4
Casos de estudo

Como forma de demonstrar a utilidade do DWeb3D, foi desenvolvida uma aplicação que faz uso de suas funcionalidades em pequenos testes pontuais. Neste capítulo essas funcionalidades são demonstradas e explora-se como isto pode ser aplicado a um escopo mais amplo.

4.1
Aplicação de demonstração

Como forma de organizar os testes e as demonstrações foram criados diversos pequenos projetos de testes, cada um focado em uma funcionalidade específica. Estas funcionalidades foram pensadas de forma a exemplificar possíveis usos do DWeb3D.

O importante aqui é demonstrar que a complexidade dessas funcionalidades ficaram escondidas no código do toolkit.

4.1.1
Grafo de Cena

Por ser a base de todo o toolkit uma discussão mais aprofundada sobre o modelo do grafo de cena é importante. Esta discussão se coloca antes das outras por ser mais básica e ajudar a compreensão das demais.

As principais funções disponíveis no pacote Model, que encapsula o grafo e a estrutura do X3D são:

- Carga de um arquivo X3D.
- Exposição da estrutura da cena.
- Renderização da estrutura alterada novamente para um arquivo.

A carga é obtida através de uma função recursiva que lê cada item do arquivo XML do X3D e o transforma num nó equivalente, respeitando a hierarquia. O processo é feito de acordo com o descrito na Figura 4.1.

Para demonstrar o funcionamento da carga do modelo podemos observar o seguinte pedaço de código:

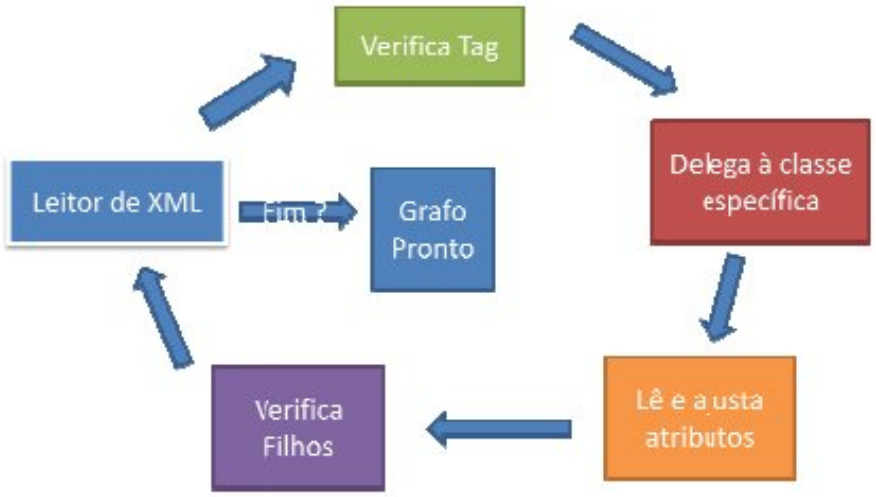

Figura 4.1: Processo de carga do modelo.

```
public static void  TestRead()
{
 // Criando o objeto para armazenar as classes
    X3DRepresentation xrep = new X3DRepresentation();
    // Carregando do arquivo
    xrep.LoadFromFile("..\\Samples\\BeckyRoadOverpass.x3d");
    // Renderizando para a tela
    Console.WriteLine(xrep.RendertoX3D());

}
```

Pode-se notar que o modelo encapsula todo o processo de carga e renderização necessário para lidar com um arquivo X3D em código .NET. No final temos um grafo hierárquico com todos os nós da cena. A renderização por sua vez é feita através de uma função recursiva em profundidade como ilustrada na Figura 4.2.

A Figura 4.3 ilustra o resultado da renderização feita pelo modelo. Ela serve como forma de mostrar a corretude da carga e da renderização.

4.1.2 Colaboração

Como feito nos capítulos anteriores, aqui também a organização se volta aos principais objetivos a serem alcançados com o DWeb3D. Para demostrar

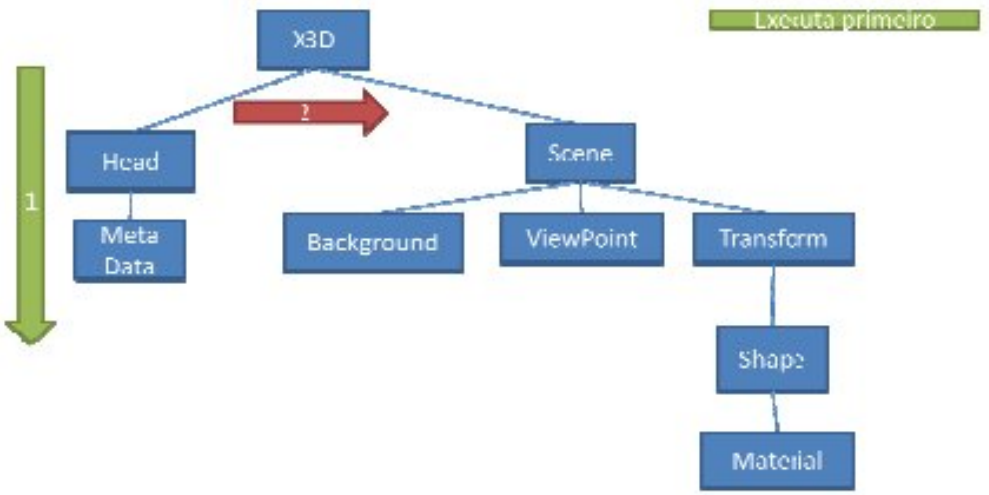

Figura 4.2: Ordem de renderização do grafo.

o quão simples é codificar uma aplicação que possa ser colaborativa foi desenvolvida uma demonstração, lembrando que o objetivo não é desenvolver uma aplicação completa, mas mostrar que é possível cortar etapas e agilizar o processo de desenvolvimento através do uso do toolkit. O conceito geral é ter algo como o ilustrado na Figura 4.4.

A primeira tarefa a ser feita é definir um servidor. Ele deve ser capaz de aceitar vários usuários e sincronizar as mensagens provenientes deles. Para isto foi definida uma cena simples que será carregada no servidor e nos dois clientes. Em seguida um dos clientes irá modificar um objeto e sincronizar. Espera-se que o segundo cliente seja capaz de ver as modificações.

O código para o servidor é o mesmo do exemplo no Capítulo 4. Para os clientes o código se parece um pouco com o do servidor e é descrito a seguir:

```
public void StartClient()
{
  // Criando o cliente na porta 1717
  Client c = new Client(1717, "127.0.0.1"');
  // Definindo e criando a cena básica que este cliente
  // vai trabalhar
  c.SceneGraph = new X3DRepresentation();
  c.SceneGraph.CreateBasicScene();
  // registrando o cliente
```

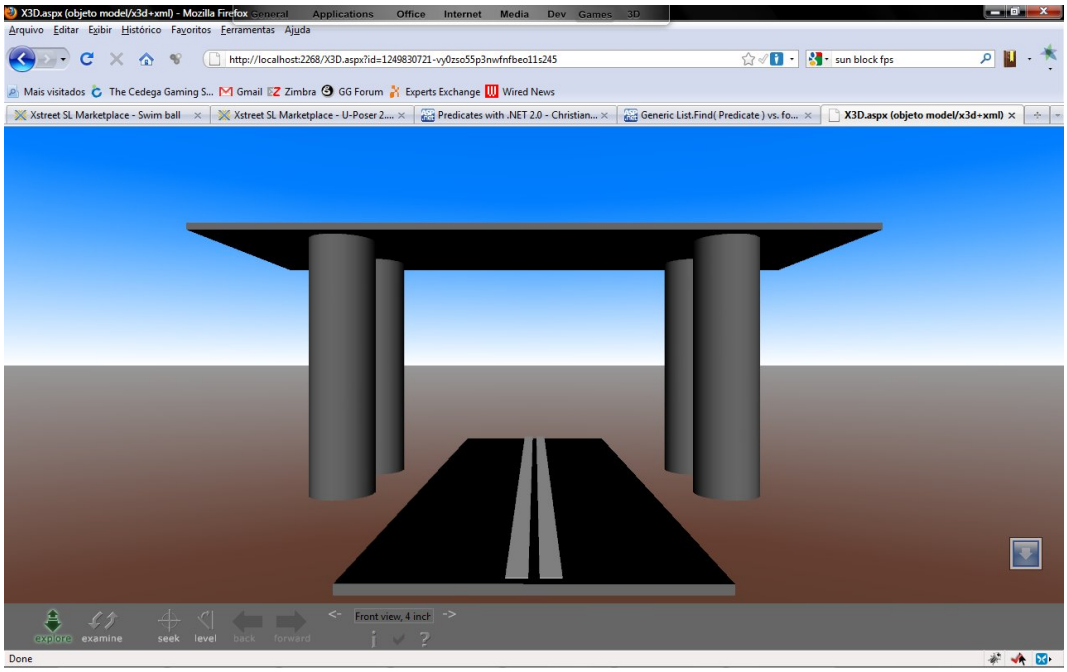

Figura 4.3: Renderização com o Flux 3D do resultado do modelo.

```
  ClientManager.Add(c);
}
```

Para o cliente a principal diferença é que ele precisa saber quem é o seu servidor. Uma vez conectados, cliente e servidor ficam quase invisíveis na aplicação. Desta forma, basta fazer as alterações necessárias e sincronizar. O único detalhe que deve ser notado é que a sincronização normalmente parte do cliente.

Como estamos comparando a eficiência do DWeb3D, mostra-se a seguir como seria o código de criação de um servidor (somente a parte de rede) sem o toolkit. Como pode-se notar ele é mais complexo.

```
public void Initiate()
{
    // Cria uma configuração para o servidor
    Config = new NetConfiguration("ObjectSync");
    Config.MaxConnections = 128;
    Config.Port = port;

    // Cria o servidor e ouve as conexões
    Server = new NetServer(Config);
    Server.SetMessageTypeEnabled
     (NetMessageType.ConnectionApproval, true);
    Server.Start();
```

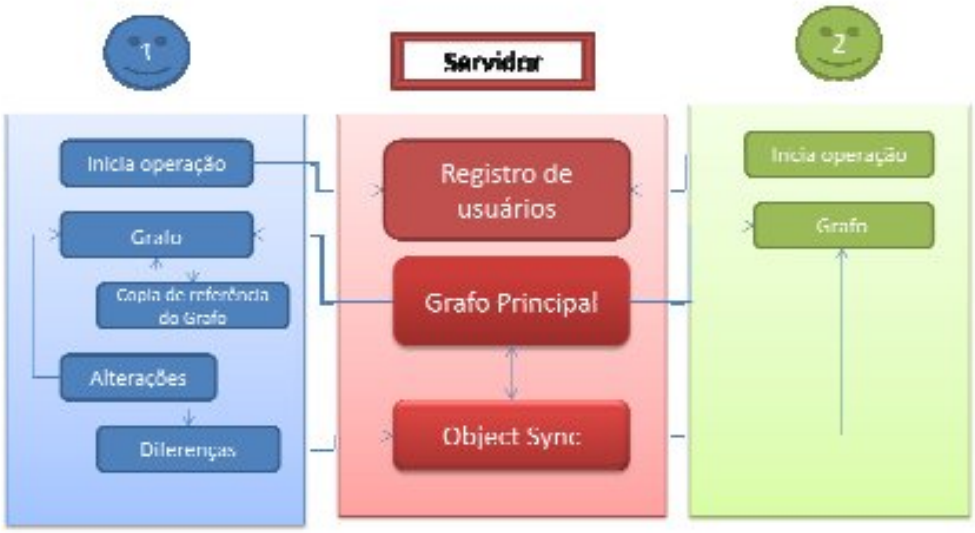

Figura 4.4: Modelo de dois usuários visualizando uma cena.

```
        //Define o canal padrão
        DefaultNetChannel = NetChannel.ReliableInOrder1;

        // Cria um buffer
        Buffer = Server.CreateBuffer();

        // Cria uma thread
        ThreadStart job = new ThreadStart(Reader);
        this.Job = new Thread(job);
        this.Job.Start();

    }
```

O código acima somente cuida da inicialização do servidor. No exemplo usando o DWeb3D, existem classes que gerenciam mais de um servidor, e gerenciam a troca de mensagens, além de cuidar de fazê-las transparente. O código equivalente sem o uso do toolkit seria portanto ainda mais complexo que o trecho acima listado.

Para finalizar, foi criado um objeto fora do grafo para que se pudesse simular uma operação de conversa entre dois usuários, e o código a seguir mostra como isso funciona.

```
// Criação de um objeto chat
var chat = new Chat();

// Definindo o objeto como um ISyncable
// (ele tem que implementar a interface)
ISyncable tmp = chat;
// Definindo um netIDválido
tmp.NetId = SyncManager.Instance.NextNetID;
// Registrando o objeto no gerente
SyncManager.Instance.Register(ref tmp );

// Eviando uma mensagem
chat.EnviarMensagem("Olá");
```

Neste código a função enviar mensagem seta uma variável e depois sincroniza. O servidor replica essa mensagem que os outros clientes poderão ler.

Para ilustrar como essa troca de mensagens seria feita sem o DWeb3D no Apêndice C está um pequeno trecho de código, responsável pelo envio e recebimento de mensagens de uma aplicação de chat.

Verifica-se que o código que têm aproximadamente 50 linhas é bastante extenso e mais complexo de ser entendido que o gerado com utilizando o DWeb3D.

4.1.3 Persistência e carga

Para se obter a persistência basta que se utilize o X3DHolder como forma de expor o código para o visualizador. Como descrito no capítulo anterior, ele será o responsável pelo registro e recuperação do cookie, e nesse aspecto não há muito o que se demonstrar, a não ser como expor o X3DHolder. Isto é feito através do pequeno pedaço de código.

```
<%@ Register NameSpace="DWeb3D.X3DHolder"
TagPrefix="X3DH" Assembly="X3DHolder"  %>

<X3DH:X3DHolder ID="XHolder" runat="server" WIDTH="300" HEIGHT="300" >
</X3DH:X3DHolder>
```

Neste aspecto a aplicação de demonstração ilustra como utilizar o segundo nível de persistência, que é a persistência em disco. Para isto, o servidor ganhou um método Save() como visto no código abaixo:

```
public void StartDefaultServer()
{
   // Criando o servidor na porta 1717
    SceneServer s = new SceneServer(1717);
    // Definindo e criando a cena básica que este servidor
    // vai trabalhar
    s.SceneGraph = new X3DRepresentation();
    s.SceneGraph.CreateBasicScene();
    // Salvando para o disco
    s.Save();
}
```

O código acima cria um servidor, define uma cena padrão e chama o método para salvá-la. Este método vai verificar a existência de uma base de dados (arquivo) padrão no disco, caso ela não exista será criada. Após isto será chamada a biblioteca db4o (Gre05) que se encarregará de lidar com as complexidades de armazenamento de objetos em disco.

Sem o uso do toolkit seria necessário lidar com a inicialização do db4o, verificação de existência da base e configurações da biblioteca, além de ter de buscar o nó correto no grafo, para que o salvamento possa ser feito com sucesso.

4.1.4 Interação com outras aplicações

Um exemplo de como podemos fazer a aplicação interagir com outras é o servidor, pois através dele a cena X3D está interagindo com o código .NET num servidor que por si só já é outra aplicação. Porém, um outro exemplo foi desenvolvido na forma do Unity3D Render. Trata-se de código que pode ser chamado no motor gráfico e assim o grafo é sincronizado lá. Desta forma podemos adicionar comportamentos específicos através de scripts atrelados aos nós na Unity3D. O exemplo abaixo demonstra como se utilizar o script.

```
public class Creator : MonoBehaviour {

// Criando a representação
X3DRepresentation xrep = new X3DRepresentation();
```

```
void Start () {

  // Carregando o arquivo
  xrep.LoadFromFile("..\\Exemplo.x3d");

  // Inicializando o renderizador
  RenderManager.Instance.InitializeGraphsRender(xrep.Scene);
  xrep.Scene.RenderToScreen();

}

// Chamado uma vez por frame.
void Update () {
xrep.Scene.RenderToScreen();
}
}
```

Para demostrar o funcionamento do plugin, uma cena bastante simples será utilizada. Trata-se de uma cena com somente um cilindro, um cubo, um viewport e algumas transformações. Sua descrição pode ser encontrada no Apêndice B, sua imagem renderizada com o Flux 3D[1] pode ser vista na Figura 4.5.

Com a implementação feita podemos acompanhar o processo de renderização da Unity3D nas Figuras 4.6 e 4.7. Nelas podemos ver o antes e o depois da renderização.

Na Figura 4.7 pode-se notar que os nós criados são selecionáveis e que eles apresentam as características dos nós da cena X3D. O processo que ocorreu foi uma conversão do modelo do grafo X3D com o grafo da Unity. Finalizando, outra característica interessante do motor gráfico é que ele permite atrelar scripts a nós. O próprio script de criação visto anteriormente é atrelado a um nó especial chamado criador e utilizando esta técnica é possível adicionar scripts que respondam a eventos específicos do nó como um click ou um usuário passar pelo nó.

Para se obter este mesmo resultado sem a utilização do DWeb3D seria necessário escrever um método de conversão do X3D no grafo da Unity. Este método teria que conter: um leitor de xml, um interpretador, uma função que transformasse essa estrutura em algo que o programa pudesse entender e uma

[1]Visualizador X3D desenvolvido pela Media Machines. http://www.mediamachines.com
.

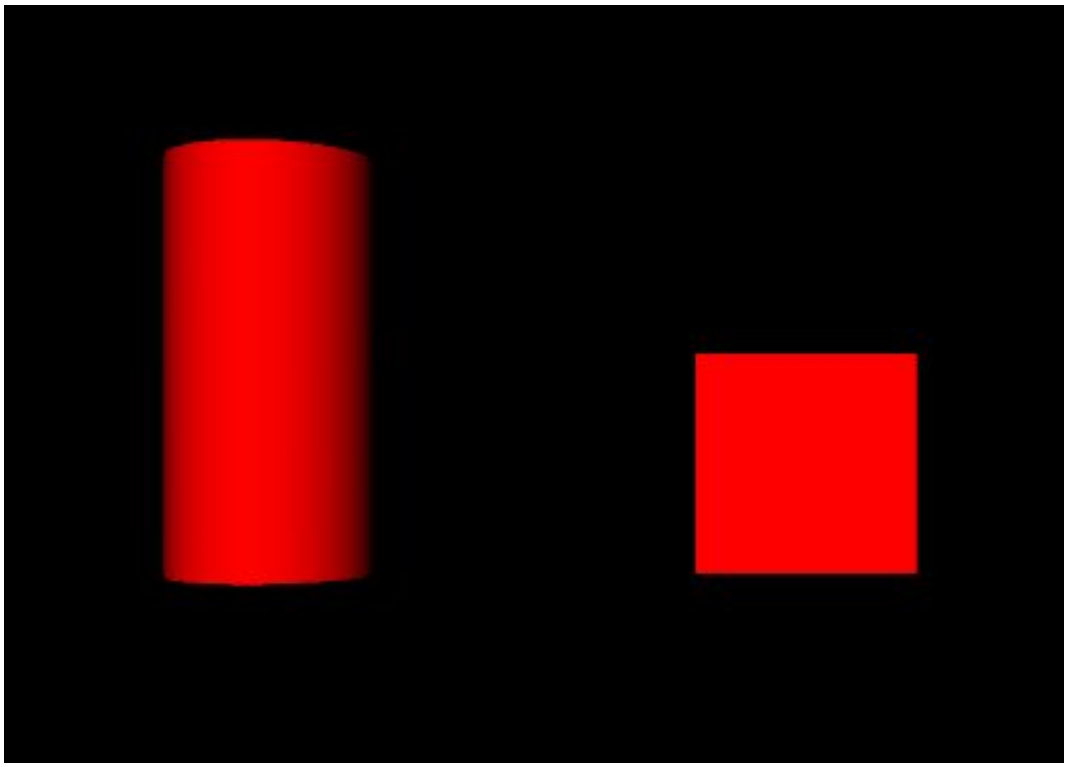

Figura 4.5: Imagem da cena básica renderizada pelo Flux.

função que transformasse a estrutura que o progama entende na estrutura da Unity. Por todo este processo ser muito extenso, será exemplificada a seguir apenas a função que faz uma parte desse processo, que é a conversão de um cubo. Lembre-se que este é apenas uma dentre muitas geometrias que teriam que ser convertidas, e que a conversão não passa só por geometrias, mas também por materiais, transformações, etc.

O trecho no Apêndice D é apenas uma pequena parte do código utilizado para a renderização. Para se refazer o trabalho do toolkit seriam necessárias muito mais linhas de código e uma complexidade bem grande.

O DWeb3D abordou o problema da interação com outras aplicações focando na integração com o Unity3D. Para uma possível interação com um outro motor gráfico, o DWeb3D implementa um esquema de plugins de renderização. Isto significa que, bastando rescrever o código de renderização, é possível aproveitar todos os outros mecanismos do toolkit como carga, manipulação do grafo, etc.

4.1.5 Interação com o GUI Web

O GUI web não deixa de ser uma outra aplicação, então neste caso estamos lidando com um caso mais específico de integração. Porém, sendo o GUI web o ambiente que irá hospedar a cena 3D, a integração é ainda mais

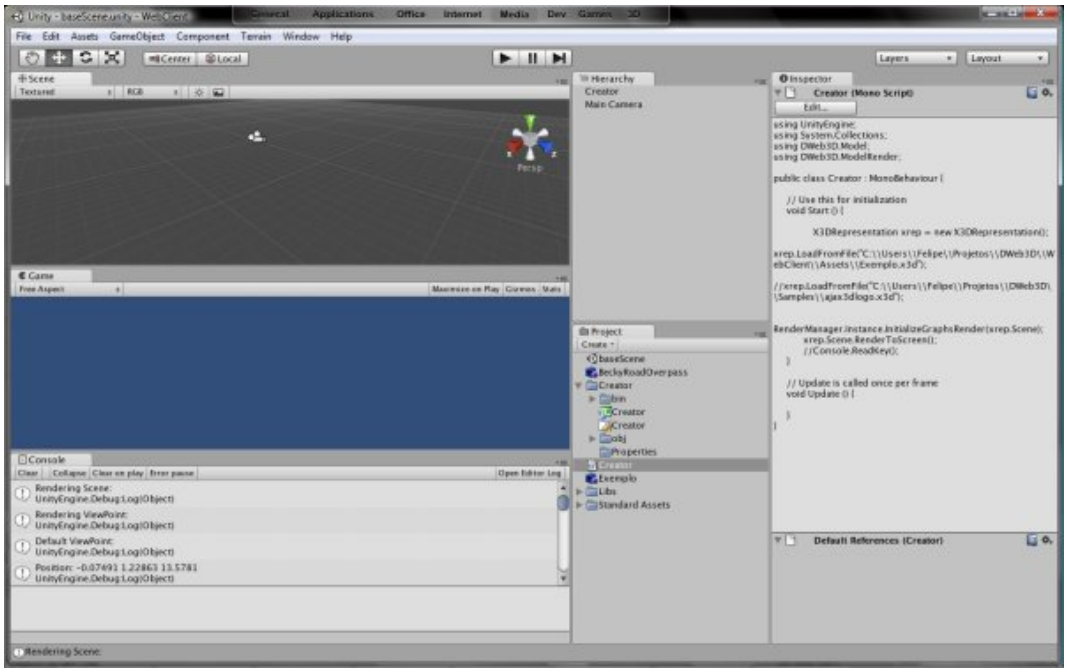

Figura 4.6: A Unity3D antes de iniciar a renderização.

importante. Sua integração pode ser feita via chamadas Ajax comuns no .Net que executam código no servidor. Este código estará no mesmo ambiente do servidor X3D, e através do gerente SceneServers pode ter acesso ao grafo do servidor e alterá-lo afetando assim a cena.

Para exemplificar esta possibilidade vamos verificar o seguinte código:

```
using System;
using DWeb3D.Model;

namespace TestWebApp
{
    public partial class _Default : System.Web.UI.Page
    {
       // Criando o objeto de armazenamento
        private X3DRepresentation x3dm;

        protected void Page_Load(object sender, EventArgs e)
        {
            x3dm = new X3DRepresentation();
            x3dm.LoadFromFile("..\\Samples\\Exemplo.x3d");
            XHolder.Representation = x3dm;
        }
```

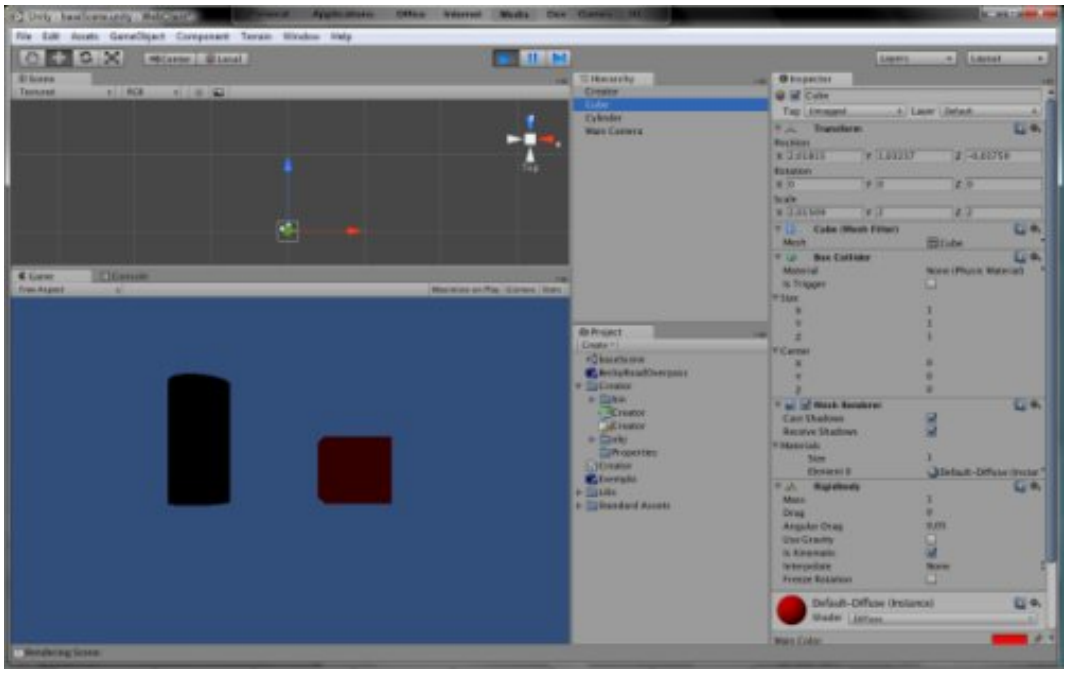

Figura 4.7: A Unity 3D depois de renderizado.

```
// Código chamado pelo botão para aumentar o box
        protected void BtAumentarCaixa_Click(object sender, EventArgs e)
        {
            ((Box)XHolder.Representation.Scene.Nodes[1]).Size
             += new Vector3D(2, 2, 2);
        }
    }
}
```

Quando o usuário clica no botão aumentar caixa o código vai no local de armazenagem (sem refresh devido ao Ajax) e soma 2 em todos valores. O antes e o depois dessa interação podem ser vistos nas Figuras 4.8 e 4.9 respectivamente.

Para se obter os mesmos resultados sem o DWeb3D, a forma mais simples seria desenvolver um código em ECMAScript que via API de browser (que muda de acordo com o cliente renderizando a cena) acessasse o nó e fizesse a modificação. O problema é que ECMAScript é complexo de depurar e os browsers mudam um pouco a API de acordo com a implementação.

4.2 Análise dos resultados obtidos

Pudemos verificar nos exemplos deste capítulo como a utilização do DWeb3D reduz a quantidade de código necessário para se obter alguns re-

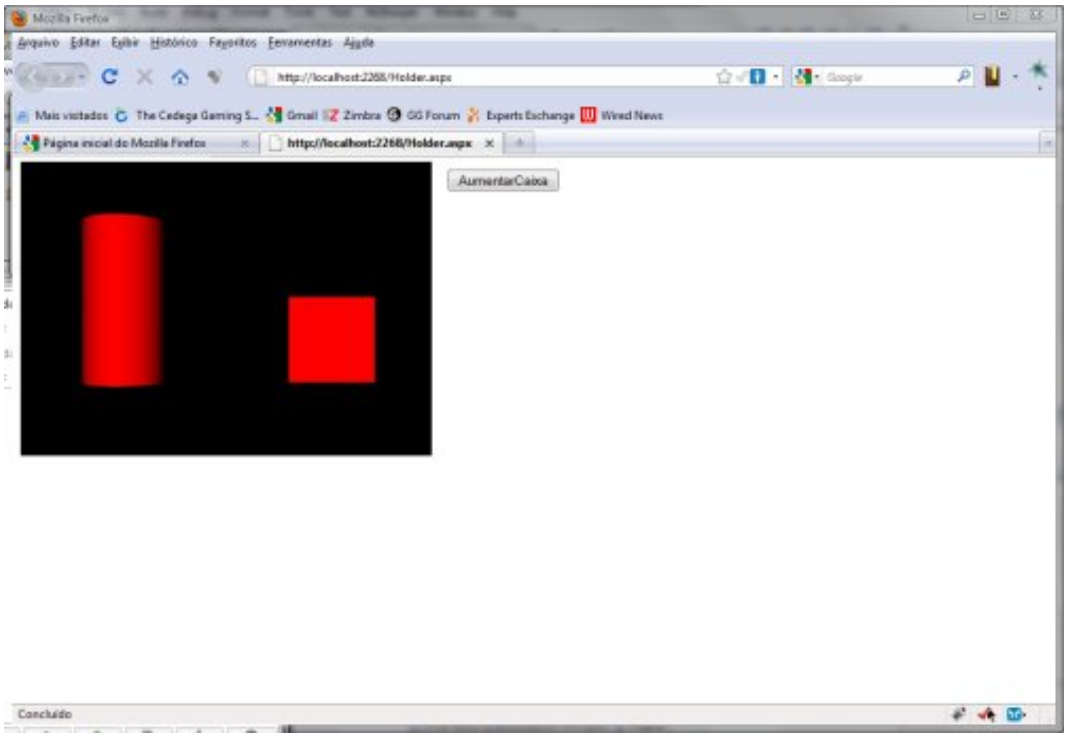

Figura 4.8: Figura simples antes do clique.

sultados. Esta redução de código não só facilita o processo de desenvolvimento inicial, mas também traz uma série de vantagens para aqueles que precisam desenvolver estes códigos. Por terem um custo menor no desenvolvimento, o risco das aplicações diminui. Assim sendo, o desenvolvedor pode arriscar mais pois o custo de um erro será menor. Outro aspecto positivo é a possibilidade de liberar os desenvolvedores a se focar nos objetivos de suas aplicações. Isto ocorre porque se os desenvolvedores não precisarem lidar com as camadas básicas, que são trabalhosas e complexas, ele poderá se focar nas funcionalidades desejadas em seu software.

A aplicação de testes nos mostrou, mesmo que de forma empírica, que este toolkit pode facilitar o processo de desenvolvimento de aplicações que tenham como requisito uma das funcionalidades propostas.

O DWeb3D pode se apresentar como uma ferramenta útil para facilitar o desenvolvimento de aplicações X3D complexas e se adequa a princípios citados anteriormente, onde um toolkit como estes, ao facilitar a vida do desenvolvedor, facilita os processos de desenvolvimento de forma generalizada e assim promove a utilização da tecnologia.

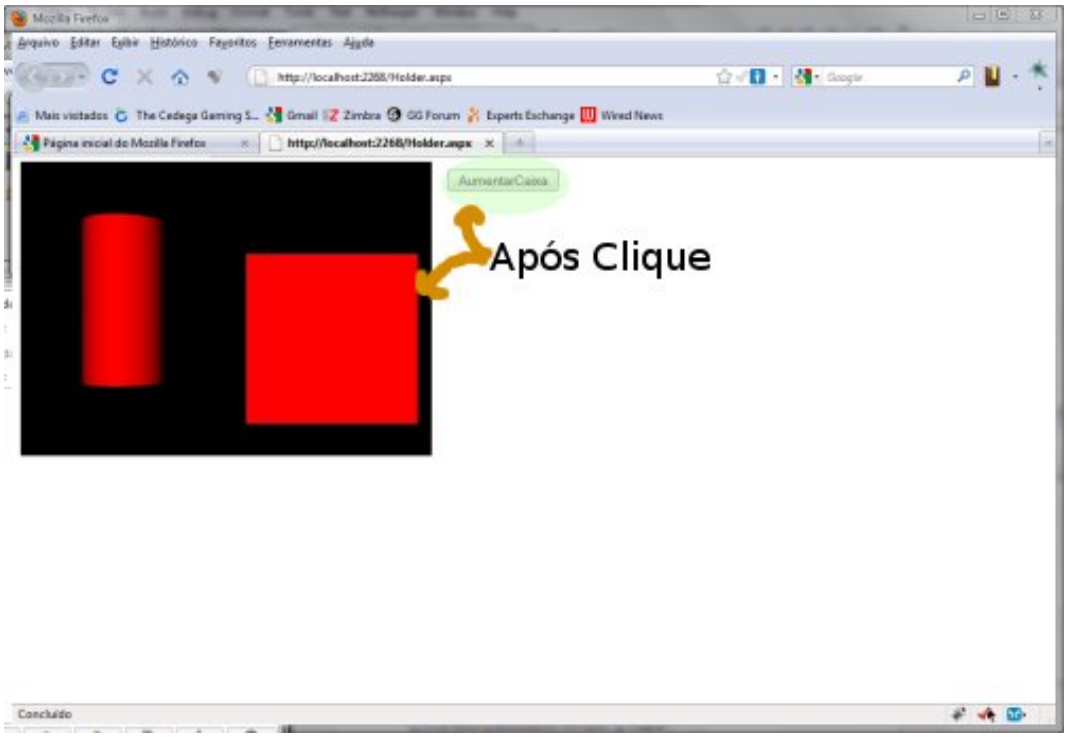

Figura 4.9: Figura simples após o clique