação procedural de texturas, entre outras. Nas próximas seções será apresentada uma breve introdução a essa linguagem.

3.1 - Introdução a ShaderLab

Para se criar um novo *Shader*, pode-se escolher *Assets->Create->Shader* do menu principal, ou duplicar um existente e trabalhar a partir do mesmo. *Shaders* na Unity3D são arquivos de texto e podem ser editados fazendo-se um duplo-clique nos mesmos na janela *Project*. Iniciaremos a explicação com um *Shader* bastante simples:

```
Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1.0,0.5,0.5,1.0)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

Esse *Shader* simples demonstra uma das formas mais simples de se renderizar um objeto da cena. É definida apenas uma propriedade de cor (*_Color*), cujo valor padrão é especificado através de seus componentes RGBA. Para renderização, existe uma única opção de *SubShader* com um passo apenas, que atribui o componente difuso na *pipeline* de função fixa com o valor especificado para a propriedade *_Color* e ligando a opção de iluminação por vértices.

O Shader a seguir define uma renderização mais completa, mas ainda baseada em iluminação por vértices:

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0.5)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On
            SeparateSpecular On
            SetTexture [_MainTex] {
                constantColor [_Color]
                Combine texture * primary DOUBLE, texture * constant
            }
        }
    }
}
```

Todo *Shader* deve iniciar com a palavra reservada *Shader* seguido de uma *string* que representa seu nome. Este será o nome exibido na janela *Inspector*. Todo o código do *Shader* ficará definido dentro de um bloco delimitado por {}.

* É interessante que o nome seja curto e descritivo, não sendo necessário corresponder com o nome do arquivo *.shader*;

* Para utilizar-se submenus, basta que se use o caractere /, por exemplo: um nome "*MeusShaders/Teste*" iria ser organizado em um submenu no *Inspector* de um Material como *MeusShaders -> Teste*. Um *Shader* é sempre composto de um bloco de propriedades seguido por um ou mais blocos de *SubShaders*, que serão explicados a seguir.

3.1.1 - Propriedades

As propriedades definidas no bloco inicial de um Shader serão aquelas que podem ser atribuídas através da janela Inspector para os materiais. O exemplo anterior apareceria da seguinte forma no editor de cena:

Figure 10 - Propriedades de um Shader (UNITY TECHNOLOGIES 2009A)

A estrutura de uma declaração de uma propriedade é mostrada na figura que segue. O primeiro termo designa o nome interno (referência à variável), enquanto os valores entre parênteses especificam a descrição (para a janela *Inspector*) e o tipo, respectivamente. O ultimo termo especifica o valor padrão para a propriedade. Encorajamos o leitor a buscar a referência completa sobre tipos de propriedades possíveis no manual de usuário da ferramenta.

Figure 11 - Declaração de uma propriedade (UNITY TECHNOLOGIES 2009A)

3.1.2 - SubShaders

Dispositivos gráficos diferentes possuem capacidades diferentes. Por exemplo, a maioria das placas gráficas atuais dão suporte a Pixel Shaders, mas muitas placas embarcadas em placas mãe baratas não. Algumas suportam 4 texturas em um único passo, enquanto outras suportam apenas 2, e assim por diante. Para permitir o funcionamento do jogo para qualquer que seja o dispositivo do usuário, um *Shader* pode conter múltiplos *SubShaders*. Para renderizar um objeto, a Unity3D passa por todos os *SubShaders* e usa o primeiro que seja completamente suportado pelo *hardware* em uso. O código a seguir ilustra a estrutura básica de um *Shader* desse tipo:

```
Shader "Structure Example" {
    Properties { /* ...shader properties... }
    SubShader {
        // ...subshader com Vertex e Pixel/fragment shaders...
    }
    SubShader {
        // ...subshader que usa 4 texturas por passo...
    }
    SubShader {
        // ...subshader que usa 2 texturas por passo...
    }
    SubShader {
        // ...subshader "feio" mas que roda em qualquer
              hardware... :)
    }
}
```

Cada *SubShader* é composto por uma coleção de passos. Para cada passo, a geometria do objeto é renderizada, portanto ao menos um passo é necessário. O exemplo *VertxLit* possui apenas um passo:

```
//...
Pass {
    Material {
        Diffuse [_Color]
        Ambient [_Color]
        Shininess [_Shininess]
        Specular [_SpecColor]
        Emission [_Emission]
    }
    Lighting On
    SeparateSpecular On
    SetTexture [_MainTex] {
        constantColor [_Color]
        Combine texture * primary DOUBLE, texture * constant
    }
}
// ...
```

Todos os comandos em um passo configuram o *hardware* gráfico para renderizar a geometria de alguma maneira específica. No exemplo acima, o bloco *Material* vincula as propriedades definidas pelo *Shader* com os parâmetros de material do sistema de iluminação em função fixa. O comando *Lighting On* liga a funcionalidade de iluminação por vértices, enquanto *SeparateSpecular On* define o uso de uma cor separada para o parâmetro de reflectância especular.

Todos esses comandos são mapeados diretamente ao modelo de *pipeline* de função fixa OpenGL/Direct3D dos dispositivos gráficos. O comando *SetTexture* é bastante importante, e especifica a forma como as propriedades mapeadas para texturas são aplicadas em conjunto

com o modelo de iluminação da *pipeline*. Esse comando é seguido por um bloco contendo a fórmula que define a equação de combinação das texturas para cada pixel/fragmento renderizado. No exemplo em questão:

```
Combine texture * primary DOUBLE, texture * constant
```

Nesse comando, o termo *texture* é referente à cor obtida pelo mapeamento da textura (nesse caso *_MainTex*). Esta cor é multiplicada (*) pela cor primária (*primary*) do vértice, computada pela equação de iluminação da pipeline fixa e posteriormente interpolada para cada pixel. Este valor é duplicado (*DOUBLE*) para intensificar a iluminação. O valor de transparência (*alpha*) é opcionalmente especificado após a vírgula, onde é computado pela multiplicação do valor *alpha* da textura com a cor constante definida para a *pipeline* (*constantColor*). Diferentes modos de combinação de textura podem ser especificados para a obtenção dos resultados desejados.

3.2 - Shaderlab Programável

A Unity permite ao desenvolvedor o uso de Shaders programados na linguagem Cg da NVidia ou em assembly. Também é possível a criação de Shaders com GLSL caso o jogo seja disponibilizado apenas para Mac OSX, uma vez que o módulo de renderização para Windows usa DirectX, que não suporta tal linguagem.

Os *Shaders* programáveis são incluídos diretamente em uma especificação *ShaderLab*, substituindo o papel de um passo de renderização em um *SubShader*. Essa integração é bastante interessante por dois motivos: simplifica a passagem de parâmetros de materiais para esses *Shaders*; permite a utilização mista de função fixa com *Shaders* programáveis em um mesmo objeto (com o uso de múltiplo passos de renderização). O código a seguir exemplifica como um passo de renderização pode ser especificado através de um código em Cg.

```
Pass {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag

      // Código para Vertex e Pixel Shaders (funções vert e frag)

      ENDCG
}
```

O exemplo a seguir, uma especificação *ShaderLab* completa, usa a informação da normal dos vértices para compor a cor exibida. O marcador *#pragma vertex vert* indica a função que será executada como o *Vertex Shader* para os objetos que usam desse Shader para renderização. De forma similar, pode-se definir o *Pixel Shader*.

```
Shader "Debug/Normals" {
SubShader {
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert

// parâmetros de entrada para o Vertex Shader
struct appdata {
    float4 vertex;
    float3 normal;
};

struct v2f {
    float4 pos : POSITION;
    float4 color : COLOR;
};
v2f vert (appdata v) {
    v2f o;
    o.pos = mul( glstate.matrix.mvp, v.vertex );
    o.color.xyz = v.normal * 0.5 + 0.5;
    o.color.w = 1.0;
    return o;
}
ENDCG
    }
}
}
```

Observa-se que nesse exemplo, não foi especificada uma função para o *Pixel Shader*, que nesse caso será uma versão padrão que usa interpolação para exibir as cores computadas por esse *Vextex Shader*. A figura a seguir mostra o resultado desse código aplicado a um objeto.

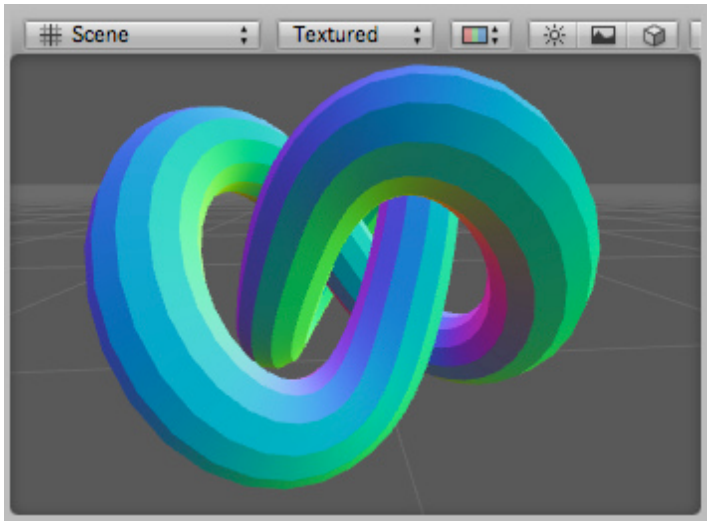


Figure 12 - Shader que exibe normais

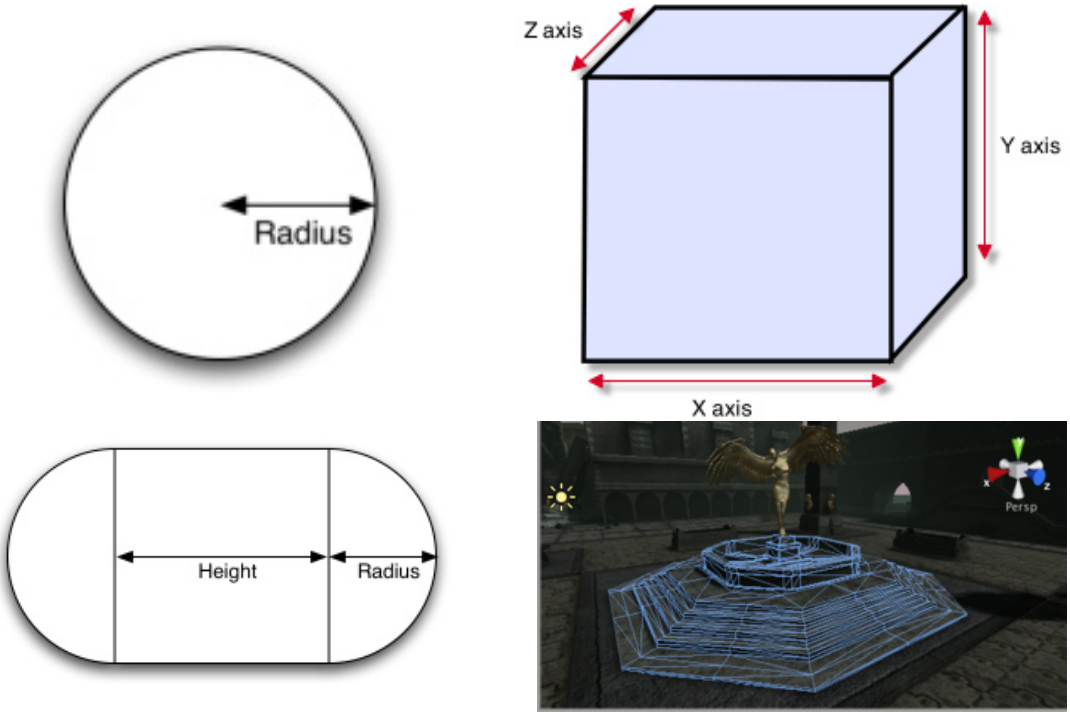
Da mesma forma que os Shader de função fixa, é possível se passar parâmetros de material que são especificados no editor de cena, usar texturas, múltiplos canais de mapeamento UV, ou qualquer outra opção que seja interessante para se atingir os resultados desejados.

4 - Sistema de Física

A Unity3D utiliza internamente o popular motor de física *PhysX* da *NVidia* para efetuar a simulação física de corpos rígidos e o tratamento de colisões. A *PhysX* é um motor de física utilizado em vários jogos populares tais como *Mass Effect, Medal of Honor: Airbone*, entre outros, sendo considerado um dos mais completos do mercado, inclusive com a possibilidade de ser executado em GPUs, o que pode acarretar em maior desempenho. Com essa integração, o desenvolvedor tem acesso simplificado a um sistema de simulação física sofisticado. Várias das funcionalidades oferecidas pela *PhysX* são manipuladas graficamente através da interface da Unity, permitindo que simulações físicas complexas sejam desenvolvidas em pouco tempo, aumentando a produtividade do desenvolvedor.

Colliders

Geometrias básicas de colisão tais como esfera, cubo, cápsula, ou precisas como um Mesh Collider, são implementados como componentes para objetos de jogo na Unity. Esses componentes podem ser anexados a um objeto da cena, que passará a fazer parte da simulação física. Os parâmetros de cada geometria de colisão estão disponíveis para alteração pelo editor de cena.



Dessa forma, pode-se tratar a simulação física de vários objetos através da utilização de uma geometria envolvente ou, caso necessário, até mesmo da geometria real do objeto, geralmente sendo utilizado em cenários estáticos, que necessitam de fidelidade de representação. Além disso, devido ao fato do cenário ser estático, várias otimizações são efetuadas a fim de garantir um bom desempenho da simulação.

Além de sua função principal na simulação física, estes componentes também podem ser utilizados como *triggers*, ou seja, elementos que ativam o processamento de um trecho de

código caso ocorra uma colisão com estes. Componentes definidos como *triggers* não são simulados como componentes de física normais durante a simulação.

Character Controller

Adicionalmente ao que já foi apresentado, a Unity também oferece acesso a um tipo especial de objeto disponível na *PhysX*: o *Character Controller*. Geralmente, o controle preciso de objetos que sofrem ação da física é bastante complicado de ser efetuado durante uma simulação. Um exemplo desse tipo de precisão seria o controle de um personagem. Como sabemos, esse tipo de simulação física é bastante complexo, pois além do tratamento das forças, deve-se tratar também o as rotações indesejadas.

Uma solução seria simplesmente ignorar a geometria de colisão do personagem, porém adotando essa solução não teríamos a interação do mesmo com os objetos do cenário. Utilizando o componente *Character Controller*, tem-se a possibilidade de controlar esse tipo de objeto facilmente, evitando todas as operações indesejadas ditas anteriormente, mantendo a interação com os objetos do cenário. Abaixo tem-se um exemplo da configuração desse objeto em um personagem.

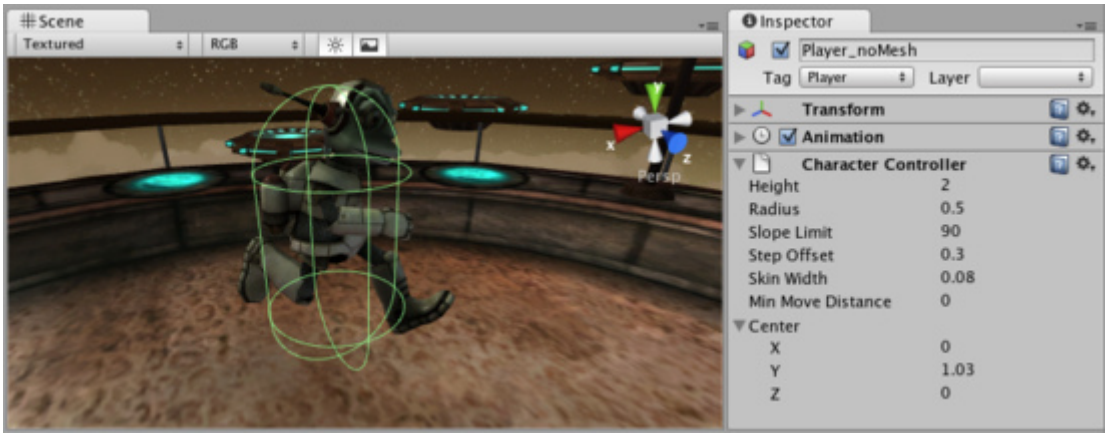


Figure 13 - Componente para personagens

Ragdolls

Além do *Character Controller*, tem-se também a possibilidade de simular *ragdolls* com o uso do componente *Character Joint*. Este componente permite que simulações de personagens inanimados (mortos?) sejam realizadas mais fielmente, tal como um “boneco de pano”. Utilizando um *Wizard*, pode-se especificar onde encontram-se os pivôs de rotação, além de vários outros parâmetros responsáveis pela simulação desse tipo de objeto. Abaixo temos uma imagem da configuração desse tipo de simulação.

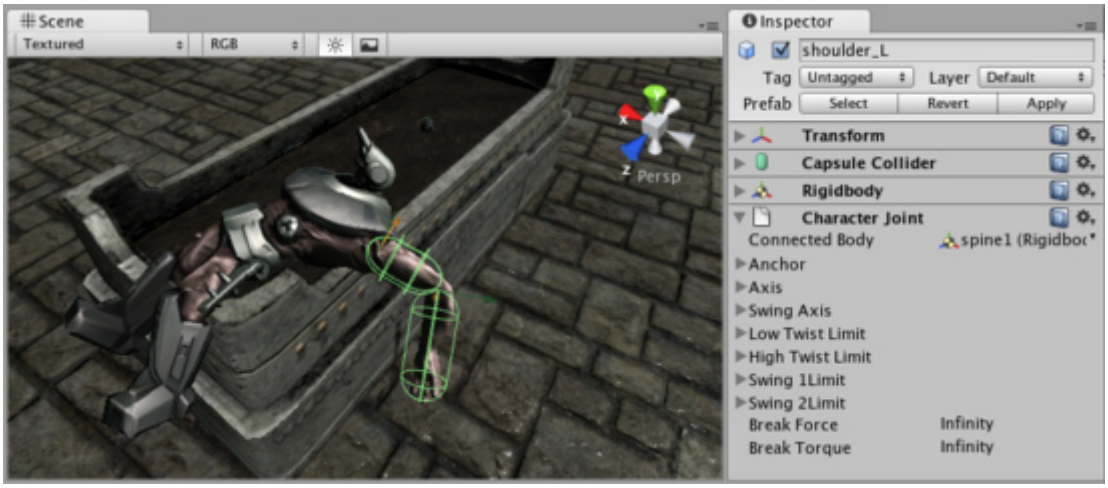


Figure 14 - Sistema de ragdoll

Wheel Colliders

Um outro recurso bastante útil oferecido pela *PhysX* através da Unity é um componente especial para simulação de veículos chamado *Wheel Collider*. Com este componente, pode-se simular força de tração, fricção, entre outras características que ocorrem normalmente em rodas de veículos. Neste caso, o movimento do carro é efetuado através da aplicação de forças nesses componentes. A utilização deste componente permite simular desde forças de fricção até o comportamento de amortecedores, conforme pode ser observado abaixo.

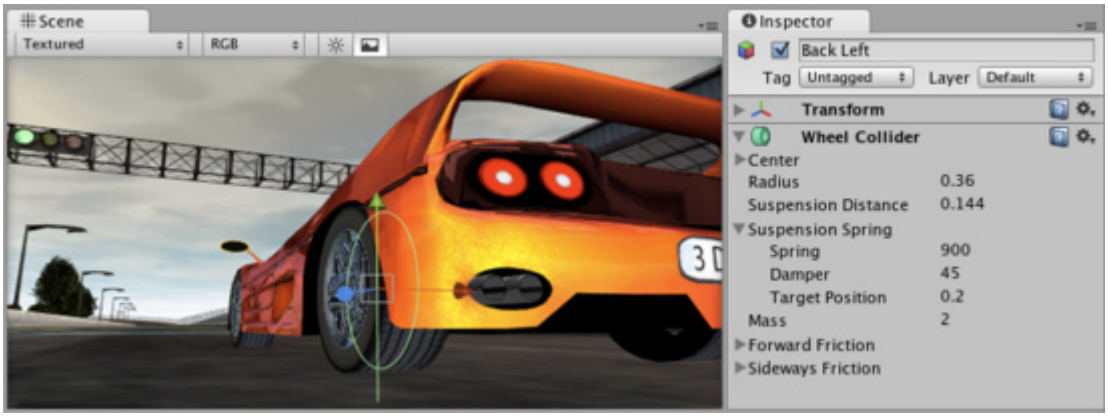


Figure 15 - Wheel Colliders

Utilizando a hierarquia entre *Game Objects* da Unity, o movimento de um veículo pode ser simulado facilmente através da configuração hierárquica destes componentes corretamente.

Joints

Uma funcionalidade muito importante disponível na Unity é a possibilidade do uso de *Joints*, ou junções, de vários tipos para a simulação de objetos conectados a outros com restrições em seu grau de liberdade. Os seguintes *Joints* estão disponíveis na PhysX através da Unity:

Hinge Joint: permite a conexão de dois objetos através da simulação de uma dobradiça. Ideal para a simulação de portas, pêndulos e outros objetos que necessitem deste tipo de conexão.

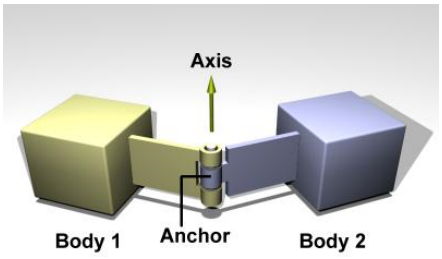


Figure 16 - Hinge Joint

Spring Joint: permite a conexão de dois abjetos através da simulação de uma mola. Objetos conectados utilizando esse tipo de *joint* possuem uma distância máxima de separação que, após soltos, tendem a voltar a sua distância de repouso.

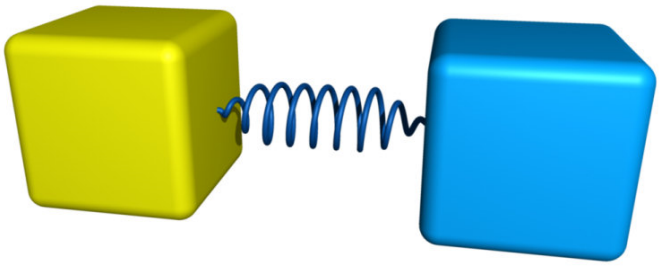


Figure 17 - Spring Joint

Fixed Joint: permite a conexão entre dois objetos de forma que os movimentos de um objeto sejam dependentes do outro. Similar a utilização das hierarquias de transformação da Unity, porém, implementado através da física. Ideal para objetos que possam ser desconectados um do outro durante a simulação.

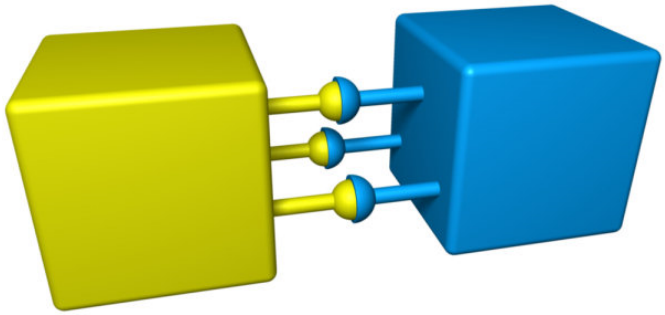


Figure 18 - Fixed Joint

Configurable Joint: esse tipo de *joint* oferece a possibilidade de customização de seu comportamento. Aqui, vários tipos de configuração podem ser efetuadas como restrição de movimento e/ou rotação e aceleração de movimento e rotação. Dessa forma, temos como construir um *joint* de acordo com a necessidade requerida.

Por fim, a *PhysX* nos permite efetuar simulações de superfícies de contatos através dos materiais físicos. Com isso, temos a possibilidade de alterar a forma como os objetos interagem através do ajuste das propriedades dos materiais que estes objetos utilizam tais como fricção e o efeito da interação entre a colisão de dois objetos.

5 - Scripting

O sistema de *scripting* da Unity3D é abrangente e flexível, o que permite o desenvolvimento de jogos completos sem a necessidade do uso de C/C++. Internamente, os *scripts* são executados através de uma versão modificada da biblioteca *Mono*, uma implementação de código aberto para o sistema *.Net*. Essa biblioteca, permite que os scripts sejam implementados em qualquer uma de três linguagens, à escolha do programador: *javascript*, *C#* ou *Boo* (um dialeto de *Python*). Não existe penalidade por se escolher uma linguagem ou outra, sendo inclusive possível se usar mais de uma delas em um mesmo jogo. A documentação oficial, entretanto, utiliza *Javascript* para a maioria dos exemplos.

De forma consistente à arquitetura desenvolvida, scripts na Unity3D são acoplados como componentes de *game objects*. Dessa forma, é importante projetar os *scripts* de maneira modular, ganhando com isso a flexibilidade do reuso. Nessa seção, iremos descrever algumas características importantes do sistema de *scripting* da Unity3D.

5.1 - Criação de scripts

Para criar um *script*, basta escolher a opção *Assets -> Create -> Javascript* no menu principal. Também é possível se criar *scripts* usando o botão direito do mouse sobre a janela *project*. Existe a opção de se criar o *script* em qualquer uma das três linguagens disponíveis.

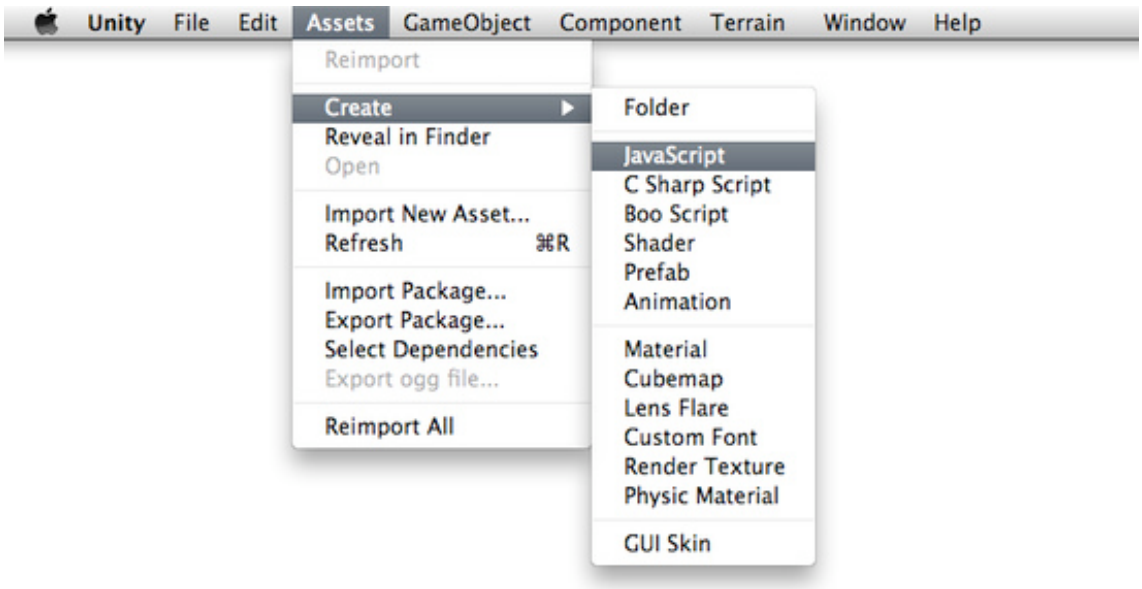


Figure 19 - Criação de um script (UNITY TECHNOLOGIES 2009A)

É possível se editar o *script* clicando duas vezes sobre o mesmo na janela *Project*. Isso irá abrir o editor padrão (UniScite no Windows ou Unitron no Mac OSX). A edição de *scripts* sempre é

feita em um programa externo e não diretamente pela Unity3D, que pode ser alterado nas preferências de usuário. Esse é o conteúdo de um *Javascript* recem-criado na Unity3D:

```
function Update () {
}
```

Um *script* novo não realiza tarefa alguma ainda, então pode-se adicionar funcionalidade ao mesmo. O código a seguir serve como um exemplo básico:

```
function Update () {
    print("Hello World");
}
```

Ao ser executado, esse código irá exibir a expressão "Hello World" no console. Mas ainda não existe nada que causa a execução desse código. É necessário se acoplar esse *script* a um *Game Object* ativo na cena para que isso ocorra. Isso pode ser feito se arrastando o arquivo do script para o objeto escolhido tanto na janela *Hierarchy,* quanto diretamente ao mesmo na janela *Scene*. Também pode-se selecionar o objeto escolhido e adicionar o script através do menu principal, como mostra a figura a seguir.

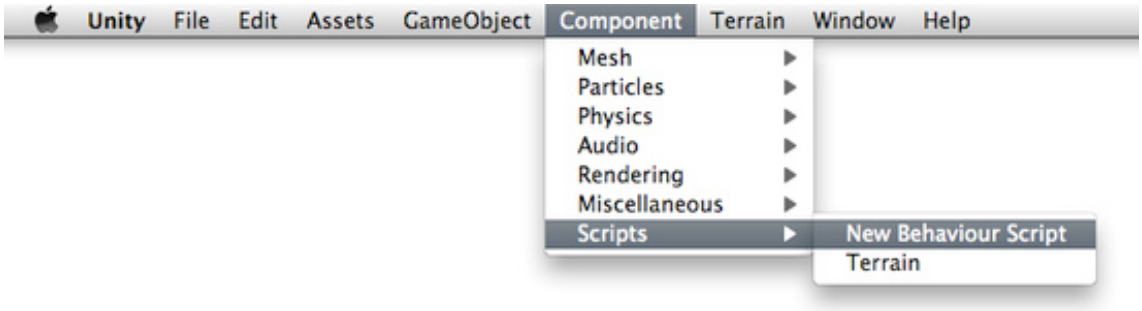


Figure 20 - Adição de um script a um objeto selecionado (UNITY TECHNOLOGIES 2009A)

Ao se selecionar o objeto ao qual o script recém-criado foi adicionado, será possível se visualizar o mesmo, indicando sua correta vinculação.

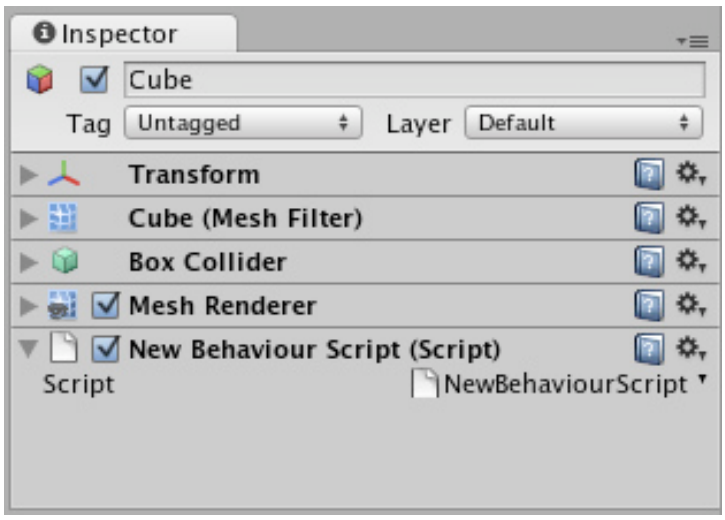


Figure 21 - Script vinculado a objeto (UNITY TECHNOLOGIES 2009A)

A função *print()* é bastante útil quando se está depurando um *script*, mas não faz nada de útil em termos de jogabilidade. O código a seguir adiciona alguma funcionalidade ao objeto que contém o *script*:

```
function Update () {
    transform.Rotate(0, 5*Time.deltaTime, 0);
}
```

Para quem é novo em linguagens de script para jogos, o código pode parecer um pouco confuso. Alguns conceitos importantes que devem ser aprendidos:

1. *function Update () {}* é a sobrescrita do método *Update*, que será executado pela Unity3D uma vez a cada frame;

2. *transform* é uma referência ao componente *Transform* do *Game Object* em questão.

3. *Rotate()* é um método existente no componente *Transform*;

4. Os parâmetros desse método representam os graus de rotação sobre cada um dos eixos em um espaço 3D: X, Y, and Z.

5. *Time.deltaTime* é um atributo da classe Time que representa o tempo que passou (em segundos) desde que o último *Update* ocorreu. Essa variável serve para garantir que o objeto em questão seja rotacionado na mesma velocidade, independente da capacidade de processamento do computador no qual o código será executado. Dessa forma, 5 * *Time.deltaTime* representa 5 graus por segundo.

Já existem referências como essa para alguns componentes que são comuns de serem utilizados em objetos. Alguns dos mais importantes são:

- transform - representa o posicionamento, orientação e escala do objeto;
- rigidbody - representa o corpo rígido para o sub-sistema de física (quando existir);
- animation - utilizado para acionar os ciclos de animação de um modelo criado em uma ferramenta de animação como o 3D Studio Max;
- renderer - componente encarregado da exibição de um objeto na cena;
- audio - fonte de efeito de áudio, vinculada a um objeto para incorporar posicionamento de áudio 3D;
- collider - geometria de colisão para o subsistema de física ou para utilização como *Trigger*.

5.2 - Acesso a outros componentes e troca de mensagens

Entretanto, muitos outros componentes pré-existentes, assim como Scripts criados especificamente para cada jogo, não têm referências especiais incluídas. Para esses, é necessária alguma maneira de se obter uma referência em tempo de execução. Isso é feito através do método *GetComponent()* existente em qualquer script. Esse método pode ser usado para se obter a referência a qualquer componente vinculado ao objeto no qual o *script* em questão está acoplado (ou a qualquer objeto que se tenha uma referência em uma variável). O exemplo a seguir mostra como obter a referência a um *script* chamado "Controlador":

```
var c : Controlador = GetComponent(Controlador);
c.MeuMetodo();
```

Nesse exemplo, a variável "c" foi usada para se guardar a referencia a instância do *script* Controlador vinculado ao mesmo objeto de jogo em questão. Em seguida, foi executado o método "MeuMétodo" nessa mesma instância.

É sempre preferível, por questões de desempenho, guardar referências aos componentes como exemplificado acima. Entretanto, existem situações onde pode ser desejável se enviar uma determinada mensagem a todos os componentes de um determinado objeto. Isso é possível através do método *SendMessage* da classe *GameObject*. Todo *script* tem acesso à instância de *GameObject* ao qual está vinculado. O seguinte exemplo tenta executar (caso exista) o método "Teste" em todos os componentes do objeto ao qual esse script for vinculado:

```
gameObject.SendMessage("Teste");
```

Um exemplo comum para o uso de SendMessage é a aplicação de "dano" a um personagem que foi atingido por uma bala, bomba, ou qualquer outro objeto do jogo que possa causar algum efeito desse tipo. A solução pode ser criar esse projétil como um objeto com componentes de física (*collider* e *rigidbody*), e incluir um *script* simples que envia uma mensagem assim que esse projétil atingir algo na cena:

```
function OnCollisionEnter(collision : Collision) {
   collision.gameObject.SendMessage("AplicarDano");
   Destroy(this.gameObject);
}
```

O código acima será executado assim que o projétil (objeto contendo representação física e esse *script*) colidir com outro objeto com física na cena. A mensagem enviada irá causar a execução do método "AplicarDano" em todos os *scripts* do objeto atingido, caso esse método exista. A linha seguinte remove o projétil da cena.

5.3 - Acesso a variáveis

Os scripts de rotação apresentados até agora giram o objeto 5 graus a cada segundo. Talvez seja interessante rotacionar a uma velocidade angular diferente dessa. Uma opção é alterar esse valor no código e salvá-lo, mas exige uma recompilação desse, ao mesmo tempo que impede que usemos o mesmo script com diferentes velocidades. Existe uma forma bem mais rápida para isso, que inclusive permite a alteração de parâmetros como esse em tempo de execução no editor de cena, e é bastante simples de ser implementada.

Em vez de digitar o valor 5 diretamente, pode-se criar um atributo *speed*, do tipo *float*, no *script* e usar esse na chamada ao método *Rotate()*. O exemplo a seguir mostra como isso pode ser feito:

```
var speed = 5.0;
function Update () {
    transform.Rotate(0, speed*Time.deltaTime, 0);
}
```

Agora observe o objeto contendo o *script* na janela *Inspector*. Pode-se notar que o valor do atributo aparece no editor de cena, como mostra a figura a seguir.

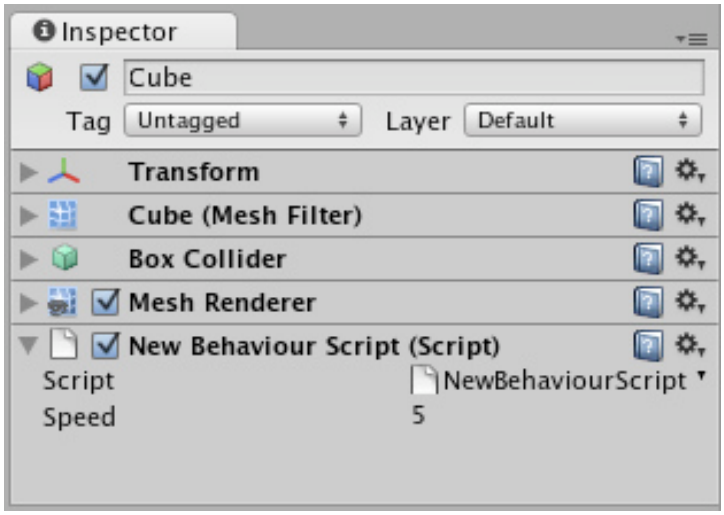


Figure 22 - Alteração do valor de um atributo (UNITY TECHNOLOGIES 2009A)

Este valor pode agora ser modificado diretamente pela janela *Inspector*, da mesma forma que pode-se renomear um arquivo. Selecionando-se a mesma, pode-se alterar seu valor. Também pode-se deslizar os valores com o mouse usando o botão direito. Pode-se alterar o valor de atributos como esse a qualquer momento, inclusive enquanto o jogo está sendo executado.

Ao apertar o botão *Play* e modificar-se o valor desse atributo, a velocidade de rotação do objeto irá ser alterada instantaneamente. Ao se sair do modo de execução, o valor volta ao anterior. Dessa forma, pode-se experimentar a vontade e ao final decidir manter o valor anterior ou alterá-lo de forma permanente (sem o botão *Play* pressionado).

Essa forma de alteração de valores em atributos também implica que é possível se usar um mesmo script em diversos objetos, cada um com um valor específico para o mesmo. Cada alteração feita irá afetar a velocidade apenas do objeto no qual a mudança foi realizada.

5.4 - Prefabs e Instanciação

A criação de *game objects* através de composição é bastante flexível, mas em alguns momentos pode ser bastante trabalhoso recriar certas dessas composições que precisam ser usadas em diversas cenas de um mesmo jogo, compartilhadas com outros desenvolvedores ou mesmo instanciadas interativamente durante a execução do jogo.

Um *Prefab* é um tipo de *asset* - um *Game Object* reusável armazenado na janela *Project*. *Prefabs* podem ser inseridos em diversas cenas, múltiplas vezes em cada uma delas. Ao se adicionar um *Prefab* a uma cena, está sendo criada uma instância do mesmo. Todas essas instâncias estão ligadas ao *Prefab* original e são no fundo clones desse. Independente de quantas instâncias existam no projeto, qualquer mudança feita ao *Prefab* original será aplicada a todas essas cópias existentes nas cenas.

Para se criar um *Prefab*, é preciso criar um container vazio para o mesmo usando o menu. Esse *Prefab* vazio não contém um *Game Object* ainda, e portanto não pode ser instanciado na cena

ainda. Após ser recheado com dados de um *Game Object*, isso pode ser feito. A figura a seguir mostra um *Prefab* recém-criado, ainda sem conteúdo (indicado pela ausência de cor em seu nome na janela *Project*).

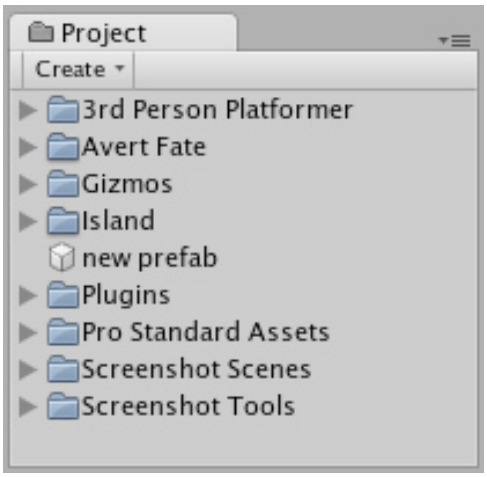


Figure 23 - Criação de um Prefab (UNITY TECHNOLOGIES 2009A)

Para preencher um prefab, deve ser usado algum objeto existente na cena atual. O seguinte roteiro explica como esse processo pode ser realizado:

1. Escolha *Assets->Create->Prefab* no menu principal e dê um nome ao mesmo;
2. Na janela *Hierarchy*, selecione o *Game Object* que se deseja guardar como um *Prefab*;
3. Arraste e solte esse objeto da janela *Hierarchy* sobre o novo *Prefab* na janela *Project*.

Após realizados esses passos, o objeto, todos os seus "filhos", componentes e valores de atributos foram copiados no *Prefab*. Agora é possível se criar diversas instâncias do mesmo arrantando-se para cena a partir da janela *Hierarchy*. O próprio objeto usado para a criação do *Prefab* foi transformado em uma instância do mesmo.

Todas as instâncias de um *Prefab* possuem a mesma estrutura e, originalmente, os mesmos valores para os atributos de seus componentes. Entretanto, é possível alterar diversas instâncias, ainda vinculadas ao *Prefab* original, os valores de alguns atributos. As alterações feitas ao *Prefab* ainda serão propagadas para essas instâncias, apenas os atributos marcados como específicos terão seus valores mantidos. A seguinte figura mostra esse procedimento, que consiste em se marcar a caixa que fica a esquerda do nome do atributo, na janela *Hierarchy* (no exemplo em questão, o atributo de nome *Repeat Trigger* não será alterado de acordo com as modificações no *Prefab*).

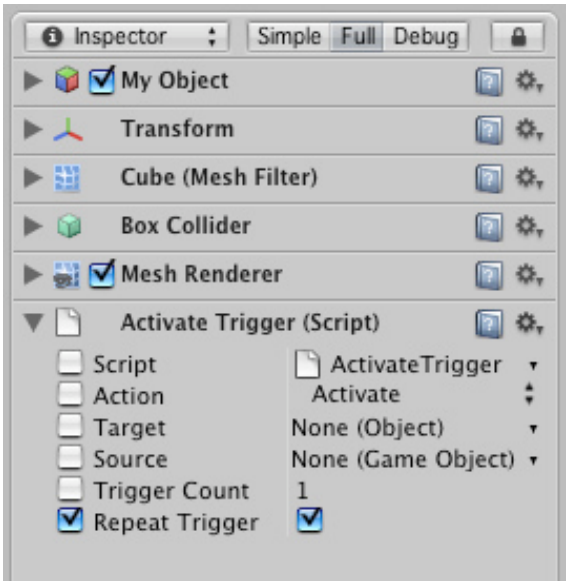


Figure 24 - Atributo desvinculado do Prefab (UNITY TECHNOLOGIES 2009A)

Em diversas situações pode ser necessário se instanciar objetos em tempo de execução. Um exemplo que ilustra esse cenário pode ser a criação de foguetes que são lançados quando o jogador pressiona uma tecla ou botão. *Prefabs* são uma ferramenta útil nessas situações, pois a criação de cópias dos mesmos através de *scripts* é bastante simples, como explicado a seguir.

A primeira tarefa é a criação de um *Prefab* contendo os componentes e valores calibrados para os atributos desejados para o referido foguete. Não iremos entrar em detalhes sobre a criação da funcionalidade do foguete em si, e sim com o processo de instanciação do mesmo. Para se ter a referência a um *Prefab* através de um *script*, basta que se crie um atributo cujo tipo seja um dos componentes existentes nesse *Prefab*. No nosso exemplo, o foguete será guiado pelo sistema de simulação física, e conseqüentemente um dos componentes do mesmo é um *Rigidbody*, dessa forma o script de instanciação deve incluir a seguinte linha:

```
var rocket : Rigidbody;
```

Isso irá permitir ao desenvolvedor arrastar o *Prefab* do foguete diretamente da janela *Project* para a janela *Hierarchy* de forma a estabelecer a referência necessária. O código a seguir cria uma instância do foguete em tempo de execução, ao mesmo tempo que adiciona uma velocidade inicial ao mesmo, referente à orientação do objeto que o criou:

```
var r : Rigidbody = Instantiate(rocket, transform.position,
                                transform.rotation);
rocket.velocity = transform.forward * speed;
```

É importante observar que o código acima independe da estrutura utilizada para o *Prefab* que representa o foguete, desde que o mesmo inclua um componente *Rigidbody*. Isso permite a criação de protótipos simples e funcionais bem antes da existência de *Assets* definitivos como modelos 3D ou sistemas de partículas. Os scripts criados inicialmente ainda serão possíveis de se usar mesmo com a evolução desse *Prefab* para uma versão mais bem acabada para a representação desse foguete.

6 - Conclusão

Esse tutorial apresentou um introdução sucinta sobre o motor de jogos Unity3D. O objetivo foi expor de maneira simplificada as principais funcionalidades dessa versátil ferramenta de desenvolvimento de jogos. Espera-se que o leitor interessado busque um maior aprofundamento nesse assunto através da bibliografia sugerida no final desse tutorial.

Os autores gostariam de agradecer às pessoas que direta ou indiretamente contribuíram para a confecção desse material. Em especial gostaríamos de agradecer às nossas famílias, pela paciência e apoio incondicional, e também à excelente comunidade de desenvolvedores e colaboradores Unity pelas excelentes informações e tutoriais disponibilizados.

Finalmente, os autores gostariam de agradecer os leitores desse tutorial e informar que a toda a equipe do UFF-MediaLab se coloca à disposição para contato através dos emails informados na capa.

Bibliografia

BILAS , S. 2002. *A data-driven game object system*. Talk at the Game Developers Conference '02.

PASSOS, E. B., SILVA, J., NASCIMENTO, G. T., KOZOVITS, L. CLUA, E. W. G. 2008. *Fast and safe prototyping of game objects with dependency injection*. Anais do Simpósio Brasileiro de Games e Entretenimento Digital. São Leopoldo, RS. 2008

STOY, C. 2006. *Game object component system*. In Game Programming Gems 6, Charles River Media, M. Dickheiser, Ed., Páginas 393 a 403.

UNITY TECHNOLOGIES. 2009 (A). *Unity 3D User Manual* [online]. Disponível em: www.unity3d.com/support/documentation/Manual [Acessado em 20 agosto de 2009].

UNITY TECHNOLOGIES. 2009 (B). *Unity 3D Community Forum* [online]. Disponível em: forum.unity3d.com/ [Acessado em 25 agosto de 2009].

UNITY TECHNOLOGIES. 2009 (C). *Unity 3D Online Tutorials* [online]. Disponível em: www.unity3d.com/support/documentation/tutorials [Acessado em 30 agosto de 2009].

UNIFY COMMUNITY. 2009. *Unity 3D Community Wiki* [online]. Disponível em: www.unifycommunity.com/wiki [Acessado em 28 agosto de 2009].