UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Ambiente 3D Web para visualização de modelos

Ricardo Jorge de Jesus Rodrigues Pesqueira

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

(2º ciclo de estudos)

Orientador: Prof. Doutor Frutuoso Silva

Covilhã, Outubro de 2012

Resumo

Os ambientes 3D são bastante utilizados na simulação de cenários por diversos motivos. Estes permitem simular cenários de forma simples e rápida, sendo por isso uma excelente ferramenta de trabalho. Com o aparecimento da tecnologia *WebgGL* que nos permite ter gráficos 2D e 3D no *browser* sem necessidade de *plugins*, tornou-se mais simples o desenvolvimento de aplicações gráficas para a Web. Com este trabalho pretendeu-se avaliar as capacidades do *WebGL* através do desenvolvimento de um ambiente 3D Web para visualização de modelos 3D em cenários *indoor*. O principal objectivo foi a criação de um ambiente 3D Web interactivo que permita ao utilizador interagir com o cenário em tempo real, por exemplo através da navegação no cenário ou da alteração das propriedades da iluminação existente. Além disso, ter ainda a possibilidade de visualizar os seus próprios modelos tridimensionais (construídos num qualquer software de modelação), desde que armazenados no formato OBJ.

Pretendeu-se ainda criar um algoritmo de iluminação global baseado no algoritmo de *ray tracing*, que permitisse o cálculo de reflexões entre os modelos do mundo virtual.

Palavras-chave

WebGL, visualização de modelos OBJ, iluminação interactiva, aplicação Web, *Three.js*, *ray tracing*

Abstract

3D environments are widely used to simulate scenarios for various reasons. They allow simulation of scenarios quickly and easily, therefore being a great simulation tool. With the appearance of the *WebGL* technology that allows having 2D and 3D graphics in the browser without plugins, it became easier to develop graphical applications for the Web. This work aims to assess the capabilities of *WebGL* through the development of a 3D Web environment to visualize models in indoor scenarios. The main objective was to create an interactive 3D environment that allows the user to interact in real time with the scenario, for example by walking through the scenario or changing properties of the lights. Besides, having the possibility to view his own models (built in any modeling tool) stored in OBJ format. It was intended to also create a global illumination algorithm based on ray tracing algorithm, enabling the calculation of reflections between the models of the virtual world.

Keywords

WebGL, viewing OBJ models, lighting interactive Web application, Three.js, ray tracing

Índice

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

WebGL	Web Graphics Library
ES	Embbeded System
API	Application Programming Interface
SL	Shading Language
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
CPU	Central Processing Unit
CG	Computação Gráfica
HTML	HyperText Markup Language
JSON	Javascript Object Notation
XML	eXtensible Markup Language

Capítulo 1

Introdução

Nos últimos anos, as tecnologias Web têm tido um grande desenvolvimento, principalmente as que são independentes da plataforma [1]. Por exemplo, na área da computação gráfica, uma dessas tecnologias é o *WebGL* (*Web Graphics Library*) [2]. Introduzido pela *Khronos*, o *WebGL* é um *standard* que permite aos *browsers* utilizarem a linguagem *OpenGL ES* (*OpenGL Embbeded System*). No fundo, o *WebGL* é uma API 3D de baixo nível, escrita em *Javascript*, que permite o desenvolvimento de aplicações gráficas tridimensionais. Este novo *standard* foi lançado em Dezembro de 2009 e conta já com vários projectos nas mais diversas áreas de simulação 3D, como por exemplo em [3], onde é apresentada uma *framework* de simulação que pode ser aplicada às áreas de animação, jogos e até medicina. Em [4], podemos até encontrar novas ideias para a educação. Também na referência [5], é apresentada uma análise sobre *standards* emergentes como o *WebGL* e o *HTML5* que removeram muitas das limitações até agora existentes, permitindo transformar a Web numa excelente plataforma de aplicações gráficas.

Direccionado para programadores de *OpenGL* familiarizados com *Javascript*, o *WebGL* permite o acesso à API (Application Programming Interface) do *OpenGL ES* que acede directamente à GPU (Graphics Processing Unit) para renderizar o resultado. Apesar de ser dependente do desempenho do *Javascript* [6], evita que o utilizador tenha que usar *plugins* no *browser* escolhido, pois é um *standard* já implementado na maioria dos *browsers*, ao contrário das tecnologias existentes anteriormente ao *WebGL* (como por exemplo Adobe Flash [7], VRML e X3D [8]). Sendo que, neste momento, apenas não é suportado pelo Internet Explorer. Mozilla Firefox, Google Chrome, Opera, são alguns dos *browsers* que já suportam este *standard* implementado nativamente nas suas distribuições. No entanto, podem ser necessárias alterações nas propriedades do *browser* antes da visualização de conteúdo 3D [9].

1.1 - Objectivos

Neste trabalho pretendeu-se avaliar as potencialidades do *WebGL* para o desenvolvimento de aplicações gráficas 3D. Para isso, o objectivo principal era a criação de um ambiente 3D interactivo que permitisse a visualização de modelos 3D no *browser*. Além disso, pretendia-se ainda que o ambiente fosse interactivo e permitisse a configuração das condições de iluminação, pois estas são fundamentais na visualização de modelos em computação gráfica.

Neste contexto, o ambiente a desenvolver teria de suportar modelos 3D num formato *standard*, como por exemplo o formato OBJ da *Wavefront* [10].
Por fim, pretendeu-se ainda o desenvolvimento de um algoritmo de iluminação global para o cálculo das reflexões entre os modelos do cenário, o qual se baseou no algoritmo de *ray tracing* [11].

1.2 - Metodologia e resultados

De modo a atingir os objectivos descritos, seria necessário obter conhecimentos sobre o *WebGL* (familiarização com a API e as suas potencialidades); depois avaliar e testar algumas *frameworks* mais populares para o *WebGL*, e identificar as suas potencialidades na visualização de um ambiente 3D.
Este trabalho resultou no desenvolvimento de uma aplicação Web interactiva em *WebGL* que corre em qualquer *browser* que suporte *WebGL*. Esta aplicação permite a visualização de modelos 3D no formato OBJ e a interacção com os mesmos. Por exemplo, permite navegar no cenário, alterar as propriedades da luz, aplicar uma textura a um modelo ou aplicar as principais transformações geométricas ao modelo em visualização.
A aplicação desenvolvida encontra-se disponível para ser testada em http://webx.ubi.pt/~a20711/.

1.3 - Estrutura do documento

Neste capítulo é feita uma introdução ao tema, apresentam-se os objectivos e a metodologia usada, bem como, é apresentada a estrutura da dissertação.
No capítulo 2 é apresentado o estado da arte, primeiro com uma breve referência à especificação do *WebGL* e as suas semelhanças com o *OpengI*, e depois é feita uma apresentação das *frameworks* mais usadas actualmente na construção de ambientes 3D para a Web.
No capítulo 3 é apresentada uma análise comparativa das *frameworks* testadas e os testes efectuados, incluindo pontos fortes e fracos. Por fim é apresentada a *framework* escolhida para o desenvolvimento do ambiente 3D.
No capítulo 4 apresenta-se a aplicação desenvolvida, com a descrição de todas as suas funcionalidades e descrição do seu desenvolvimento, conseguido através da *framework* escolhida.

No capítulo 5 apresenta-se o algoritmo criado com base no algoritmo de *ray tracing* para o cálculo das reflexões, uma breve introdução à linguagem utilizada, implementação e testes, e conclusões sobre os resultados obtidos.
Por último, no capítulo 6, são apresentadas algumas conclusões, e propostas de trabalho futuro.

Capítulo 2

Estado da Arte

2.1 - Especificação *WebGL*

Como o *WebGL* é baseado no *OpenGL ES* 2.0 [12], o *pipeline* de renderização é semelhante ao *pipeline* de renderização de *shaders* do *OpenGL* (ver Fig. 1), com algumas excepções, por exemplo, no *WebGL* não existem *display lists* ou "*glBegin"* e "*glEnd"*.

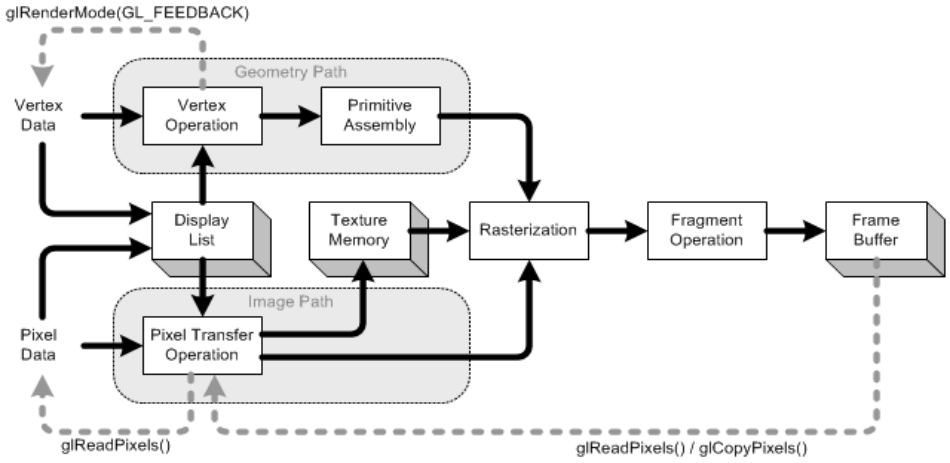

Figura 1 - *Pipeline OpenGL* [13]

Numa primeira fase, são utilizados *buffers* que contêm as posições dos vértices, e as suas propriedades, tais como normais, cores e texturas dos modelos virtuais. Esta parte faz parte da API, escrita em *Javascript*, com métodos preparados para criar os referidos *buffers* [14], e também definir e aplicar as respectivas matrizes de transformação para cada modelo. Depois, de modo a serem definidas as cores da imagem final, o *WebGL* permite que sejam carregados, compilados e atribuídos aos modelos virtuais, programas escritos na linguagem *OpenGL SL* (*Shading Language*), que depois são executados na GPU (*Graphics Processing Unit*), de modo a que as operações de *shading* sejam executadas em paralelo.

Na Fig. 2 pode-se verificar, assinalado a laranja, os dois tipos de *shaders* à disposição do programador: o *vertex shader* e o *fragment shader*. Nas subsecções seguintes apresentamos as diferenças entre os dois tipos de *shaders* programáveis mencionados.

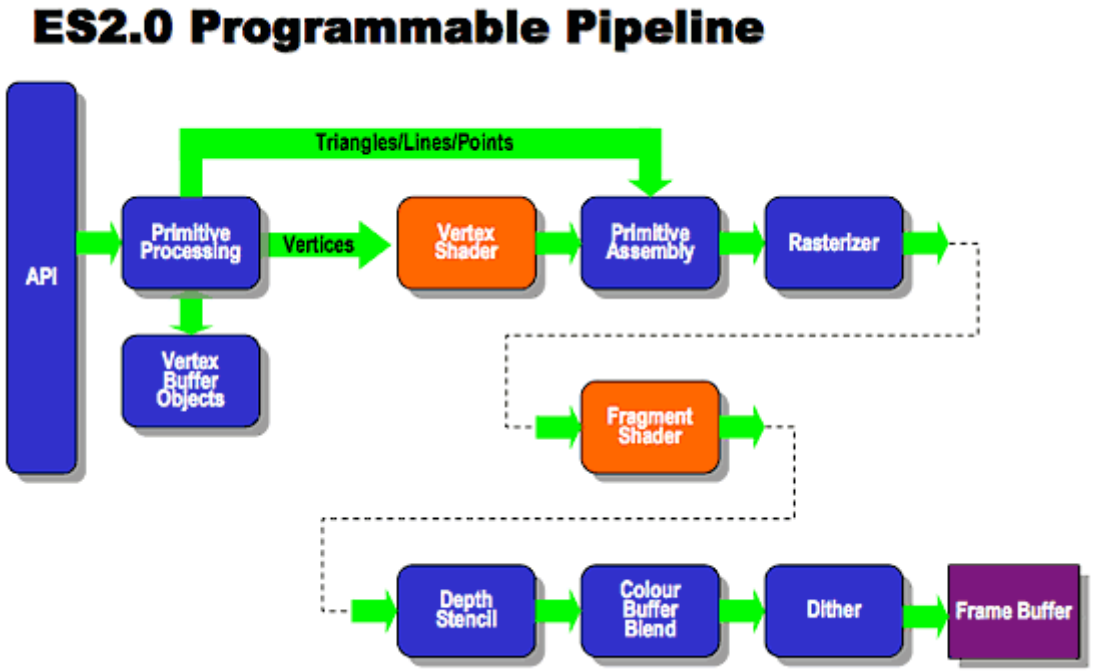

Figura 2 - *Pipeline WebGL* (*OpenGL ES 2.0*) [15]

2.1.1 - *Vertex Shader*

Este tipo de *shader* providencia um método geral de programar os vértices de uma dada geometria. Como podemos ver na Fig. 2, a API *WebGL* passa como parâmetro os dados dos vértices (*Vertex Buffer Objects*), que são:

- *Attributes*: Armazena informações sobre cada vértice, que podem ser, por exemplo, coordenadas de textura (*UV's*) ou as normais.
- *Uniforms*: Permitem armazenar dados constantes para poderem ser utilizadas nos cálculos (por exemplo matrizes).
- *Samplers:* Um tipo específico de *uniforms* que armazenam texturas.
- *Shader program:* Código fonte do programa que descreve as operações que pretendemos efectuar sobre os vértices.

Os resultados do processamento são denominados *varyings*. Estas variáveis permitem-nos passar a informação do *vertex shader* para o *fragment shader* (por exemplo, para calcular a normal da face com base na normal dos seus vértices). Para além disso, no código fonte do programa, é necessário usar a variável (já pré-declarada na linguagem *GLSL*) denominada *gl_position* que define a posição final do vértice em coordenadas ecrã.

2.1.2 - *Fragment Shader*

Depois do *vertex shader*, na fase de *primitive assembly* e *rasterizer* (ver Fig. 2) são gerados os fragmentos associados aos triângulos, quadrados, ou linhas, passamos então para o *fragment shader* para podermos definir como queremos tratar os mesmos.
Um a um, todos os fragmentos vão ser tratados por este programa, recebendo os seguintes parâmetros de entrada:

- Variáveis *varying:* Variáveis calculadas no *vertex shader*.
- *Uniforms*: Que armazenam dados constantes para serem utilizados no programa.
- *Samplers*: Um tipo especifico de *uniforms* que armazena texturas para serem utilizadas.
- *Shader program*: Código fonte do programa que descreve as operações a serem realizadas para cada fragmento.

O objectivo do programa *fragment shader* é o de calcular o valor da cor final de cada pixel (i.e. de cada fragmento) que é armazenada na variável *gl_FragColor*.

2.2 – *Frameworks*

É importante notar que, a API do *WebGL* permite o acesso a métodos de baixo nível, de modo a ser possível o acesso à GPU. Apesar de ser uma grande vantagem, acaba por requerer um maior nível de conhecimento por parte do programador, pois terá primeiro que implementar por ele mesmo algumas funcionalidades mais comuns do *OpenGL* [16].
Recentemente, têm surgido várias *frameworks* para o *WebGL*, em conjunto com o já referido *Javascript* em constante crescimento (recentemente foram introduzidos novos motores de *Javascript*, como por exemplo: *Google V8*, *Apple SquirrelFish* e *Microsoft Chakra*) [17] o que tornou possível a criação da próxima geração de aplicações 3D para a Web. Estas *frameworks* têm a vantagem de mascarar os níveis mais baixos da API do *WebGL,* facilitando o trabalho ao programador. *Shaders* pré-programados, primitivas, editores gráficos, possibilidade de carregar modelos em vários formatos, ou seja, modelados utilizando softwares de modelação (como por exemplo, o *Blender* [18]), são apenas algumas das vantagens de utilizar este tipo de API's, o que torna o desenvolvimento mais fácil e rápido. Além disso, estas *frameworks* providenciam funções para efectuar as operações mais básicas, como por exemplo, uma simples rotação de um objecto. As *frameworks* mais avançadas têm também funções para efectuar animações, iluminação e sombras, nível de detalhe, detecção de colisões ou selecção de objectos no ambiente 3D. Nesta secção apresentamos o estado da arte em termos das *frameworks* mais relevantes para o *WebGL* neste momento [19].

2.2.1 - *C3DL*

C3DL (*Canvas 3D JS Library*) é uma *framework* em *Javascript* que fornece funcionalidades 3D básicas para programadores Web que desejem desenvolver conteúdos 3D para o *browser*, simplificando a API de baixo nível do *WebGL*, bem como estruturas matemáticas necessárias (i.e. matrizes de transformação e respectivas operações) [20].

O C3DL tem as seguintes características:

- Carregamento de modelos guardados no formato *Collada* [21].
- Selecção de objectos no ambiente 3D utilizando o rato.
- Sistema de iluminação.
- Sistema de câmara.
- Sistema de partículas.

Possui também um sistema de efeitos que possibilita a troca de *shader* a ser aplicado para alterar a sua aparência. Com este sistema, permite que sejam aplicados os seguintes efeitos:

- *Cartoon*
- Escala de cinza
- Cor sólida
- Sépia

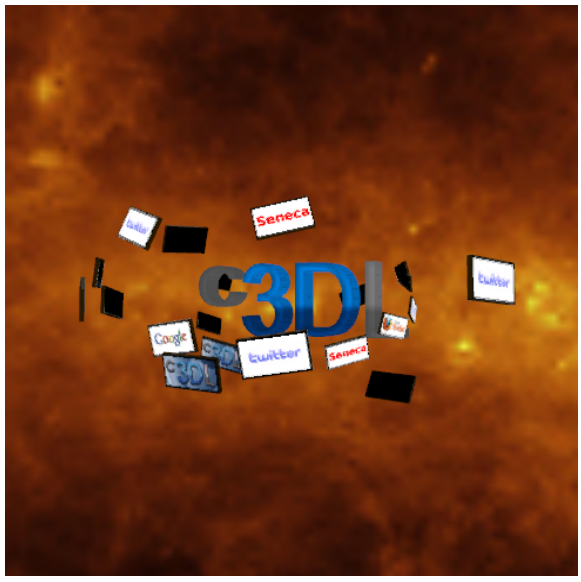

Figura 3 - Exemplo *C3DL* [22]

2.2.2 – *CopperLicht*

Desenvolvido pela Ambiera [23], *Copperlicht* [24] é uma *framework* especialmente criada para o desenvolvimento de jogos e aplicações 3D no *browser*. Esta API tem a vantagem de incluir um completo editor 3D, de nome *CopperCube*, que suporta todas as características necessárias para criar o ambiente virtual / jogo 3D.

Ao contrário de outras *frameworks*, os modelos tridimensionais são compilados para um pequeno ficheiro binário, reduzindo consequentemente o tempo necessário no carregamento da página, possibilitando ao *browser* começar a renderização o mais rápido possível. Isto é conseguido utilizando o *Coppercube*, importando os ficheiros 3D, e publicando o cenário como ficheiro *copperlicht* (.ccbjs). São suportados vários formatos de ficheiros 3D, tais como: *3ds, obj, x, lwo, b3d, csm, dae, dmf, oct, irrmesh, ms3d, my3D, mesh, lmts, bsp, md2, stl, ase, ply, dxf, cob* e *scn*.

Na utilização dos modelos, estes devem ser atribuídos a um grafo de cena hierárquico, que define a ligação entre todos os elementos. Por exemplo, num ambiente 3D com um automóvel, um condutor, e uma casa, o condutor pode ser atribuído ao mesmo nó do grafo que o automóvel, o que possibilita que todas as transformações aplicadas ao automóvel, podem ser também aplicadas ao condutor, facilitando a organização do ambiente 3D. Para além destas vantagens, temos também as seguintes características:

- Na criação do ambiente 3D, existem também vários materiais e *shaders* pré-definidos e prontos a usar bem como caminhos (*paths*) e *splines*. Estes *shaders* e materiais podem também ser personalizados no *CopperCube*.
- Preparado para a renderização ou animação em cenários 3D complexos.
- Suporte para pré-cálculo optimizado da iluminação bem como de transparência.
- "*Billboards*" – imagens sempre viradas para o utilizador.
- "*Skyboxes*" – técnica de criação de um fundo para o ambiente 3D.
- Animação por esqueleto e de texturas, também facilmente importável para a ferramenta *CopperCube*.
- Redução automática de re-renderização: em vez de redesenhar todo o cenário 3D em todas as *frames* (o que se pode tornar lento em alguns sistemas mais antigos), esta API tenta redesenhar o cenário apenas quando é absolutamente necessário, por exemplo, apenas quando a câmara muda de posição ou o cenário é alterado.
- Sistema de detecção de colisão e de selecção de objectos no ambiente 3D.
- Gestão de "*input*": Visto que a entrada recebida pelo rato e teclado não funciona da mesma maneira em todos os *browsers*, *Copperlicht* implementa um nível extra de abstracção de modo a permitir compatibilidade.

Apesar de todas as vantagens desta *framework*, a maior parte destas características são disponibilizadas apenas utilizando a ferramenta *CopperCube*. No entanto, esta ferramenta

está apenas disponível para utilização gratuita durante 14 dias, sendo depois necessário adquirir uma licença comercial.

Figura 4 - Exemplo *Copperlicht* [25]

2.2.3 – *PhiloGL*

Ao contrário das *frameworks* mais comuns, esta *framework* tem como objectivo disponibilizar ao programador as ferramentas necessárias para construir aplicações 3D avançadas de visualização de dados. Existe grande variedade de tutoriais e apresenta um desenvolvimento activo desde o seu aparecimento. Para além da informação disponível sobre como utilizar a API, existem também tutoriais sobre *WebGL* podendo, deste modo, ser mais fácil entender o relacionamento da API com o respectivo nível inferior. As suas funcionalidades principais são:

- Módulo de funções matemáticas (vectores, matrizes e respectivas operações).
- Boa abstracção do nível do *WebGL*.
- Algum suporte também para o desenvolvimento de jogos, no entanto o suporte para animação é básico, não havendo, por exemplo, possibilidade de animação por *keyframes*.
- Possui vários módulos para serem utilizados na construção de apresentações gráficas 3D.
- *Open Source.*

- Módulo para tratamento de eventos (teclado e rato).
- Módulo de pós-processamento de imagem (ainda em fase inicial).
- Módulo "*Workers*" que permite que vários *scripts* possam ser executados ao mesmo tempo.

Em suma, esta *framework* tenta reduzir, através dos vários módulos existentes, a programação necessária para que possamos visualizar rapidamente o resultado final. Podemos ver na Fig. 5 um exemplo da visualização de dados de tráfego aéreo.

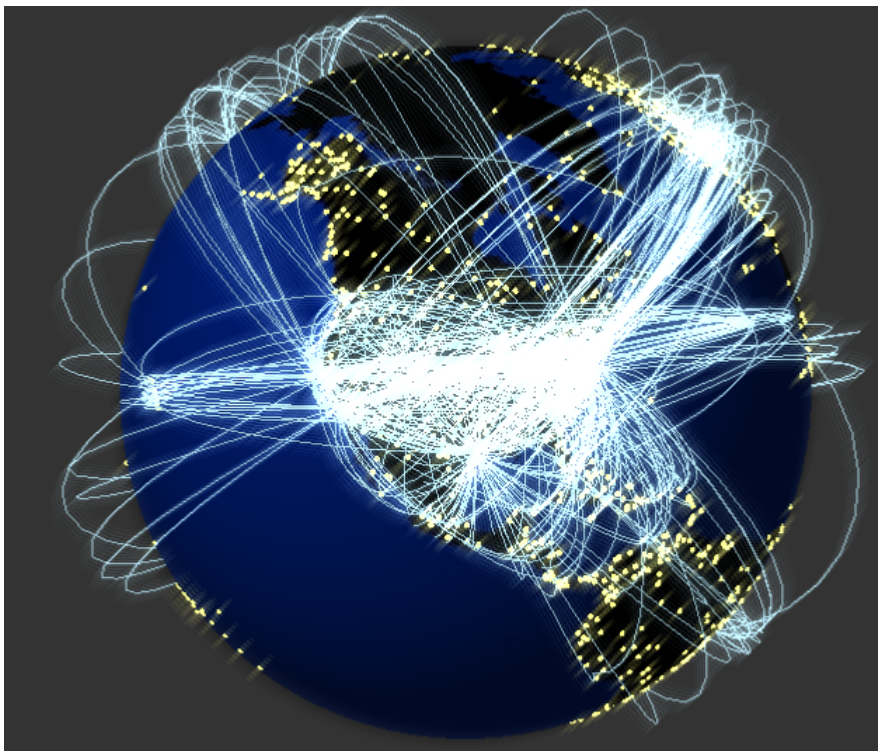

Figura 5 - Exemplo *PhiloGL* de visualização de dados sobre tráfego aéreo [26]

2.2.4 - *GLGE*

Esta *framework* procura abstrair os programadores do contexto do *WebGL*, e simplificar o processo de programação para criar conteúdo o mais completo possível para a Web.
O *GLGE* utiliza o formato em XML (*eXtensible Markup Language*) para definir malhas, animações, materiais e qualquer outro elemento que possa fazer parte do ambiente 3D. Os cenários do *GLGE* são geridos através de um grafo de cena, em que cada objecto que pertence ao cenário é colocado em um nó respectivo do grafo.

De momento, o *GLGE* providencia o melhor conjunto de características, quando comparado às outras *frameworks* existentes, para o desenvolvimento de jogos em *WebGL*. Apesar disso, ainda não se compara a motores de jogo de *desktop* [27].
Existe também a vantagem de se poder utilizar facilmente o software *Blender* em conjunto com o *GLGE* através de um *script* que permite ao *Blender* exportar completamente um cenário criado (completo com texturas, iluminação, e outras características) para o formato XML do *GLGE*, que contém o grafo de cena. Existem várias versões de *scripts* para este efeito, como por exemplo o *script* implementado em [28].

Em resumo, as características mais importantes do *GLGE* [29] são:

- Animação por *keyframes*.
- Sistema de iluminação com luz direccional, luz de foco e pontual.
- Mapeamento de normais.
- Animação por esqueleto dos modelos, facilmente importável.
- Carregamento de modelos através do formato *Collada*. É possível carregar animações neste formato também.
- Mapeamento *Parallax* – tipo de mapeamento de normais, mais detalhado.
- Renderização de texto.
- Sistema de nevoeiro.
- Sistema de profundidade de sombras.
- Selecção de objectos (baseado em *shader*).
- Reflexões / Refracções e mapeamento do ambiente 3D.

Outra característica do *GLGE* é que tem implementado na própria API vários *shaders* de uso comum que são "escondidos" do programador para que não seja necessário ter isso em conta. Podem ser utilizados vários tipos de luz, normais, sombras, nevoeiro, etc. Isto é a principal vantagem do *GLGE*, pois liberta o programador para poder criar conteúdo, jogos e aplicações com o *WebGL*.

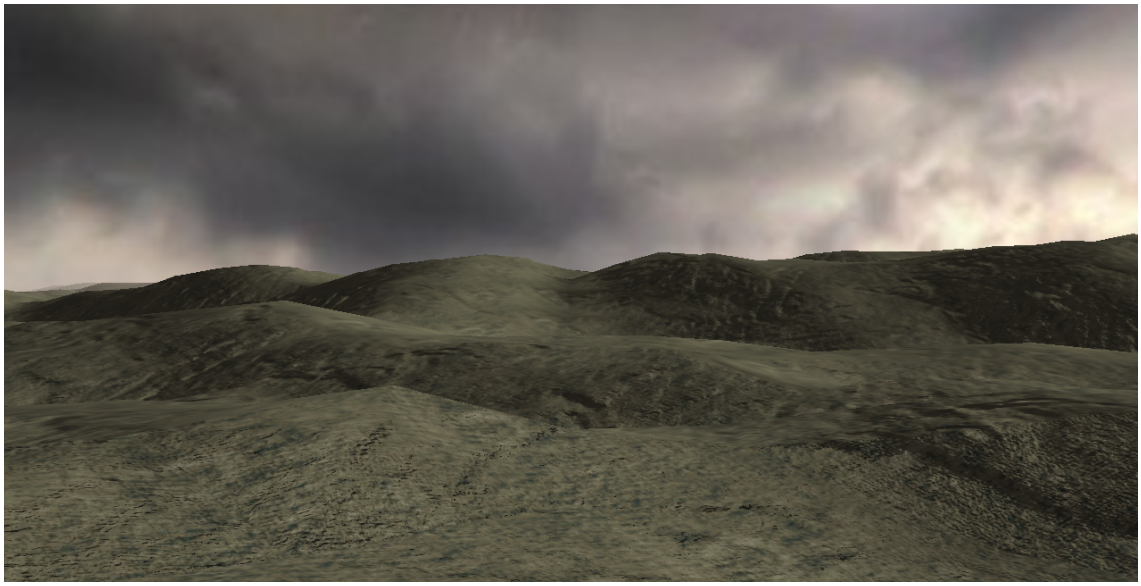

Figura 6 - Exemplo de um ambiente renderizado com *GLGE* [30]

2.2.5 – *O3D*

Originalmente desenvolvida pela Google como um *plugin* para *browser*, com o aparecimento do *WebGL* esta API foi então implementada como nova *framework*, deixando de ser necessário o *plugin* e passando a ser uma biblioteca *Javascript* construída no topo do *WebGL* convertendo as funcionalidades existentes no *plugin*, para as existentes no *WebGL*.
Sendo também uma API *open-source*, qualquer programador pode identificar erros e propor correcções ou propor métodos para implementar novas funcionalidades.
O O3D fornece um conversor de formato *Collada*, que pode ser utilizado para importar ficheiros neste formato, e convertê-los no formato reconhecido pelo O3D. Para os outros formatos, existe a possibilidade do programador escrever o seu próprio conversor. O O3D também fornece uma API para utilizar um grafo de cena, à semelhança de outras *frameworks*.
Uma funcionalidade bastante útil que está presente nesta API é a possibilidade de alterar o código da aplicação e poder visualizar a mudança em tempo real (i.e. remover um modelo).
Outras funcionalidades principais são as seguintes:

- Sistema de partículas.
- Sistema de física.
- Vários materiais (incluindo Lambert, Phong, etc.).
- Diferentes tipos de luzes (direccional, pontual, ambiente, etc.).
- Sistema de câmara com opção ortogonal ou perspectiva, e também com controladores de navegação através do teclado e rato já definidos.
- *Shaders* personalizados (por exemplo sombras e nevoeiro), e editáveis em tempo real.
- Sistema de detecção de colisão e animação.

Em resumo, esta API permite criar aplicações 3D bastante apelativas visualmente. No entanto, o ponto fraco continua a ser a documentação algo limitada, pois baseia-se em exemplos que foram convertidos directamente do *plugin* para *WebGL*, havendo ainda funcionalidades por converter. Para além disso, a última actualização desta *framework* foi em 2010 [31]. Um exemplo de um jogo interactivo (ver Fig. 7) já construído com a base em *WebGL* pode ser encontrado em [32].

Figura 7 - Exemplo de um jogo renderizado através da *framework O3D* [32]

2.2.6 – *SpiderGL*

A filosofia presente por detrás desta *framework* é fornecer estruturas típicas e algoritmos para renderização em tempo real para serem utilizados nas aplicações 3D na Web, sem forçar o programador a seguir um certo paradigma (como por exemplo, grafos de cena), ou prevenir que aceda ao nível mais abaixo, ou seja, o *WebGL* [33]. Segundo descrito pelos autores em [34], a API *SpiderGL* foi construída tendo em conta 3 qualidades fundamentais:

- Eficiência: Com Javascript e *WebGL*, a eficiência não é apenas a eficácia dos algoritmos, mas a habilidade de encontrar o mecanismo mais eficiente a ser implementado, como por exemplo o carregamento assíncrono de dados e passagem de parâmetros para os *shaders*, sem sobrecarga do CPU (*Central Processing Unit*).
- Simplicidade e pouco tempo de aprendizagem: Os utilizadores devem ser capazes de reutilizar os conhecimentos sobre o assunto e tirar vantagem da *framework* o mais rápido possível. Por esta razão, o *SpiderGL* tenta evitar um nível demasiado alto de abstracção, oferecendo ao utilizador, por exemplo correspondências entre as funções comuns em *OpenGL*.

- Flexibilidade: Esta API não tenta "esconder" funções do *WebGL* implementadas nativamente, em vez disso, tenta fornecer funcionalidades de alto nível que preenchem as necessidades mais comuns de um programador de CG (Computação Gráfica), o que torna a utilização desta API e do *WebGL* ao mesmo tempo, praticamente transparente.

Outra característica interessante sobre esta API é a possibilidade de utilizar uma aplicação Web, de nome *MeShade* que possibilita ao utilizador carregar um modelo 3D, criar um *shader* personalizado, e exportar em código JSON (*Javascript Object Notation*) e HTML (*HyperText Markup Language*) que irão criar uma página Web que irá fornecer visualização interactiva do objecto utilizando o *shader* programado. Na Fig. 8 podemos visualizar uma demonstração desta *framework*.

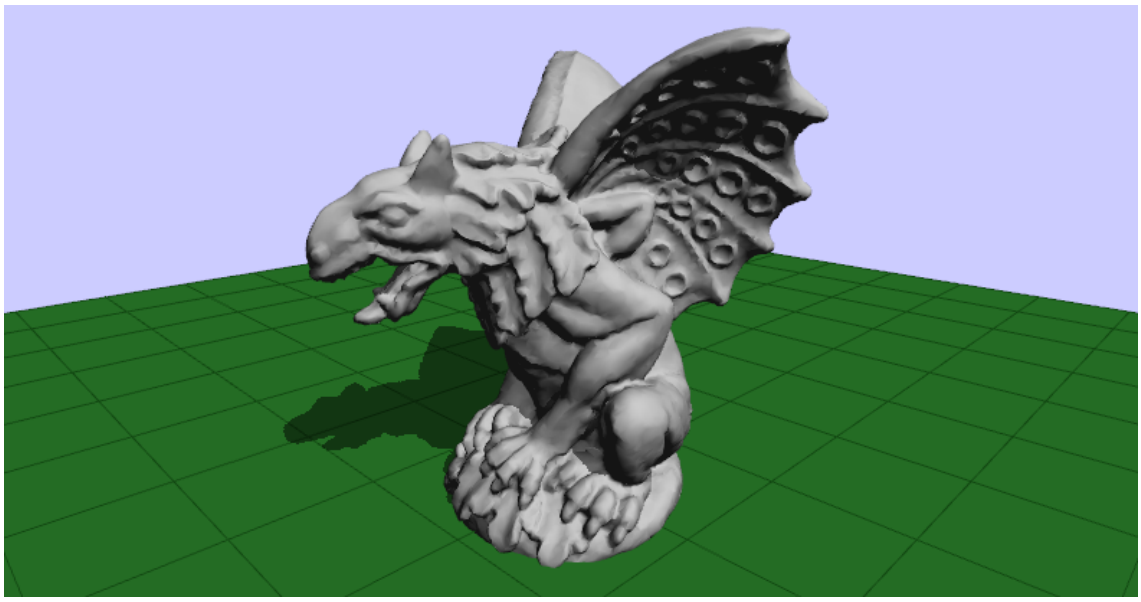

Figura 8 - Demonstração da *framework SpiderGL* com a visualização de um modelo com sombra [35]

2.2.7 – *Three.js*

Dotado de um motor 3D com muitas funcionalidades e um nível muito alto de abstracção, esta *framework* é a mais popular do momento não só devido à grande facilidade de utilização e aprendizagem, mas também pelo facto de ser constantemente alterada com implementação novas funções. Tem já algumas utilizações em jogos [36], investigação e aplicações 3D [37].
Esta *framework* tem a vantagem de o seu desenvolvimento ter atingido um dinamismo incrível neste momento, actualmente cerca de 50 programadores trabalham em conjunto com o autor [38]. No entanto, a documentação [39] não cobre ainda todas as funcionalidades (apesar de estas estarem a serem progressivamente cobertas a cada novo lançamento), sendo que a

aprendizagem desta *framework* é conseguida apenas através dos exemplos disponíveis, o que acaba por não ser uma desvantagem visto o n.º de exemplos ser suficiente para qualquer necessidade. Algumas características são:

- Incluídas bibliotecas matemáticas com todos os tipos de dados necessários, e praticamente todas as operações possíveis implementadas (i.e. matrizes e vectores).
- Objectos pré-definidos (primitivas) e *parsers* de geometria.
- Vários materiais (incluindo Lambert, Phong, etc.).
- Diferentes tipos de luzes (direccional, pontual, ambiente, etc.).
- Sistema de câmara com opção ortogonal ou perspectiva, e também com controladores de navegação através do teclado e rato já definidos.
- *Shaders* personalizados (por exemplo sombras e nevoeiro).
- Sistema de detecção de colisão e animação.
- Suporte a carregamento de modelos através do formato *Collada*, *Wavefront Obj* [10] e JSON (este último segundo uma estrutura implementada pelos autores) [40].
- Grafo de cenário, sendo possível adicionar e remover objectos dinamicamente.
- Animação por *keyframes*.

Novas funcionalidades aparecem bastante frequentemente, no entanto, muitas vezes a documentação da API deixa de estar actualizada e um projecto construído em uma versão anterior pode necessitar de ser revisto devido a mudanças nos parâmetros das funções.
Existem neste momento várias demonstrações de trabalhos realizados através desta *framework*, em que é possível visualizar as suas capacidades de processamento de materiais e iluminação mais complexas, como podemos ver na Fig. 9.

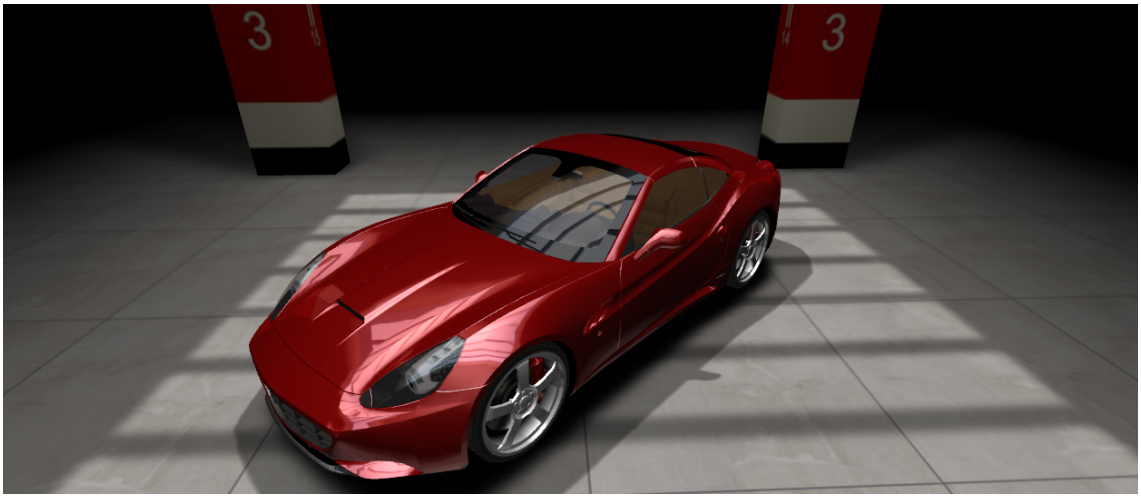

Figura 9 - Aplicação para visualização de modelos construída com a *framework Three.js* [41]

Capítulo 3

Escolha da *framework*

3.1 - Metodologia de investigação

Depois de efectuar uma aprendizagem com recurso a tutoriais online [42] e tendo conhecimento da existência de todas estas *frameworks*, passamos então à fase de testes e desenvolvimento prático sobre o *WebGL* de modo a explorar funcionalidades das *frameworks*. Nesta fase, testamos as *frameworks* que apresentam características mais promissoras em termos de iluminação e processamento de sombras, com vista a saber qual a melhor para posterior visualização de um cenário 3D com a iluminação interactiva que iremos construir.
Para isso, foi modelado utilizando o software *Blender* (versão 2.59), um pequeno cenário de teste como mostra a figura 10.

Figura 10 - Renderização no *Blender* do cenário de teste

O cenário de teste é composto pelos seguintes elementos (Total de cerca de 28000 vértices):

- Um modelo de uma mesa, com uma textura aplicada (cerca de 9000 vértices).
- Um modelo de um estetoscópio com dois materiais aplicados (cerca de 8000 vértices).
- Duas formas primitivas, um anel ("*torus*") e uma esfera com um material com componente especular e uma cor atribuída.
- Um plano (representando o chão) com uma textura aplicada e repetida.

Nas secções seguintes, apresentamos o trabalho efectuado na tentativa de importar este cenário para o *WebGL* e verificar o respectivo resultado de renderização. Para este efeito, foi utilizado o IDE (*Integrated Development Environment*) *NetBeans* (versão 7.0) [43] em conjunto com o *Apache Tomcat* (versão 6.0) [44] de modo a podermos testar localmente as aplicações. O *browser* escolhido para renderizar o ambiente 3D foi o *Google Chrome* [45] e é importante referir que foi neste *browser* que conseguimos obter os melhores resultados, quer em termos de imagem final, como em termos de velocidade de construção da mesma.

3.2 - Testes Desenvolvidos

3.2.1 – Teste com *Three.js*

A primeira *framework* testada foi a *Three.js*. Com recurso a exemplos encontrados no repositório *Github* [46] [47] e respectivo estudo da API foi decidido utilizar o *add-on* que os autores da *framework* desenvolveram para o software *Blender*, que permite que seja exportado directamente do *Blender* para um ficheiro *JSON*, as informações do modelo desejado. Essas informações são:

- Escala
- Materiais
- Lista de vértices
- Lista de normais
- Lista de cores
- Lista de UV's
- Lista de faces

Depois dos modelos prontos, o programa de teste com o *Three.js* tem, resumidamente, a seguinte estrutura:

- Criar um cenário chamando o contexto do *WebGL*.
- Inicializar variáveis com posição da câmara, luzes e opções de detalhe das sombras.
- Carregar cada um dos ficheiros JSON e implementar a respectiva rotina de *callback* (executada quando é terminado o carregamento da geometria dos modelos) onde é possível alterar os atributos carregados a partir de ficheiro, como por exemplo escala e material.
- Renderizar o resultado.

No entanto, quando é feita a aplicação de texturas com geração de coordenadas UV automáticas no *Blender*, não é possível exportar para o ficheiro JSON estas coordenadas. Como resultado os objectos com texturas (no caso da mesa e do plano), não serem renderizados correctamente no *browser*. A solução para este problema foi realizar o mapeamento UV manualmente no *Blender*, executar *"save image"* depois de concluído, marcar as coordenadas da textura como *"UV"* em vez de *"generated"*, e guardar o ficheiro de projecto do *Blender* e ao exportar já obtivemos as coordenadas UV no ficheiro JSON.
Para além disso, a função de renderizar do *Three.js* é assíncrona, logo pode acontecer que a imagem da textura não seja carregada quando o objecto é renderizado. Para resolvermos este problema do tempo da renderização, utilizamos a função *setTimeout()* para esperar um certo tempo (em milisegundos) antes de chamar a função de renderizar.
A *framework Three.js* carrega então correctamente os materiais definidos no *Blender* (Phong, Lambert, etc.), bem como o tipo de *shading* (*flat* ou *smooth*).
Sobre as sombras, o sistema do *Three.js* apenas processa as sombras para um tipo de luz, que é o *SpotLight* (semelhante à luz direccional, mas a posição não é normalizada), apesar de permitir outros tipos de luzes, como luz direccional e ambiente, estas não incluem processamento de sombras. Isto pode mudar nas versões futuras desta *framework*. A implementação das sombras no *Three.js* é bastante personalizável e sensível às diferenças nos vários atributos, por exemplo, onde exactamente está o foco de luz, tamanho do volume de visualização da luz, e o elemento *bias* para evitar artefactos [48]. Por outras palavras, é necessário que o volume de visualização da luz englobe apenas o da câmara e nada mais. A ajuda dos autores nos fóruns da API foi determinante neste ponto. Alguns pontos importantes sobre o processamento das sombras no *Three.js* são:

- *renderer.shadowCameraNear*
- *renderer.shadowCameraFar*
- *renderer.shadowCameraFov*
- *renderer.shadowMapBias*

Apesar da alteração constante das versões da *framework*, há uma melhoria constante nos parâmetros das funções. A figura 11 mostra o resultado final obtido com a versão de 20/11/2011:

Figura 11 - Renderização do cenário de teste, utilizando *Three.js*

3.2.2 – Teste com *Copperlicht*

A segunda *framework* testada foi a *Copperlicht*. Foi efectuada uma breve aprendizagem sobre a API e o software comercial *CopperCube* (disponível em demo durante 14 dias), utilizando os tutoriais disponíveis no site oficial [49].

O editor 3D é facilmente instalável na plataforma Windows, e é bastante *"user-friendly"*, conforme mostrado na figura 12.

Utilizando *Coppercube*, foi importado o cenário de teste, no formato *Collada*. No entanto, no *CopperCube* não são carregadas as cores dos materiais, pois não existe suporte para a definição de cores sólidas directamente, foi então necessário definir uma textura com a cor, e aplicar aos objectos (i.e. seleccionar o objecto -> *Materials* -> Seleccionar textura). De seguida, nestes mesmos materiais, é necessário definir o tipo de sombras e comportamento do material. Em relação à iluminação, apenas está disponível um tipo de luz no *Coppercube*, que é o tipo pontual, para o qual é necessário definir no material se a luz é dinâmica, ou estática. No caso de estática, é necessário definir os parâmetros no menu "*Light Mapping*" de resolução da sombra, *subsampling*, nível de luz ambiente, e escolher o modo que pode ser entre sombras, cor difusa, iluminação global, ou apenas branco. No nosso caso, seleccionamos estática com uma resolução de 512x512, em modo de sombras com *subsampling* x16.

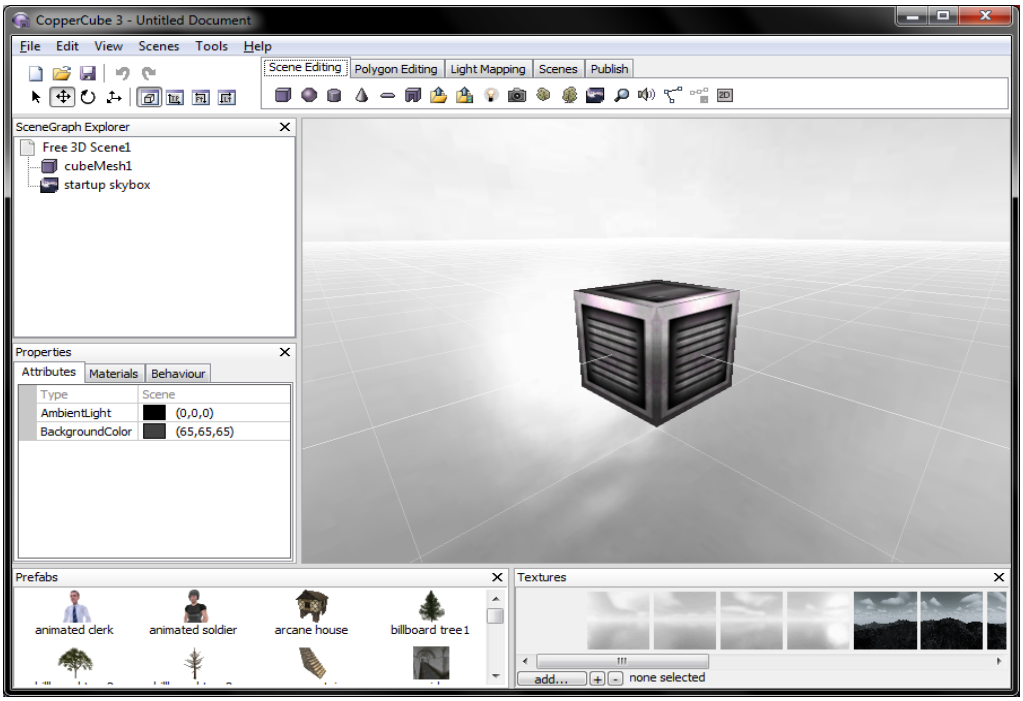

Figura 12 - *CopperCube*

É necessário depois clicar em "*calculate!*" para que sejam calculados os efeitos estáticos, conforme a luz pontual inserida. No caso de luz dinâmica, esta é calculada aquando a renderização. No entanto, são calculadas as sombras apenas no objecto em si, não existindo projecção das sombras em outros objectos, até à data de realização deste teste. Em relação aos materiais, podemos ajustar se queremos por exemplo um material reflectivo, sólido, etc. No entanto, não é possível ajustar os parâmetros destes atributos.

Por último, definimos a posição da luz (luz direccional, directamente acima dos objectos), e definimos um tipo de câmara. Esta API já tem várias câmaras pré-definidas, como por exemplo uma câmara na terceira pessoa (que segue sempre um certo objecto), uma câmara simples de visualização de modelos (que orbita á volta dos objectos), uma câmara livre (que o utilizador pode controlar com o rato e teclado), etc. No nosso caso, escolhemos uma câmara simples, de modo a olhar apenas fixamente para o nosso cenário de teste.

Tudo isto pode ser conseguido facilmente, sem ser necessário qualquer tipo de programação. No caso de ser necessário utilizar a API, esta tem uma boa documentação, e é também possível escrever os nossos próprios *shaders*.

Depois de termos o cenário desejado, renderizamos então o cenário através de *Tools -> Test as WebGL*. É também suportada a renderização com *Flash*, aplicação Windows (.exe) ou para Mac. Os objectos são então convertidos para o formato binário .ccbjs que tem a vantagem de ser um ficheiro bastante pequeno e rápido de transferir, mesmo em cenário mais complexos. A renderização final do nosso cenário de teste pode ser vista na figura 13.

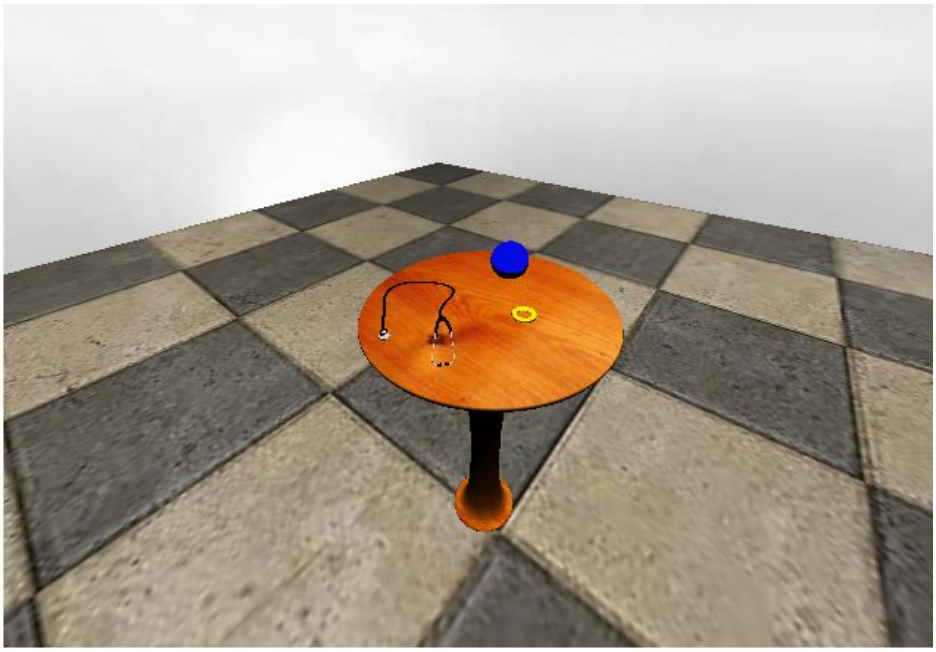

Figura 13 - Cenário de teste renderizado com o *CopperCube*

A utilização do *CopperCube* oferece um nível de abstracção do *WebGL* o mais alto possível, visto que, utilizando apenas o editor, é possível importar modelos e configurar todo o cenário pretendido. No entanto, a única maneira de carregar um modelo ou fazer os cálculos de iluminação mais avançados, é utilizando o *CopperCube*, que passa a necessitar de uma licença comercial ao fim de 14 dias.

3.2.3 – Teste com *GLGE*

A terceira *framework* testada foi a conhecida *GLGE*. Depois de uma breve familiarização com a API, utilizando os tutoriais [50] e [51]. Outros exemplos muito úteis podem ser encontrados também em [52].

O *GLGE* procura oferecer também um nível bastante alto de abstracção do *WebGL* ao programador e tem a vantagem de se poder utilizar ficheiros declarativos XML, com uma estrutura pré-definida, para carregar modelos e definir o ambiente 3D.

Foi optado por utilizar o formato *Collada* para exportar os objectos construídos no software *Blender*. A estrutura do ficheiro XML criado é a seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<glge>
<camera id="maincamera" loc_x="0" loc_y="20" loc_z="-35"/>
<scene id="mainscene" camera="#maincamera" >
                  <collada document="estetoscopio.dae" />
                  <collada document="esfera.dae" id = "esfera"/>
                  <collada document="mesa.dae" />
                  <collada document="ring.dae" />
                  <collada document="plane.dae"/>
<light id="spotlight"
loc_x="0" loc_y="200" loc_z="0" rot_x="-1.570796326794897"
buffer_height="1024" buffer_width="1024"
shadow_bias="1.05"
cast_shadows="TRUE"
attenuation_quadratic="0.000000001"
attenuation_linear="0.000001"
attenuation_constant="2"
type="L_SPOT" />
</scene>
</glge>
```

É importante notar que, ao contrário da maioria das API's, no *GLGE* os ângulos são definidos em radianos e não em graus. Outra coisa igualmente importante é que no *GLGE* não é possível definir a intensidade da luz, apenas é possível configurar a sua atenuação e o tipo de sombra. A intensidade varia com a posição onde colocamos a luz e também com o tipo de luz que escolhemos, que pode ser pontual, direccional, ou foco. No nosso caso, é um foco de luz, pois até à data, é o único tipo de luz desta API que projecta sombras.
Depois do ficheiro XML criado, simplesmente é carregado este ficheiro utilizando a *Document.load()*, sendo "*Document*" uma variável do tipo *GLGE.Document*. O ficheiro é então processado e o motor *GLGE* renderiza o cenário 3D. É possível também alterar cada um dos atributos XML definidos, dinamicamente, ou adicionar novos elementos (por exemplo outro tipo de luz, numa certa posição), na própria API em *Javascript*. No entanto, o *GLGE* não possui quaisquer formas primitivas (por exemplo um simples cubo) definidas na sua API.

A imagem final do cenário de teste, renderizada em *WebGL* através do GLGE, está representada na figura 14.

Figura 14 - Cenário de teste renderizado com a *framework GLGE*

É possível também utilizar alguns *add-ons* disponíveis para o *Blender* para exportar um cenário totalmente criado para um ficheiro XML, pronto para ser lido e renderizado. Alguns exemplos podem ser vistos em [53] e [54]. Alguns autores criam até o seu próprio exportador *Blender-GLGE*, tal como referido anteriormente em [28].

3.2.4 – Conclusões

Optámos por testar apenas as *frameworks* mais promissoras tendo em vista os objectivos traçados, que foram neste caso o *Copperlicht, GLGE* e *Three.js*. Todas elas apresentam um nível de abstracção bastante elevado ao programador, no entanto, existem várias vantagens e desvantagens em cada uma delas. Nas tabelas 1, 2 e 3 estão sumarizados os pontos fortes e fracos destas 3 API's.

Three.js	
Pontos Fortes	**Pontos Fracos**
Bons exemplos e suporte	Constantes alterações na API
Excelente qualidade de sombras e iluminação	Documentação muitas vezes não é actualizada entre versões de API
Facilidade de importação de modelos	Poucos formatos de importação (quando comparado com outras API's)
Vários tipos de materiais, *shaders* facilmente utilizáveis, mesmo em objectos importados	-
Vários tipos de modelos primitivos	-
Facilidade de importação de modelos no formato *Collada* ou *JSON* (facilmente exportável a partir do *Blender*)	-
Nível muito alto de abstracção do *WebGL*	-
Framework mais utilizada no momento	-

Tabela 1 - Pontos fortes e fracos da *framework Three.js*

GLGE	
Pontos Fortes	**Pontos Fracos**
Fácil importação de cenários criados no *Blender*	Ausência de formas primitivas
O ambiente 3D pode ser construído apenas a partir do ficheiro XML	Iluminação limitada (pouca parametrização)
Nível muito alto de abstracção do *WebGL*	Pouco detalhe de sombras
Implementados vários *shaders* de uso comum, libertando o programador dessa tarefa	Apenas suporta o formato *Collada* para além do XML
Boa documentação da API e exemplos	-

Tabela 2 - Pontos fortes e fracos da *framework GLGE*

Copperlicht	
Pontos Fortes	**Pontos Fracos**
Ferramenta 3D de construção do ambiente, o *CopperCube*	*CopperCube* não é gratuito (apenas durante 14 dias)
Excelente nível de abstracção do *WebGL*, sendo possível criar toda a mecânica do ambiente 3D apenas utilizando o *CopperCube*	Sem suporte para cores, as cores dos modelos são definidas apenas por uma textura
Suporte de vários formatos de ficheiro para o carregamento de modelos	Até à data, sem suporte para sombras dinâmicas entre os modelos, apenas no próprio modelo
Boa documentação da API	-
Vários tipos de câmara pré-definidos	-
Grafo de cena	-

Tabela 3 - Pontos fortes e fracos da *framework Copperlicht*

Tendo em conta todos estes parâmetros, e avaliando os resultados obtidos na renderização, decidimos escolher a *framework Three.js* para a continuação do trabalho. Decidimos não escolher a *framework Copperlicht* devido a ser necessária uma licença para poder utilizar todas as características necessárias. A *framework GLGE* apresenta muitas características interessantes, mas a qualidade de iluminação e parametrização da mesma é um pouco limitada. Além disso, o resultado da renderização do cenário de teste na *framework Three.js*, foi factor determinante na decisão, devido às suas qualidades em questões de iluminação.

Capítulo 4

Aplicação Desenvolvida (*Indoor OBJ Loader*)

4.1 - Descrição

Utilizando HTML5, *JavaScript* e a *framework Three.js* pretendeu-se desenvolver uma aplicação Web interactiva que permite carregar modelos 3D e onde é possível inspeccioná-los através da iluminação e navegação no cenário. O formato escolhido para o carregamento de modelos 3D, já referido, foi o formato *Wavefront OBJ* [10], devido a este ser um dos formatos *standard* mais utilizado.
Esta aplicação Web permite demonstrar as capacidades gráficas das aplicações Web com base no *WebGL*, bem como, da *framework* escolhida no carregamento de cenários 3D e das ferramentas de interacção disponíveis.

4.2 - Funcionamento da *framework Three.js*

Escrita em *Javascript*, esta *framework* faz o processamento gráfico com base no esquema apresentado na figura 15.
Através do grafo de cena, podemos adicionar as várias partes que compõem o cenário com os seus respectivos parâmetros. Estas partes incluem a câmara, as luzes e os modelos. No caso dos modelos, estes têm uma geometria associada com a informação das faces (que podem ser triângulos ou quadrados, *THREE.Face3* ou *Three.Face4)*, vértices e normais. Para além disso, os modelos podem também ter materiais associados, que por sua vez podem ter uma textura associada, carregada através de uma imagem.
Toda esta informação é passada a um objecto *THREE.WebGLRenderer* que faz a ligação com o nível mais baixo, ou seja, o *WebGL* nativo do *browser*. São construídos os *buffers* de faces e vértices ao qual se vai aplicar o respectivo *vertex shader* e *fragment shader*, executado pelo GPU da máquina. Estes dois programas são construídos dinamicamente pela *framework*. Isto é conseguido através da junção de cadeias de caracteres (*strings*) que contêm o código *OpenGL SL* a ser compilado. Estas *strings* podem ser, por exemplo, informações sobre os vários tipos de material e luzes, que são seleccionadas e adicionadas ao programa final (*WebGLProgram*) que determina a cor final dos pixéis a serem desenhados. Os sucessivos *renders* da imagem,

de modo a proporcionar interactividade, são conseguidos através do método *requestAnimationFrame*, presente nos *browsers* suportados.

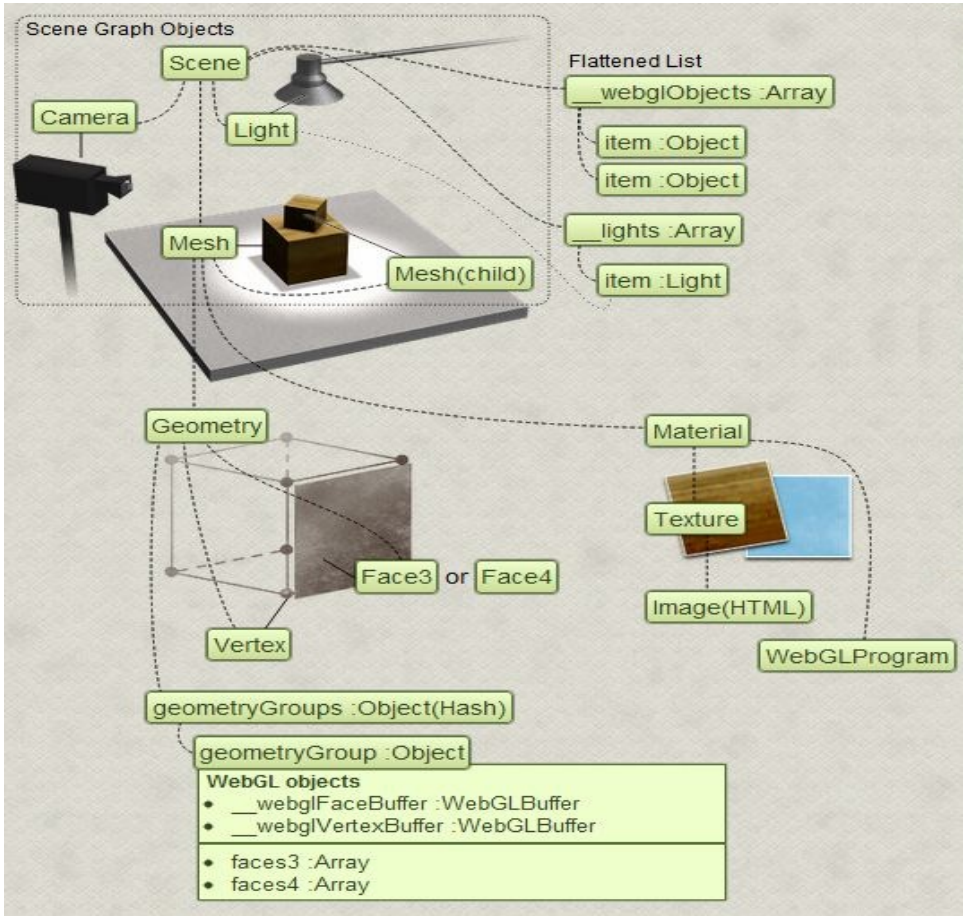

Figura 15 - Mapa de processamento *Three.js* [55]

Durante o progresso de construção desta aplicação, foram lançadas novas versões da *framework*, com novas funcionalidades e melhorias em relação às existentes. Estas são marcadas, no já referido repositório em [46], como revisões, sendo que a revisão utilizada na aplicação foi a r49. Existe também documentação sobre as modificações necessárias à programação de acordo com a revisão em causa [56].

4.3 - Construção da aplicação

Apresenta-se agora as fases que levaram à criação da aplicação final. Cada uma destas fases é descrita em detalhe nas subsecções seguintes. As fases foram as seguintes:

- ✓ Construção do cenário 3D.
- ✓ Escolha do formato a utilizar para exportar/importar os modelos criados.
- ✓ Inicialização da *framework* e outras ferramentas.

- ✓ Importação dos modelos para o cenário.
- ✓ Implementação da iluminação.
- ✓ Carregador *OBJ* e interacção.

4.3.1 - Construção do cenário 3D

O cenário 3D, que suporta a aplicação Web desenvolvida, foi modelado através da aplicação *open-source*, *Blender* (versão 2.63). Parte da construção deste cenário foi baseado no tutorial de modelação mencionado em [57]. O cenário contém 13 objectos diferentes, com um total de 15620 vértices e 15337 faces.

4.3.2 - Formato de exportação

De modo a ser possível carregar os modelos criados na aplicação Web desenvolvida, é necessário exportar os mesmos para o formato suportado pela *framework*. Para facilitar este processo, os autores da referida *framework* desenvolveram um *plugin* para o software *Blender*, que permite adicionar a funcionalidade de importar/exportar ficheiros no formato *JSON*. Os ficheiros e instruções para instalação do referido *plugin* podem ser encontrados em [58]. No entanto, basta criar uma directoria com os *scripts* disponibilizados na directoria de *addons* do Blender, e depois activar o *plugin* no Blender em *User preferences -> Addons -> Import-Export -> Three.js*. Depois de activado, a opção de importar/exportar para o *Three.js* fica disponível no *Blender* e permite-nos seleccionar as informações do objecto que pretendemos escrever no ficheiro *JSON*. Tais informações podem ser:

- ➢ Geometria (vértices, faces, normais).
- ➢ Materiais (Phong ou Lambert), cores e UVs.
- ➢ Fontes de luz e câmaras (de momento em fase experimental).

4.3.3 - Inicialização da *framework*

Visto a *framework* ter sido escrita na linguagem *Javascript*, começamos por juntar à página Web que suporta a aplicação o ficheiro que a contém (*Three.js*) através da tag <*script*>. Por exemplo:

```
<script src="lib/Three.js"></script>
```

A partir desse momento, podemos então inicializar os 3 elementos fundamentais:

- ✓ *Three.Scene*: Objecto que permite que sejam adicionados todos os elementos do cenário, quer seja modelos, câmaras ou fontes de luz.
- ✓ *Three.PerspectiveCamera*: Tal como é habitual em bibliotecas gráficas, neste objecto podemos definir os parâmetros de visualização, nomeadamente: ângulo de abertura (definido a 45º), aspecto (definido em função da altura e largura da janela do utilizador), *near* (definido a 1), *far* (definido a 10000) e a posição do observador.
- ✓ *Three.WebGLRenderer*: Este objecto processa toda a *Three.Scene* e *Three.PerspectiveCamera*, construindo a *frame* final. Tem vários atributos disponíveis para alterar a forma como é feito o processamento da imagem final. No entanto, estes já têm valores pré-definidos e por isso não necessita que nenhum deles seja definido para podermos visualizar o ambiente 3D. No nosso caso, definimos os atributos:
  - *antialiasing: true* (pode não ser suportado em *hardware* mais antigo).
  - *shadowMapEnabled: true* (activar processamento de sombras, utilizado em algumas partes do cenário).
  - *shadowMapSoft: false* (activar processamento extra para suavização de sombras)
  - *shadowCameraFov: 50* (grau de abertura da sombra em relação à câmara)

Inicialização de outras ferramentas

Para além do *script* inicializado, *Three.js*, são também utilizados na página Web, 3 outros *scripts* que podem ser encontrados também no já referido repositório oficial desta *framework* [46], e que são:

- ❖ *Detector.js* - Detecta se o *browser* que o utilizador está a usar para aceder à aplicação, suporta a tecnologia *WebGL* ou não. Caso não suporte o *WebGL* é apresentada uma mensagem de erro, com hiperligações para as localizações onde pode encontrar um *browser* que suporta.
- ❖ *Stats.js* - Permite fazer a contagem das *fps* (*frames per second*), da aplicação Web. É muito simples de utilizar pois basta ser inicializado e posicionado.
- ❖ *OBJLoader.js* - *Script* para carregamento de ficheiros no formato *OBJ*. Recebe como parâmetro uma referência para um ficheiro, e processa o ficheiro com base em padrões de caracteres encontrados. No final, cria uma instância de

THREE.Object3D que contém toda a geometria do objecto pronta para ser adicionada ao cenário.

Por último, foi também feita a inicialização de uma biblioteca de nome *"dat.GUI"*. Escrita também em *Javascript*. Esta biblioteca permite criar menus com a particularidade de os eventos associados à interacção com esse menu, estarem associados directamente a variáveis programadas em *Javascript*, sendo assim fácil de conseguir imediatamente o procedimento desejado, ideal para ambientes de demonstração e teste.
No âmbito desta aplicação Web, optamos por utilizar esta biblioteca para inicializar dois menus, um menu para a interacção com as fontes de luz do cenário, e outro menu para o carregamento de modelos *OBJ*, e a respectiva interacção com o mesmo. Assim através da programação de eventos com esta biblioteca foi possível criar as variáveis a serem alteradas pelo utilizador de modo a interagir com o ambiente.
Foi efectuada uma breve aprendizagem através do tutorial indicado em [59]. O *script* completo para a biblioteca pode ser encontrado em [60].

4.3.4 - Importação dos modelos criados

Os modelos criados e armazenados em ficheiros *JSON*, são carregados para a *framework* inicializando o objecto correspondente, ou seja, *THREE.JSONLoader().* De seguida, utiliza-se o método *load()* para ler sequencialmente cada ficheiro. Este método tem os seguintes parâmetros: caminho do ficheiro (relativo ao servidor), função a executar, e caminho para as texturas. Neste caso, a função a executar recebe automaticamente a geometria do modelo depois de lido, e para cada modelo esta função é executada e cria-se uma instância de *THREE.Mesh()*, que recebe como parâmetros a geometria do modelo e o material a aplicar. Neste caso, aplica-se aos modelos carregados o material *THREE.MeshPhongMaterial* (modelo de *Phong*) e o material *THREE.MeshFaceMaterial* que aplica ao modelo as cores e suas propriedades (como por exemplo a opacidade) tal como foram criadas e depois exportadas do *Blender*. A descrição completa sobre as propriedades dos materiais existentes pode ser encontrada na já referida documentação em [39]. Em relação às texturas, foi efectuado no *Blender* o mapeamento UV, que permite também exportar as coordenadas da textura no ficheiro *JSON*, tendo de ter as texturas o tamanho de uma potência de 2.
Com a instância de *THREE.Mesh()* criada, alteramos a propriedade *map* para o caminho da textura correspondente, relativa ao servidor (necessário para que seja aplicado o mapeamento UV, lido do ficheiro, à imagem pretendida), e então adicionamos o modelo ao cenário para ser visualizado na próxima renderização.
Por último, definimos também as propriedades *doubleSided*, *castShadow* e *receiveShadow*, de acordo com o modelo em causa. Isto permite que aquando da renderização a *framework* possa ter em conta os modelos com estas propriedades activas e juntar o programa *GLSL*

necessário. No nosso caso, definimos a propriedade *doubleSided* como *true* para todos os modelos e as propriedades *castShadow* e *receiveShadow* para os modelos: "*monkey"*, "mesa" e "cadeira".

4.3.5 - Implementação da iluminação

Na aplicação desenvolvida optou-se por utilizar 3 fontes de luz no cenário, sendo:

- 2 x *THREE.PointLight* : Este tipo de fonte de luz tem como parâmetros a cor, intensidade, distância e posição. A cor é definida em hexadecimal e a intensidade e distância são numéricas. Este tipo de luz emite em todas as direcções, em forma de esfera, segundo a distância definida, e afecta os modelos com material de tipo *Lambert* ou *Phong*.
- 1 x *THREE.SpotLight :* Esta fonte de luz tem como parâmetros a cor, intensidade, distância, posição, ângulo e expoente. Este tipo de luz funciona de maneira semelhante à anterior, mas emite numa dada direcção para um ângulo de abertura especificado o que permite produzir sombras no ambiente. O ângulo (expresso em radianos) define o tamanho circular que a luz irá afectar e o expoente define o seu raio.

No âmbito do ambiente 3D, dispomos de três modelos que simbolizam as fontes de luz, sendo dois deles esferas, e um outro modelo de foco de luz. Optámos por colocar as duas luzes pontuais dentro das esferas que representam os candeeiros no tecto, activar a propriedade *transparent* e definir no material (Phong) a propriedade *emissive* que nos permite que as faces das esferas emitam luz, da mesma cor que a fonte de luz associada. Aos valores por defeito corresponde a cor branca, ou seja, com intensidade de 1.0 e uma distância de 50.0. Podemos verificar o efeito de uma destas luzes pontuais e o resultado na Fig. 16.
Para o terceiro modelo de luz utilizamos o tipo *Spotlight*, logo abaixo do modelo de holofote (ver Fig. 17). Os parâmetros utilizados neste caso foram:

- Cor branca: definida em hexadecimal 0xFFFFFF
- Intensidade: 2
- Distância: 30
- Ângulo: π
- Expoente: 10

Figura 16 - Luz pontual

Para além destes parâmetros, é importante notar que é necessário também definir um ponto alvo para esta fonte de luz, através da propriedade *target*, que nos permite definir a direcção da luz. Neste caso, escolhemos um dos pontos de um modelo por baixo da luz, o modelo do "*monkey".*

Para podermos verificar o resultado do processamento das sombras, necessitamos também de definir as propriedades da própria sombra que vai ser criada por esta fonte de luz. As propriedades são as seguintes:

- *castShadow* : activar o cálculo de sombras para esta luz.
- *shadowMapWidth*: largura da textura de sombra (em pixéis). Definido a 2048.
- *shadowMapHeight*: largura da textura de sombra (em pixéis). Definido a 1024.
- *shadowCameraNear, shadowCameraFar, shadowCameraFov*: Volume de visualização para os cálculos das sombras. Valores definidos a 1, 1000 e 50, respectivamente.

Depois de adicionar esta fonte de luz ao cenário podemos obter um resultado como o apresentado na Fig. 17, onde é possível ver a sombra dos vários modelos no ambiente.

O ambiente criado permite a interacção com as fontes de luz. Para isso o utilizador pode seleccionar as fontes de luz, através de um clique do rato, e alterar as suas propriedades. Quando ocorre um clique no cenário, é disparado um evento *Javascript*, de nome *onDocumentMouseDown*. É então criada uma nova instância do objecto *THREE.Ray* e do objecto *THREE.Projector*. O primeiro permite que sejam efectuados testes de intercepção de um "raio" com os modelos existentes no cenário. Para que isto seja possível, é necessário definir dois pontos para que possa ser calculada a direcção do vector deste raio. Neste caso, o primeiro ponto é a posição da câmara, e o segundo ponto é calculado através da fórmula apresentada a seguir.

Figura 17 - SpotLight

```
X: (utilizadorX / largura_janela) * 2 - 1
Y: - (utilizadorY / altura_janela) * 2 - 1
Z: 0.5
```

Em que, "utilizador" é a posição 2D do pixel clicado da imagem. A coordenada Z é fixa em 0.5 devido a não ser possível saber a "profundidade" do clique em relação ao ambiente 3D. No entanto, esta posição está projectada apenas no plano 2D da imagem, e é necessário que seja transformada para coordenadas do espaço 3D, isto é possível através do objecto *THREE.Projector*, através do método *unprojectVector*, obtemos as coordenadas do clique no espaço 3D, em relação à câmara definida.

De seguida, é verificado o resultado do método *ray.intersectObjects*, se for não vazio, o resultado contém o modelo interceptado. De modo a ser mais eficaz de detectar intercepções, foi criada uma lista de cubos (utilizando *THREE.CubeGeometry*) na mesma posição das fontes de luz, ou seja, uma *bounding box* para cada luz. Obtemos então a referência para o cubo que foi seleccionado, alteramos a propriedade *visible* para *true* e de imediato na renderização seguinte aparecerá o cubo visível, o que permite dar feedback ao utilizador da selecção efectuada. Para sabermos a que fonte de luz o cubo se refere, verificamos uma propriedade *name* previamente atribuída, e conforme o seu valor sabemos que fonte de luz o utilizador pretende alterar. São então disponibilizadas as opções no menu, dependendo do tipo de luz que foi seleccionado, e estas propriedades podem depois ser alteradas quando é disparado o evento *onChange* de cada um dos menus. Um exemplo da

interacção com uma luz pontual pode ver-se na Fig. 18. Nesta pode ver-se a luz seleccionada e também as opções disponíveis no menu para este tipo de luz.

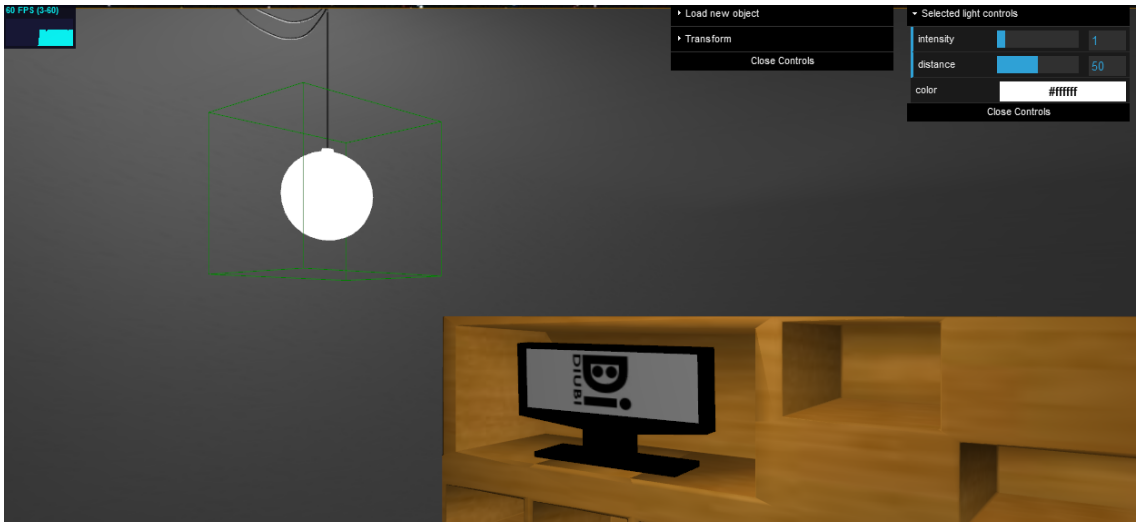

Figura 18 - Interacção com uma fonte de luz pontual.

Resumidamente, o algoritmo executado para a detecção do modelo seleccionado pelo utilizador funciona do seguinte modo:

1. Converter coordenada 2D do clique efectuado para o espaço 3D através do objecto *THREE.Projector.*
2. Criar uma instância de *THREE.Ray*, para calcular um vector vai desde a posição da câmara até ponto obtido no passo 1, e normalizar esse vector.
3. Verificar o resultado de intercepção com as *bounding boxes* existentes no cenário.
4. Se houver uma intercepção com uma *bounding box*, verifica-se o nome da mesma e torna-se visível. Depois adicionam-se as opções de interacção ao menu "*Selected light controls"* associadas às propriedades da luz seleccionada. Se não foi obtida intercepção, passa-se ao passo seguinte.
5. Se não foi obtida nenhuma intercepção, remover opções do menu (caso existam) e torna-se invisível a *bounding box* previamente seleccionada (caso esteja visível).

4.3.6 - Visualização e manipulação de modelos OBJ

Conforme referido anteriormente, a aplicação Web desenvolvida tem a funcionalidade de visualizar e manipular modelos 3D no formato *Wavefront OBJ*. Esta funcionalidade engloba 3 opções: ler modelo 3D a partir de um ficheiro, aplicar uma textura com base numa imagem fornecida pelo utilizador e ainda aplicar transformações geométricas ao modelo.

- Ler modelo 3D de um ficheiro OBJ

A visualização de um modelo 3D a partir de um ficheiro OBJ encontra-se disponível na opção "*Load OBJ file"* do menu "*Load new object"*. A selecção desta opção abre uma janela que permite ao utilizador seleccionar o ficheiro pretendido no seu computador. Por questões de segurança implementadas nos principais *browsers* Web, não é possível obter o caminho directo para o ficheiro na máquina local do utilizador, apenas é disponibilizada uma referência para o conteúdo. Utilizando essa referência, efectuamos a leitura com recurso a uma instância do objecto *FileReader*, que é um objecto *JavaScript* para processamento de ficheiros, com vários métodos de leitura incorporados, bem como eventos programáveis.
Depois do conteúdo do ficheiro lido, inicia-se então o seu processamento através de um objecto da *framework*, *THREE.OBJLoader*. A informação é então processada linha a linha procurando os padrões característicos do ficheiro, e é devolvida a geometria completa do modelo. Note-se que o ficheiro pode conter várias geometrias, ou seja, várias partes pertencentes a um mesmo modelo, de modo que optou-se por processar todas as geometrias em separado, e adicionar todas elas ao cenário depois de criada(s) a(s) malha(s) através de *THREE.Mesh*. Foram também implementadas mensagens de erro para o caso de não ser possível processar o ficheiro, ou a geometria devolvida estar incompleta ou inválida. No caso da leitura do ficheiro com sucesso, são activados os menus de interacção e é mostrada uma mensagem de sucesso ao utilizador. O modelo é então desenhado no centro do cenário dentro de um cubo transparente, e é definida a propriedade *name* com o nome do modelo para poder identificar o objecto. Isto é necessário visto que o "contentor" de modelos do cenário, *THREE.Scene*, não providencia o acesso directo a cada modelo em separado, a não ser que exista uma referência directa ao mesmo.
No entanto, ao modelo carregado são aplicadas as transformações geométricas necessárias para que qualquer modelo possa ser representado dentro do cubo transparente pré-definido. Foi criado por isso um algoritmo que processa os vértices do modelo e normaliza as suas coordenadas de acordo com as dimensões do cubo, assegurando que o modelo é desenhado dentro do mesmo. Um exemplo de um modelo carregado pode ser visualizado na Fig. 19.

- Aplicação de uma textura com base numa imagem

De modo similar ao descrito no ponto anterior, mas neste caso através da opção "*Apply Texture",* o utilizador pode seleccionar uma imagem do computador para aplicar como textura ao modelo já carregado, tendo em conta as coordenadas UV especificadas no ficheiro OBJ, isto é, se o modelo já for texturizado. A textura a aplicar pode ter qualquer formato de imagem, mas recomenda-se que a largura e altura da imagem sejam uma potência de 2, conforme especificado na tecnologia *WebGL*.

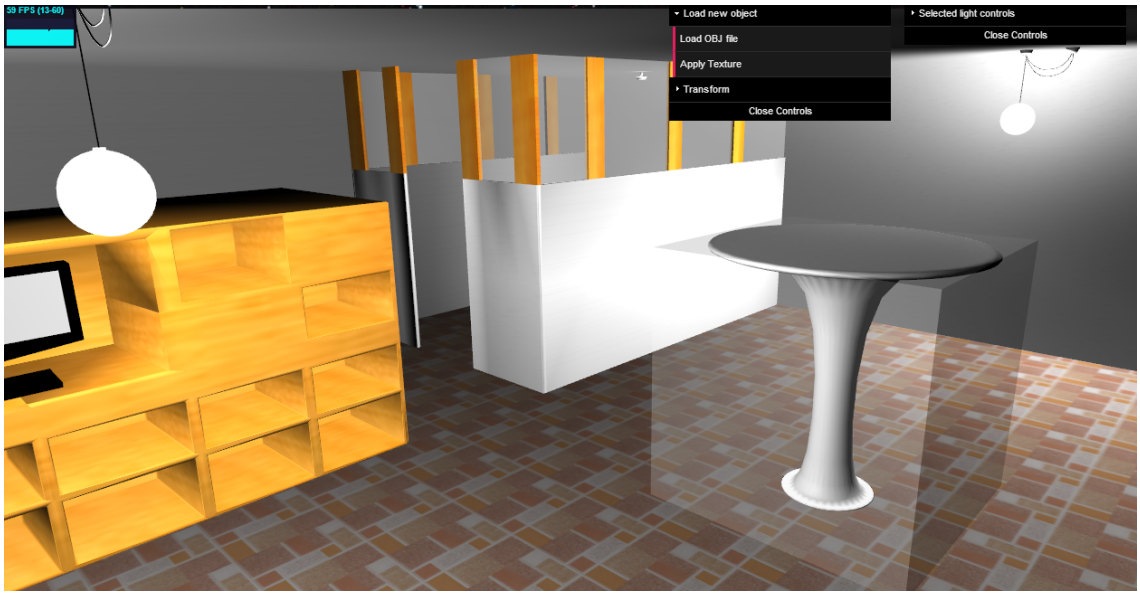

Figura 19 - Visualização de um modelo OBJ

Para aplicar a textura ao modelo são necessários os seguintes passos:

1. Ler ficheiro seleccionado, utilizando o objecto *FileReader*, através do método *readAsDataURL*.
2. Criar um novo elemento de imagem, criando uma instância do objecto *THREE.Texture*.
3. Para cada elemento do cenário:
    a. Pesquisar por nome e retirar do cenário todos os modelos que tenham como propriedade *name* igual à do modelo carregado anteriormente.
4. Para cada elemento carregado (i.e. geometria) a partir do ficheiro OBJ:
    a. Atribuir textura à propriedade *map* do material do modelo.
    b. Definir como *true*, a propriedade *needsUpdate*, da propriedade *map*, do material. Isto é necessário para que a *framework* faça a respectiva actualização do material do modelo na próxima renderização.
5. Adicionar novamente o modelo ao cenário, normalizando as coordenadas para ficar dentro do cubo.

Depois de executados estes passos, o modelo é então desenhado novamente pela *framework*, desta vez com a textura aplicada conforme o mapeamento UV existente no ficheiro. Podemos visualizar uma textura aplicada ao modelo na Fig. 20.

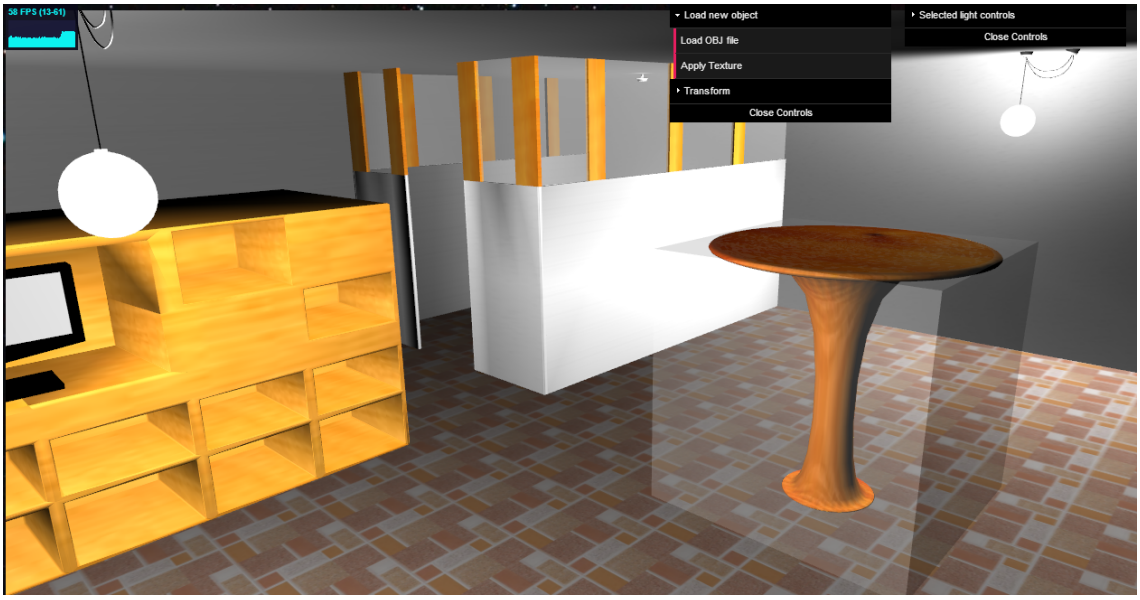

Figura 20 - Aplicação de uma textura a um modelo carregado

- Transformações ao modelo

Ao modelo carregado do ficheiro podemos ainda aplicar algumas transformações disponíveis no menu "*Transforms"*, que são activados após o carregamento do modelo OBJ. As operações disponíveis são:

- ✓ Rodar o modelo segundo os eixos X, Y ou Z.
- ✓ Mover o modelo em X, Y ou Z.
- ✓ Alterar a escala do modelo.

A alteração destas opções pelo utilizador permitem alterar as transformações aplicadas ao modelo, as quais passam a ser notadas na próxima renderização, visto que a *framework* actualiza automaticamente as matrizes de transformação aplicadas a cada modelo. Podemos verificar uma alteração da escala do modelo na Fig. 21, em que foi definido no menu "*Transforms*", a propriedade *scale* para 0.4.

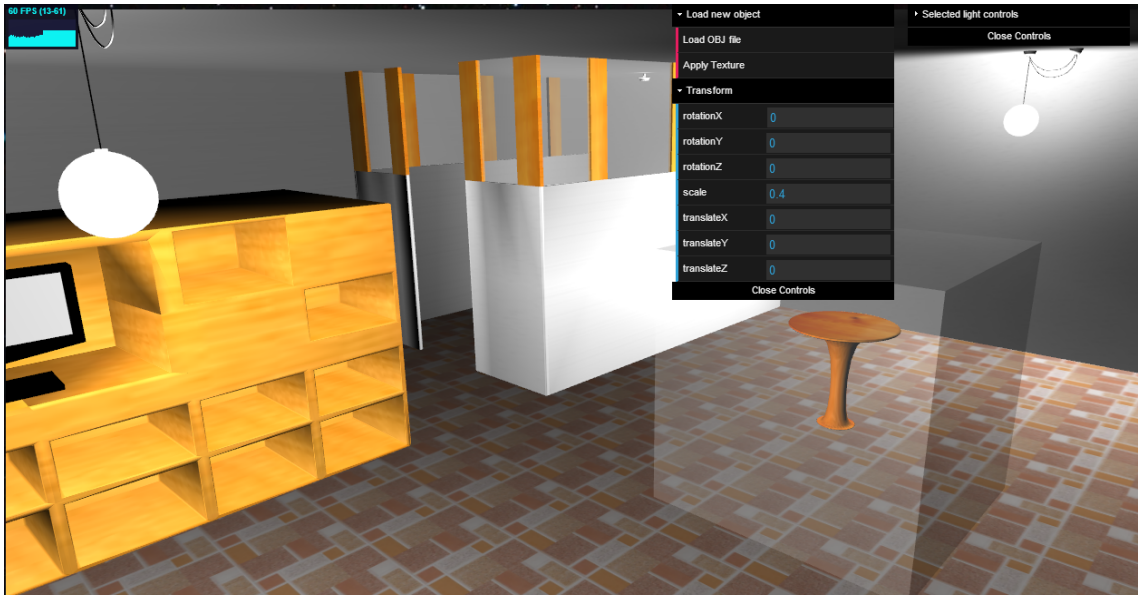

Figura 21 - Aplicação de uma transformação ao modelo

Capítulo 5

Algoritmo Desenvolvido

5.1 - Descrição

Parte integrante deste trabalho foi também a tentativa de criação e teste de um algoritmo de *ray tracing* [11], utilizando as ferramentas disponíveis na *framework* escolhida, e a linguagem *GLSL* (*OpenGL Shading Language*). Este capítulo encontra-se organizado conforme o seguinte:

- ✓ GLSL no âmbito da *framework* escolhida.
- ✓ Algoritmo desenvolvido: *ray tracing*.
- ✓ Conclusões.

5.2 - GLSL no âmbito da *framework* escolhida

No âmbito da *framework Three.js*, podemos também definir o nosso próprio código fonte quer para o *vertex shader* quer para o *fragment shader*. O código fonte destes dois programas é identificado no código HTML como um *script* do tipo *vertex* ou *fragment*, respectivamente. Por exemplo:

```
<script id = “vertex_shader” type = “x-shader/x-vertex”
//Declaração de variáveis
//Função main()
//Código GLSL
</script>
<script id = “fragment_shader” type = “x-shader/x-fragment”
//Declaração de variáveis
//Função main()
//Código GLSL
</script>
```

A sintaxe de programação é baseada na linguagem C, com os tipos mais comuns já implementados, como por exemplo *vec3* (vector de 3 dimensões). No entanto, não existe nesta linguagem a possibilidade de utilizarmos memória dinâmica ou ponteiros.
Fora da função *main*, podemos definir os parâmetros utilizados no *vertex shader*, que neste caso são *varyings* (para serem processadas e de seguida serem passadas ao *fragment shader*), e também *uniforms* para serem utilizadas (apenas no *shader* em que estão definidas). Na função *main* são então definidas as operações a executar nos vértices/fragmentos.
De notar que, o programa é executado uma vez para cada vértice (ou para cada pixel do fragmento, no caso do *fragment shader*), não sendo possível guardar dados entre essas execuções, visto que são executados paralelamente pela GPU.
No entanto para que a *framework* possa utilizar estes dois programas, o *vertex shader* e o *fragment shader*, é necessário definir o modelo que se pretende utilizar através da definição de um material do tipo *THREE.MeshShaderMaterial*. Este tipo de material, recebe como parâmetro um objecto *Javascript* com as seguintes propriedades: *uniforms*, *atributtes*, *vertexShader*, *fragmentShader*.

Uniforms: Um objecto *Javascript* com as *uniforms* que pretendemos que sejam passadas para os *shaders*. Estes valores são constantes e não poderão ser alterados ao nível do *shader*.

Segundo a referência [61], neste objecto é obrigatório indicar o tipo de constante, e o seu valor. A partir desta definição, a *framework* irá criar os *buffers* ao nível do *WebGL* necessários para que estas constantes sejam atribuídas e passadas para os *shaders* (em que também terão de estar declarados com o mesmo nome, no código *GLSL*).

Atributtes: Um objecto *Javascript* com constantes (por exemplo uma cor) para serem aplicadas individualmente aos vértices, estando por isso apenas acessível ao *vertex shader*. Este objecto é definido da mesma maneira que o objecto de *uniforms* acima descrito. No nosso caso não é necessário definir quaisquer *atributtes* visto que o cálculo da cor é feito apenas no *fragment shader*.

vertexShader e *fragmentShader*: Uma *string* contendo o código fonte dos dois programas. Como o código fonte é inserido na página Web através da *tag* <*script*>, esta é obtida da seguinte maneira:

```
vertexshader = document.getElementById('vertex_shader').innerHTML
fragmentshader = document.getElementById('fragment_shader').innerHTML
```

Depois do material aplicado a um modelo (definido através de uma instância *THREE.Mesh*), os *shaders* são então processados pela *framework* e compilados ao nível do *WebGL* produzindo a imagem final. Um exemplo disso é apresentado na Fig. 22, sendo uma simulação de iluminação em uma textura para o caso de um cubo. Mais informação sobre a programação dos *shaders* foi também conseguida consultando alguns tutoriais sobre iluminação [62] e [63]. O livro referenciado em [64] foi também essencial para a compreensão deste novo modelo de programação baseado em *shaders*.

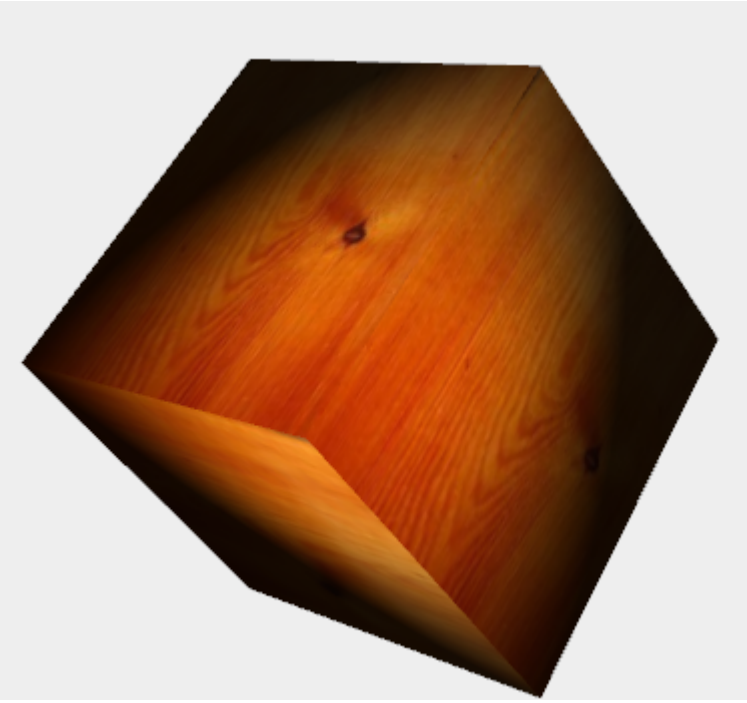

Figura 22 - Renderização de um cubo usando os *shaders*

5.3 - Algoritmo desenvolvido: *ray tracing*

Nesta secção descreve-se um algoritmo desenvolvido para calcular a iluminação global com base no algoritmo *ray tracing* tradicional. Começamos por uma breve descrição do funcionamento do algoritmo de *ray tracing*, de seguida explica-se a sua adaptação ao caso em questão tendo em conta os recursos disponíveis pela *framework* escolhida e finalmente descreve-se o algoritmo criado e os resultados obtidos.

5.3.1 - Descrição geral

Um algoritmo de *ray tracing* é uma técnica que permite gerar uma imagem com elevado grau de realismo tendo em conta as fontes de luz e a sua interacção com os modelos.
Este algoritmo pode ser descrito como uma tentativa de imitar a natureza, pois a ideia geral é que os raios, provenientes de uma fonte de luz, sejam reflectidos no cenário até chegar ao observador, dando-nos a percepção das sombras, reflexões e iluminação do mundo à nossa volta.
No âmbito da computação gráfica, este algoritmo é conhecido como um algoritmo de iluminação global, que é aplicado a todo o cenário virtual tendo em conta a localização do observador (câmara). No entanto, existe uma diferença fundamental em relação ao que acontece no mundo real. Em vez de serem lançados raios da fonte de luz e verificar quais chegam até à câmara (depois de terem sido processadas as respectivas reflexões no cenário), é feito precisamente o contrário. Ou seja, são lançados raios da posição do observador (i.e. da câmara) para o cenário onde são processadas as reflexões (um certo número máximo de vezes) e é verificado se o raio chega até uma fonte de luz. Se não chegar, pode significar por exemplo, que os sítios por onde passou estarão em sombra, ou no caso de chegar tem de se calcular a intensidade da luz tendo em conta o material dos modelos por onde passou. De igual modo, podem ser calculadas também as reflexões entre os modelos dependendo das propriedades do material que cada um tiver. O algoritmo é feito ao contrário do mundo real visto que poderia haver um grande número de raios desperdiçados (que nunca chegam a atingir a câmara), consumindo tempo de processamento desnecessário. No nosso caso, pretendemos criar um algoritmo similar para calcular as reflexões causadas entre os modelos virtuais existentes no cenário.

5.3.2 - *Ray tracing* e a *framework*

Um dos problemas encontrados foi o facto da *framework* apenas permitir atribuir os *shaders* a um modelo de cada vez, ou seja, não haver a possibilidade de aplicar o mesmo *shader* a toda a geometria presente no cenário. Logo a execução de um *shader* para um modelo não permite o acesso à geometria dos restantes modelos do cenário. Assim não era possível calcular a reflexão de um modelo noutro. A solução passou por criar uma estrutura de dados com toda a geometria do cenário e definir esta estrutura como um valor constante na compilação do *shader*, ou seja, do tipo *uniform*.
De modo a simplificar a estrutura, foi decidido que todos os modelos do cenário teriam que ter as faces triangulares. A estrutura de dados foi então definida como dois vectores unidimensionais. Um vector contendo os vértices dos triângulos (definidos consecutivamente,

de modo a podermos identificar a face), e um outro vector contendo as normais de cada triângulo.

5.3.3 - Algoritmo

De acordo com o descrito anteriormente decidiu-se implementar um tipo de *ray tracing* por modelo, que consiste resumidamente nos seguintes passos:

1. Inicializar *fragment shader* com a informação geométrica completa do cenário.
2. Para cada fragmento do objecto:
   2.1 - Construir um raio com origem na posição da câmara, e que passa pelo fragmento a ser processado (i.e. pixel em processamento).
   2.2 - Calcular reflexão do raio através do seu vector normal do fragmento em processamento. Esta normal é calculada automaticamente pela *framework* no *vertex shader*.
3. Com o raio calculado no passo 2:
   3.1 - Para cada triângulo presente no cenário: Verificar se o raio intersecta algum triângulo. Se sim, ir para o passo 4, se não, continuar para o próximo triângulo.
4. Se houve intersecção, definir cor deste fragmento como sendo a cor do triângulo interceptado. Se não houve, definir cor deste fragmento com a cor pré-definida para o modelo em processamento.

Podemos agora separar os dois algoritmos que constituem a aplicação. O algoritmo *Javascript* que inicializa o cenário de teste e as constantes a serem passadas para o nível *GLSL*, e o algoritmo dos *shaders GLSL* que processa os modelos efectuando o método de *ray tracing*. Para testar o algoritmo optou-se por utilizar um cenário simples de modo a podermos verificar o seu funcionamento.

Algoritmo Javascript

1. Inicializar o *WebGL*, ou seja, criar uma instância de *THREE.WebGLRenderer*.
2. Inicializar grafo de cena, criando uma instância de *THREE.Scene*.
3. Inicializar câmara, criando uma instância de *THREE.PerspectiveCamera*.
   a. Activar controlos de câmara, criando uma instância de *THREE.TrackballControls* (câmara que nos permite orbitar à volta do cenário).
   b. Definir posição da câmara.

4. Criar geometria, um cubo e um plano:
    a. Criar uma instância de *THREE.CubeGeometry* e *THREE.PlaneGeometry*.
    b. Converter as duas geometrias para triângulos, utilizando o método *triangulateQuads*.
    c. Criar duas novas instâncias de *THREE.Mesh,* com as geometrias e adicionar os dois modelos ao grafo de cena.
5. Inicializar *uniforms* a serem utilizadas pelo *fragment shader* ao processar o cubo. Um objecto com as propriedades:
    a. "*id*" - um inteiro, de valor 0, para no *shader* podermos identificar que estamos a processar o cubo.
    b. Dimensões da janela.
    c. Matriz inversa da matriz de projecção da câmara, matrizes de transformação dos modelos.
    d. Estrutura de dados com os vértices e vectores de normais.
6. Inicializar *uniforms* a serem utilizadas pelo *fragment shader* ao processar o plano. Um objecto com as mesmas propriedades descritas no passo 5, excepto no passo a), em que o valor do "id" é definido a 1, de modo a que depois na programação do *shader* possamos identificar qual é o objecto que está a ser processado.
7. Criar materiais do tipo *THREE.ShaderMaterial* com as *uniforms* descritas.
8. Atribuir materiais aos modelos.
9. Iniciar *loop* de renderização.

Um dos problemas encontrados aquando a execução de um *shader* com as *uniforms* acima descritas foi que, na maior parte das GPU's, existe uma limitação sobre o número de *uniforms* que podem ser declaradas. No nosso caso, o limite encontrado foi de 232 variáveis.
De modo a ultrapassar este problema, decidimos baixar o n.º de vértices da geometria gerada, de modo a não atingir este limite.
Apresenta-se agora o algoritmo detalhado, na parte da linguagem *GLSL* (*fragment shader*).

Algoritmo - *Fragment Shader*

1. Converter fragmento actual, de coordenadas do espaço 2D (ecrã) para coordenadas do espaço 3D (mundo virtual) [65].
2. Calcular vector (raio), entre a posição da câmara, e o fragmento actual.
3. Calcular reflexão do raio calculado no passo 2.
4. Alterar origem do raio para o fragmento actual, e direcção para o resultado do passo 3 (normalizada).
5. Se estivermos a processar o modelo do cubo:
    5.1 - Definir cor final do fragmento para azul.

5.2 - Para cada triângulo do plano:

5.2.1 - Verificar se o vector de raio, do passo 4, intersecta esse triângulo [66]. Se sim, definir cor final do fragmento para amarelo e ir para o passo 7. Se não, continuar para o próximo triângulo.

6. Se estivermos a processar o modelo do plano:

6.1 - Definir cor final do fragmento para vermelho.

6.2 - Para cada triângulo do cubo:

6.2.1 - Verificar se o vector de raio, do passo 4, intersecta esse triângulo [66]. Se sim, definir cor final do fragmento para amarelo e ir para o passo 7. Se não, continuar para o próximo triângulo.

7. Devolver cor final do pixel.

No passo 1, é necessária a conversão 2D para 3D, pois o como a execução é feita pixel a pixel, as coordenadas deste são coordenadas apenas em duas dimensões da janela de visualização, sendo por isso necessário convertê-las para o espaço 3D do nosso mundo virtual. Este procedimento é efectuado através da matriz inversa de projecção da câmara, tendo em conta também as dimensões da janela e *viewport*. Podemos visualizar o resultado final para o cenário de teste utilizado nas Fig. 23, 24 e 25. Note-se que a alteração da posição do observador produz diferentes resultados como seria de esperar.

Um aspecto importante foi o facto de ter sido necessário definir uma propriedade que permitisse identificar unicamente o modelo, pois quando o programa *GLSL* é executado pela GPU, não é possível saber a que modelo está a ser aplicado o processamento.

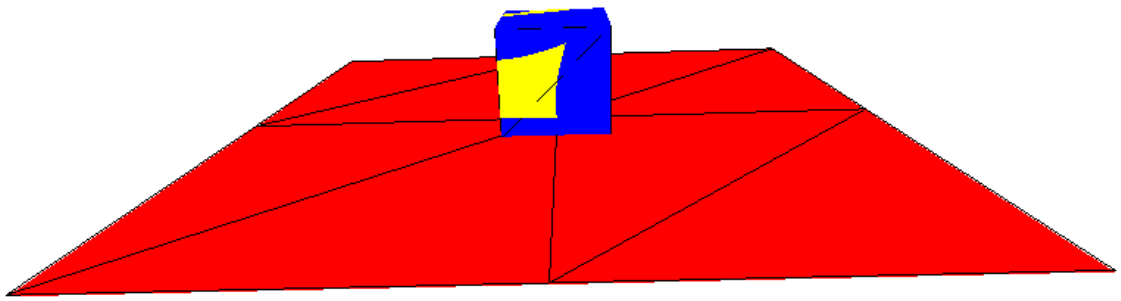

Figura 23 - Vista de frente do algoritmo em execução

Para o processamento dos erros de compilação usou-se uma extensão para o *browser Google Chrome*, denominada *WebGL Inspector* [67], que permite capturar a imagem final que foi renderizada e verificar o conteúdo dos *buffers* e texturas ao nível do *WebGL*, bem como ver os erros de compilação dos programas *GLSL*.

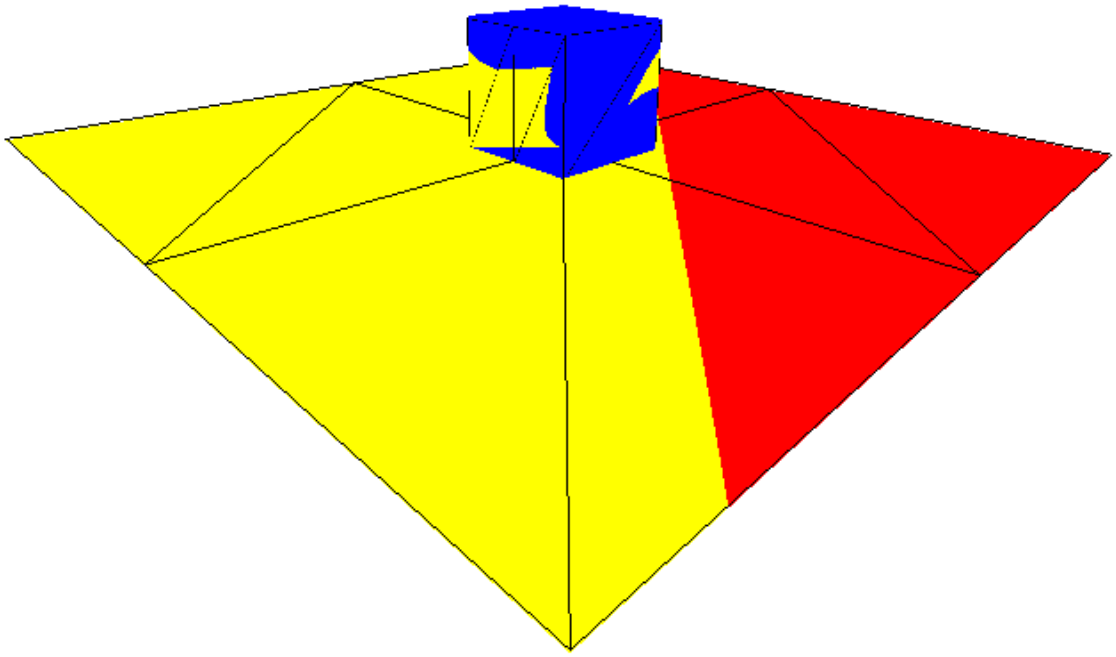

Figura 24 - Vista de perspectiva do algoritmo em execução

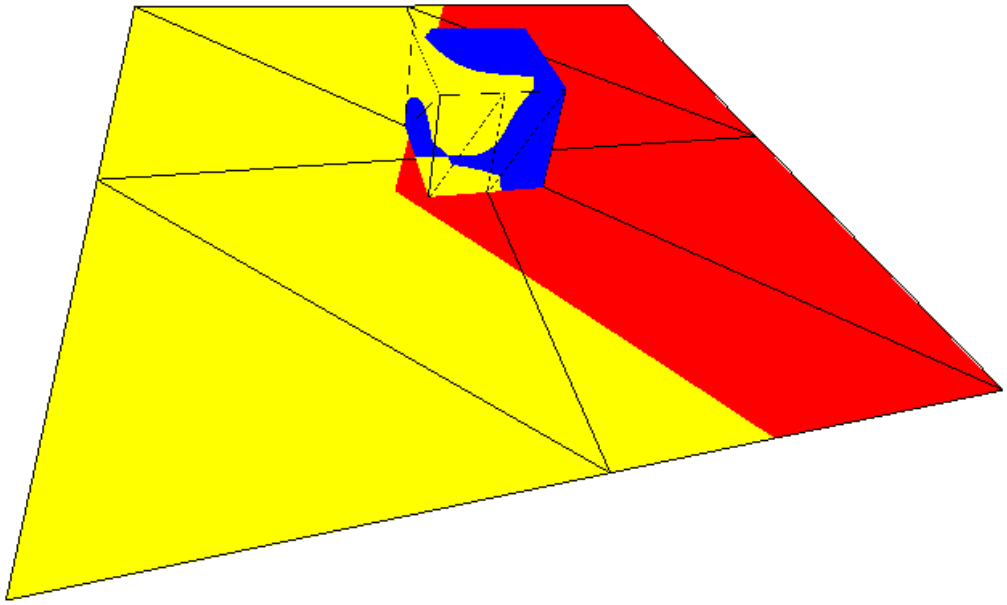

Figura 25 - Uma outra vista de perspectiva do algoritmo em execução

5.5 - Conclusões

O objectivo deste algoritmo é a visualização das reflexões entre os dois objectos. No entanto, como se pode verificar nas Figs. 23, 24 e 25, os resultados obtidos foram inconclusivos devido às reflexões (marcadas a amarelo), não serem consistentes com a perspectiva da câmara. Isto foi verificado em várias perspectivas pois foi implementada a câmara interactiva presente na *framework*, o que nos permitiu observar o processamento em tempo real das reflexões dos raios com a mudança de câmara.

Uma das grandes desvantagens da linguagem *GLSL*, é o facto de não ser possível aceder a vectores através de um índice não constante, sendo necessárias rotinas para percorrer toda a estrutura de dados criada quando é necessária uma informação numa certa posição. Esta limitação existe de modo a que seja compatível com a maior parte das GPU's existentes.

Tal como referido também anteriormente, existe a limitação sobre a quantidade de dados constantes que podem ser utilizados pela GPU. A solução para este problema envolve utilizarmos a memória de vídeo da GPU para armazenar os dados, ou seja, texturas. Neste caso, como os dados constantes são informação da geometria, que são pontos 3D, podem ser codificados e normalizados como valores RGB, esta solução não foi extensivamente testada pelo que será objecto de trabalho futuro.

De notar ainda a dificuldade encontrada no processo de *debug* ao nível do *GLSL*. Visto que o código fonte é compilado e executado na GPU, não existe um método directo que nos permite saber os valores dos cálculos, num determinado momento, ou usar uma interface de saída para saber os mesmos, vistos que os valores de saída são apenas as cores dos pixéis finais.

Concluímos então que devido a estas limitações, não nos foi possível verificar com certeza o comportamento correcto do algoritmo. No entanto os trabalhos já efectuados nesta área baseiam-se nas primitivas geométricas disponíveis directamente ao nível do *GLSL*. Utilizando uma estrutura de dados, ou sendo possível o acesso à geometria do mundo virtual, seria possível o processamento de quaisquer tipo de modelos construídos (ou até importados através da aplicação desenvolvida), em vez de apenas formas primitivas pré-definidas.

Note-se que ainda não existem soluções para o cálculo da iluminação global baseada em *shaders*. No entanto, começam a surgir algumas tentativas como, por exemplo em [68] onde temos um exemplo interactivo de condições de iluminação realísticas utilizando *shaders GLSL*. O cenário tem várias formas geométricas pré-definidas para visualização. De igual modo, construído pelo mesmo autor, podemos encontrar em [69] uma outra simulação de reflexões entre objectos e água. Em [70] pode ser visto também um exemplo simplificado sobre reflexões entre objectos simples pré-definidos e com animação. Além disso, em [71] podemos encontrar um projecto também sobre *ray tracing* que utiliza a mesma *framework*. Neste caso, o autor implementou uma técnica conhecida por *photon scattering*.

Capítulo 6

Conclusões e trabalho futuro

Pretendeu-se com esta dissertação a construção de um cenário 3D interactivo, utilizando a tecnologia *WebGL*, com base numa *framework* escolhida tendo em conta os testes realizados com as diversas *frameworks* disponíveis. O objectivo principal era a construção de um cenário 3D para a visualização e simulação de condições de iluminação onde fosse possível carregar modelos 3D num formato standard. Este objectivo foi conseguido e a aplicação desenvolvida permite visualizar modelos no formato OBJ e permite também interagir com as fontes de iluminação e com os modelos.

Esta dissertação contribuiu particularmente para melhorar os meus conhecimentos sobre computação gráfica, e em particular da tecnologia *WebGL* que permite a criação de gráficos 3D no *browser*, e que será cada vez mais importante no futuro. Para além disso, foi também muito importante na aprendizagem da linguagem *Javascript* e *GLSL*, tendo sido o primeiro contacto com esta linguagem e com os *shaders*.

Outro objectivo desta dissertação foi a implementação de um algoritmo baseado no *ray tracing* que pudesse ser aplicado ao cenário 3D e permitisse simular condições mais realistas de iluminação. Apesar de esse objectivo não ter sido atingido completamente (pelas razões já invocadas no capítulo 5) foi desenvolvido um algoritmo capaz de processar toda a geometria do cenário de teste e avaliar as reflexões entre os objectos.

Como trabalho futuro, sugerem-se as seguintes funcionalidades ou melhoramentos à aplicação desenvolvida:

- Suporte para outros formatos standard.
- Possibilidade de processar também modelos animados.
- Disponibilizar mais opções ao utilizador, como por exemplo definir novas fontes de luz.

Em relação ao algoritmo para o cálculo das reflexões vemos como trabalho futuro a sua conclusão, a qual poderá passar pelo uso do suporte para texturas como forma de armazenamento de dados (geometria), de modo a tirar partido da memória de vídeo da GPU. E quem sabe a possibilidade de disponibilizar o algoritmo como parte integrante da *framework Three.js* se os resultados forem os esperados.

Referências

[1] Tommi Mikkonen, Antero Taivalsaari, "*Apps vs. Open Web: The Battle of the Decade*", 2nd Workshop on Software Engineering for Mobile Application Development (MSE'2011, Santa Monica, California, USA, October 27, 2011), pp. 22-26

[2] "*WebGL - OpenGL ES 2.0 for the Web*", http://www.khronos.org/WebGL/, visitado em Outubro 2011

[3] T. Halic, W. Ahn, S. De, "*A Framework for 3D Interactive Applications on the Web*", SIGGRAPH Asia 2011 Posters, Dec 11-15, Hong Kong, China, article no. 58

[4] Otto J. Wittner, "*HTML5 in the Norwegian Higher Education Institutions*", http://www.ecampus.no/wp-uploads/2011/05/html5_norwegian_hei.pdf, visitado em Janeiro 2012

[5] Antero Taivalsaari, Tommi Mikkonen - "*The Web as an Application Platform: The Saga Continues*" - 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, Aug. 30 2011-Sept. 2 2011, pp. 170-174

[6] Peter Paulis, Ján Lacko - "*3D Webpages*", Študentská vedecká konferencia FMFI UK, Bratislava, 2010, pp. 316–327

[7] "*Animation Software, Animation Movies | Adobe Flash Professional CS6*", http://www.adobe.com/products/flash.html, visitado em Setembro 2012

[8] "*X3D and VRML, The most widely used 3D formats | Web3D Consortiúm*", http://www.web3d.org/realtime-3d/x3d-vrml/x3d-vrml-most-widely-used-3d-formats, visitado em Setembro 2012

[9] Xinping Yang - "*Navigating / Browsing in 3D with WebGL*", Student thesis, University of Gävle, Faculty of Engineering and Sustainable Development, Department of Industrial Development, IT and Land Management, June 2010

[10] "*Obj Specification*", http://www.martinreddy.net/gfx/3d/OBJ.spec, visitado em Agosto de 2012

[11] "*Ray Tracing*", http://pt.wikipedia.org/wiki/Ray_tracing, visitado em Março 2011

[12] "*OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*", http://www.khronos.org/opengles/, visitado em Outubro 2010

[13] "*OpenGL rendering pipeline*", http://www.songho.ca/opengl/files/gl_pipeline.gif, visitado em Novembro 2011

[14] "*Getting Started – WebGL Public Wiki*", http://www.khronos.org/WebGL/wiki/Getting_Started, visitado em Novembro 2011

[15] "*ES 2.0 Programmable Pipeline*", http://vip.cs.utsa.edu/classes/cs5113s2007/lectures/images/opengles_20_pipeline.gif, visitado em Dezembro 2011

[16] Matti Anttonen, Arto Salminen, Tommi Mikkonen, Antero Taivalsaari, "*Transforming the web into a real application platform: new technologies, emerging trends and missing pieces.*" SAC 2011: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 800-807

[17] Antero Taivalsaari, Tommi Mikkonen, Matti Anttonen, Arto Salminen - "*The Death of Binary Software: End User Software Moves to the Web*", 2011 Ninth International Conference on Creating, Connecting and Collaborating through Computing, pp. 17-23

[18] "*Blender Software*", http://www.blender.org/, visitado em Novembro 2011

[19] "*User Contributions – WebGL*", http://www.khronos.org/WebGL/wiki/User_Contributions, visitado em Outubro 2011

[20] Catherine Leung, Andor Salga – "*Enabling WebGL*", WWW 2010 · Developers Track April 26-30 · Raleigh · NC · USA

[21] "*COLLADA – Digital Asset and FX Exchange Schema*", https://collada.org/mediawiki/index.php/COLLADA_-_Digital_Asset_and_FX_Exchange_Schema, visitado em Dezembro 2011

[22] "*Canvas 3D JS Library*", http://www.c3dl.org/, visitado em Dezembro 2011

[23] "*Ambiera Software*", http://www.ambiera.com/ , visitado em Novembro 2011

[24] "*Copperlicht – JavaScript 3D Engine using WebGL*", http://www.ambiera.com/copperlicht/features.html, visitado em Novembro 2011

[25] "*Copperlicht Screenshots*", http://www.ambiera.com/copperlicht/screenshots.html, visitado em Novembro 2011

[26] "*PhiloGL - World Flights*", http://www.senchalabs.org/philogl/PhiloGL/examples/worldFlights/, visitado em Novembro 2011

[27] André F. S. Barbosa, Frutuoso G. M. Silva - "*Serious Games - Design and Development of OxyBlood*" – ACE '11 Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology, Article No. 15

[28] Nils Hering, Martin Rünz, Lubosz Sarnecki, Lutz Priese – "*3DCIS: A Real-time Browser-rendered 3D Campus Information System Based On WebGL*" - MSV'11 & CGVR'11: July 18, 2011, Institute of Computational Visualistics, University of Koblenz-Landau, Koblenz, Germany, pp 10-16

[29] "*About GLGE Library*", http://www.glge.org/about/, visitado em Janeiro 2012

[30] "*WebGL Demo: Sky and new fog options in GLGE*", http://www.glge.org/demos/skydemo/, visitado em Novembro 2011

[31] "*Downloads - o3d - WebGL Implementation of O3D - Google Project Hosting*", http://code.google.com/p/o3d/downloads/list, visitado em Dezembro 2011

[32] "*O3D WebGL pool game*", http://o3d.googlecode.com/svn/trunk/samples_webgl/o3d-webgl-samples/pool.html, visitado em Novembro 2011

[33] "*SpiderGL - Home*", http://spidergl.org/, visitado em Dezembro 2011

[34] Marco Di Benedetto, Federico Ponchio, Fabio Ganovelli, Roberto Scopigno, "*SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW*", Web3D '10 Proceedings of the 15th International Conference on Web 3D Technology, pp. 165-174

[35] "*SpiderGL Shadow Mapping*", http://spidergl.org/example.php?id=6, visitado em Novembro 2011

[36] Bijin Chen, Zhiqi Xu - "*A Framework for Browser-based Multiplayer Online Games using WebGL and WebSocket*", Multimedia Technology (ICMT), 2011 International Conference on 26-28 July 2011, pp. 471 - 474

[37] Wenling Hu, Hao Yuan, Jiangong Wang, Liang Wang - "*The Research and Application of Power System Visualization Based on HTML5*", Electric Utility Deregulation and Restructuring and Power Technologies (DRPT), 2011 4th International Conference on 6-9 July 2011, pp. 1562 - 1565

[38] "*Przeglad frameworków WebGL*" - http://3dgames.pl/blog/2011/11/przeglad-frameworkow/, visitado em Janeiro 2012

[39] "*three.js - documentation*", http://mrdoob.github.com/three.js/docs/51/, visitado em Setembro 2012

[40] "*JSON Model Format 3.1*", https://github.com/mrdoob/three.js/wiki/JSON-Model-format-3.1, visitado em Julho 2012

[41] "*+360º - Car Visualizer - Three.js*", http://carvisualizer.plus360degrees.com/threejs/, visitado em Julho 2012

[42] "*Learning WebGL*", http://learningWebGL.com, visitado em Outubro 2011

[43] "*Netbeans IDE*", http://netbeans.org/, visitado em Outubro 2011

[44] "*Apache Tomcat*", http://tomcat.apache.org/, visitado em Novembro 2011

[45] "*Google Chrome*", http://www.google.de/chrome/?hl=pt-PT, visitado em Outubro 2011

[46] "*mrdoob/Three.js Github*", https://github.com/mrdoob/three.js/, visitado em Outubro 2011

[47] "*Basics of Three.js*", http://fhtr.org/BasicsOfThreeJS, visitado em Outubro 2011

[48] "*Shadow Mapping with GLSL*", http://fabiensanglard.net/shadowmapping/index.php, visitado em Novembro 2011

[49] "*Copperlicht Tutorials*", http://www.ambiera.com/copperlicht/tutorials.html, visitado em Novembro 2011

[50] "*Hands-on WebGL: Basic GLGE Tutorial*", http://www.rozengain.com/blog/2010/06/23/hands-on-WebGL-basic-glge-tutorial/, visitado em Dezembro 2011

[51] "*GLGE tutorials: Camlittle.com*", http://camlittle.com/glge-tutorials/, visitado em Dezembro 2011

[52] "*GLGE Github Examples*", https://github.com/supereggbert/GLGE/tree/master/examples, visitado em Dezembro 2011

[53] "*Blender-WebGL-exporter*", http://code.google.com/p/blender-WebGL-exporter/wiki/InstallScript, visitado em Dezembro 2011

[54] "*Blender-Glge Exporter*", https://github.com/lubosz/blender-glge-exporter, visitado em Dezembro 2011

[55] "*Three.js Walking Map*", http://ushiroad.com/3j/, visitado em Setembro 2012

[56] "*Migration*", https://github.com/mrdoob/three.js/wiki/Migration, visitado em Agosto 2012

[57] "*Modelling a Modern Interior Scene in Blender*", http://cg.tutsplus.com/tutorials/blender/modeling-a-modern-interior-scene-in-blender/, visitado em Julho 2012

[58] "*Three.js Blender Import/Export*", https://github.com/mrdoob/three.js/tree/master/utils/exporters/blender, visitado em Junho 2012

[59] "*dat.GUI workshop*", http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage, visitado em Agosto 2012

[60] "*dat-gui - A lightweight controller library for JavaScript*", http://code.google.com/p/dat-gui/, visitado em Agosto 2012

[61] "*Uniforms Types mrdoob/three.js wiki*", https://github.com/mrdoob/three.js/wiki/Uniforms-types, visitado em Março 2012

[62] "*WebGL lesson 7 - basic directional and ambient lighting*", http://learningWebGL.com/blog/?p=684, visitado em Fevereiro 2012

[63] "*WebGL lesson 12 - Point Lighting*", http://learningWebGL.com/blog/?p=1359, visitado em Fevereiro 2012

[64] Aaftab Munshi, Dan Ginsburg, Dave Shreiner - "*OpenGL ES 2.0 Programming Guide*", Addison-Wesley, July 2008

[65] "*gluUnproject variant for WebGL*", http://www.jeshua.me/blog/gluUnprojectvariantforWebGL, visitado em Junho de 2012

[66] "*Undernones: GPU Ray tracing with GLSL*", http://undernones.blogspot.pt/2010/12/gpu-ray tracing-with-glsl.html, visitado em Maio 2012

[67] "*WebGL Inspector*", http://benvanik.github.com/WebGL-Inspector/, visitado em Abril 2012

[68] "*WebGL Path Tracing*", http://madebyevan.com/WebGL-path-tracing/, visitado em Junho 2012

[69] "*WebGL Water*", http://madebyevan.com/WebGL-water/, visitado em Junho 2012

[70] "*Ray tracer using WebGL*", http://people.mozilla.com/~sicking/WebGL/ray.html, visitado em Março 2012

[71] "*WebGL*", http://iamnop.co.cc/WebGL/, visitado em Julho 2012