Marcelo Sousa

Desenvolvimento de um Simulador de Robô Móvel

Validação da Adaptação do OpenAR a OpenGL 3.3

Tese de mestrado em Engenharia Electrotécnica e de Computadores

Setembro/2015

· U C ·

UNIVERSIDADE DE COIMBRA

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Electrotécnica e de Computadores

Desenvolvimento de um Simulador de Robô Móvel

Validação da Adaptação do OpenAR a OpenGL 3.3

Marcelo Alfredo Ramos de Carvalho da Cunha de Sousa

Coimbra, 2015

Desenvolvimento de um Simulador de Robô Móvel

Validação da Adaptação do OpenAR a OpenGL 3.3

Orientador:
Prof. Paulo Jorge Carvalho Menezes

Júri:
Prof. Rui Paulo Pinto da Rocha
Prof. João Filipe de Castro Cardoso Ferreira
Prof. Paulo Jorge Carvalho Menezes

Marcelo Alfredo Ramos de Carvalho da Cunha de Sousa

Dissertação submetida para obtenção do grau de Mestre em Engenharia Electrotécnica e de Computadores

Departamento de Engenharia Electrotécnica e de Computadores

Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Setembro 2015

Agradecimentos

Gostaria de agradecer ao Professor Paulo Menezes, Bruno Patrão, Luís Almeida, João Seabra, Carlos Cortinhas e Samuel Pedro e outros colegas do laboratório de Robótica Móvel no Instituto de Sistemas e Robótica, pela sua ajuda. Gostaria de deixar um agradecimento especial à Marília Carvalho que fez a imagem da capa desta dissertação e que é minha irmã. Gostaria de agradecer em especial aos meus pais e irmãs pelo seu apoio nos meus estudos, à minha mulher Ana também pelo seu apoio e à minha, ainda pequena, filha Leonor por existir.

Resumo

A simulação de robots visa simular o comportamento destes numa aplicação de computador poupando tempo e dinheiro. Esta dissertação teve como objectivo construir um protótipo de robô e drone com a utilização da biblioteca de realidade aumentada OpenAR e com as capacidades gráficas que a API OpenGL 3.3 proporciona, e ainda com a visualização de malhas poligonais a partir de ficheiros do formato COLLADA que foi implementada na biblioteca. A biblioteca de realidade aumentada OpenAR é uma framework 3D orientada a objectos para aplicações de realidade aumentada.

A dissertação procura em primeiro lugar transmitir conhecimento sobre o que são as malhas poligonais e como são as estruturas em que estas se armazenam. Em segundo lugar pretende mostrar a contribuição do autor para a biblioteca OpenAR e a construção do protótipo de simulação de robô e drone. A inovação mais importante implementada na biblioteca é a utilização de *shaders*, que permite renderizar cada objecto 3D de uma forma diferente. Outra inovação foi realizar deformações de malhas poligonais com a utilização de um esqueleto articulado.

Palavras chave: Computação Gráfica, Simulação, Modelos Tridimensionais, Malhas Poligonais, OpenGL, COLLADA

Abstract

Robot simulation aims to simulate the beaviour of robots in a computer aplication saving time and money. This dissertation had as a goal to build a prototype of a robot and drone using the augmented reality library OpenAR with the graphics capabilities of the OpenGL 3.3 API, also with the visualization of meshes from COLLADA file format implemented in OpenAR. OpenAR is a 3D framework object oriented for augmented reality applications.

This dissertation has as a first objective to pass on knowledge about what a mesh is and how are made the structures where they are stored in. As a second objective aims to show the author contribution to the OpenAR framework and the build of the robot and drone simulator application. The most important innovation implemented in the library is the use of shaders which allows to render each 3D object with it's own rendering. Other innovation was to be able to have in OpenAR framework mesh deformation using an articulated skeleton.

Key words: Computer Graphics, Simulation, Three-dimensional models, polygon meshes, OpenGL, COLLADA

Índice

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas

API - Application Programming Interface

COLLADA - COLLAborative Design Activity

GLSL - OpenGL Shading Language

ISR - Instituto de Sistemas e Robótica

LOD - Level of Detail

OpenAR - framework C++ para desenvolvimento de aplicações de realidade aumentada (ISR)

OpenGL - Open Graphics Library

Capítulo 1: Introdução

Um fenómeno real pode ser descrito através de um processo que servirá de base para construir um software de simulação. O programa de simulação permite realizar uma determinada acção sem se executar essa acção realmente, mas reproduzindo-a virtualmente nos seus exactos termos.

Um simulador de robô é um programa de computador que permite criar um ambiente em que o robô se desloca em ambientes fechados ou em campo aberto sem a necessidade de usar o robô propriamente dito, possibilitando que se poupem recursos financeiros. Nos simuladores de robôs são simulados os comportamentos destes num mundo virtual em que podem interagir com outros robôs e/ou com corpos rígidos. Os robôs são programados para interagir nesse mundo virtual. A simulação em robótica permite a quem faz o desenvolvimento testar o seu código de programação e verificar se o design mecânico está de acordo com o design proposto nos requisitos.

Existem vantagens[36,37] e desvantagens em usar simuladores de robôs, as vantagens superam largamente as desvantagens. Existem assim as seguintes vantagens: com a utilização dos simuladores conseguem-se baixos custos financeiros para produzir um robô desde o seu desenho inicial; o design do robô pode também ser redesenhado sem custos financeiros; o código programado pode ser testado de acordo com as especificações que foram estabelecidas; qualquer componente do robô pode ser testado individualmente; se estivermos perante um projecto complexo o robô pode ser simulado em etapas; uma simulação completa pode determinar se o robô vai de encontro às especificações; praticamente todos os softwares de simulação são compatíveis com várias linguagens de programação e o tempo que decorre entre o início de um projecto e o seu fim pode diminuir. Em relação a desvantagens pode-se começar por dizer que o mundo real pode sujeitar um robô a uma variedade de situações que um mundo virtual não sujeitaria e outra situação também importante é que um robô simulado só simula aquilo que foi programado para simular.

As características[35] mais importantes dos simuladores de robôs são: prototipagem rápida do robô (é possível com um simulador de robôs rapidamente criar um protótipo de um robô), alguns simuladores apresentam "renderização" realista tridimensional (existem simuladores com a funcionalidade de modelação 3D ou então com a possibilidade de usar ferramentas de terceiros para criar modelos e ambientes virtuais), existem simuladores que permitem a utilização de motores ou bibliotecas de Física para obter movimentos realistas, é possível ter portabilidade do código da plataforma de simulação para a plataforma real[1], que é uma das características mais importantes. Existe ainda outra característica destes simuladores que é dinâmica dos corpos dos robôs ser definida por scripts em diversas linguagens como, por exemplo: C, C++, Perl, Python, Java e MATLAB.

1.1 Estado da Arte

Existem simuladores de robô *open source* e comerciais. Vamos analisar alguns simuladores *open source* mais representativos deste tipo de software de simulação.

Player/Stage

Stage é um simulador de múltiplos robôs que faz parte do projecto Player/Stage[2], Player é um servidor de rede para controlo de robôs. Com o Stage podemos criar clientes de dispositivos virtuais robóticos para se ligarem ao servidor Player. O Stage tem capacidade de simulação 2D tanto de plataformas robóticas como de uma grande variedade de sensores, permite que potencialmente milhares de robôs sejam simulados simultaneamente e fornece a possibilidade de se evitarem colisões e permite também a geração de mapas. O Stage é usado geralmente como um *plug-in* do Player, os controladores de robôs desenvolvidos no Stage precisam de poucas ou nenhumas modificações para serem colocados em robôs reais, mas o Stage pode ser usado como um programa autónomo para simulação de robôs. Existe também a biblioteca libstage em C++ com a qual se podem criar e executar simulações usando o Stage. O Stage pode ser instalado em sistemas Linux e OS X e pode ver-se um ecrã deste *software* na Figura 1.

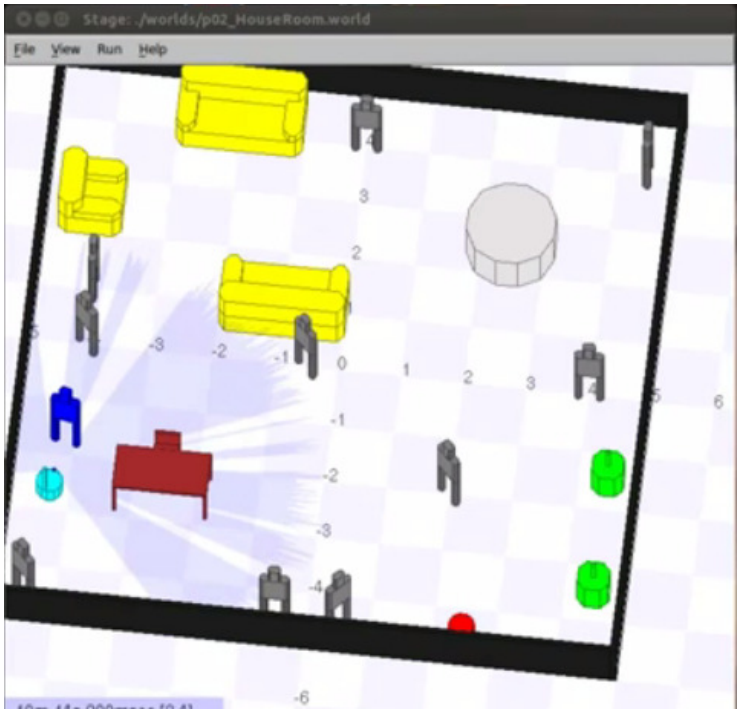

Figura 1: Ecrã do Simulador Stage

Gazebo

Gazebo[3] é uma plataforma de simulação de robôs capaz de simular uma população de robôs, sensores e objectos num mundo virtual tridimensional. Este simulador de robôs utiliza o motor de "renderização" de gráficos 3D orientado por objectos OGRE[4], que serve para "renderizar" os modelos dos robôs, ambientes e objectos. Utiliza também uma biblioteca de simulação de dinâmica de corpos rígidos, ODE[5], nesta biblioteca usam-se corpos rígidos com formas como cubos, esferas

e cilindros e em que cada uma dessas formas tem propriedades físicas como massa e fricção, a parte física da simulação do robô define-se com estas formas e com juntas. O Gazebo é compilado com suporte para ODE, mas podem usar-se outros motores de física como Bullet, Simbody e DART. Este simulador é compatível com Player e o popular Robot Operating System (ROS)[6]. Os controladores de robô escritos para o simulador Stage podem ser usados geralmente sem modificação no Gazebo (e vice-versa). No Gazebo existe um formato SDF que é um formato de ficheiro XML onde se definem os ambientes e modelos. Programas externos podem interagir com o Gazebo com bibliotecas disponibilizadas nas linguagens C e Python. O Gazebo foi desenvolvido para funcionar em sistemas operativos Linux, mas pode funcionar também em OS X.

MORSE

MORSE[7] é um simulador genérico usado em robótica a um nível académico, é *software open source* e utiliza o motor de *software* Blender[8] para modelação e "renderização" 3D. A simulação física é baseada no motor Bullet[9], que é utilizado em conjunto com o Blender. Este simulador de robô permite uma simulação realista tridimensional de robôs em campo aberto e em ambientes fechados, permite a interação com humanos e também é possível ter sistemas com múltiplos robôs. Fornece a possibilidade de usar uma grande quantidade de sensores, actuadores e plataformas robóticas. Permite a integração com os *middlewares* ROS, YARP[10], Pocolibs[11] e MOOS[12]. O Blender suporta vários formatos 3D, como p.e., 3ds Max, COLLADA, VRML, Wavefront OBJ, etc, assim os modelos que estejam nestes formatos podem ser usados no MORSE para conceber os modelos de robôs e ambientes virtuais. É importante dizer que o Blender utiliza *shaders*[39] na "renderização" dos objectos tridimensionais e por isso o MORSE beneficia desta funcionalidade. Neste simulador as simulações são geradas a partir de *scripts* na linguagem Python onde se descrevem os modelos dos robôs e os ambientes onde estes se deslocam. O MORSE funciona em sistemas operativos Linux e OS X.

USARSim - Unified System for Automation and Robot Simulation

USARSim[13] é um simulador 3D baseado no motor de jogos Unreal Tournament[15], e é utilizado para investigação e educação. Foi eleito para ser utilizado na execução da competição de robôs virtuais da iniciativa de competição internacional de robótica Robocup[14]. Este simulador é utilizado em simulações de robôs para busca e salvamento em ambientes urbanos. Os modelos podem ser criados com a ferramenta UnrealEd que vem incluída no motor Unreal, ou importados formatos de modelação 3D compatíveis com o software usado correntemente na modelação tridimensional. O USARSim beneficia dos contínuos aperfeiçoamentos do motor Unreal e isso permite-lhe ter uma "renderização" bastante realista e uma simulação física de alta performance. Existem avaliações quantitativas que mostram uma correspondência estreita de resultados entre o USARSim e um sistema do mundo real. O código de controlo para controlar os robôs pode ser escrito usando: interface GameBot (protocolo de comunicação do motor Unreal), MOAST[16], interface para Player ou a toolbox de Matlab para USARSim[17]. O código de controlo dos robôs pode ser movido tal como foi escrito para robôs reais e vice-versa. Este simulador suporta uma variedade de sensores tais como sensores de toque, sensores de som, câmaras e lasers, usados habitualmente pelos investigadores de Robótica. O USARSim pode funcionar em sistemas operativos Windows, Linux e OS X. Na Figura 2 pode ver-se um ecrã do USARSim em que se

simula um *drone*.

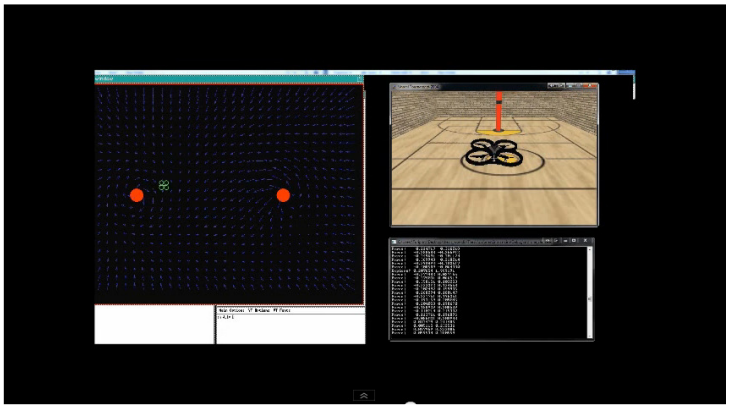

Figura 2: Ecrã do Simulador USARSim

Como foi referido anteriormente uma das características dos simuladores em Robótica é a visualização 3D próxima do real, para isto é necessário usar motores de "renderização" ou motores de jogos de computador. A renderização 3D é um processo que permite converter modelos 3D em imagens 2D de uma forma realista[18]. Um motor gráfico 3D é o software que realiza esse processo.

Na tabela 1 podem ver-se algumas características de alguns motores gráficos 3D e motores de jogos de computador, onde se incluem os motores referentes aos simuladores de robô apresentados. O motor de jogo Unity não está associado a nenhum simulador de robô que seja relevantemente utilizado, mas é uma plataforma para a criação de conteúdo tridimentsional na sua essência para utilização em jogos de computador mas também pode ser utilizado para a simulação, logo em simulação de robôs. O Unity[19,20,21,22] por si só é comparável aos simuladores de robô referidos anteriormente.

O Unity é uma plataforma muito interessante para a simulação em robótica, pois apresenta algumas vantagens[21] em relação a alguns simuladores de robô que são de difícil utilização devido à sua complexidade. O Unity contém uma documentação muito completa. Existe uma comunidade de utilizadores que pode ajudar novos utilizadores a iniciarem-se na plataforma. O Unity têm uma ferramenta de edição de cenas 3D com maior facilidade de utilização do que, p. e., o editor UnrealEd pertencente ao Unreal Tournament. No editor do Unity podem associar-se, *scripts* já existentes, aos objectos da cena 3D poupando tempo em termos de programação. Esses *scripts* podem dar comportamentos interactivos aos objectos. O Unity utiliza o motor PhysX[74], que é um motor de Física de referência. O Unity suporta várias APIs de renderização tais como Direct3D e OpenGL, proporciona também a utilização de *shaders* Cg/HLSL[40,41]. Os *shaders* GLSL do OpenGL podem ser gerados a partir da compilação dos *shaders* Cg/HLSL. E finalmente o Unity pode ser distribuído em várias plataformas, Windows e OS X e em web *browsers*. Fazendo uma pesquisa na *Internet* podem encontrar-se alguns projectos *open source* e académicos relacionados com a simulação de robôs, realizados em Unity, como são os casos do VirtualVEX[23], SARGE[24], Botnavsim[25], Excavator[26]. Excavator é um simulador de uma escavadora construído com o Unity, mas no entanto tem algumas semelhanças com a simulação de robôs.

Tabela 1.1: Motores gráficos 3D / motores jogos de computador

Nome	Linguagem Programação	Freeware/ Comercial	Suporte shaders	Formatos 3D	Motor Física	Sims. Robô
Blender Engine	Python	freeware	OpenGL GLSL	3ds Max, COLLADA, .OBJ, VRML, etc	Bullet	MORSE
OGRE	C++, wrappers Python e Java	freeware	DirectX HLSL, OpenGL GLSL	ferramentas convertem Blender e Maya para OGRE		Gazebo
OpenAR	C++	Propriedade ISR / UC	OpenGL GLSL	3ds Max, Blender, COLLADA, Wavefront .OBJ, etc	Bullet	
Unity	C#, Unity script, Boo	comercial, versão livre	Cg/HLSL	3ds Max, COLLADA,.fbx, .OBJ, etc	PhysX	
Unreal Tournament	C++, UnrealScript	comercial e SDK livre	UT3/4 DirectX HLSL e OpenGL GLSL	meshes Maya e 3ds Max importadas para UnrealEd	UT2 UT3 Kamal; UT4 PhysX	USARSim

1.2 Motivação, Objectivos e Contribuição

A utilização de simuladores de robôs permite ao investigador que trabalha em Robótica ter uma ferramenta poderosa que lhe permite escapar aos constrangimentos físicos que a máquina lhe impõe. No entanto os simuladores que existem possuem um determinado grau de complexidade o que dificulta a sua utilização. Procurando ir na senda de encontrar uma forma simples e realista para simular a utilização de um robô móvel a deslocar-se num cenário virtual, nesta tese experimentou-se realizar um simulador de robô que estivesse o mais próximo possível da realidade em termos de visualização e a adaptação da biblioteca OpenAR à versão 3.3 do OpenGL[27,28,29]. Essa visualização incluí o desenho dos modelos 3D e o movimento destes em termos das Leis da Física. Um cenário virtual que se pretenda que seja semelhante à realidade deve ter, p.e., aplicada aos

objectos a força da gravidade. A API OpenGL a partir da versão 3.3, particularmente com a utilização dos *vertex* e *fragment shaders*, juntamente com os motores de Física *open source* como é o caso do Bullet[9], usados actualmente, oferecem possibilidades de visualização de objectos 3D de uma forma próxima do real como seria expectável em jogos de computador comerciais ou cinema, com efeitos especiais produzidos em computador. A API OpenGL e os motores de Física *state-of-the-art* são as ferramentas fundamentais, que existem na actualidade para tentar alcançar o objectivo de obter uma peça de software como é o caso dos simuladores 3D e em particular um simulador de robô móvel.

Foi ainda definido como objectivo a integração de suporte para importação de modelos descritos num dos formatos mais genéricos e estáveis, o formato COLLADA[31]. Este formato é um standard aberto para troca de activos digitais entre aplicações de software[30] e permite obter modelos gerados a partir de aplicações de modelação 3D. Existem muitos pacotes de software que exportam neste formato como é o caso do 3ds Max, Blender, Cinema 4D, Maya, etc. Existe também um grupo alargado de motores de jogos de computador que utiliza este formato. Este formato é baseado num esquema XML[38] à semelhança dos ficheiros .sdf utilizados no simulador de robô Gazebo.

O OpenAR é uma framework 3D orientada a objectos para aplicações de realidade aumentada desenvolvida na linguagem C++, propriedade do Instituto de Sistemas e Robótica da Universidade de Coimbra. Actualmente suporta geometria básica e modelos baseados em malhas poligonais. Quando este trabalho foi iniciado no OpenAR estava implementada a versão 1 do OpenGL e podiam visualizar-se malhas poligonais de ficheiros apenas no formato Wavefront .OBJ[32]. O OpenAR utiliza o motor de Física Bullet[9] para podermos visualizar os efeitos da Física. Existem métodos implementados, em termos de Física, no OpenAR para definir corpos rígidos, corpos moles (soft-bodys), restrições físicas entre corpos rígidos, etc.

Esta tese inova no aspecto em que permite ter na biblioteca OpenAR maiores capacidades gráficas do que tinha anteriormente. A implementação da API OpenGL 3.3 e particularmente a utilização de *shaders* GLSL[33,34] por cada objecto 3D permite renderizar esses objectos de muitas e diferentes formas, utilizando texturas. Outra inovação é poderem visualizar-se malhas poligonais no formato COLLADA, além disso em conjunto com a implementação do formato COLLADA foi implementada a possibilidade de se visualizarem malhas poligonais de outros formatos como 3ds Max, Blender, LightWave, etc. Podemos ver também na Tabela 1 as características do OpenAR das quais os *shaders* GLSL e o suporte de modelos 3D em COLLADA e outros formatos foram realizados nesta tese. Estas novas funcionalidades permitiram construir um protótipo para simulação de robô e *drone* muito realista. Outra inovação foi a deformação de malhas poligonais utilizando um esqueleto articulado.

1.3 Estrutura da Tese

A tese está estruturada em quatro capítulos e alguns anexos. O capítulo 2 pretende transmitir em primeiro lugar conhecimento sobre o que são as malhas poligonais, como são as estruturas que as guardam e ainda algumas das características principais do formato COLLADA. A secção 2.2 explica algumas estruturas de dados que existem para armazenar malhas poligonais e que são: Malhas vértice-vértice, Malhas face-vértice, Malhas *Winged-edge* e *Triangle Strip*. Na secção 2.3 faz-se uma pequena descrição do formato *Wavefront* (secção 2.3.1) e nas secções 2.3.2 faz-se uma descrição das principais características do formato COLLADA: Geometria, Cena 3D, Luzes,

Câmara, Animações, *Skinning* e Física.

O capítulo 3 é uma descrição da contribuição do trabalho realizado pelo autor na biblioteca OpenAR e a construção do protótipo de simulador de robô e *drone*. Na secção 3.1 faz-se uma explanação da utilização de *shaders* para visualização de malhas poligonais e uma descrição do funcionamento das classes escritas em C++ para visualização das malhas e carregamento destas a partir dos ficheiros COLLADA. Na secção 3.2 faz-se uma descrição do funcionamento do protótipo de simulador de robô e *drone*.

O capítulo 4 é um capítulo de resultados no qual estão referenciadas as modificações que foram operadas no OpenAR (secção 4.1) e finalmente tiram-se conclusões tendo em conta os objectivos que foram propostos inicialmente e os resultados que se conseguiram atingir, propõe ainda novas funcionalidades para a biblioteca OpenAR (secção 4.2).

Capítulo 2: Malhas poligonais

2.1 Introdução

A Modelação Geométrica[42] é o campo da Matemática que discute os métodos matemáticos que estão por trás da modelação de objectos reais usados em Computação Gráfica e Desenho Assistido por Computador e Fabrico Assistido por Computador (CAD/CAM). Esses métodos descrevem formas, formas essas que podem ser 2D ou 3D, as formas que interessam para este estudo são as formas tridimensionais, pois permitem representar objectos num ambiente de realidade virtual ou aumentada. Um modelo tridimensional representa um objecto tridimensional usando uma colecção de pontos no espaço ligados por várias entidades geométricas como triângulos, linhas, superfícies curvas, etc[48].

Os tipos de modelos geométricos[43,44] que existem são os seguintes: modelos wireframe (arestas 3D), modelos de sólidos por composição de primitivas, modelos obtidos por Geometria Construtiva de Sólidos (CSG), modelos obtidos por modelação por varrimento (sweep), modelação por superfície e modelação por volume. Neste capítulo o estudo vai incidir nos modelos superfície. Os modelos superfície subdividem-se em modelos implícitos e malhas poligonais. Nos modelos implícitos existem as quádricas e as superfícies paramétricas.

A construção de modelos geométricos pode ser feita com sistemas de Desenho Assistido por Computador (CAD) e sistemas de aquisição de dados. A aquisição de dados 3D é a geração tridimensional ou espaço temporal de modelos a partir de dados de sensores. Algumas formas de representar objectos 3D com aquisição de dados estão descritas resumidamente nos Anexos desta dissertação. Sistemas de software de Desenho Assistido por Computador (CAD) são sistemas de software de computador que permitem projectar e realizar desenho técnico. Estes programas têm a capacidade de se poder desenhar em 2D e 3D. São usados para aumentar a produtividade do *Designer* e melhorar a qualidade do *design* do produto.

As aplicações dos modelos geométricos são como já vimos anteriormente o CAD/CAM, a Realidade Virtual e Aumentada, Processamento de Imagens em Medicina, Jogos de Computador, Sistemas de Informação Geográfica, Imagens Médicas com Sólidos, etc.

É mais simples fazer modelação com modelos superfície do que com os modelos sólidos. Pretendendo-se nesta tese realizar a visualização de modelos geométricos esta dissertação vai debruçar-se no estudo das malhas poligonais que são um subgrupo dos modelos superfície.

Uma *mesh* ou malha poligonal[45] é um conjunto de vértices, arestas e faces que definem a forma de um poliedro em Computação Gráfica 3D, como se pode ver na Figura 3. Vértice é um ponto partilhado por duas arestas. Uma aresta é uma ligação entre dois vértices. Uma face é um conjunto de arestas, em que uma face triangular tem três arestas, e uma face quadrilátero tem quatro arestas. Um polígono é conjunto co planar de faces.

Uma malha poligonal é constituída por faces, faces essas que mais usualmente são triângulos como se pode ver na Figura 4. Consoante o número de triângulos que constituem uma malha poligonal pode ter-se uma melhor definição do objecto 3D a representar. Na Figura 5 podem ver-se modelos simplificados de uma malha poligonal. A malha poligonal original é constituída por 69451 triângulos (a), e nas figuras (b), (c), (d) e (e) podem ver-se malhas poligonais simplificadas com 17364, 4341, 1086, e 272 triângulos, respectivamente[46]. Quanto mais triângulos existirem

maior será o nível de detalhe (LOD) de uma malha poligonal, mas também mais espaço essa malha ocupa em memória.

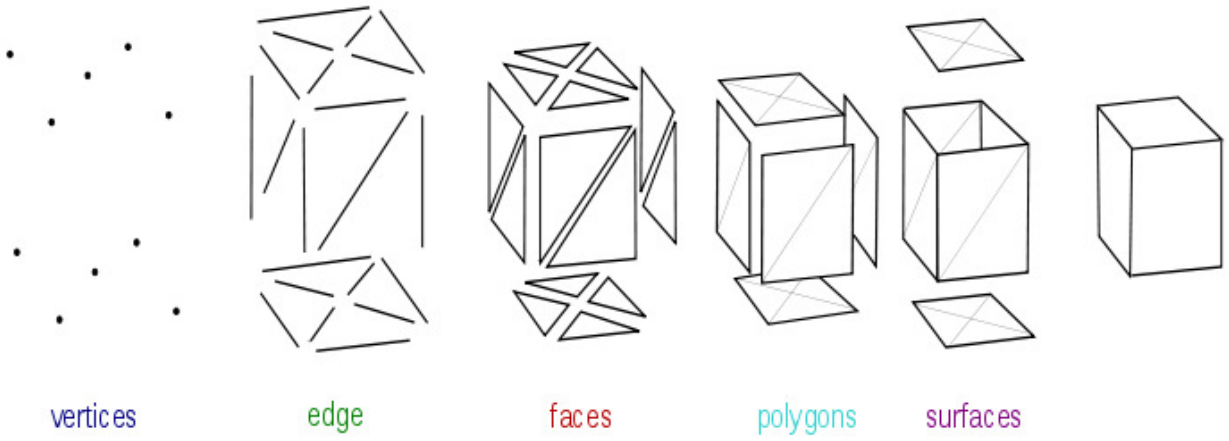

Figura 3: Elementos de uma malha poligonal (vértices, arestas, faces, polígonos e superfícies)

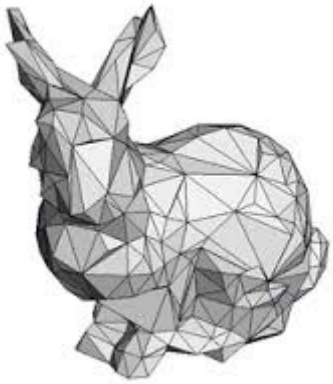

Figura 4: Malha poligonal de triângulos

Existem aplicações em que vértices, arestas e faces ou polígonos constituem a informação a guardar numa estrutura de dados, para se poder representar uma malha poligonal. As coordenadas dos vértices são a informação geométrica a guardar. A informação topológica (conectividade) é a informação que diz como se organizam as faces e as arestas. Existem ainda propriedades adicionais que se podem armazenar, como vector normal a cada face/vértice e cor ou coordenadas de textura[47].

Nas secções seguintes são expostas várias estruturas de dados para armazenar malhas poligonais e faz-se ainda uma descrição do formato de ficheiro COLLADA.

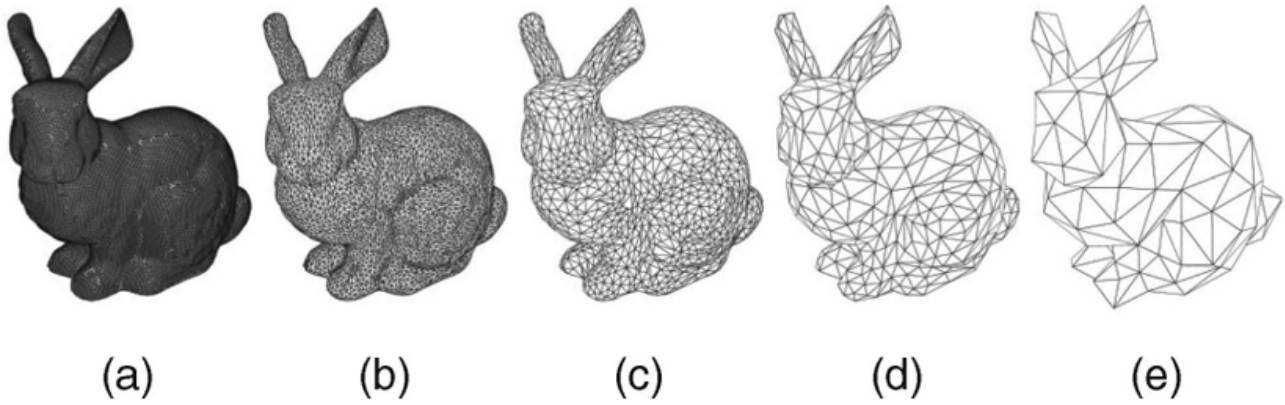

Figura 5: Malhas poligonais simplificadas

2.2 Formas de representar malhas poligonais

A escolha de uma estrutura de dados para uma malha poligonal tem que ver com o compromisso que deve existir entre velocidade de renderização que se pretende e a facilidade de transformação da malha. Existem vários métodos correspondentes a diferentes estruturas de dados para armazenar a informação dos polígonos das malhas poligonais, ou seja, vértices, arestas e faces. Estes métodos são explicados nas secções seguintes.

É importante referir os *Index Arrays*[50] como uma base para representar malhas poligonais como a estrutura face-vértice que se pode ver na secção 2.2.2 e também como base para a estrutura *Triangle Strip* na secção 2.2.4. Nos *Index Arrays* uma malha poligonal é representada por dois *arrays*, um que guarda as coordenadas dos vértices e outro que guarda conjuntos de 3 índices que definem um triângulo como se pode ver na Figura 6[58].

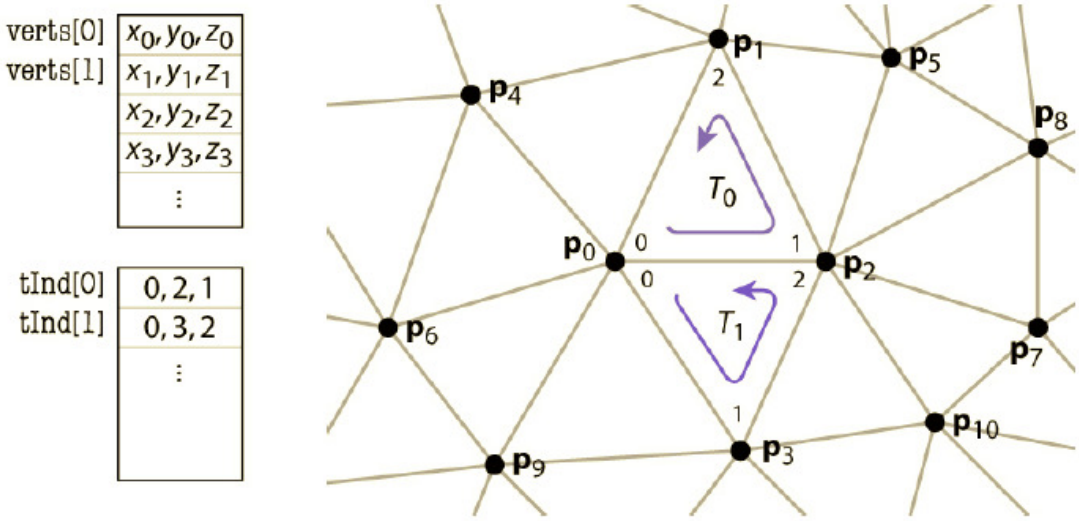

Figura 6: Triângulos indexados (Index Array)

Os *Index Arrays* são utilizados em OpenGL e DirectX. Em OpenGL existe a primitiva *glDrawElements* que suporta os *Index Arrays* quando se usa um *Vertex Buffer Object* (VBO)[52]. Quando se utiliza o VBO podem existir outros *arrays* para coordenadas de textura, vectores normais e juntas de esqueleto (no caso de malhas poligonais com animação). No OpenAR foi implementado *Vertex Buffer Object* para fazer a visualização de malhas poligonais, quando se procedeu à alteração para OpenGL 3.3. Os *Index Arrays* também podem ser referidos como triângulos indexados.

2.2.1 Malhas vértice-vértice ou Malha VV

As malhas poligonais vértice-vértice[44,45] são uma estrutura de dados que representa um objecto de uma forma simples e ocupa pouco espaço de armazenamento em comparação com outras estruturas de dados. Um objecto representa-se como um conjunto de vértices ligados entre si, cada vértice indexa os vértices vizinhos. Esta representação tem a desvantagem de fazer perder tempo de processamento devido ao facto de existirem arestas que são desenhadas duas ou mais vezes, como se pode ver na tabela da Figura 7, a aresta v1,v5 é desenhada como se pode ver na primeira linha da tabela e depois é desenhada outravez na 7ª linha da tabela. Outra desvantagem é a informação das faces e das arestas ser implícita, ou seja, é necessário pesquisar os dados para obter as faces e as arestas. Na Figura 7 pode ver-se um exemplo de um cubo com a estrutura de malhas vértice-vértice.

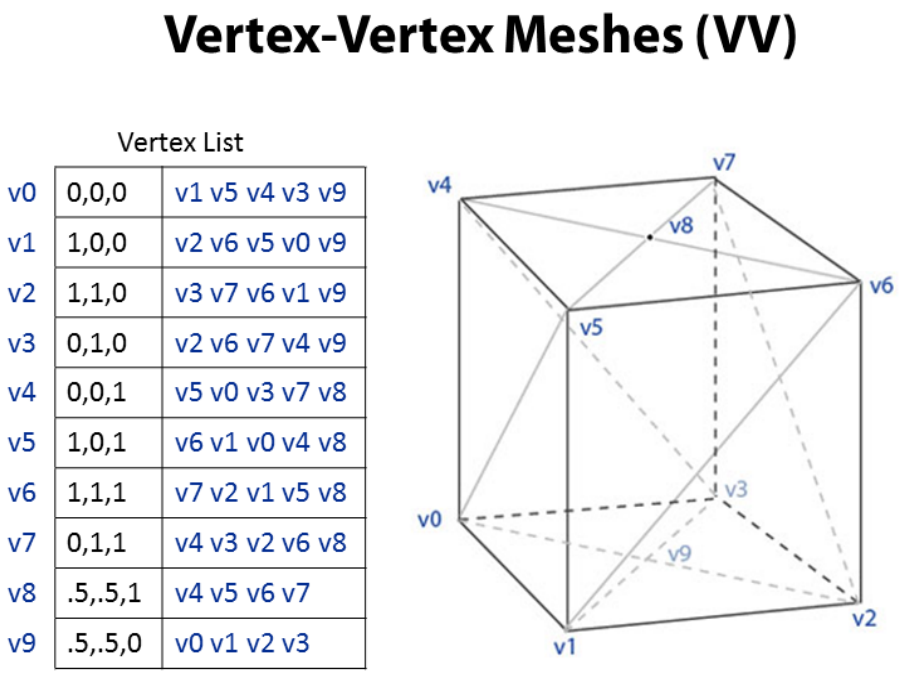

Figura 7: Malhas poligonais vértice-vértice

2.2.2 Malhas face-vértice ou Malha FV

Na estrutura de dados de malhas poligonais face-vértice[44,45], um objecto é representado com

uma lista de vértices e uma lista de faces. As malhas face-vértice permitem uma pesquisa explícita dos vértices de uma face e das faces rodeiam um vértice, no entanto a informação das arestas é implícita, é necessário fazer uma pesquisa nos dados para as obter. Esta estrutura de dados é usada para "renderização" quando não se pretende modificar a geometria do modelo. É uma estrutura usada no formato de ficheiro Wavefront .OBJ. Na Figura 8 pode ver-se um exemplo de um cubo com a estrutura de dados malha FV.

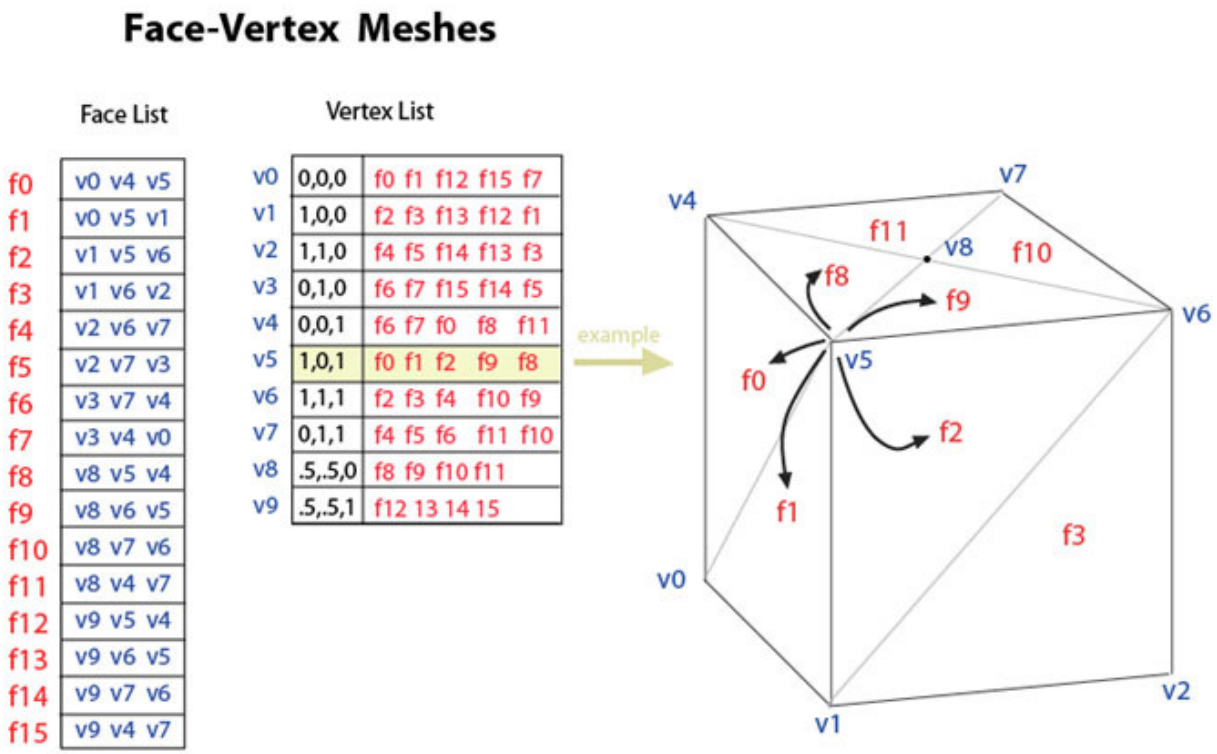

Figura 8: Malhas poligonais face-vértice

2.2.3 Malha Winged-edge ou Malha de aresta alada

As malhas de aresta alada têm este nome porque a cada aresta estão associadas duas faces, fazendo uma analogia com uma borboleta percebe-se este conceito imaginando que o centro da borboleta é a aresta e as asas são as faces adjacentes. Nas malhas de aresta alada[44,45] os vértices, faces e arestas são obtidos de forma explícita. Esta estrutura permite a modificação dinâmica da geometria de uma malha por isso esta representação é usada em programas de modelação. Essa modificação dinâmica é possível porque as operações de separação e junção são feitas com rapidez. Tem as desvantagens de necessitar de muito espaço de armazenamento e ter uma complexidade que vai aumentando devido a terem que se manter muitos índices. Na Figura 9 pode ver-se o exemplo de um cubo com a estrutura de dados de aresta alada.

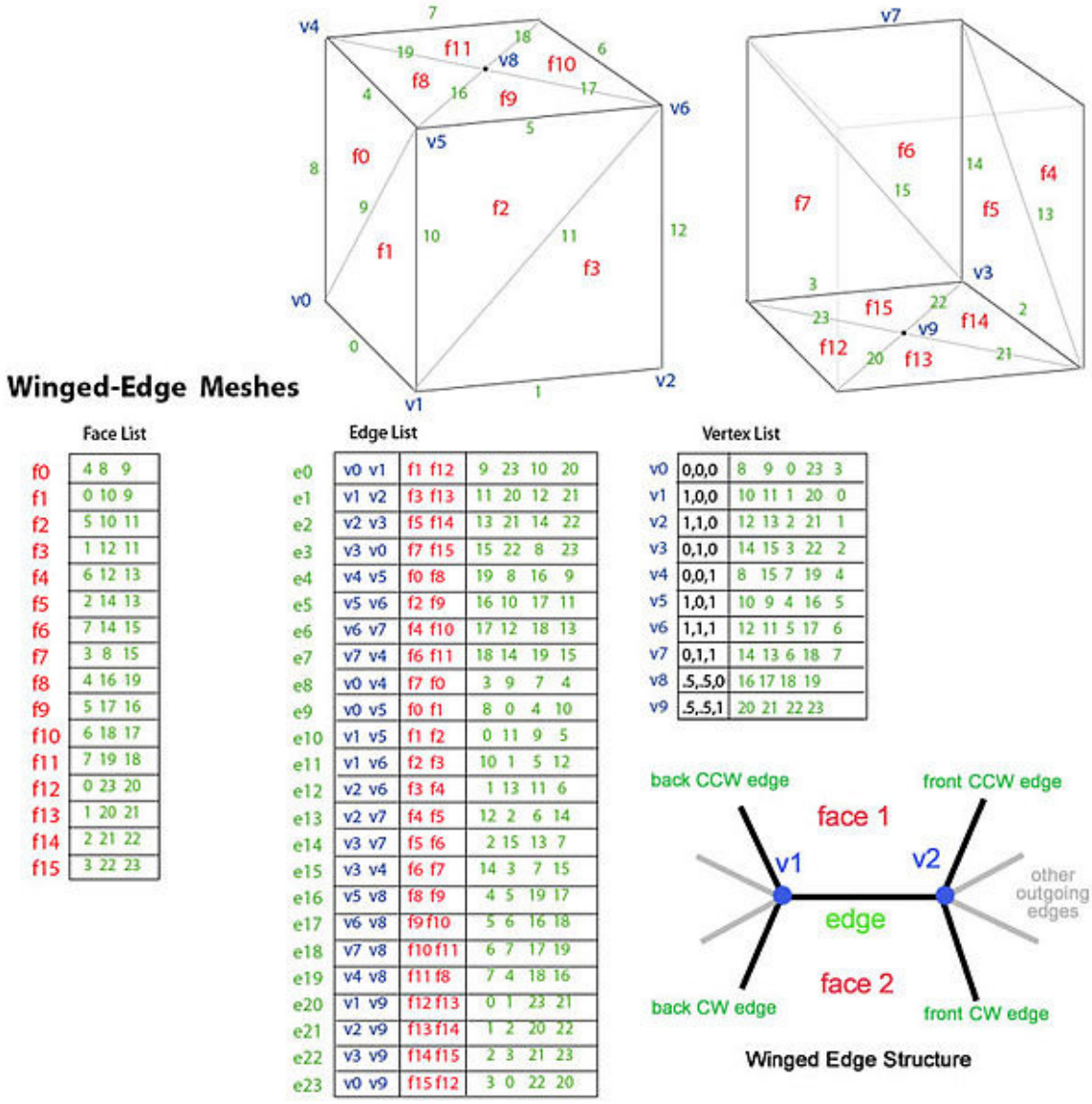

Figura 9: Malhas poligonais de aresta alada

2.2.4 Triangle Strip

Na estrutura de dados para representação de objectos 3D, *Triangle strip*[50,51], cada triângulo partilha uma aresta com o triângulo vizinho e assim sucessivamente. Na Figura 10, podemos ver um exemplo de como funciona esta estrutura de dados. Se não utilizássemos os *Triangle strips* construiríamos os triângulos da seguinte forma: ABC, CBD, CDE e DEF. Com *Triangle strips* tem-se a seguinte sequência ABCDEF. Uma vez que há partilha de uma aresta o número de vértices

guardados em memória é reduzido de 3N para N+2, para se desenharem N triângulos. O espaço ocupado em disco é também menor do que noutras estruturas de dados e é mais rápido carregar a malha poligonal para a memória do computador. O método de utilização de *Triangle strips* é utilizado em DirectX e OpenGL. Este método tem a desvantagem de ter um custo proibitivo se se pretender alterar a conectividade da malha[49].

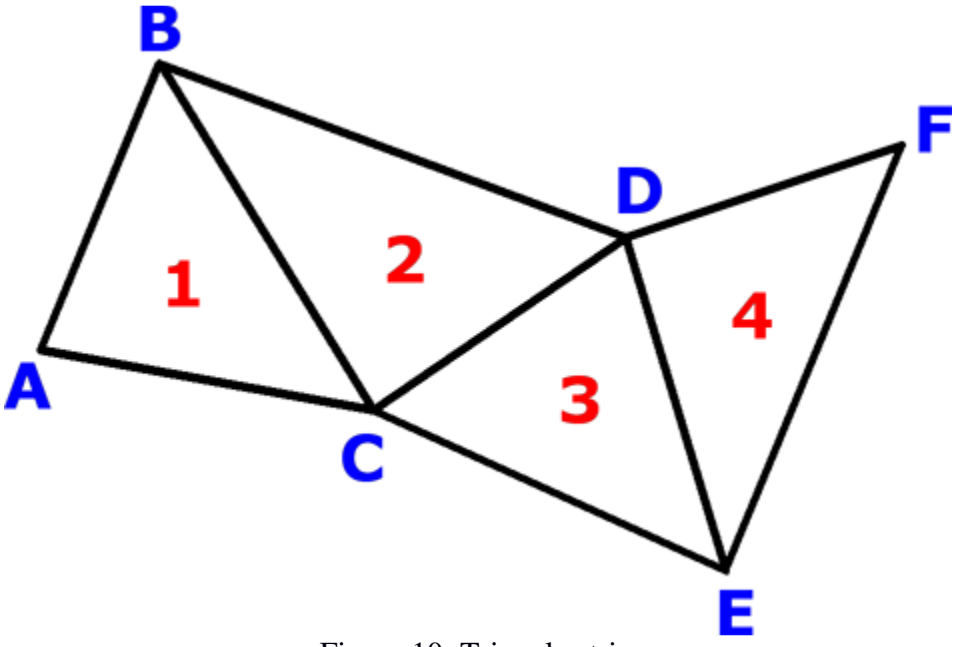

Figura 10: Triangle strip

2.3 Malhas, formatos de ficheiro

As malhas poligonais podem ocupar bastante espaço de memória assim é necessário armazená-las em disco. Existem bastantes formatos de ficheiro para guardar os dados das malhas poligonais[45]. Nesta secção faz-se uma breve explicação sobre o formato de ficheiro Wavefront .obj, mas o que é fundamental na secção 2.3 é o formato de ficheiro de activos digitais COLLADA.

2.3.1 Formato de ficheiro Wavefront .obj

O Wavefront .OBJ[53] era o formato de ficheiro que se podia utilizar no OpenAR com OpenGL versão 1. A seguir faz-se uma breve descrição deste formato. Este formato utiliza a estrutura de dados malhas poligonais face-vértice visto na secção 2.2.2. O formato com extensão .OBJ é um formato aberto e aceite universalmente, representa a geometria 3D de um objecto, a posição de cada vértice, a posição UV de cada vector com coordenada de textura, vectores normais e faces.

As texturas são *arrays* bidimensionais constítuidos por valores de cores. As coordenadas de textura são a localização da textura e servem para associar imagens aos polígonos que formam as malhas poligonais por forma a visualizarem-se os modelos 3D com mais realismo. O mapeamento UV é o processo que projecta um mapa de textura num modelo 3D, ou seja, é o processo que fornece as coordenadas de textura aos vértices do modelo 3D. As letras "U" e "V" denotam os eixos

da textura 2D porque "X", "Y" e "Z" são usados para denotar os eixos do modelo 3D no espaço[54]. Na Figura 11 pode ver-se este processo.

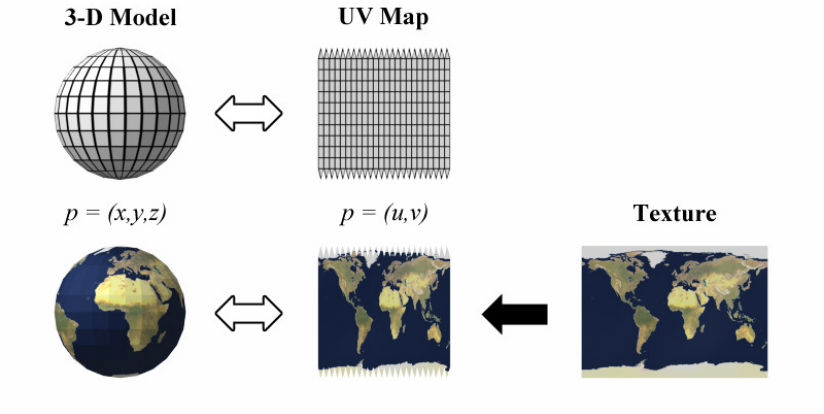

Figura 11: Aplicação de textura UV a um modelo 3D

Um vector normal é mais um atributo de um vértice tal como as coordenadas de textura. Em OpenGL quando a luz está activada os vectores normais são usados para determinar quanta luz é recebida por um vértice específico ou superfície[65].

A seguir podem ver-se exemplos do conteúdo do formato Wavefront .OBJ, os elementos do ficheiro escrevem-se com seguinte notação[53]:

Lista dos vértices com coordenadas (x,y,z[,w]), w é igual a 1.0 e é opcional:

```
v 0.123 0.234 0.345 [1.0]
v ...
```

Lista de coordenadas de textura em coordenadas UV, (u, v[,w]), w é igual a 0 e é opcional:

```
vt 0.500 1 [0]
vt ...
```

Lista de vectores normais nas coordenadas (x, y, z):

```
vn 0.707 0.000 0.707
vn ...
```

As faces são definidas usando listas com os índices dos vértices, texturas e normais.

Lista com índices dos vértices:

```
f v1 v2 v3 ....
```

Lista com índices dos vértices e índices das coordenadas de textura:

```
f v1/vt1 v2/vt2 v3/vt3 ...
```

Lista com índices dos vértices, índices das coordenadas de textura e índices de vectores normais:

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 ....
```

Lista com índices dos vértices e índices de vectores normais:

```
f v1//vn1 v2//vn2 v3//vn3 ...
```

2.3.2 Formato de ficheiro COLLADA

COLLADA[55] significa COLLAborative Design Activity. O formato COLLADA surgiu como solução para um problema com bastantes anos e que era o problema de se conseguir fazer intercâmbio de conteúdos 3D entre aplicações devido a existirem muitos formatos proprietários. É um formato de ficheiro para conteúdos digitais em aplicações com gráficos tridimensionais utilizando um esquema XML[56] standard aberto. É um formato identificável pela extensão .DAE, mas também existe uma extensão .ZAE que indica que é um ficheiro COLLADA com compressão.

Um documento COLLADA é um documento XML, por isso é necessário ter em consideração algumas definições[59]:

1. Elemento:
   - um documento XML consiste em vários elementos;
   - os elementos estão encapsulados formando uma estrutura hierárquica;
   - o primeiro elemento é chamado raiz (root);
   - cada elemento tem uma *tag* de início, um conteúdo e uma *tag* de fim como se pode ver no exemplo: `<elemento> conteúdo </elemento>`;
   - um elemento pode ter conteúdo vazio, nesse caso pode escrever-se apenas `<elemento/>`;

2. Atributo:
   - um elemento pode conter um ou mais atributos como se pode ver no exemplo:
   `<elemento atrib1="valor1" atrib2="valor2" > conteúdo </elemento>`
   - o valor tem que estar entre aspas
   - os atributos contêm *meta-data* acerca do elemento

Exemplo de um documento COLLADA:

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA
xmlns="http://www.collada.org/2005/11/COLLADASchema"
version="1.4.0">
</COLLADA>
```

Este exemplo contém em primeiro lugar o cabeçalho que indica que se trata de um documento XML. Contém ainda o elemento `<COLLADA>`, onde se pode ver o atributo `xmlns` que indica o *namespace schema* e que é um esquema COLLADA 1.4 e o atributo `version` que indica que se trata da especificação COLLADA com a versão 1.4.0[60]. O elemento `<COLLADA>` é o elemento raiz do documento COLLADA e contem vários elementos filhos e atributos, como se pode ver na Figura 12.

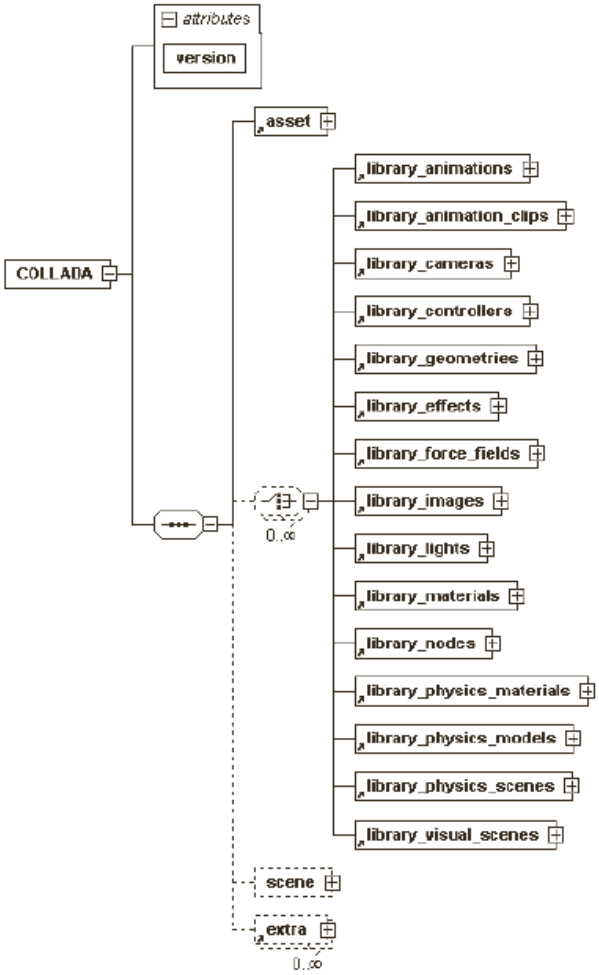

Figura 12: Elemento COLLADA, atributos e nós filhos

Algumas das características mais importantes do formato COLLADA 1.4.0 são:

- Geometria de malhas
- Materiais
- Texturas
- Luzes
- Câmaras
- Animação

- Skinning[61] - processo de ligação dos vértices de uma malha ao esqueleto da malha para animação
- Morphing[62] - o Morphing de uma malha é a criação de uma versão da malha deformada em que se guardam as posições dos vértices (exemplo: animação 3D da face humana)
- Animação
- Física (propriedades físicas, interacção entre corpos rígidos)
- Efeitos - com um efeito podem colorir-se objectos de uma cena, é também possível ter esse efeito descrito em diferentes plataformas correspondentes a diferentes perfis de linguagens de *shaders* ou APIs (Cg, GLSL, GLES)
-

Algumas características importantes do formato COLLADA 1.5.0 são:

- Geometria B-Rep - As representações por fronteira (B-Rep) são uma forma de obter modelos de objectos 3D tal como as malhas poligonais mas o que se tem em conta são as arestas e superfícies em vez de se guardar cada ponto[63,64].
- Cinemática - Numa cena 3D podem ter-se modelos com propriedades cinemáticas. Um modelo cinemático é controlado por um mais sistemas articulados.
- Arquivos com compressão (extensão .ZAE)
- Efeitos - perfil GLES2

Seria muito exaustivo explicar todas estas características nesta dissertação no entanto todas estas e outras características ou funcionalidades podem ser encontradas com grande pormenor nas especificação 1.5.0 do formato COLLADA[57] e no livro *Collada: Sailing the Gulf of 3d Digital Content Creation*[59].

Nesta dissertação optou-se por dar mais importância às características que pudessem ser mais importantes para se poder representar visualmente uma malha poligonal, ou seja, Geometria das malhas, Cena 3D, Luzes, Câmara, Animações, *Skinning* e Física. A descrição destas características foi pensada tendo em vista uma possível implementação de classes em C++ que pudessem ler esta informação de um ficheiro em formato COLLADA.

Geometria

Como se pode ver na Figura 12, existem várias bibliotecas, onde a que contém os dados geométricos corresponde ao elemento `<library_geometries>`.

Os dados guardam-se em *arrays*, assim foram definidos os seguintes tipos de *arrays*:

- `<float_array>` para guardar números de vírgula flutuante
- `<int_array>` para guardar números inteiros
- `<bool_array>` para guardar valores booleanos
- `<IDREF_array>` para guardar referências a IDs
- `<Name_array>` para guardar nomes que poderão corresponder a identificadores (SIDs) de alguns elementos

Cada *array* tem um atributo `count` obrigatório que serve para pré-alocar espaço de memória.

Vamos tomar em consideração o seguinte exemplo para se verificar como se descreve a geometria de uma malha para representar um cubo[57].

```
<library_geometries>
	<geometry id="box" name="box">
	<mesh>
		<source id="box-Pos">
			<float_array id="box-Pos-array" count="24">
				-0.5 0.5 0.5
				0.5 0.5 0.5
				-0.5 -0.5 0.5
				0.5 -0.5 0.5
				-0.5 0.5 -0.5
				0.5 0.5 -0.5
				-0.5 -0.5 -0.5
				0.5 -0.5 -0.5
			</float_array>
			<technique_common>
				<accessor source="#box-Pos-array" count="8" stride="3">
					<param name="X" type="float" />
					<param name="Y" type="float" />
					<param name="Z" type="float" />
				</accessor>
			</technique_common>
		</source>
		<source id="box-0-Normal">
			<float_array id="box-0-Normal-array" count="18">
				1.0 0.0 0.0
				-1.0 0.0 0.0
				0.0 1.0 0.0
				0.0 -1.0 0.0
				0.0 0.0 1.0
				0.0 0.0 -1.0
			</float_array>
			<technique_common>
				<accessor source="#box-0-Normal-array" count="6" stride="3">
					<param name="X" type="float"/>
					<param name="Y" type="float"/>
					<param name="Z" type="float"/>
				</accessor>
			</technique_common>
		</source>
		<vertices id="box-Vtx">
			<input semantic="POSITION" source="#box-Pos"/>
		</vertices>
		<polygons count="6" material="WHITE">
			<input semantic="VERTEX" source="#box-Vtx" offset="0"/>
			<input semantic="NORMAL" source="#box-0-Normal" offset="1"/>
			<p>0 4 2 4 3 4 1 4</p>
			<p>0 2 1 2 5 2 4 2</p>
			<p>6 3 7 3 3 3 2 3</p>
			<p>0 1 4 1 6 1 2 1</p>
			<p>3 0 7 0 5 0 1 0</p>
			<p>5 5 7 5 6 5 4 5</p>
		</polygons>
		</mesh>
	</geometry>
</library_geometries>
```

O primeiro elemento é o elemento `<geometry>` e que tem como atributos um *id* e *name* onde se identifica o objecto que se pretende representar. Assim tem-se:

```
<geometry id="box" name="box">
```

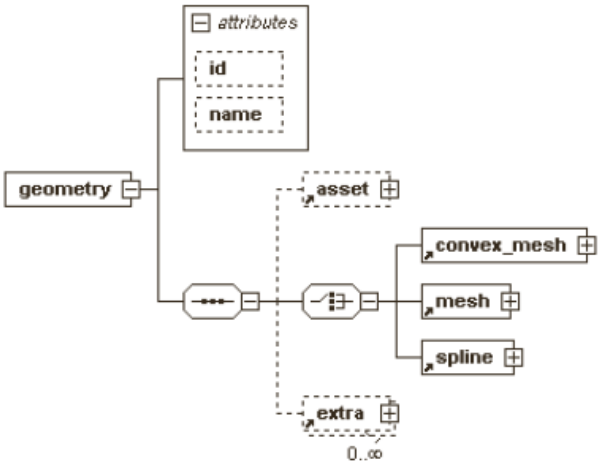

Figura 13: Elemento geometry

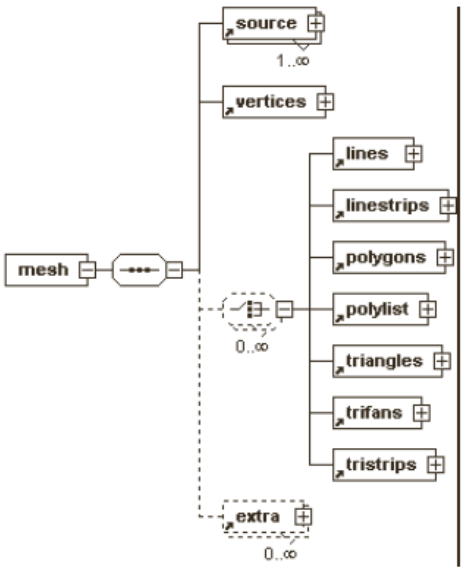

Figura 14: Elemento mesh

O elemento `<geometry>` é o elemento que contém toda a informação que descreve a forma geométrica. Pode com este elemento descreverem-se malhas poligonais mas também outras geometrias com se pode ver na Figura 13, *splines* e *convex meshes*. O elemento *convex mesh* serve para representar uma malha física que se utiliza com a Física COLLADA e que se pode visualizar. O elemento *spline* serve para representar uma curva spline multisegmento[59].

A seguir, no exemplo, aparece o elemento `<mesh>`. Este elemento não tem atributos e

guarda os dados não tratados em um ou mais elementos <source> como se pode ver no exemplo. Na Figura 14 podemos ver uma representação do elemento <mesh>. Neste elemento podemos criar uma malha poligonal com diferentes primitivas geométricas, <lines>, <linestrips>, <polygons>, <polylist>, <triangles>, <trifans> e <tristrips>. No exemplo é usada a primitiva <polygons> [59].

A seguir surge o primeiro elemento <source> :

```
<source id="box-Pos">
```

está identificado com o valor box-Pos. As coordenadas dos vértices do cubo estão no *array* de vírgula flutuante indicado pelo elemento <float_array>. A seguir a este *array* encontra-se o elemento <technique_common>, que indica que se trata de informação para um elemento específico de um perfil comum. A seguir vem o elemento <accessor> que declara que a *source* em causa tem coordenadas tridimensionais X, Y e Z e a que tipo de dados se refere, que neste caso é *float*. No elemento <accessor>[59] existem dois atributos importantes, o atributo count que indica o número de vezes que o *array* vai ser acedido, no exemplo existem 8 vértices logo é acedido 8 vezes. Existe também o atributo stride que indica o número de valores que devem ser considerados como unidade em cada acesso, neste caso é igual a 3 porque cada vértice tem 3 coordenadas. Repetindo apenas este bloco com os dados dos vértices do cubo tem-se:

```
<source id="box-Pos">
	<float_array id="box-Pos-array" count="24">
		-0.5 0.5 0.5
		0.5 0.5 0.5
		-0.5 -0.5 0.5
		0.5 -0.5 0.5
		-0.5 0.5 -0.5
		0.5 0.5 -0.5
		-0.5 -0.5 -0.5
		0.5 -0.5 -0.5
	</float_array>
	<technique_common>
		<accessor source="#box-Pos-array" count="8" stride="3">
			<param name="X" type="float" />
			<param name="Y" type="float" />
			<param name="Z" type="float" />
		</accessor>
	</technique_common>
</source>
```

A seguir aparece outro bloco *source* com atributo identificador box-0-Normal e neste bloco estão os vectores normais, que são vectores perpendiculares às faces do cubo. Os vectores normais servem para determinar quanta luz é recebida num vértice ou numa face[65].

```
<source id="box-0-Normal">
	<float_array id="box-0-Normal-array" count="18">
		1.0 0.0 0.0
		-1.0 0.0 0.0
		0.0 1.0 0.0
		0.0 -1.0 0.0
		0.0 0.0 1.0
		0.0 0.0 -1.0
```

```
            </float_array>
            <technique_common>
                  <accessor source="#box-0-Normal-array" count="6" stride="3">
                        <param name="X" type="float"/>
                        <param name="Y" type="float"/>
                        <param name="Z" type="float"/>
                  </accessor>
            </technique_common>
      </source>
```

No elemento `<accessor>` pode ver-se que existem 6 vectores normais em que cada vector é normal à face do cubo correspondente. Existem 6 faces por isso `count` é igual a 6 e o atributo `stride` é igual a 3 porque também aqui as coordenadas são X, Y e Z.

A seguir no exemplo apresentado aparece o elemento `<vertices>`[59] em que os vértices da malha poligonal são declarados. Este elemento contém o elemento `<input>` onde se declara que trata de informação posicional e também onde está a localizada essa informação dos vértices.

```
            <vertices id="box-Vtx">
                  <input semantic="POSITION" source="#box-Pos"/>
            </vertices>
```

A seguir declara-se a geometria que vai descrever a malha poligonal e no exemplo a primitiva geométrica utilizada são polígonos de n-lados, tem-se assim o elemento `<polygons>`. Para uma representação mais eficiente devem usar-se outras primitivas como `<triangles>` ou `<polylist>`[57].

```
      <polygons count="6" material="WHITE">
            <input semantic="VERTEX" source="#box-Vtx" offset="0"/>
            <input semantic="NORMAL" source="#box-0-Normal" offset="1"/>
            <p>0 4   2 4   3 4   1 4</p>
            <p>0 2   1 2   5 2   4 2</p>
            <p>6 3   7 3   3 3   2 3</p>
            <p>0 1   4 1   6 1   2 1</p>
            <p>3 0   7 0   5 0   1 0</p>
            <p>5 5   7 5   6 5   4 5</p>
      </polygons>
```

O atributo `count` diz quantos polígonos vão formar o modelo 3D que no exemplo são 6 correspondendo às 6 faces do cubo. O atributo `material` indica que os polígonos que formam o cubo estão associados a um material identificado por `WHITE`. Cada elemento `<p>` corresponde a um polígono. Os elementos `<input>` indicam que se tratam de dados com informação de vértices e normais. O conteúdo do elemento `<p>` é um *array* de inteiros que são os índices dos vértices e normais. O exemplo termina com o fecho das *tags* da malha poligonal, geometria e biblioteca de geometrias:

```
            </mesh>
      </geometry>
</library_geometries>
```

A seguir pode ver-se outro exemplo abreviado de uma malha poligonal onde se pode verificar como

está organizada a informação do polígono. Neste exemplo existe apenas um polígono que é um quadrado. Para além de dados dos vértices e normais também existem coordenadas de textura e tangentes à textura, assim com esta informação nesta malha poligonal poderia ter-se Mapeamento Normal que é uma técnica aplicada às texturas que permite ter mais detalhe sem acrescentar polígonos[66].

```
<mesh>
	<source id="position"/>
	<source id="normal"/>
	<source id="tex-coord"/>
	<source id="tex-tangent"/>
	<vertices id="verts">
		<input semantic="POSITION" source="#position"/>
	</vertices>
	<polygons count="1" material="Bricks">
		<input semantic="VERTEX" source="#verts" offset="0"/>
		<input semantic="NORMAL" source="#normal" offset="1"/>
		<input semantic="TEXCOORD" source="#tex-coord" offset="2" set="0"/>
		<input semantic="TEXTANGENT" source="#tex-tangent" offset="3" set="0"/>
		<p>0 0 0 1   2 1 2 0   3 2 1 2   1 3 3 3</p>
	</polygons>
</mesh>
```

No seguinte exemplo de uma malha poligonal declarada no formato COLLADA a primitiva geométrica utilizada são triângulos, elemento `<triangles>`. Os dados têm informação de vértices, normais e coordenadas de textura. A malha poligonal tem dois triângulos definidos.

```
<mesh>
	<source id="position"/>
	<source id="normal"/>
	<source id="textureCoords"/>

	<vertices id="verts">
		<input semantic="POSITION" source="#position"/>
	</vertices>

	<triangles count="2" material="Bricks">
		<input semantic="VERTEX" source="#verts" offset="0"/>
		<input semantic="NORMAL" source="#normal" offset="1"/>
		<input semantic="TEXCOORD" source="#textureCoords" offset="2" set="1" />
		<p>
			0 0 0  1 3 2  2 1 3
			0 0 0  2 1 3  3 2 2
		</p>
	</triangles>
</mesh>
```

Cena

Cada documento COLLADA pode conter uma cena. Uma cena pode ser um objecto 3D ou vários objectos 3D, pode ser uma personagem animada ou simplesmente um cubo, etc. Uma cena 3D são os objectos que se visualizam e as simulações físicas associadas a esses objectos. Em COLLADA é com o elemento `<scene>` que se declara uma cena 3D.

As razões que determinaram que um elemento COLLADA apenas pudesse conter uma

cena[59] são:

- cada documento é a unidade de armazenamento e transmissão do conteúdo codificado em XML;
- é mais fácil discernir o conteúdo que está no documento que representa a cena 3D;
- as ferramentas de *software* podem mais facilmente carregar o conteúdo de uma só cena;
- a relação de um para um entre documento e cena 3D foi projectado tendo em vista a existência de simplicidade.

Exemplo do elemento `<scene>`[59] :

```
<scene>
       <instance_physical_scene url="#earthly_physics"/>
       <instance_visual_scene url="#earthly_visuals"/>
       <extra type="Weather"/>
</scene>
```

O elemento `<instance_physical_scene>` instancia a simulação física, o elemento `<instance_visual_scene>` instancia a cena com os objectos visuais. Estas instanciações são declaradas efectivamente nos elementos biblioteca correspondentes:

```
<library_physics_scenes>
       <physics_scene id="earthly_physics">
       </physics_scene>
</library_physics_scenes>

<library_visual_scenes>
       <visual_scene id="earthly_visuals">
       </visual_scene>
</library_visual_scenes>
```

O elemento `<visual_scene>` contém a informação visual da cena. Cada elemento `<visual_scene>` tem que ter pelo menos um elemento filho `<node>`, estes elementos contêm a informação sobre geometria, luzes e câmara que é a informação que constitui uma cena 3D.

No exemplo seguinte podemos ver a cena 3D criada para visualizar o cubo visto num exemplo anterior. Esse exemplo mostrava como declarar a geometria do cubo. Neste exemplo da declaração de uma cena pode ver-se como se instancia a geometria do cubo e como se define a cena visual.

```
<library_visual_scenes>
       <visual_scene id="DefaultScene">
              <node id="Box" name="Box">
                     <translate> 0 0 0</translate>
                     <rotate> 0 0 1 0</rotate>
                     <rotate> 0 1 0 0</rotate>
                     <rotate> 1 0 0 0</rotate>
                     <scale> 1 1 1</scale>
                     <instance_geometry url="#box">
                            <bind_material>
                                   <technique_common>
                                          <instance_material symbol="WHITE"
                                           target="#whiteMaterial"/>
                                   </technique_common>
```

```
                        </bind_material>
                    </instance_geometry>
            </node>
      </visual_scene>
</library_visual_scenes>
<scene>
      <instance_visual_scene url="#DefaultScene"/>
</scene>
```

Luzes

A luz é uma fonte que ilumina uma cena. O elemento `<light>` declara a fonte de luz com que se pretende iluminar a cena 3D. Sem luz na cena 3D os objectos seriam "renderizados" a preto, e não se compreenderia o que se pretendesse visualizar. Quanto mais fontes de luz foram usadas maior será a qualidade da "renderização" da cena 3D. No formato COLLADA podem existir vários tipos de luz: luz ambiente, ponto de luz, luz direcional e *spotlight* que é uma luz como uma luz de uma lanterna[57,59].

A luz ambiente é uma fonte de luz que irradia de todas as direcções e ao mesmo tempo. A intensidade deste tipo de luz não é atenuada pela distância ou ângulo de visão. A seguir pode ver-se um exemplo de luz ambiente declarada em COLLADA[57,59]:

```
<light id="blue">
      <technique_common>
            <ambient>
                  <color>0.1 0.1 0.5</color>
            </ambient>
      </technique_common>
</light>
```

Ponto de luz é uma fonte de luz que irradia para todas as direcções a partir de uma localização conhecida no espaço. A intensidade é atenuada com o aumento da distância em relação à fonte de luz. Exemplos deste tipo de luz são velas ou lâmpadas. A seguir pode ver-se um exemplo de um ponto de luz declarado em COLLADA[57,59]:

```
<library_lights>
      <light id="lamp">
            <technique_common>
                  <point>
                        <color> 1.0 1.0 0.2 </color>
                        <linear_attenuation>0.3</linear_attenuation>
                  </point>
            </technique_common>
      <light>
</library_lights>

<node>
      <translate> 50.0 30.0 20.0 </translate>
      <instance_light url="#lamp"/>
</node>
```

Podem definir três tipos de atenuação opcionais: constante, linear e quadrática. Se estes valores não forem indicados são usados valores por defeito. A atenuação é dada pela fórmula :

A = A1 + A2 + A3 em que A1 é igual à *constante* de atenuação, A2 é igual à *Distância* X *Atenuação linear* e A3 é igual à *Distância* X *Distância* X *Atenuação Quadrática*.

A luz direccional é uma luz que irradia luz em uma direcção, a fonte de luz está infinitamente distanciada. A luz não é atenuada com a distância. Exemplos deste tipo de luz são o Sol, a Lua e as estrelas. A seguir pode ver-se um exemplo de luz direccional declarada em formato COLLADA[57,59]:

```
<light id="blue">
	<technique_common>
		<directional>
			<color>0.1 0.1 0.5</color>
		</directional>
	</technique_common>
</light>
```

A direcção deve ser indicada no nó onde a luz é instanciada.

Spotlight é uma fonte de luz que irradia em uma direcção a partir de uma localização conhecida no espaço. A luz irradia da fonte com um raio em forma de cone. A intensidade da luz é atenuada com a distância à fonte de luz. Exemplo deste tipo de luz são lanternas, holofotes e faróis. A seguir pode ver-se um exemplo de uma *spotlight* declarada em formato COLLADA[57,59]:

```
<light id="blue">
	<technique_common>
		<spot>
			<color>0.1 0.1 0.5</color>
			<linear_attenuation>0.3</linear_attenuation>
		</spot>
	</technique_common>
</light>
```

Neste tipo de luz também se pode definir a atenuação da mesma maneira que se viu para o ponto de luz, mas podem ainda acrescentar-se mais dois elementos que são `<falloff_angle>` e `<falloff_exponent>` que permitem definir a forma do raio de luz.

Câmara

A câmara personifica o ponto de vista do utilizador que olha para a cena 3D. O elemento `<library_camera>` é uma biblioteca onde se podem colocar vários elementos `<camera>`. O elemento `<camera>` declara uma vista para a cena 3D que o utilizador irá visualizar. Existem propriedades de óptica e de imagem que são configuradas para se obter uma visualização. O elemento `<optics>` descreve o campo de visão e o *viewing frustum*. Campo de visão[68] é medido como um ângulo e é a extensão de mundo que se consegue visualizar no ecrã. *Viewing frustum*[67] é a região do espaço com o mundo ou objectos a modelar que aparecerá no ecrã, como se pode ver Figura 15. No elemento `<optics>` pode definir-se que tipo de projecção se pretende ter para a câmara no perfil comum, elemento `<technique_common>`. Esses tipos são o tipo ortográfico que corresponde ao elemento `<ortographic>` e o tipo de projecção em perspectiva que corresponde ao elemento `<perspective>`. O tipo de projecção ortográfica está dentro do

grupo das projecções paralelas, assim a imagem resultante deste tipo de projecção não mostra profundidade, é uma imagem. Como o que se deseja é mostrar profundidade o tipo de projecção mais utilizado é a projecção em perspectiva. No entanto em COLLADA podem existir outros tipos de projecção mas têm que ser criados pelo utilizador no elemento `<technique>`.

O elemento `<imager>` representa o sensor da câmara que pode ser a película de filme ou o sensor CCD[70] das câmaras digitais e é de utilização opcional. Este elemento não tem propriedades comuns por isso quando é utilizado tem que se usar o elemento filho `<technique>` para definir as propriedades.

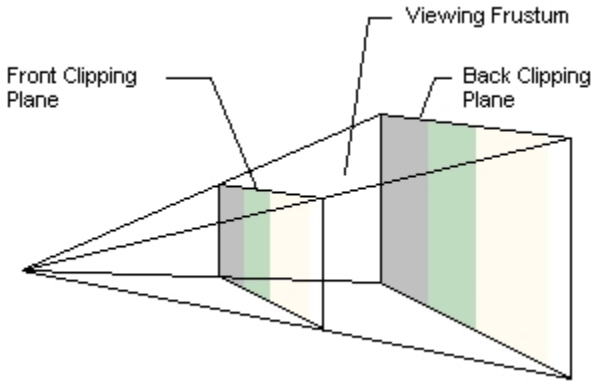

Figura 15: Viewing frustum

A seguir podemos ver um exemplo da declaração da vista de uma câmara em COLLADA:

```
<camera name="eyepoint">
	<optics>
		<technique_common>
			<perspective>
				<yfov>45.0</yfov>
				<aspect_ratio>1.33333</aspect_ratio>
				<znear>0.1</znear>
				<zfar>32767.0</zfar>
			</perspective>
		</technique_common>
	</optics>
</camera>
```

No exemplo pode ver-se que a câmara tem o tipo de projecção em perspectiva, o campo de visão FOV é de 45 graus, o *aspect ratio*[69] é 1.333 e que é a relação proporcional entre a largura e a altura do ecrã, e pode ainda ver-se a definição das distâncias à origem dos planos de corte que definem o *Viewing frustum*: *znear* e *zfar*.

No próximo exemplo pode ver-se a utilização de uma câmara criada na biblioteca de câmaras que é instanciada no elemento `<node>` com identificação `Camera` de uma cena visual, `<visual_scene>`.

```
<node id="Camera">
	<lookat>
		2.0 0.0 4.0 <!-- eye position (X,Y,Z) -->
		0.0 0.0 0.0 <!-- interest position (X,Y,Z) -->
		0.0 1.0 0.0 <!-- up-vector position (X,Y,Z) -->
```

```
</lookat>
<instance_camera url="#camera1"/>
...
```

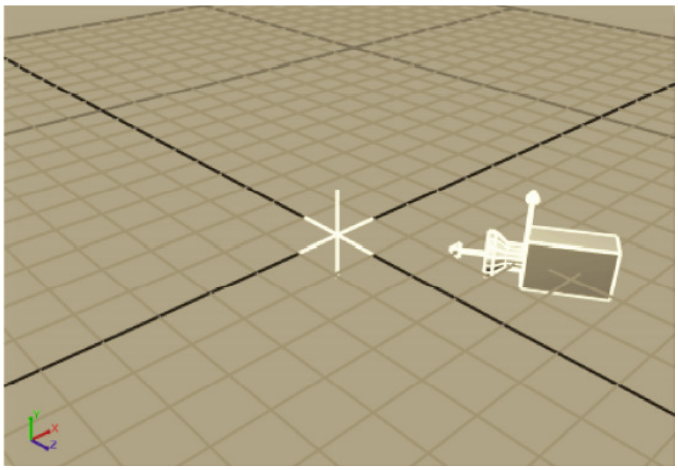

Figura 16: Câmara, posicionamento e ponto de interesse

No elemento `<lookat>` define-se em primeiro lugar as coordenadas em que a câmara está localizada, no caso está em (2.0, 0.0, 4.0), o ponto de espaço para onde está a apontar que é a origem dos eixos, e finalmente a indicação do eixo que fica para cima que no caso é o eixo dos yy.

Animações

A animação usualmente é a criação da ilusão de movimento. A animação descreve a transformação de um objecto ou valor ao longo do tempo. Uma técnica comum para realizar animações é a animação por *key frame*, em que se faz amostragem de dados a duas dimensões. A primeira dimensão é a entrada e usualmente é o tempo. A segunda dimensão é a saída e representa o valor que está a ser animado ao longo do tempo. O objectivo é obter esse valor para um determinado instante de tempo. As chaves são os valores do tempo em segundos. Se o tempo corresponde exactamente a uma das *keys*, o valor retornado é exactamente o valor associado com essa *key*. Se o tempo está entre duas amostras, utiliza-se um conjunto de *key frames* e um algoritmo de interpolação, os valores intermédios são calculados para os tempos entre *key frames*, produzindo um conjunto de valores de saída no intervalo entre *key frames*. O conjunto de *key frames* e a interpolação entre *key frames* define uma função 2D, que é chamada curva de animação e desta forma obtêm-se transições suaves na animação[57].

Em COLLADA o elemento `<animation>` é o elemento usado para declarar uma ou mais animações. Uma animação pode ser composta por outras animações. O elemento `<animation>` é constituído por 3 elementos principais: `<source>`, `<sampler>` e `<channel>`. O elemento `<source>` é utilizado para guardar um ou mais valores em *arrays*, da mesma forma que se viu anteriormente para a Geometria. A seguir pode ver-se um exemplo com *sources* para o movimento de um antebraço de uma personagem animada em que a primeira *source* contém os valores de entrada que é o tempo e é expresso em segundos.

```
<source id="node-ValveBiped.Bip01_R_Forearm_matrix-input">
```

```
      <float_array id="node-ValveBiped.Bip01_R_Forearm_matrix-input-array" count="101">0
0.03333333 0.06666667 0.1 ... </float_array>
      <technique_common>
        <accessor source="#node-ValveBiped.Bip01_R_Forearm_matrix-input-array"
count="101" stride="1">
            <param name="TIME" type="float"/>
        </accessor>
      </technique_common>
</source>
```

A *source* seguinte é a saída e contém matrizes 4x4 com a transformação do movimento do antebraço.

```
<source id="node-ValveBiped.Bip01_R_Forearm_matrix-output">
      <float_array id="node-ValveBiped.Bip01_R_Forearm_matrix-output-array"
count="1616">0.7389092 0.673805 0 12.88987 ... </float_array>
      <technique_common>
        <accessor source="#node-ValveBiped.Bip01_R_Forearm_matrix-output-array"
count="101" stride="16">
            <param name="TRANSFORM" type="float4x4"/>
        </accessor>
      </technique_common>
</source>
```

A próxima *source* contém o tipo de interpolação que é usada e que é a interpolação linear, mas existem outros tipos de interpolação que podem ser usadas.

```
<source id="node-ValveBiped.Bip01_R_Forearm_matrix-interpolation">
      <Name_array id="node-ValveBiped.Bip01_R_Forearm_matrix-interpolation-array"
count="101">LINEAR LINEAR LINEAR LINEAR ... </Name_array>
      <technique_common>
        <accessor source="#node-ValveBiped.Bip01_R_Forearm_matrix-interpolation-array"
count="101" stride="1">
            <param name="INTERPOLATION" type="name"/>
        </accessor>
      </technique_common>
</source>
```

O elemento `<sampler>` cria valores calculados a partir dos valores guardados nos *arrays*, ou seja define uma função de interpolação para a animação. Os pontos de amostragem são descritos pelos elementos `<input>` que correspondem aos elementos `<source>`. O atributo `semantic` pode ser do tipo INPUT, OUTPUT, INTERPOLATION, ou outros. Os valores da *source* de interpolação são enviados para o *sampler* através do elemento `<input>` com `semantic= "INTERPOLATION"`[57]. A seguir podemos o *sampler* para o exemplo do movimento de antebraço.

```
<sampler id="node-ValveBiped.Bip01_L_Forearm_matrix-sampler">
      <input semantic="INPUT" source="#node-ValveBiped.Bip01_L_Forearm_matrix-input"/>
      <input semantic="OUTPUT" source="#node-ValveBiped.Bip01_L_Forearm_matrix-output"/>
      <input semantic="INTERPOLATION" source="#node-ValveBiped.Bip01_L_Forearm_matrix-
interpolation"/>
</sampler>
```

O elemento `<channel>` serve para ligar os valores de saída às estruturas de dados que recebem

esses dados. Existem dois atributos obrigatórios `source` e `target`. O atributo `source` serve para referir o *sampler* utilizado e o atributo `target` vai ter a referência da estrutura de dados que vai ser preenchida com a saída do *sampler*.

A seguir podemos ver os vários canais que existem do exemplo da personagem animada para as diferentes partes que têm movimento incluindo o antebraço.

```
<channel source="#node-ValveBiped.Bip01_Spine1_matrix-sampler" target="node-
ValveBiped.Bip01_Spine1/matrix"/>
<channel source="#node-ValveBiped.Bip01_Spine2_matrix-sampler" target="node-
ValveBiped.Bip01_Spine2/matrix"/>
<channel source="#node-ValveBiped.Bip01_Head1_matrix-sampler" target="node-
ValveBiped.Bip01_Head1/matrix"/>
<channel source="#node-ValveBiped.Bip01_L_UpperArm_matrix-sampler" target="node-
ValveBiped.Bip01_L_UpperArm/matrix"/>
<channel source="#node-ValveBiped.Bip01_L_Forearm_matrix-sampler" target="node-
ValveBiped.Bip01_L_Forearm/matrix"/>
<channel source="#node-ValveBiped.Bip01_R_UpperArm_matrix-sampler" target="node-
ValveBiped.Bip01_R_UpperArm/matrix"/>
<channel source="#node-ValveBiped.Bip01_R_Forearm_matrix-sampler" target="node-
ValveBiped.Bip01_R_Forearm/matrix"/>
```

Skinning

Skinning é uma técnica de deformação da malha poligonal (*mesh*), em que a *mesh* ou *skin* está ligada a um esqueleto, como se pode ver na Figura 17. O movimento do esqueleto faz mudar as posições dos vértices da *mesh* surgindo assim o efeito de animação. Um esqueleto pode ser utilizado em diferentes *meshes*.

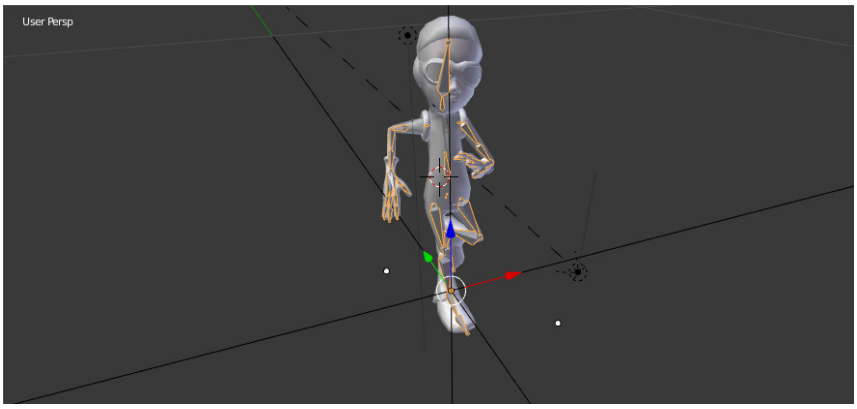

Figura 17: Visualização de uma mesh com esqueleto (software de modelação Blender)

Um esqueleto é um conjunto hierárquico de ossos interligados usados para animação[71]. Para se utilizar esta técnica de animação em COLLADA tem que se usar um controlador, elemento `<controller>`, na biblioteca de controladores, elemento `<library_controllers>`. O controlador é instanciado num elemento `<node>`. Um controlador controla um objecto e neste caso vai ser a geometria de uma malha poligonal utilizando a técnica de *Skinning*. O elemento `<controller>` tem dois elementos filhos: elemento `<skin>` para a técnica de *Skinning* e o elemento `<morph>` para a técnica de animação Morphing[62]. O elemento `<skin>` tem um

atributo `source` em que se tem uma referência para a malha poligonal declarada na geometria. Dentro do elemento `<skin>` existem vários elementos importantes. O primeiro é elemento `<bind_shape_matrix>` e consiste em uma matriz 4x4 com informação sobre a posição da malha poligonal antes da ligação ao esqueleto. Este elemento é opcional e se for omitido será usada a matriz identidade. A seguir surgem três elementos `sources`, a primeira *source* com um *array* com nomes de juntas (joints) do esqueleto, a segunda com um *array* de matrizes inversas de ligação e a terceira `source` com um *array* de pesos. A matriz inversa de ligação é a matriz que é aplicada a um vértice e é expressa no sistema de coordenadas das juntas[59]. Cada osso do esqueleto está ligado a duas juntas e cada junta está ligada a pelo menos um osso, como se pode ver na Figura 18[72]. Um vértice é influenciado por uma ou mais juntas, essa influência é o peso que uma determinada junta tem no vértice que está ligado a essa junta. A soma dos pesos que cada junta exerce sobre um vértice deve ser igual a 1.

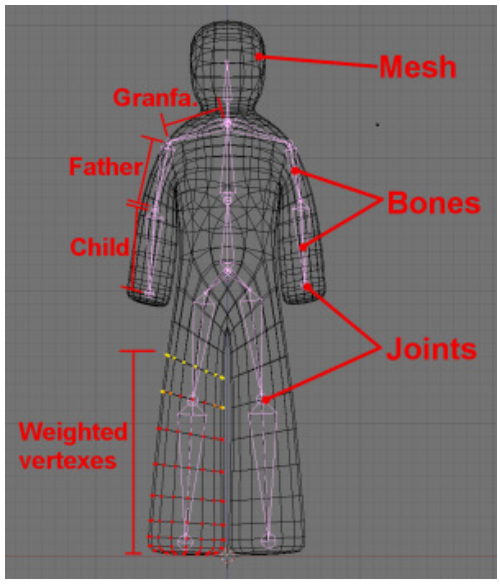

Figura 18: Malha com esqueleto onde se podem ver os ossos e as juntas

O elemento `<joints>` serve para associar o *array* de juntas com o *array* de matrizes inversas de ligação. Este elemento contem dois elementos `<input>`, um com atributo `semantic="JOINT"` que corresponde às juntas e outro com atributo `semantic="INV_BIND_MATRIX"` que corresponde às matrizes inversas de ligação[59].

O elemento `<vertex_weights>` contem uma lista de juntas que têm uma influência e o peso dessa influência. Tem dois elementos `<input>`, um com atributo `semantic="JOINT"` que se refere às juntas e outro atributo `semantic="WEIGHT"` que se refere aos pesos. O elemento `<vcount>` é um *array* de inteiros que indica quantas juntas influenciam cada vértice. A seguir o elemento `<v>` contem uma lista com o índice da cada junta e o índice do peso referentes aos *arrays* dos elementos `<input>`[59].

A seguir pode ver-se um exemplo prático da declaração de um controlador para a técnica de *Skinning*:

```
<library_controllers>
    <controller id="geom-SMDImport-skin1">
      <skin source="#geom-SMDImport">
        <bind_shape_matrix>1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1</bind_shape_matrix>
        <source id="geom-SMDImport-skin1-joints">
          <Name_array id="geom-SMDImport-skin1-joints-array" count="37">joint0 joint1
joint2 joint3 joint4 joint5 joint6 joint7 joint8 ...</Name_array>
          <technique_common>
            <accessor source="#geom-SMDImport-skin1-joints-array" count="37" stride="1">
              <param name="JOINT" type="name"/>
            </accessor>
          </technique_common>
        </source>
        <source id="geom-SMDImport-skin1-bind_poses">
          <float_array id="geom-SMDImport-skin1-bind_poses-array" count="592">1 0 0 0 0
3.13916e-7 1 -42.51616 0 -1 3.13916e-7 -0.5882514 0 0 0 1 0.04088901 ... </float_array>
          <technique_common>
            <accessor source="#geom-SMDImport-skin1-bind_poses-array" count="37"
stride="16">
              <param name="TRANSFORM" type="float4x4"/>
            </accessor>
          </technique_common>
        </source>
        <source id="geom-SMDImport-skin1-weights">
          <float_array id="geom-SMDImport-skin1-weights-array" count="1211">1 0.75 0.25
0.5 0.5 0.5 0.5 0.5 0.5 0.75 0.25 0.5 0.5 0.75 0.25 0.75 0.25 0.75 ... </float_array>
          <technique_common>
            <accessor source="#geom-SMDImport-skin1-weights-array" count="1211"
stride="1">
              <param name="WEIGHT" type="float"/>
            </accessor>
          </technique_common>
        </source>
        <joints>
          <input semantic="JOINT" source="#geom-SMDImport-skin1-joints"/>
          <input semantic="INV_BIND_MATRIX" source="#geom-SMDImport-skin1-bind_poses"/>
        </joints>
        <vertex_weights count="5402">
          <input semantic="JOINT" source="#geom-SMDImport-skin1-joints" offset="0"/>
          <input semantic="WEIGHT" source="#geom-SMDImport-skin1-weights" offset="1"/>
          <vcount>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
</vcount>
          <v>12 0  12 0  12 0  12 0  12 0  12 0  12 0  12 0  12 0  12 0  12 0  12 0
...</v>
        </vertex_weights>
      </skin>
    </controller>
  </library_controllers>
```

O controlador deve depois ser instanciado na cena visual onde é necessário utilizar o elemento `<skeleton>` que deve apontar para o primeiro elemento das juntas, que no caso do exemplo apresentado é `joint0`. Na cena visual, `<visual_scene>`, são declaradas todas as juntas (nomes das juntas, Ids e Sids nos elementos `<node>`), como se pode ver a seguir:

```
<node id="node-ValveBiped.Bip01_Pelvis" name="ValveBiped.Bip01_Pelvis" sid="joint0"
type="JOINT">
        <matrix>1 0 0 0 0 3.13916e-7 -1 -0.588238 0 1 3.13916e-7 42.51616 0 0 0
1</matrix>
```

```
        <node id="node-ValveBiped.Bip01_L_Thigh" name="ValveBiped.Bip01_L_Thigh"
sid="joint1" type="JOINT">
          <matrix>0.04088902 6.07937e-5 0.9991636 4.733477 -0.9978216 0.05181661
0.04083093 0.002311707 -0.05177082 -0.9986566 0.002179392 1.64986e-4 0 0 0 1</matrix>
          <node id="node-ValveBiped.Bip01_L_Calf" name="ValveBiped.Bip01_L_Calf"
sid="joint2" type="JOINT">
            <matrix>0.9956897 0.02341793 0.08974407 18.20104 0.08971927 0.002110331 -
0.9959648 -0.05045688 -0.02351283 0.9997235 1.95578e-7 -9.53674e-7 0 0 0 1</matrix>
            <node id="node-ValveBiped.Bip01_L_Foot" name="ValveBiped.Bip01_L_Foot"
sid="joint3" type="JOINT">
              <matrix>0.427656 0.9037328 -0.01944345 18.89762 0.05116213 -0.002724119
0.9986869 1.90735e-6 0.9024926 -0.4280891 -0.04740186 4.76837e-7 0 0 0 1</matrix>
            </node>
          </node>
        </node>
        <node id="node-ValveBiped.Bip01_R_Thigh" name="ValveBiped.Bip01_R_Thigh"
sid="joint4" type="JOINT">
...
```

Nos elementos <node> declaram-se as juntas e são colocadas as matrizes de transformação de cada junta.

Física

A Física no formato COLLADA visa dar suporte de propriedades físicas aos objectos na cena 3D. Existem *middlewares* também chamados bibliotecas de Física ou conhecidos como motores de Física que permitem associar atributos físicos aos modelos tridimensionais. Alguns desses motores ou bibliotecas de Física são: Bullet Physics Library[9], PhysX[74], Havok[75], ODE[5] e PAL[76]. O formato COLLADA fornece um conjunto de elementos que permite guardar esse conteúdo relacionado com a Física. Uma dessas propriedades, e que é a aplicação das Leis de Newton, é a gravidade, pode associar-se a uma esfera uma determinada massa e definir-se a direcção da gravidade em relação ao sistema de eixos no ficheiro COLLADA e assim visualizar com a utilização do motor de Física a esfera a cair verticalmente simulando a força da gravidade.

A seguir podem ver-se os principais elementos do formato COLLADA para a Física:

```
<library_visual_scenes>
      <visual_scene>
      <node>

<library_physics_scenes>
      <physics_scene>
            <instance_physics_model>
                  <instance_rigid_body>
                  target= “”
                  <instance_rigid_constraint>
                  <instance_force_field>

<library_physics_models>
      <physics_model>
            <rigid_body>
            <rigid_constraint>
            <instance_physics_material>

<library_physics_materials>
      <physics_material>
```

```
<library_force_fields>
	<force_field>

<scene>
	<instance_physics_scene>
	<instance_visual_scene>
```

Como se viu anteriormente a cena 3D é inicialmente criada no elemento `<scene>`, e neste elemento instacia-se a cena visual e a cena física. A cena física é declarada com o elemento `<physics_scene>`, e neste elemento podem instanciar-se: corpos rígidos, restrições entre corpos rígidos e campos de forças. Os corpos rígidos e as restrições entre estes são efectivamente criados no elemento `<physics_model>`, onde são instaciados os materiais físicos que são criados no elemento biblioteca `<library_physics_materials>`.

É importante para o estudo da Física no formato COLLADA distinguir entre malhas côncavas e malhas convexas. Uma malha com uma forma côncava é uma malha oca ou arredondada no seu interior. Uma malha com forma convexa é uma malha arredondada no seu exterior similar ao exterior de uma esfera. Na Figura 19[73] podem ver-se exemplos das formas côncava e convexa.

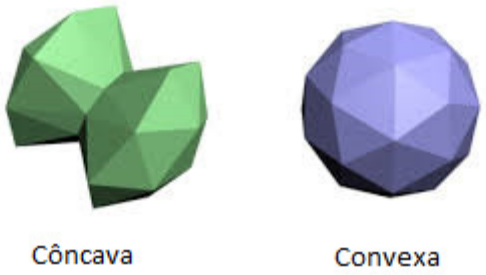

Figura 19: Malhas côncava e convexa

O objectivo de uma simulação é calcular a posição e orientação de todos os corpos rígidos em um determinado instante de tempo, e o formato COLLADA fornece elementos que permitem isto[59]. Em COLLADA existe suporte para se definirem várias formas físicas simples e que são esferas, paralelepípedos, cilindros, cones, cápsulas.

Esfera:

```
<sphere>
	<radius> 1.0 </radius>
</sphere>
```

Caixa ou paralelepípedo:

```
<box>
	<half_extents>
		3.0 1.0 2.0
	</half_extents>
</box>
```

O tamanho da caixa é dado pelo elemento <half_extents> com 3 valores dos tamanhos referentes a cada um dos 3 eixos.

Plano:

```
<plane>
	<equation>
		a  b  c  d
	</equation>
</plane>
```

Um plano infinito é definido pela equação $a \cdot x + b \cdot y + c \cdot z + d = 0$. No elemento <equation> indicam-se os coeficientes da equação do plano.

Cilindro:

```
<cylinder>
	<height> 4.0 </height>
	<radius> 2.0 1.0 </radius>
</cylinder>
```

A altura do cilindro define-se com o elemento <height> e os raios em relação aos eixos dos xx e eixo dos zz define-se em <radius>, pois o eixo que fica para cima é o eixo dos yy.

Se não for possível usar uma das formas simples para fazer a aproximação à malha visual pode usar-se uma malha convexa. O elemento <convex_mesh> permite utilizar a geometria definida para outra malha poligonal, usando o atributo convex_hull_of para fazer a ligação à malha de visualização. Esta malha convexa pode ser utilizada para visualização e para malha física. Pode assim definir-se:

```
<geometry id="minhaMalha">
	<mesh>
		...
	</mesh>
</geometry>

<convex_mesh convex_hull_of="#minhaMalha"/>
```

O elemento <shape> é o elemento que permite definir a forma de um objecto físico e tem como elementos filhos os elementos já vistos: <sphere>, <box>, <plane>, <cylinder> e <capsule>, etc. Tem como elemento pai o elemento <rigid_body>, corpo rígido, que pode conter vários elementos <shape> desde que sejam compostos por diferentes partes que tenham as suas propriedades físicas distintas. Um corpo rígido é um objecto não deformável com forma (geometria) e propriedades de massa que interagem com outros corpos rígidos de acordo com as Leis básicas de Newton para a Física[57], definido pelo elemento <rigid_body>. O elemento <shape> tem ainda informação sobre a massa do objecto físico e que são os elementos filhos <mass>, <density> e <hollow>. O elemento <mass> contém um número de vírgula flutuante que pode indicar a massa em *Kg* da forma física. O elemento <density> contém a densidade da forma física que pode ser usado em vez do elemento <mass>, a Densidade = Massa /

Volume. Se a massa for indicada o que estiver no elemento <density> será ignorado. O elemento <hollow> tem um valor booleano e se for verdadeiro indica que a massa está distribuída na superfície da forma. A seguir pode ver-se um exemplo de uma forma física com forma geométrica de uma esfera e com um determinado valor para a massa.

```
<physics_model>
	<rigid_body sid="meuCorpoRigido">
		<technique_common>
		<shape>
			<mass> 0.25 </mass>
			<sphere>
				<radius> 2.0 </radius>
			</sphere>
		</shape>
	</technique_common>
	</rigid_body>
</physics_model>
```

As malhas poligonais para serem usadas no elemento <shape> têm que ser instanciadas e para isso usa-se o elemento <instance_geometry> que instancia uma malha definida no elemento <mesh> ou <convex_mesh>.

Existem propriedades físicas que podem ser colocadas no elemento <shape>, para isso usa-se o elemento <physics_material> mas também se pode usar uma instância com o elemento <instance_physics_material> desde que os materiais sejam declarados na biblioteca de materiais físicos. O elemento <physics_material> declara as propriedades físicas de um objecto e tem três elementos filhos que são <dynamic_friction>, <restitution> e <static_friction>. O elemento <dynamic_friction> é um elemento que guarda o coeficiente de fricção dinâmica e que surge quando existem objectos que se estão a deslocar e tocam uns nos outros. O elemento <static_friction> guarda o coeficiente de fricção estática que surge quando não existe movimento entre os objectos mas é preciso iniciar esse movimento e a força que o inicia fica sujeita à fricção estática. O elemento <restitution> guarda um coeficiente que é a proporção de energia cinética preservada num impacto. A seguir pode ver-se um exemplo de uma bola saltitante[57]:

```
<rigid_body id="bouncy_ball">
	<shape>
		<sphere> <radius> 1 </radius> </sphere>
		<physics_material id="bouncy_material">
			<technique_common>
				<dynamic_friction> 0.12 </dynamic_friction>
				<restitution> 0.05 </restitution>
				<static_friction> 0.23 </static_friction>
			</technique_common>
		</physics_material>
	</shape>
</rigid_body>
```

O elemento <rigid_body> tem também elementos filhos para propriedades físicas e que são o elemento <mass>, o elemento <dynamic>, o elemento <mass_frame> e o elemento <inertia>. O elemento <mass> é a massa física do corpo rígido que poderá ser expressa em *Kg*

e se existirem várias *shapes* dentro do elemento <rigid_body>, a massa será a soma das massas dessas *shapes*. Não será necessário indicar o valor da massa dentro dos elementos <shape>. O elemento <dynamic> contém um valor booleano que caso seja verdadeiro indica que o corpo rígido pode mover-se e caso seja falso o corpo rígido não se move. O elemento <mass_frame> indica o centro de massa e por defeito é a matriz identidade que nos diz que o centro de massa está na origem local e os eixos principais são os eixos locais. O elemento <inertia> contém um vector com 3 números de vírgula flutuante com o momento de inércia que é uma grandeza física que indica a dificuldade de alterar o estado de movimento de um corpo em rotação[57,59,77].

As restrições entre corpos rígidos permitem ligar vários corpos rígidos e formar objectos articulados[59]. Para isto existe o elemento <rigid_constraint> que permite declarar essa restrição rígida. É necessário ter em consideração os seguintes elementos filhos <ref_attachment>, <attachment> e <technique_common>. O elemento <ref_attachment> serve para fazer uma ligação da restrição rígida a um corpo rígido que será o primeiro corpo rígido. O elemento <attachment> serve para fazer uma ligação da restrição rígida ao segundo corpo rígido. O elemento <technique_common> declara informação do corpo rígido no perfil comum. Na restrição rígida é necessário ter em conta o grau de liberdade ou DOF (*degree of freedom*) que especifica o movimento em relação a um eixo de translação ou em relação a um eixo de rotação[59]. Assim no perfil comum existem os vários elementos a ter em consideração[57,59]:

- o elemento <enabled>, contém um valor booleano verdadeiro por defeito que caso seja falso significa que não existe ligação entre os corpos rígidos;
- o elemento <interpenetrate> contém um valor booleano que por defeito é verdadeiro e indica que os corpos rígidos se podem interpenetrar;
- o elemento <limits> permite especificar os limites da restrição em termos de graus de liberdade ou games. Tem os seguintes elementos filhos:
  - o elemento <linear> que indica os limites em cada eixo em que <min> e <max> contém um vector com 3 números de vírgula flutuante com os limites em cada eixo;
  - o elemento <swing_cone_and_twist> descreve os limites angulares ao longo de cada eixo de rotação em que <min> e <max> contém um vector com 3 números de vírgula flutuante com os limites em cada eixo;
- o elemento <spring> permite ter maior suavidade nas restrições rígidas, é uma mola entre os corpos rígidos e tem como elementos filhos o elemento <linear> e o elemento <angular> os quais contêm os seguintes elementos:
  - o elemento <target_value> é um valor com a mola na posição de repouso;
  - o elemento <stifness> é o coeficiente da mola com unidades de Força / Distância;
  - o elemento <damping> é um facto usado para criar resistência ao movimento.

A seguir pode ver-se um exemplo do elemento <limits>[57]:

```
<limits>
       <swing_cone_and_twist>
              <min sid="...">
                     -15.0 -15.0 -INF
```

```
            </min>
            <max sid="...">
                  15.0 15.0 INF
            </max>
      </swing_cone_and_twist>
      <linear>
            <min sid="...">
                  0 0 0 </min>
            <max sid="...">
                  0 0 0 </max>
      </linear>
</limits>
```

A seguir pode ver-se um exemplo do elemento <spring>[57]:

```
<spring>
      <angular>
            <stiffness>5.4544</stiffness>
            <damping>0.4132</damping>
            <target_value>90</target_value>
      </angular>
</spring>
```

Como já foi referido anteriormente o elemento <physics_scene> é onde se instanciam os objectos físicos, assim é importante referir dois elementos do perfil comum ou seja em <physics_scene>/<technique_common> e que são o elemento <gravity> que é um vector que representa a força da gravidade e que sendo o eixo dos yy para cima por defeito, tem o seguinte valor (0,-9.8,0) N/Kg. O outro elemento é <time_step> e é o intervalo de tempo, em segundos, em que a simulação tem que ser calculada.

Capítulo 3: Contribuição

3.1 Trabalho desenvolvido

Neste trabalho foi decidido implementar o OpenGL versão 3.3, uma vez que se podem usar *shaders* e a linguagem GLSL introduzidos com o OpenGL 2.0, e é uma versão que já está bastante divulgada mas não tanto como a versão 1. O OpenGL 3.3 é uma grande revisão na norma que reflectia o poder de processamento crescente nas placas gráficas nessa altura, ano de 2010. GLSL significa, OpenGL *Shading Language* e é uma linguagem de alto nível para *shaders*[34]. Um *shader* é, em termos de Computação Gráfica, um programa usado para fazer *shading* (sobreamento). É um programa que diz como se deve desenhar alguma coisa de uma forma específica e única[39]. Estes programas definem um estilo de "renderização". Como a GPU tem muitos processadores, os *shaders* executam paralelamente, a "renderização" decorre em paralelo. No OpenAR foi implementada a possibilidade de cada objecto 3D ter associados os seus *vertex shader* e *fragment shader*.

A contribuição que foi dada com este trabalho foi implementar o OpenGL 3.3 no OpenAR, foi permitir carregar *meshes* guardadas em ficheiros, com a preocupação de usar *meshes* armazenadas no formato COLLADA e criar um módulo do OpenAR com um simulador de *robot* e *drone* terrestre. Neste projecto foi iniciado ainda o suporte de modelos articulados usando um esqueleto no interior de uma *mesh.* Uma *mesh* criada num programa de modelação como o *Maya* ou *Blender* pode ser visualizada num módulo do OpenAR através da utilização de uma classe que lê a *mesh* animada ou estática de um ficheiro onde está armazenada, e a seguir faz-se a sua visualização (*render*). O desenvolvimento foi feito no sistema operativo Linux, usando como linguagem de programação a linguagem C++ na biblioteca OpenAR e nos módulos que a utilizam. É muito importante dizer que o programador dos módulos do OpenAR não tem que alterar os módulos que tenha desenvolvido na versão do OpenAR com OpenGL 1, porque os métodos mantêm-se os mesmos que eram usados anteriormente.

Definiram-se no *engine* do OpenAR as matrizes *Projection*, *View* e *Model* do OpenGL. A matriz *Projection* é uma matriz 4x4 que define a projecção 3D para 2D de uma cena.

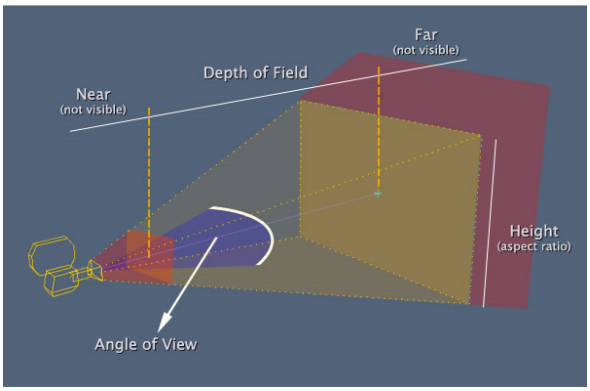

Figura 20: Câmara de projecção

Esta matriz é definida usando o ângulo FOV - *Field of View* que significa campo de visão (na Figura 20 aparece como *Angle of View*) , através da distância à câmara dos planos Near e Far e

ainda do *aspect ratio* que é a relação proporcional entre a largura e a altura do ecrã, na Figura 20 apenas aparece a altura.

A matriz *View* é uma matriz 4x4 que determina a posição da câmara no espaço. É uma matriz global que afecta tudo e que coloca tudo relativamente à posição da câmara. Nesta matriz definem-se as coordenadas com a posição onde está a câmara, as coordenadas para onde a câmara aponta e a definição do sistema de eixos.

A matriz *Model* é também uma matriz 4x4 e é uma matriz local que afecta os modelos. Se se quiserem fazer translações, rotações ou modificar a escala de um objecto usa-se a matriz *Model* correspondente ao objecto.

As matrizes *Projection*, *View* e *Model* são definidas no construtor da classe oaEngine. Essas matrizes são passadas por pârametro para a classe oaObject. Se existirem rotações, translações ou alterações na escala do objecto a desenhar fazem-se essas operações usando a posição base do objecto, posição corrente e factor de escala. A seguir as matrizes são enviadas para a memória da placa gráfica através instruções do OpenGL 3.3. As classes oaSphere, oaCylinder, oaCone, oaTorus e oaBox já existiam no OpenAR com a versão OpenGL 1 e foram reprogramadas para OpenGL 3.3., são desenhados respectivamente, esferas, cilindros, cones, toroides, e paralelepípedos. Estas classes têm em comum os métodos GenBuffer() e build() que são onde se constroem e desenham os objectos 3D.

A seguir mostra-se um exemplo da criação de uma esfera num módulo do OpenAR:

```
std::string vs = DATADIR "/shaders/dirlightdiffambpix.vert";
std::string fs = DATADIR "/shaders/dirlightdiffambpix.frag";

oaSphere * sphere1;
sphere1=new oaSphere(50);
sphere1->setname("Sphere1");
sphere1->loadShaders(vs.c_str(), fs.c_str());
Pose posecSphere1(200,890,-40,0,1,0,360);
sphere1->setpose(posecSphere1);
engine.InsertObject(sphere1);
```

Neste exemplo temos um método que não era usado anteriormente no OpenAR com OpenGL 1, que é o método `loadShaders`, e que serve para dar a possibilidade de cada objecto criado poder ter os seus *shaders*, *vertex shader* e *fragment shader*. Este método devolve um valor inteiro que é o *ProgramID*, cada *mesh* vai ficar com o seu *ProgramID* correspondente. O *ProgramID* vai sendo incrementado para cada objecto criado obtendo-se assim um *ProgramID* único para cada *mesh*. Os *shaders* executam em paralelo na GPU e permitem que cada objecto 3D tenha a sua própria "renderização". É a inovação mais importante neste trabalho, pois abre grandes possibilidades de "renderizar" objectos 3D de muitas formas diferentes.

O *shader* de vértices executa para cada vértice da *mesh*. Recebe como entrada variáveis por vértice chamadas variáveis atributo e variáveis por polígono chamadas variáveis uniformes[78]. Nas variáveis especificam-se as coordenadas do vértice em questão. Assim pode alterar-se a geometria do objecto.

A seguir mostra-se o código de exemplo de um *vertex shader* e um *fragment shader* muito simples e devidamente comentado:

O *vertex shader*:

```
#version 330 core

// Dados de entrada dos vértices, posições e coordenadas de textura,
// diferentes em cada execução do shader
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;

// Dados de saída; vão ser interpolados para cada fragmento, coordenadas de textura
out vec2 UV;

// Valores que se mantêm constantes para toda a mesh, MVP é o produto das matrizes Model,
View e Projection
uniform mat4 MVP;

void main(){
         // Saída com a posição do vértice: MVP * posição
         gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

         // Coordenadas UV do vértice
         UV = vertexUV;
}
```

O *fragment shader*:

```
#version 330 core

// Valores interpolados provenientes do vertex shader
in vec2 UV;

// Dados de saída
out vec3 color;

// Valores que são constantes para toda a mesh, sampler2D é um tipo de amostragem da
textura relacionado com tipo formato de imagem da textura, significa textura 2D
uniform sampler2D myTextureSampler;

void main(){
      // Cor de saída = cor da textura para a coordenada UV especificada
      color = texture2D( myTextureSampler, UV ).rgb;
}
```

O *fragment shader* é executado para cada *pixel* no ecrã onde aparecem os polígonos da *mesh*. O *fragment shader* é responsável pela colocação da cor final naquela porção da *mesh*. Os *shaders* de fragmentos podem ter modelos de luz[78]. Os modelos de luz já existiam no OpenGL 1 mas eram implementados via instruções OpenGL. Um modelo de iluminação lida com a forma como a luz reflecte para fora de um material ou se transmite através desse material e em que direcção o faz. Existem vários modelos de iluminação. No OpenAR com OpenGL 1, o motor era iniciado por defeito com um modelo de iluminação Phong.

A seguir podemos ver um par de *shaders* de vértices e fragmentos com vectores normais às faces da *mesh* e luz direccionada. Exemplo de *vertex shader* com vectores normais:

```
#version 330

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
```

```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 normal;
out vec4 vertexPos;
out vec2 TexCoord;
out vec3 Normal;

void main()
{

   Normal = normalize(vec3(view * model * vec4(normal,0.0)));
   TexCoord = vec2(vertexUV);
   gl_Position = projection * view * model * vec4(vertexPosition_modelspace,1.0);
}
```

Exemplo de *fragment shader* com vectores normais à superfície (*mesh)* e luz :

```
#version 330

uniform sampler2D myTextureSampler;

in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;
out vec4 outputColor;
out vec3 color;

void main()
{

  vec4 color;
  vec4 amb;
  float intensity;
  vec3 lightDir;
  vec3 n;

  lightDir = normalize(vec3(1.0,1.0,1.0));
  n = normalize(Normal);
  intensity = max(dot(lightDir,n),0.0);

  color = texture(myTextureSampler, TexCoord);
  amb = color * 0.33;
  outputColor = (color * intensity) + amb;

}
```

Exemplo de um *vertex shader* para fazer *Skinning*, utilizado nas *meshes* animadas[79]:

```
#version 330

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

// Entrada de dados
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 normal;
layout (location = 3) in ivec4 BoneIDs;
layout (location = 4) in vec4 Weights;
```

```
// Saída de dados
out vec4 vertexPos;
out vec2 TexCoord;
out vec3 Normal;

const int MAX_BONES = 100;

uniform mat4 gBones[MAX_BONES];

void main()
{
    // Matriz de transformação dos ossos do esqueleto
    mat4 BoneTransform = gBones[BoneIDs[0]] * Weights[0];
    BoneTransform     += gBones[BoneIDs[1]] * Weights[1];
    BoneTransform     += gBones[BoneIDs[2]] * Weights[2];
    BoneTransform     += gBones[BoneIDs[3]] * Weights[3];

    Normal = normalize(vec3(view * model * BoneTransform * vec4(normal,0.0)));

    TexCoord = vec2(vertexUV);

    // A posição dos vértices resulta da matriz BoneTransform
    // O movimento do ossos do esqueleto é que faz com que os vértices da mesh
    // mudem para uma nova posição
    gl_Position = projection * view * model * BoneTransform *
    vec4(vertexPosition_modelspace,1.0);
}
```

Quando se programa com GLSL podemos encontrar várias armadilhas. Isto tem a ver com limitações impostas por se estar a programar uma GPU e devido a esta tecnologia ser ainda recente. A primeira questão é que não existe um *debugger* para programas que corram na GPU. Também não é possível escrever mensagens de erro. Isto faz com que seja muito difícil fazer *debug* em *shaders*. Uma forma de dar a volta a esta questão é usar um *if* e estabelecer a cor de saída, p.e., para vermelho. Outra questão onde pode haver problemas é no uso de variáveis, se deixarmos uma variável com um valor indefinido puderam ocorrer erros que podem ser difíceis de detectar[78].

Nos ficheiros de *shaders* devemos indicar obrigatoriamente a versão do *shader*. Para isso existe a directiva `#version`. A directiva `#version` deve iniciar o ficheiro do *shader*. Esta directiva define-se da seguinte forma:

```
#version number profile
```

em que *number* é um inteiro com a versão do GLSL. Temos os seguintes exemplos para o *number*: 330 para versão 3.30 do GLSL, 420 para versão 4.20 do GLSL. O elemento *profile* é opcional e tem como valores possíveis *core* ou *compatibility*. O valor por defeito *core* diz que o *profile* contém funcionalidades da versão indicada em *number* e *compatibility* contém funcionalidades da versão indicada e das versões abaixo dessa. Em GLSL existe a macro __VERSION__ que contém a versão do GLSL. Para usar no mesmo ficheiro de *shader* código para diferentes versões de GLSL deve fazer-se o seguinte:

```
#if __VERSION__ == 330
#version 330
...
#endif
```

```
#if __VERSION__ == 420
#version 420
...
#endif
```

Devem utilizar-se *Ifs* para cada versão do GLSL.

Outra contribuição do autor para o OpenAR foi a possibilidade de se lerem *meshes* em muitos formatos. Criando um objecto da classe `oaMeshObj` podemos ler ficheiros com *meshes* dos formatos: Collada (.dae), Blender 3D (.blend), 3ds Max 3DS (.3ds) ,3ds Max ASE (.ase),Wavefront Object (.obj),Industry Foundation Classes (IFC/Step) (.ifc), XGL (.xgl,.zgl), Stanford Polygon Library (.ply), AutoCAD DXF (.dxf),LightWave (.lwo),LightWave Scene (.lws),Modo (.lxo), Stereolithography (.stl),DirectX X (.x),AC3D (.ac),Milkshape 3D (.ms3d), Quake I (.mdl), Quake II (.md2),Quake III Mesh (.md3),Quake III Map/BSP (.pk3), Return to Castle Wolfenstein (.mdc),Doom 3 (.md5*),etc. É possível utilizar todos estes formatos devido à inclusão da biblioteca Assimp[80].

A seguir pode ver-se um exemplo de como criar um objecto em C++ com uma *mesh* a partir de um ficheiro COLLADA num módulo do OpenAR:

```
std::string vs = DATADIR "/shaders/dirlightdiffambpix.vert";
std::string fs = DATADIR "/shaders/dirlightdiffambpix.frag";

oaMeshObj meshExample;
meshExample.loadShaders(vs.c_str(), fs.c_str());
meshExample. readOBJ("Spider-Man_Sensational/Spider-Man_Sensational.dae");
meshExample.scale(60.35);
meshExample.setMass(10000);
Pose meshPose(100,690,100,0,0,1,20);
meshExample.setpose(meshPose);
engine.InsertObject(&meshExample);
```

O processo de carregar uma *mesh* a partir de um ficheiro e visualizá-la no ecrã começa por ler esse ficheiro usando uma classe da biblioteca Assimp. Obtemos um objecto que é a cena 3D que pode ter uma *mesh* ou várias *meshes*. Dessa *mesh* obtêm-se vários *arrays* com coordenadas de vértices, coordenadas de vectores normais, coordenadas de textura e um *array* com os índices obtidos a partir das faces (triângulos) da *mesh*. Obtidos estes *arrays* eles são enviados para a memória da placa gráfica através de instruções OpenGL 3.3 apropriadas. Finalmente faz-se a vinculação (*bind*) da textura à *mesh* e o *render* dos triângulos que estão em memória com uma instrução apropriada OpenGL 3.3, fazendo-se assim a visualização do objecto 3D. Este processo para visualização da *mesh* é realizado na classe oaMeshObj.

A seguir mostra-se um exemplo de como criar um objecto em C++ com uma *mesh* com animação a partir de um ficheiro COLLADA num módulo do OpenAR:

```
std::string vs = DATADIR "/shaders/skinning.vert";
std::string fs = DATADIR "/shaders/skinning.frag";
oaMeshObjAnim meshAnim1;
meshAnim1.loadShaders(vs.c_str(), fs.c_str());
meshAnim1. readOBJ("marcus/marcus.dae");
Pose meshAnim1Pose(0,890,100,0,0,1,20);
meshAnim1.scale(4.5);
meshAnim1.setpose(meshAnim1Pose);
engine.InsertObject(&meshAnim1);
```

Temos neste exemplo o *shader* de vértices com a transformação das posições dos vértices usando um esqueleto em que se faz o *Skinning* visto no Capítulo 2 que é o processo em que cada junta está associada a um grupo de vértices. O processo para carregar uma *mesh* animada é similar ao que se faz no caso de uma *mesh* estática, a classe oaMeshObjAnim tem os mesmos métodos da classe oaMeshObj e mais alguns para o movimento do esqueleto.

No OpenAR existem alguns tipos de *shaders* conforme se queiram "renderizar" os objectos 3D, existem *shaders* para "renderizar" só com cores, sem textura, existem *shaders* para renderizar com textura e existem *shaders* para usar em *meshes* animadas que usam um esqueleto. Mas como o programador dos módulos é que define que *shaders* utiliza então também pode utilizar *shaders* programados por si, tendo apenas que usar as variáveis de entrada e saída como estão definidas nos *shaders* incluídos no OpenAR. Existem muitos tipos de *shaders* que se podem encontrar em livros e na *Internet*, o programador de *shaders* pode usar *normal map shaders*, *reflection shaders*, *refraction shaders*, *color shaders*, etc. Há muitas possibilidades para "renderizar" os objectos 3D.

Os *shaders* permitem trabalhar a luz usada na cena 3D, existe luz nas cenas 3D do OpenAR no entanto não foi possível explorar muito a sua utilização neste trabalho. Com a sua utilização poderíamos ter, p. e., vários focos de luz ou uma cena 3D escurecida e um foco de luz a focar uma porção de um objecto 3D.

Na figura seguinte podemos ver o diagrama de classes UML para as classes mais representativas do motor gráfico do OpenAR, em que as classes oaBox, oaTorus, oaCone, oaCylinder, oaSphere, oaMeshObj e oaMeshObjAnim herdam as propriedades e métodos da classe oaObject que por sua vez está ligada à classe oaEngine. No *engine* do OpenAR podemos definir *n* objectos em teoria, na prática definiu-se um máximo de 20.

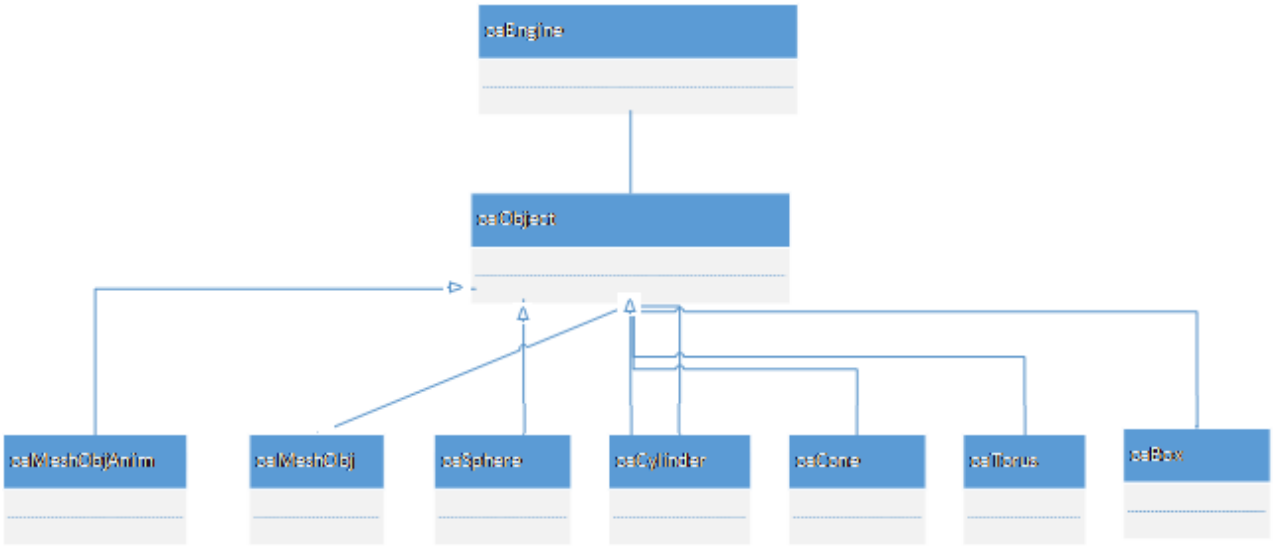

Figura 21: Diagrama de classes UML do motor do OpenAR

Nos anexos podemos encontrar uma descrição mais detalhada do funcionamento dos métodos, que foram alterados para fazer a implementação do OpenGL 3.3.

Se se pretender desenhar um objecto 3D, *mesh* estática ou *mesh* com animação é necessário

declarar no código se a cena 3D contém animações através de uma propriedade do *engine*. No caso do objecto 3D ser uma *mesh* estática é desenhado apenas uma vez mas se o objecto que se pretende desenhar for uma animação ou uma cena 3D onde exista pelo menos uma *mesh* com animação então os objectos têm que ser desenhados continuamente e o método drawScene da classe oaEngine deve estar dentro de um ciclo infinito no módulo.

3.2 Simulador de Robot e Drone Terrestre

As funcionalidades do OpenGL 3.3 e a possibilidade de se lerem *meshes* em muitos formatos de ficheiros e os *shaders* por objecto 3D dão-nos ferramentas poderosas para construirmos aplicações próximas da realidade no OpenAR. Podemos então construir aplicações com as funcionalidades que o OpenAR nos disponibiliza, a essas aplicações chamamos módulos do OpenAR.

Neste trabalho foi iniciado um módulo, que é um protótipo, que tem em vista a realização de um simulador de *robot*. Este módulo é constituído por uma *mesh* lida de um ficheiro com uma sala virtual. Esta sala foi modelada no *software* de modelação *Maya*. Existe um robô (*mesh* também lida de um ficheiro) que se pode ver na Figura 22. Este robô pode ser conduzido livremente pela sala através da utilização das teclas do computador. Pode mover-se para a frente e para trás accionando as teclas das setas do computador para cima e para baixo. Tem a câmara do OpenGL (imagem que o utilizador vê) montada na parte de cima do robot, essa câmara pode ser rodada num plano horizontal accionando as teclas das setas esquerda e direita. O robot desloca-se para a frente e para trás na direcção do plano horizontal em que a câmara estiver apontada. A câmara pode ainda mover-se num plano vertical utilizando as teclas W (para cima) e S (para baixo).

Figura 22: Imagem do Robot virtual

As *meshes* neste módulo são lidas a partir de ficheiros e cada *mesh* tem a sua textura. Nas Figuras 23 e 24 podemos ver através da câmara do nosso *robot* que este está posicionado noutros locais da sala. Temos definido um ângulo de visão da câmara com 60 graus como é comum por exemplo nos jogos de computador onde se pretende simular a visão humana.

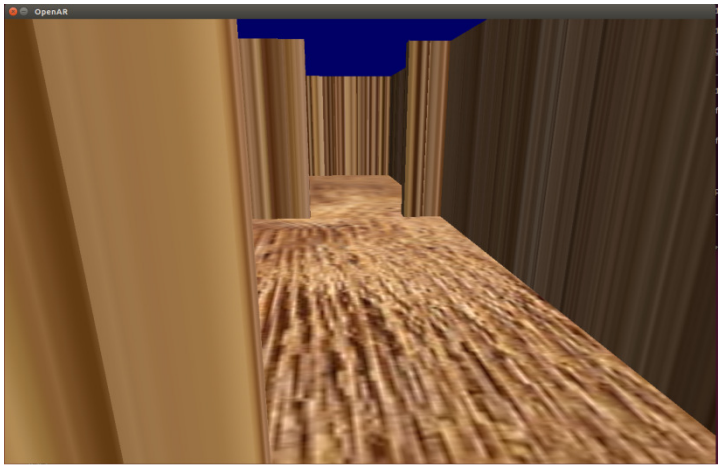

Figura 23: Sala onde se pretende fazer deslocar o *robot*

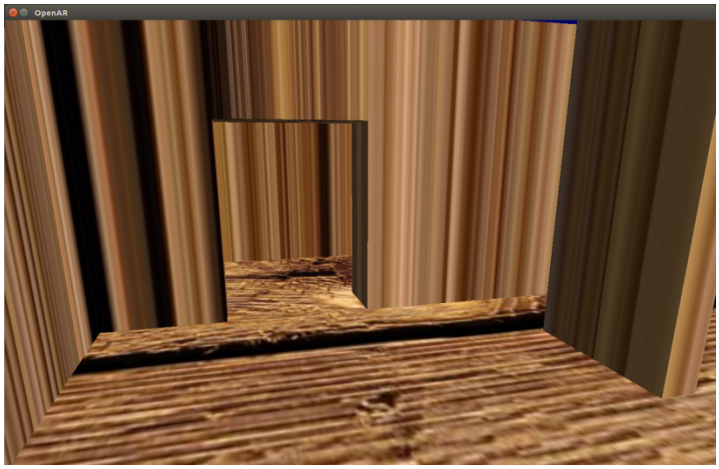

Figura 24: Sala com vista para uma entrada, nova posição do *robot*

Não foi possível realizar atempadamente a simulação de *drone*. Era necessário encontrar um modelo 3D para o *drone*, no entanto é dfícil encontrar um modelo de *drone* adequado nos modelos 3D de *download* livre disponíveis na *Internet*. Teria que se realizar também o controlo do *drone*. Na Figura 25 podemos ver uma vista de cima da sala que pode dar uma ideia do que seria o simulador de *drone*. Se alterássemos a forma de controlar a câmara do simulador de *robot* para que esta se deslocasse na vertical, que no caso é o eixo dos yy, e o modelo de *drone* mudasse também a sua posição na vertical acompanhando a câmara poderíamos ter um simulador de *drone*. Se tivéssemos um dispositivo de hardware ligado ao computador em que pudéssemos deslocar o nosso *drone* para a frente e para trás, para cima e para baixo e ainda que rodasse num plano horizontal seria o ideal para termos a sensação de estar a conduzir uma máquina voadora, no caso um *drone*, em vez de terem que se usar as teclas do computador. Seria também interessante através do motor de Física Bullet ter o efeito de oscilação no ar provocado pelas hélices do *drone*.

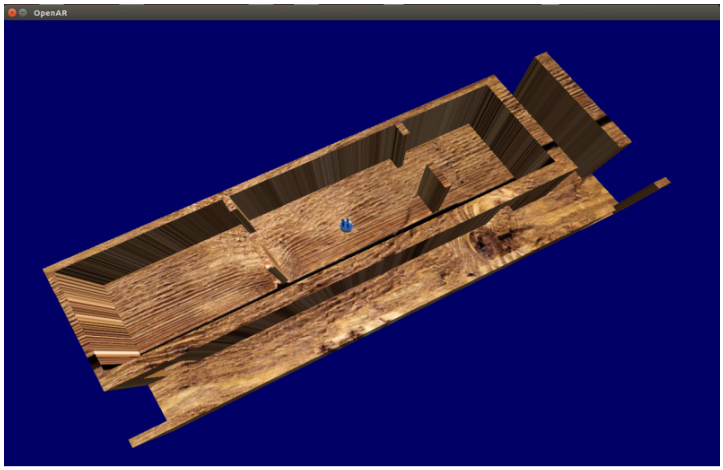

Figura 25: Sala vista de cima com um pequeno ponto azul que é o *robot*

Para o simulador de *robot* ser usado por outras pessoas em aplicações de simulação de *robots* é necessário ter em conta que este software foi realizado em ambiente Linux e programado em C/C++. É necessário compilar o OpenAR no computador onde se está a desenvolver a aplicação e ter instaladas todas as bibliotecas invocadas pelo OpenAR. Para este simulador de *robot* ser aproveitado para outras aplicações ou desenvolvido com mais detalhes é necessário ter em conta o que está realizado. O que está realizado é o carregamento de ficheiros com malhas poligonais em COLLADA ou outros formatos e a sua visualização. Este simulador é composto por duas *meshes* fundamentais: a sala e o *robot*. O que importa explicar mais é como é feito o movimento do *robot* e da câmara que tem que acompanhar o movimento do *robot*. O algoritmo que realiza o movimento pode ser visto no código fonte do simulador no Anexo E. O *robot* e câmara deslocam-se no plano XZ por isso são as coordenadas x e z é que têm que ser afectadas para haver movimento. O *robot* desloca-se no plano XZ na direcção que a câmara estiver a "olhar". O vector que representa a direcção da câmara é (lx, ly, lz) , o vector com a posição da câmara é (x,y,z) e a posição do *robot* é (robotPose.tx, robotPose.ty, robotPose.tz). Quando se pretende mover o *robot* com a câmara usam-se as teclas das setas para cima e para baixo. Quando se usa a seta para cima significa deslocar o *robot* com a câmara para a frente, assim afectam-se os vectores da seguinte forma:

```
x += lx * fraction;
z += lz * fraction;
robotPose.tx += lx * fraction;
robotPose.tz += lz * fraction;
```

Quando se usa a seta para baixo significa deslocar o *robot* com a câmara para trás, assim afectam-se os vectores da seguinte forma:

```
x -= lx * fraction;
z -= lz * fraction;
robotPose.tx -= lx * fraction;
robotPose.tz -= lz * fraction;
```

É necessário ter em atenção a constante **`fraction`.** Existem as teclas das setas para a direita e para a esquerda que permitem rodar a câmara na horizontal e assim é necessário incrementar ou

decrementar o ângulo `angleY`. Para movimentar a câmara na vertical usam-se as teclas W e S que o que fazem é incrementar o ângulo do plano vertical `angleX`. É depois necessário afectar as coordenadas da direcção da câmara em função destes ângulos:

```
lz = cos(angleY) * cos(angleX);
lx = sin(angleY) * cos(angleX);
ly = sin(angleX);
```

e finalmente mover a câmara e o *robot* com os métodos para isso definidos.

Capítulo 4: Resultados e Conclusão

4.1 Resultados

As *meshes* que se mostram neste documento são personagens de banda desenhada pois são os modelos 3D mais fáceis de encontrar na Internet de *download* livre.

Os resultados da implementação da API OpenGL 3.3 e a possibilidade de carregar *meshes* a partir de ficheiros pode ser verificada na criação de módulos do OpenAR. É possível "renderizar" superfícies vulgarmente utilizadas na Matemática como esferas, cilindros, cones, paralelepípedos e toróides como se pode ver na figura seguinte:

Figura 26: Desenho 3D com textura, de um toróide, um cilindro, um cone e uma esfera

Podemos carregar *meshes* estáticas em COLLADA e em outros formatos e também podemos ter texturas de muitos formatos de imagem. Cada *mesh* ou objecto 3D pode ter o seu par de *shaders* de vértices e fragmentos. Na Figura 27 podemos ver 4 objectos 3D em que cada um tem o seu par de *shaders*. A personagem de BD maior tem um *shader* de vértices em que uma matriz com transformação da posição do esqueleto no seu interior permite modificar a posição dos vértices obtendo-se uma animação, em função do tempo. O pequeno cubo que se pode ver tem um *shader* que apenas permite ter cores nas suas faces e não texturas ao contrário do paralelepípedo onde estão poisadas as personagens que têm textura porque têm um *shader* de vértices que está programado para que tenha textura. A *mesh* que é a personagem homem aranha é uma *mesh* estática com um *shader* de vértices com textura.

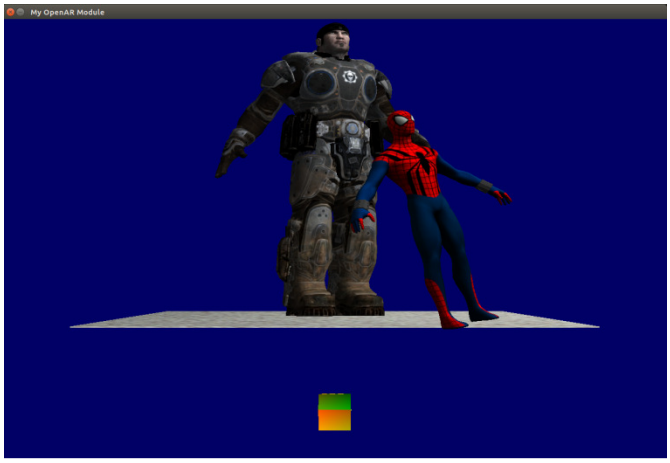

Figura 27: Cena 3D em que cada objecto tem o seu par de shaders de vértices e fragmentos

É possível visualizar animações com esqueleto carregando ficheiros COLLADA e/ou outros formatos. Uma animação é uma deformação de uma *mesh*. Existem outras formas de deformação de *meshes*. Nas figuras seguintes podemos ver uma sequência de imagens que mostram uma animação. Vê-se uma personagem (*mesh*) em que a posição dos vértices se modifica em função de um esqueleto no interior da *mesh* constituindo isto a deformação da *mesh* cujo resultado é podermos ver a personagem animada. A Figura 28 mostra uma sequência de imagens para que se possa ter uma ideia de movimento que permite ter a animação.

Foi iniciada a implementação da Física do formato COLLADA, mas não foi possível apresentar resultados nesta dissertação. É possível nos módulos do OpenAR criar uma *mesh* física a partir da *mesh* criada para visualização. É também possível utilizar métodos do motor de Física Bullet, como por exemplo, para aplicar a força da gravidade às *meshes* visualizadas. No entanto não foi possível utilizar os dados da Física que estejam num ficheiro de formato COLLADA. A biblioteca Assimp usada para a visualização ignora os dados da Física, por isso para utilizar esta informação teria que se fazer essa implementação em classes à parte.

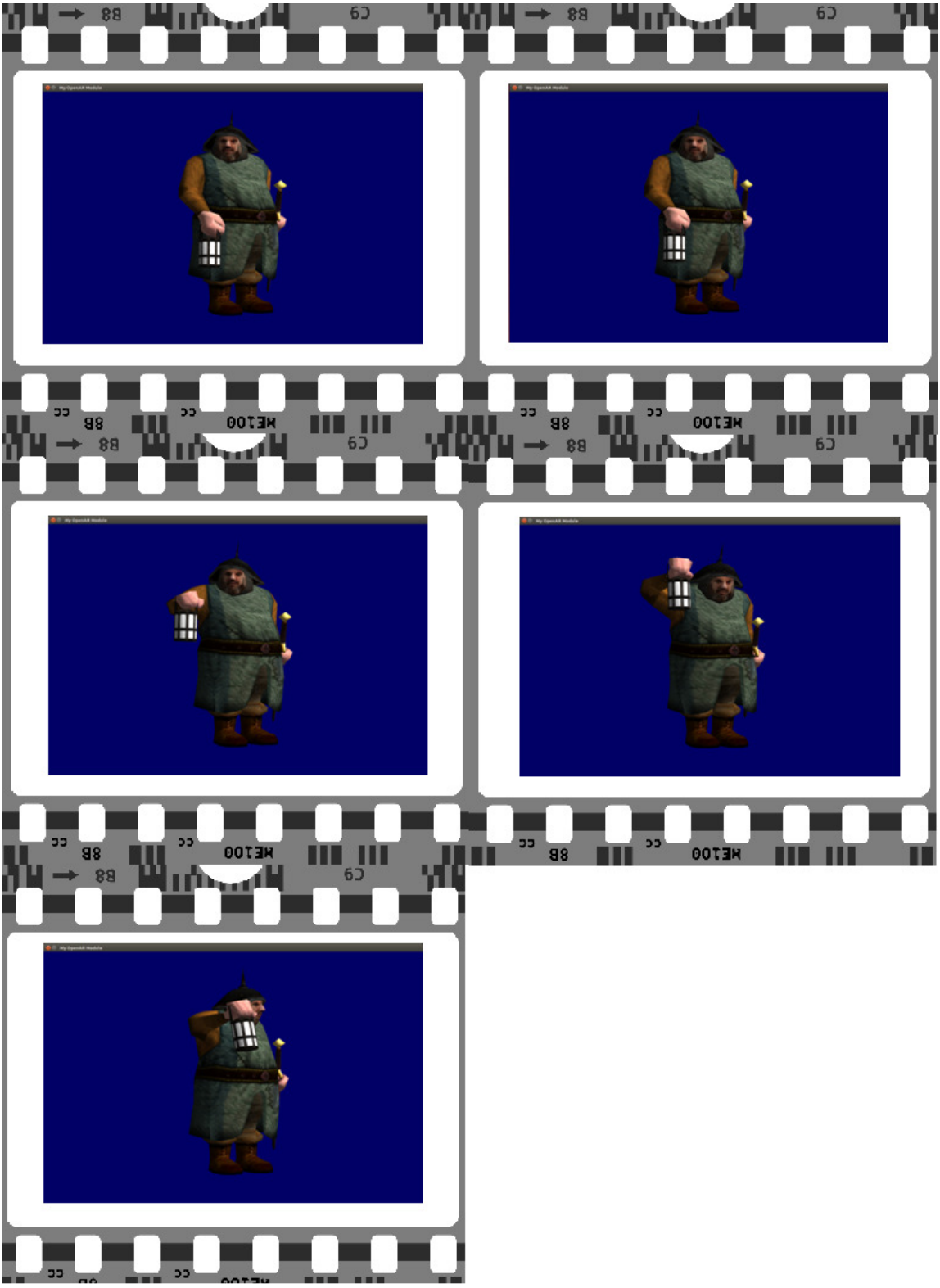

Figura 28: Sequência de animação

4.2 Conclusão

O objectivo deste trabalho proposto inicialmente foi implementar o OpenAR com a versão 3.3 do OpenGL. Foi também estabelecido o objectivo de construir um módulo do OpenAR com um protótipo de simulador de *robot* e *drone*. Outro objectivo foi importar *meshes* do formato COLLADA, outro objectivo ainda, foi fazer deformação de *meshes*. O objectivo da implementação de OpenGL 3.3 foi conseguido, podem visualizar-se *meshes* com a utilização das primitivas gráficas do OpenGL 3.3. Podemos importar *meshes* do formato COLLADA e de muitos outros formatos. O objectivo da construção de um simulador de *robot* e *drone* também foi conseguido pois foi efectuado o protótipo explicado na secção 3.2. O objectivo da deformação da *mesh* foi conseguido pela via da deformação da *mesh* com a utilização de um esqueleto no interior da *mesh*.

Quando este trabalho foi iniciado o OpenAR tinha a versão 1 do OpenGL implementada e apenas lia *meshes* de alguns ficheiros Wavefront .OBJ. No final deste trabalho foi implementada a versão 3.3 do OpenGL 3.3 com utilização de *shaders* por objecto 3D e visualização de *meshes* a partir de múltiplos formatos de ficheiros incluindo *meshes* com animações.

O OpenAR pode ser utilizado em muitos campos que não só a simulação de *robots* ou *drones*, o OpenAR pode ser usado para jogos de computador (mais concretamente como motor gráfico para jogos de computador), cinema de animação, simuladores de acidentes de automóvel, simuladores de avião, imagens usadas em Medicina, etc.

Para poder ter no OpenAR simulações mais realistas poderia melhorar-se a biblioteca gerando terreno virtual, trabalhando melhor a utilização da luz, obtendo texturas com mais qualidade com recurso a outras técnicas como *normal mapping*, *shadow mapping*, *parallax mapping*, etc. Poderia incluir-se uma colecção de *shaders* para o programador de módulos do OpenAR utilizar, poderia fazer-se a implementação de *Tesselation shaders*, que é uma funcionalidade do OpenGL 4, para obter nova geometria que proporciona maior detalhe. Poderia também implementar-se a extracção dos dados da Física do formato COLLADA.

Anexos

A. Formas de representar objectos 3D baseados em aquisição de dados

Existem outras formas de representar objectos 3D. Estas formas são baseadas em aquisição de dados utilizando os seguintes tipos de sensores: ópticos, acústicos, "*scanning*" por laser, térmicos e sísmicos.

A seguir estão resumidas duas formas de representar objectos 3D usando aquisição de dados que são as nuvens de pontos e *voxels* e que para poderem ser trabalhadas em Computação Gráfica são transformadas em *meshes* poligonais.

Nuvem de pontos (*Point Cloud)*

Uma nuvem de pontos é um conjunto de pontos num sistema de coordenadas tridimensional, normalmente definidos por coordenadas X, Y e Z, e o objectivo é representar a superfície externa de um objecto.

As nuvens de pontos podem ser criadas recorrendo a *scanners* 3D. Estes dispositivos medem um grande número de pontos na superfície de um objecto, e produzem uma nuvem de pontos em ficheiro de dados.

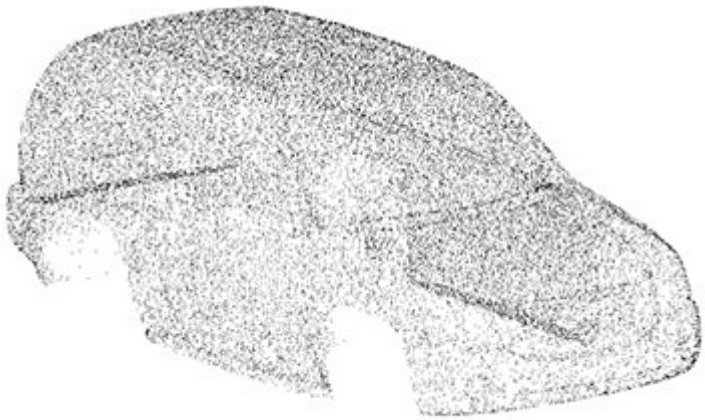

Figura : Exemplo de uma Nuvem de pontos

As nuvens de pontos não podem ser usadas directamente pelas aplicações 3D, e por isso têm que ser convertidas para modelos de *meshes* de polígonos ou *meshes* de triângulos ou superfícies NURBS ou ainda modelos CAD.

Existem várias técnicas para converter uma nuvem de pontos numa superfície 3D. Algumas técnicas como a triangulação *Delaunay*, formas *Alfa* e *ball pivoting*, constroem uma rede de triângulos “em cima” dos vértices existentes na nuvem. Outras técnicas convertem a nuvem de pontos para um

campo de distância volumétrica e depois um algoritmo “Marching cubes” converte para a superfície.

Uma aplicação em que as nuvens de pontos são directamente usadas é na indústria da metrologia, a tomografia industrial computadorizada. A nuvem de pontos de uma peça manufacturada pode ser comparada com um modelo CAD para serem verificadas as diferenças. Estas diferenças podem ser mostradas com mapas de cores que dão um indicador visual do desvio entre a peça manufacturada e o modelo CAD. As nuvens de pontos podem ser usadas para representar dados volumétricos usados, por exemplo, na imagiologia médica. Nos sistemas de informação geográfica, as nuvens de pontos são uma das fontes para fazer um modelo de elevação de terreno. As nuvens de pontos também são aplicadas para gerar modelos 3D de ambiente urbano.

Unlimited Detail da empresa Euclideon

É um motor gráfico 3D que faz uso de um algoritmo de pesquisa numa nuvem de pontos para renderizar imagens. Este motor chama-se *Unlimited Detail* e a empresa que o representa diz que fornece “poder gráfico ilimitado” suplantando a necessidade de usar "renderização" baseada em *meshes* de polígonos. Pode ser visto em http://www.euclideon.com/products/geoverse/.

Unlimited Detail é descrito pela empresa *Euclideon* como uma forma de motor de indexação para pesquisa em nuvem de pontos, que usa um grande número de pontos individuais para criar modelos, ao contrário da tradicional *mesh* poligonal. De acordo com a descrição da empresa, o motor usa um algoritmo de pesquisa para determinar quais desses pontos são visíveis no ecrã, e depois só exibe esses pontos. Por exemplo, num ecrã 1024x768, o motor apenas exibiria 786432 pontos visíveis em cada *frame*. À medida que o motor está a mostrar o mesmo número de pontos em cada *frame*, o nível de detalhe geométrico fornecido é limitado apenas pela quantidade de memória necessária para guardar os dados da nuvem de pontos, e a velocidade de renderização está limitada apenas pela resolução do ecrã.

PCL (Point Cloud Library)

É uma *framework open-source* que inclui numerosos algoritmos para nuvens de pontos de dimensão n e processamento de geometria 3D. Esta biblioteca foi escrita em C++ e os seus algoritmos podem ser usados para Percepção em Robótica para segmentar partes relevantes de uma cena, extrair pontos-chave e computar descritores para reconhecer objectos. Mais informação pode ser encontrada em http://www.pointclouds.org.

Voxel

Os *voxels* são adquiridos usando sensor de dados como *scanners* de ultras sons ou MRI (Magnetic resonance imaging). Um *Voxel* representa um valor numa grelha regular num espaço tridimensional. Um *Voxel* é uma combinação de volume e *pixel.* A posição de um *Voxel* é inferida a partir da sua posição relativamente a outros *Voxels* (é a sua posição na estrutura de dados que faz uma única imagem volumétrica). Os *Voxels* são bons a representar espaços regulares que não são preenchidos homogeneamente.

Os *Voxels* são frequentemente usados na visualização e análise de dados médicos e científicos. Alguns dispositivos de exibição volumétrica usam *voxels* para descrever a sua resolução

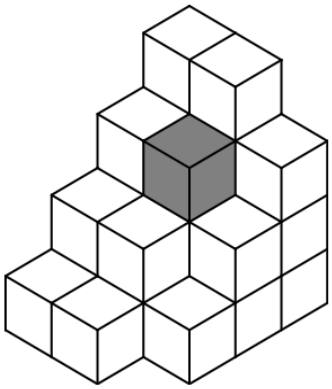

Figura :Voxel

Um *voxel* representa apenas uma só amostra, ou ponto de dados, numa grelha tridimensional espaçada regularmente. Este ponto de dados pode consistir em uma pequena porção de dados tal como opacidade, ou múltiplas porções de dados, tais como cor em conjunto com opacidade. Um *voxel* representa um só ponto nesta grelha *(grid)*, e não um volume, o espaço entre cada *voxel* não é representado num conjunto de dados baseados em *voxels*. Dependendo do tipo de dados e em que se pretender usar o conjunto de dados, a informação que estiver em falta pode ser reconstruída ou aproximada, por exemplo, por interpolação.

Os *voxels* podem conter vários valores escalares, por exemplo, no caso de ecografias com *B-mode* e dados *Doppler,* densidade e taxa de densidade volumétrica são capturadas em canais de dados separados relacionados com as mesmas posições de *voxels*.

Os *voxels* fornecem-nos precisão e visualização da profundidade da realidade, mas são compostos por grandes conjuntos de dados que são pesados de gerir dada a largura de banda dos computadores mais comuns. Mas através de compressão eficiente e manipulação de grandes ficheiros de dados a visualização interactiva é conseguida nos computadores pessoais.

Outros valores podem ser úteis para renderização 3D, tais como normal de um vector e cor.
O uso mais comum dos *voxels* inclui imagem volumétrica e representação de terreno em jogos e simulações.

B. Matrizes

É importante ter algum conhecimento sobre as matrizes usadas em OpenGL. Na visualização dos gráficos 3D usam-se matrizes 4x4, porque estas matrizes permitem transformarmos os nossos

vértices (x, y, z, w) e isso faz-se multiplicando a matriz pelo vector dos vértices, como se pode ver na Figura seguinte:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax+by+cz+dw \\ ex+fy+gz+hw \\ ix+jy+kz+lw \\ mx+ny+oz+pw \end{bmatrix}$$

Figura : Multiplicação de uma matriz por um vector

Se pretendermos fazer translações usamos a matriz:

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura : Matriz translação

Se quisermos fazer a translação do vector (10, 10, 10, 1) em 10 unidades no eixo dos xx, obtemos o vector (20, 10, 10, 1) :

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*10+0*10+10*1 \\ 0*10+1*10+0*10+0*1 \\ 0*10+0*10+1*10+0*1 \\ 0*10+0*10+0*10+1*1 \end{bmatrix} = \begin{bmatrix} 10+0+0+10 \\ 0+10+0+0 \\ 0+0+10+0 \\ 0+0+0+1 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

Figura : Translação do vector (10, 10, 10, 1)

É importante lembrar que 1 significa posição e não direcção, se multiplicássemos o vector (0, 0, -1, 0) por esta matriz de translação obteríamos o próprio vector, ou seja, o vector (0, 0, -1, 0) representa ir em direcção ao eixo -z.

A matriz identidade é importante como se vai ver a seguir porque se a multiplicarmos por um dado vector A, o resultado é o vector A:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1*x+0*y+0*z+0*w \\ 0*x+1*y+0*z+0*w \\ 0*x+0*y+1*z+0*w \\ 0*x+0*y+0*z+1*w \end{bmatrix} = \begin{bmatrix} x+0+0+0 \\ 0+y+0+0 \\ 0+0+z+0 \\ 0+0+0+w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Figura : Matriz identidade

Temos a matriz de escala e queremos escalar um vector 2 unidades em todas as direcções, obtemos o resultado que se pode ver na Figura 15.

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura : Matriz de escala

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 2*x+0*y+0*z+0*w \\ 0*x+2*y+0*z+0*w \\ 0*x+0*y+2*z+0*w \\ 0*x+0*y+0*z+1*w \end{bmatrix} = \begin{bmatrix} 2*x+0+0+0 \\ 0+2*y+0+0 \\ 0+0+2*z+0 \\ 0+0+0+1*w \end{bmatrix} = \begin{bmatrix} 2*x \\ 2*y \\ 2*z \\ w \end{bmatrix}$$

Figura : Multiplicação da matriz de escala 2 unidades nos eixos x, y e z por um dado vector

É com esta matriz que aumentamos ou diminuímos a escala de um objecto 3D.

Temos as seguintes matrizes das Figuras seguintes consoante pretendermos fazer rotações em relação ao eixo dos xx, dos yy ou dos zz em que o nosso ângulo de rotação é θ.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Figura : Matriz rotação em relação ao eixo dos xx

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Figura : Matriz rotação em relação ao eixo dos yy

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Figura : Matriz rotação em relação ao eixo dos zz

C. Métodos da classe oasimpleobj.cpp

Alguns métodos com as construções geométricas das superfícies esféricas, cilíndricas e cónicas no ficheiro oasimpleobj.cpp:

```
int oaSphere::SphereGenBuffer( float radius, int slices, int stacks )
{
    using namespace glm;
    using namespace std;

    const float pi = 3.14159265358979323846264338327795f;
    const float _2pi = 2.0f * pi;

    vector<vec3> positions;
    vector<vec3> normals;
    vector<vec2> textureCoords;

    for( int i = 0; i <= stacks; ++i )
    {
        // V texture coordinate.
        float V = i / (float)stacks;
        float phi = V * pi;

        for ( int j = 0; j <= slices; ++j )
        {
            // U texture coordinate.
            float U = j / (float)slices;
            float theta = U * _2pi;

            float X = cos(theta) * sin(phi);
            float Y = cos(phi);
            float Z = sin(theta) * sin(phi);

            positions.push_back( vec3( X, Y, Z) * radius );
            normals.push_back( vec3(X, Y, Z) );
            textureCoords.push_back( vec2(U, V) );
        }
    }
```

```
    // Now generate the index buffer
    vector<GLuint> indices;

    for( int i = 0; i < slices * stacks + slices; ++i )
    {
        indices.push_back( i );
        indices.push_back( i + slices + 1  );
        indices.push_back( i + slices );

        indices.push_back( i + slices + 1  );
        indices.push_back( i );
        indices.push_back( i + 1 );
    }

    //GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    GLuint vbos[4];
    glGenBuffers( 4, vbos );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[0] );
    glBufferData( GL_ARRAY_BUFFER, positions.size() * sizeof(vec3),
positions.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 0 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[1] );
    glBufferData( GL_ARRAY_BUFFER, normals.size() * sizeof(vec3),
normals.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 1, 3, GL_FLOAT, GL_TRUE, 0, (void*)0 );
    glEnableVertexAttribArray( 1 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[2] );
    glBufferData( GL_ARRAY_BUFFER, textureCoords.size() * sizeof(vec2),
textureCoords.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 2, 2, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 2 );

    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vbos[3] );
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(GLuint), indices.data(), GL_STATIC_DRAW );

    int numberIndices = ( slices * stacks + slices ) * 6;

    glBindVertexArray( 0 );
    glBindBuffer( GL_ARRAY_BUFFER, 0 );
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 );
```

```
    return numberIndices;

}
int oaCylinder::CylinderGenBuffer(float height, float radius, int slices,
int stacks, bool op) {

    using namespace glm;
    using namespace std;

    const float PI = 3.14159265358979323846264338327795f;

    std::vector<glm::vec3> vertices;
    std::vector<glm::vec3> normals;
    std::vector<glm::vec2> textureCoords;

    int i,j;

    // cylinder sides
    for (j = 0; j < stacks; j++) {

        double z1 = (height/stacks) * j;
        double z2 = (height/stacks) * (j+1);

        for (i = 0; i <= slices; i++) {

            double longitude = (2*PI/slices) * i;
            double sinLong = sin(longitude);
            double cosLong = cos(longitude);
            double x = cosLong;
            double y = sinLong;

            normals.push_back( vec3(x, y, 0) );
            textureCoords.push_back( vec2( 1.0/slices * i, 1.0/stacks *
(j+1)) );
            vertices.push_back( vec3(radius*x, radius*y, z2) );

            textureCoords.push_back( vec2( 1.0/slices * i, 1.0/stacks *
j) );
            vertices.push_back( vec3(radius*x, radius*y, z1) );

        }

    }

    // cylinder top and bottom
    if (!op) {
        int rings;
        rings = 1;

        normals.push_back( vec3(0,0,1) );
```

```
        for (j = 0; j < rings; j++) {
            double d1 = (1.0/rings) * j;
            double d2 = (1.0/rings) * (j+1);

            for (i = 0; i <= slices; i++) {
                double angle = (2*PI/slices) * i;
                double s = sin(angle);
                double c = cos(angle);

                textureCoords.push_back( vec2( 0.5*(1+c*d1),
0.5*(1+s*d1)) );
                vertices.push_back( vec3(radius*c*d1, radius*s*d1,
height) );

                textureCoords.push_back( vec2( 0.5*(1+c*d2),
0.5*(1+s*d2)) );
                vertices.push_back( vec3(radius*c*d2, radius*s*d2,
height) );
            }
        }

        normals.push_back( vec3(0,0,-1) );
        for (j = 0; j < rings; j++) {
            double d1 = (1.0/rings) * j;
            double d2 = (1.0/rings) * (j+1);

            for (i = 0; i <= slices; i++) {
                double angle = (2*PI/slices) * i;
                double s = sin(angle);
                double c = cos(angle);

                textureCoords.push_back( vec2( 0.5*(1+c*d2),
0.5*(1+s*d2)) );
                vertices.push_back( vec3(radius*c*d2, radius*s*d2, 0) );

                textureCoords.push_back( vec2( 0.5*(1+c*d1),
0.5*(1+s*d1)) );
                vertices.push_back( vec3(radius*c*d1, radius*s*d1, 0) );
            }
        }
    }

    std::vector<unsigned short> indices;
    std::vector<glm::vec3> indexed_vertices;
    std::vector<glm::vec2> indexed_textureCoords;
    std::vector<glm::vec3> indexed_normals;

    indexVBO(vertices, textureCoords, normals, indices, indexed_vertices,
indexed_textureCoords, indexed_normals);
```

```
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    GLuint vbos[4];
    glGenBuffers( 4, vbos );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[0] );
    glBufferData( GL_ARRAY_BUFFER, indexed_vertices.size() *
sizeof(glm::vec3), indexed_vertices.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 0 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[1] );
    glBufferData( GL_ARRAY_BUFFER, indexed_textureCoords.size() *
sizeof(glm::vec2), indexed_textureCoords.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 1 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[2] );
    glBufferData( GL_ARRAY_BUFFER, indexed_normals.size() *
sizeof(glm::vec3), indexed_normals.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 2, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 2 );

    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vbos[3] );
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(unsigned short), indices.data(), GL_STATIC_DRAW );

    // unbind buffers
    glBindVertexArray(0);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,0);

    return indices.size();
}


int oaCone::ConeGenBuffer( float height, float radius, int slices, int
stacks, bool op ) {

    using namespace glm;
    using namespace std;

    const float PI = 3.14159265358979323846264338327795f;

    std::vector<glm::vec3> vertices;
    std::vector<glm::vec3> normals;
    std::vector<glm::vec2> textureCoords;

    int i,j;
```

```
    for (j = 0; j < stacks; j++) {
        double z1 = (height/stacks) * j;
        double z2 = (height/stacks) * (j+1);

        for (i = 0; i <= slices; i++) {
            double longitude = (2*PI/slices) * i;
            double sinLong = sin(longitude);
            double cosLong = cos(longitude);
            double x = cosLong;
            double y = sinLong;
            double nz = radius/height;
            double normLength = sqrt(x*x+y*y+nz*nz);

            normals.push_back( vec3(x/normLength, y/normLength,
nz/normLength) );

            textureCoords.push_back( vec2( 1.0/slices * i, 1.0/stacks *
(j+1)) );
            vertices.push_back( vec3((height-z2)/height*radius*x,
(height-z2)/height*radius*y, z2) );

            textureCoords.push_back( vec2( 1.0/slices * i, 1.0/stacks *
j) );
            vertices.push_back( vec3((height-z1)/height*radius*x,
(height-z1)/height*radius*y, z1) );
        }

    }

    if (!op) {
        int rings = 1;
        if (rings > 0) {

            normals.push_back( vec3(0,0,-1) );
            for (j = 0; j < rings; j++) {
                double d1 = (1.0/rings) * j;
                double d2 = (1.0/rings) * (j+1);

                for (i = 0; i <= slices; i++) {
                    double angle = (2*PI/slices) * i;
                    double s = sin(angle);
                    double c = cos(angle);

                    textureCoords.push_back( vec2( 0.5*(1+c*d2),
0.5*(1+s*d2)) );
                    vertices.push_back( vec3(radius*c*d2, radius*s*d2, 0)
);

                    textureCoords.push_back( vec2( 0.5*(1+c*d1),
0.5*(1+s*d1)) );
```

```
                    vertices.push_back( vec3(radius*c*d1, radius*s*d1, 0)
);
                }
            }
        }
    }

    std::vector<unsigned short> indices;
    std::vector<glm::vec3> indexed_vertices;
    std::vector<glm::vec2> indexed_textureCoords;
    std::vector<glm::vec3> indexed_normals;

    indexVBO(vertices, textureCoords, normals, indices, indexed_vertices,
indexed_textureCoords, indexed_normals);

    glGenVertexArrays( 1, &vao );

    glBindVertexArray( vao );

    GLuint vbos[4];
    glGenBuffers( 4, vbos );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[0] );
    glBufferData( GL_ARRAY_BUFFER, indexed_vertices.size() *
sizeof(glm::vec3), indexed_vertices.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 0 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[1] );
    glBufferData( GL_ARRAY_BUFFER, indexed_textureCoords.size() *
sizeof(glm::vec2), indexed_textureCoords.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 1 );

    glBindBuffer( GL_ARRAY_BUFFER, vbos[2] );
    glBufferData( GL_ARRAY_BUFFER, indexed_normals.size() *
sizeof(glm::vec3), indexed_normals.data(), GL_STATIC_DRAW );
    glVertexAttribPointer( 2, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );
    glEnableVertexAttribArray( 2 );

    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vbos[3] );
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(unsigned short), indices.data(), GL_STATIC_DRAW );

    // unbind buffers
    glBindVertexArray(0);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,0);

    return indices.size();
```

```
}
```

D. Métodos do OpenAR com a implementação do OpenGL 3

classe oaEngine

→ alteração do construtor da classe
No construtor desta classe são definidas as matrizes Projection, View e Model. Nessa definição é usada a biblioteca GLM, OpenGL Mathematics. A matriz Projection é uma matriz 4x4 que define a projecção 3D para 2D de uma cena, normalmente é pré-calculada a partir de outra função, no nosso caso, usa-se GLM para isso. A matriz View é uma matriz 4x4 que determina a posição da câmara no espaço. É uma matriz global que afecta tudo e que coloca tudo relativamente à posição da câmara. A matriz Model é também uma matriz 4x4 e é uma matriz local que afecta os modelos. Se se quiserem fazer translações, rotações ou modificar a escala de um objecto tem que se usar a matriz Model correspondente ao objecto.

→ setCamPose
Este método que já existia em OpenGL 1 foi modificado para OpenGL 3 e em que é afectada a matriz View, para qual são passadas as coordenadas da câmara.

→ SetViewingAngle
Este método também já existia e foi modificado para suportar OpenGL 3 onde a matriz Projection é modificada e onde o parâmetro viewingangle é o ângulo do campo de visão horizontal, ou seja, é o zoom da câmara.

→ drawScene
Este método limpa ecrã com a instrução glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT), limpa o *color buffer* que faz com que efectivamente o ecrã fique limpo, limpa o *depth buffer* que faz com que um objecto que está atrás de outro seja ocluído. A seguir chama o método DrawObjects e a depois faz SwapBuffers que significa uma troca entre dois *buffers front* e *back*, que são dois *color buffers,* isto permite evitar efeitos indesejáveis no ecrã como por exemplo *flickering*.

→ DrawObjects
No construtor da classe oaEngine é inicializado o *array* objlist, este *array* guarda quantos objectos são inseridos na nossa cena 3D. Cada objecto é inserido no oaEngine invocando o seu método InsertObject. No método DrawObjects chama-se o método drawGL3 da classe oaObject para cada objecto que foi inserido no oaEngine.

newshader.cpp
→ LoadShaders
Este método que está no ficheiro newshader.cpp recebe como parâmetros ponteiros para os ficheiros com os *shaders* de vértices e de fragmentos. Lê esses ficheiros, compila-os (vão ser compilados no

runtime dos módulos) e liga-os a um ProgramID que vai ser usado noutros métodos para se poder fazer o *render* dos objectos 3D.

classe oaObject

→ drawGL3
Este é um método central para o desenho dos objectos 3D, pois aqui é que se chama o método build() que é onde efectivamente se faz o *render* dos objectos. Nas classes das meshes e dos objectos oaMeshObj, oaMeshObjAnim, oaSphere, oaCylinder, oaCone, oaTorus e oaBox é feito o *override* ao método build(). Neste método são feitas todas as operações de translação, rotação e escala sobre as posições dos objectos, as suas coordenadas no espaço, que permite que os efeitos físicos possam ser visualizados. É verificado se se pretende ter na nossa cena 3D, animações, se a propriedade do oaEngine hasAnimations for verdadeira então o build() deve ser sempre chamado, se a propriedade hasAnimations tiver valor *false* então chama-se o build() apenas uma vez. Neste método drawGL3 as matrizes Projection, View e Model são enviadas para a memória da placa gráfica.

→ loadShaders
Este método invoca o método LoadShaders do ficheiro newshader.cpp, passa os ponteiros dos ficheiros de shaders da classe oaObject para newshader. Temos aqui uma característica pouco vista nos exemplos que existem pela Internet de utilização de *shaders* em OpenGL. Podemos ter *shaders* de vértices e fragmentos por cada objecto, ou seja, podemos ter "renderizações" completamente diferentes de objecto para objecto.

→ getProgramID
Este método devolve o ProgramID do OpenGL para algum método que o pretenda usar, como é o caso do método LoadMesh da classe oaMeshObjAnim, em que é necessário quando se quer enviar um esqueleto para o *shader*.

→ defineTexBMPGL3
Este método permite ler uma imagem de um ficheiro .BMP e atribuí-la como textura a um objecto 3D. Este método deve ser usado para podermos ter texturas em objectos instanciados a partir das classes oaSphere, oaCylinder, oaCone, oaTorus e oaBox.

→ defineTexDDSGL3
Este método permite ler uma imagem de um ficheiro .DDS e atribuí-la como textura a um objecto 3D. Este método deve ser usado para podermos ter texturas em objectos instanciados a partir das classes oaSphere, oaCylinder, oaCone, oaTorus e oaBox.

→ applySphereTexture
Este método deve ser usado quando se cria uma esfera com a classe oaSphere e permite ter uma textura esférica.

→ setpose
Este método permite-nos posicionar o nosso objecto 3D numa determinada localização do espaço.

classe oaMeshObj

Esta classe herda os métodos e propriedades da classe oaObject. Usa a biblioteca Assimp (Open Asset Import Library), ver em http://assimp.sourceforge.net/. Permite ler ficheiros em muitos formatos de *meshes*, mas o formato que nos interessa particularmente é o COLLADA. A biblioteca Assimp lê as características geométricas do ficheiro COLLADA, as características físicas não é possível ler com esta biblioteca.

→ readOBJ : chama LoadMesh
Este método recebe o ponteiro para o ficheiro da mesh a carregar, e invoca o método LoadMesh passando-lhe o nome do ficheiro com a indicação que está na directoria models do OpenAR. Este método já existia em OpenGL 1, por isso foi deixado para manter a compatibilidade com o que já estava.

→ LoadMesh
Este método recebe como parâmetro o ficheiro da mesh a carregar, e através de uma classe da biblioteca Assimp carrega o conteúdo desse ficheiro para um objecto do tipo aiScene. A seguir invoca o método InitFromScene.

→ InitFromScene
Neste método chama-se o método InitMesh onde se obtêm os *arrays* de vértices, normais, coordenadas de textura e índices. A seguir invoca o método InitMaterials que carrega texturas e materiais indicadas no ficheiro da mesh. A seguir a estes dois métodos serem chamados, a memória da placa gráfica é carregada com os *arrays* de vértices, normais, coordenadas de textura e índices.

→ InitMesh
Neste método obtêm-se os array de vértices, normais, coordenadas de textura e índices a partir de um objecto do tipo aiMesh da biblioteca Assimp que é passado por parâmetro.

→ InitMaterials
Este método carrega texturas e/ou materiais a partir de um objecto do tipo aiMaterial obtido do nosso objecto Scene. No caso das texturas o caminho e nome do ficheiro de imagem com a textura é obtido do objecto da biblioteca Assimp e é passado a um método Load da classe OaMeshObjTexture. No caso dos materiais, quando obtidos são colocados na memória da placa num buffer do tipo GL_UNIFORM_BUFFER.

→ build (override): chama Render
Este método foi deixado para manter a compatibilidade com o que já estava implementado para o OpenGL 1, uma vez que o método build na classe oaMeshObj faz o override do método com o mesmo nome da classe oaObject. Invoca o método Render.

→ Render
Neste método é onde efectivamente é desenhado ("renderizado") o que está na memória da placa gráfica. É de referir que antes de serem "renderizados" os objectos com a instrução

glDrawElementsBaseVertex, deve-se “vincular” a textura, por isso é invocado o método Bind da classe OaMeshObjTexture.

→ métodos que utilizam o motor de Física (Bullet)
Foram mantidos os métodos oaStaticMesh, oaConvexMesh e buildCollisionShape para podermos ter estas *meshes* com OpenGL 3.3 e Física a funcionar. Podemos visualizar o efeito da gravidade.
classe OaMeshObjTexture

→ Load
Este método carrega o ficheiro de imagem para textura que for passado por parâmetro com a biblioteca ImageMagick que suporta múltiplos ficheiros de imagem. Para melhorar a qualidade da imagem da textura foi utilizada uma filtragem trilinear, mas também se poderia ter usado uma filtragem anisotrópica mas esta filtragem só teria interesse para superfícies que estejam em ângulos de visão oblíqua como no caso da figura seguinte:

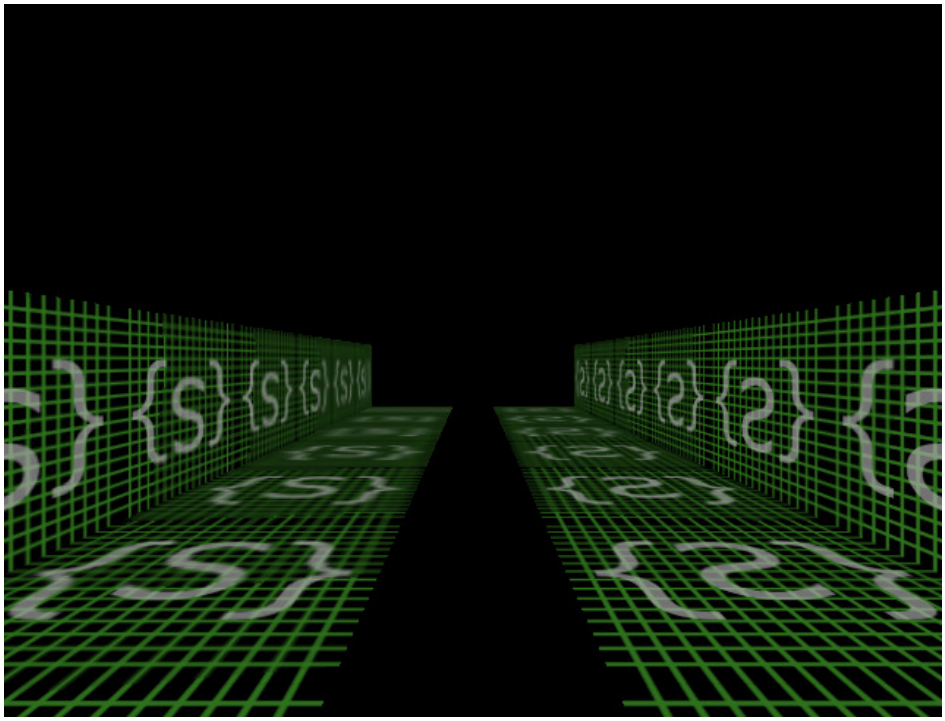

Figura : Imagem de programa teste para utilização de texturas com filtragem anisotrópica

→ Bind
Este método vincula a textura ao objecto 3D.

classe oaMeshObjAnim

Esta classe é similar à classe oaMeshObj e herda também as propriedades e métodos da classe oaObject. Usa a biblioteca Assimp (Open Asset Import Library), ver em http://assimp.sourceforge.net/. Permite ler ficheiros em alguns formatos de meshes com animações,

mas para o formato que nos interessa é o COLLADA. Este formato permite ter meshes com animações.

→ readOBJ : chama LoadMesh
Este método recebe o ponteiro para o ficheiro da *mesh* a carregar, e invoca o método LoadMesh passando-lhe o nome do ficheiro com a indicação que está na directoria models do OpenAR. Este método já existia em OpenGL 1, por isso foi deixado para manter a compatibilidade com o que já estava.

→ build (override): chama RenderAnimation
Este método foi deixado para manter a compatibilidade com o que já estava implementado no OpenGL 1. Invoca o método RenderAnimation.

→ LoadMesh
Este método foi deixado para manter a compatibilidade com o que já estava implementado no OpenGL 1, uma vez que o método build na classe oaMeshObj faz o override do método com o mesmo nome da classe oaObject. Invoca o método Render.

→ InitFromScene
Neste método chama-se o método InitMesh onde se obtêm os *arrays* de vértices, normais, coordenadas de textura, ossos e índices. A seguir invoca o método InitMaterials que carrega texturas e materiais indicadas no ficheiro da mesh. A seguir a estes dois métodos serem chamados, a memória da placa gráfica é carregada com os *arrays* de vértices, normais, coordenadas de textura, ossos e índices.

→ InitMesh
Neste método obtêm-se os arrays de vértices, normais, coordenadas de textura, ossos e índices a partir de um objecto do tipo aiMesh da biblioteca Assimp que é passado por parâmetro. Neste método é invocado o método LoadBones.

→ LoadBones
Preenche o array bones que fica em memória a partir do array do bones que vem do ficheiro com a *mesh* animada. Invoca o método AddBoneData.

-> AddBoneData
Este método constrói os arrays IDs (Identificação dos ossos) e Weights que são os pesos dos ossos.

→ InitMaterials
Este método carrega texturas a partir de um objecto do tipo aiMaterial obtido do nosso objecto Scene. O caminho e nome do ficheiro de imagem com a textura é obtido do objecto da biblioteca Assimp e é passado a um método Load da classe OaMeshObjTexture.

→ Render
Neste método é onde efectivamente é desenhado (renderizado) o que está na memória da placa gráfica. É de referir que antes de serem renderizados os objectos com a instrução

glDrawElementsBaseVertex, deve-se "vincular" a textura, por isso é invocado o método Bind da classe OaMeshObjTexture.

→ FindRotation, FindScaling, CalcInterpolatedPosition, CalcInterpolatedRotation, CalcInterpolatedScaling, ReadNodeHeirarchy, BoneTransform, FindNodeAnim, SetBoneTransform

Estes métodos são os métodos cruciais desta classe e servem para obtermos uma matriz resultado com as posições dos vértices em função do movimento dos ossos num determinado espaço de tempo.

→ renderAnimation
Este método invoca o método Render em função do tempo definido no ficheiro da mesh e do movimento do esqueleto (da transformação do esqueleto que provoca uma transformação nas coordenadas dos vértices da mesh).

→ métodos que utilizam o motor de Física (Bullet)
Foram mantidos os métodos constructTriMesh, oaStaticMesh, oaConvexMesh e buildCollisionShape para podermos ter estas meshes com OpenGL 3.3 e Física a funcionar. Podemos visualizar o efeito da gravidade.

→ classe math_3d (explicação da utilização desta classe em vez de GLM)
Foi usada esta classe para podermos trabalhar com matrizes 4x4 porque não foi possível implementar com sucesso o GLM.

ficheiro oasimpleobj.cpp
classes: oaSphere, oaCylinder, oaCone, oaTorus, oaBox

Esta classes permitem fazer o *rendering* respectivamente de esferas, cilindros, cones, toróides e paralelepípedos. Já estavam implementadas no OpenAR na versão 1 do OpenGL e usavam a biblioteca *glut*. Em OpenGL 3 foram reescritas e pode ver-se a utilização das coordenadas esféricas e cilíndricas no desenho de superfícies como as esferas, cilindros e cones. (Ver código oasimpleobj.cpp nos Anexos)

E. Código Fonte do Simulador de *Robot*

```
using namespace std;
#include <iostream>
#include <fstream>
#include <ostream>

#include <stdio.h>
#include <cstdlib>
#include <math.h>
#include <getopt.h>
#include "system.h"
#include "debuginfo.h"
#include "cv.h"
#include "highgui.h"
#include "oaengine.h"
#include "oaobject.h"
#include "oameshobjanim.h"
#include "oameshobj.h"
#include "ar.h"
#include <btBulletDynamicsCommon.h>

using namespace OpenAR;

int main(int argc, char **argv){

    int viewangle = 60;
    int scale = 2;
    float fraction = 0.1f;
    float angleX = 0.0f;
    float angleY = 0.0f;

    // actual vector representing the camera's direction
    float lx=-0.881227f,lz=0.437478f,ly = -0.179030f;

    // XZ position of the camera
    float x=6.0f,z=2.0f;

    bool plan = false;

    System::Init(1280,800,false);

    std::string vs1 = DATADIR "/shaders/dirlightdiffambpix.vert";
    std::string fs1 = DATADIR "/shaders/dirlightdiffambpix.frag";

    std::string vs2 = DATADIR "/shaders/skinning.vert";
```

```
std::string fs2 = DATADIR "/shaders/skinning.frag";

oaEngine engine;
engine.SetViewingAngle(viewangle);

oaMeshObj meshRoom;
meshRoom.loadShaders(vs1.c_str(), fs1.c_str());
meshRoom.oaConvexMesh("sala/salaTEX.obj");
meshRoom.scale(2);
meshRoom.setMass(0);
Pose pose1(0,0,0,0,1,0,30);
meshRoom.setpose(pose1);
engine.InsertObject(&meshRoom);

oaMeshObj meshRobot;
meshRobot.loadShaders(vs1.c_str(), fs1.c_str());
meshRobot.oaConvexMesh("sala/robotTEX.obj");
meshRobot.scale(2);
meshRobot.setMass(0);
Pose robotPose(0,0,0,0,1,0,30);
meshRobot.setpose(robotPose);
engine.InsertObject(&meshRobot);

for(;;) {

    SDL_Event ev;

    while(SDL_PollEvent(&ev)) {

        switch(ev.type) {
        case SDL_QUIT:
            return 0;
            break;
        case SDL_MOUSEBUTTONDOWN:
            switch (ev.button.button){
            case SDL_BUTTON_LEFT:
                break;
            case SDL_BUTTON_RIGHT:
                break;
            }
            break;
        case SDL_MOUSEBUTTONUP:
            switch (ev.button.button){
            case SDL_BUTTON_LEFT:
                break;
            case SDL_BUTTON_RIGHT:
```

```
            break;
        }
        break;

    case SDL_MOUSEMOTION:
        break;
    case SDL_KEYDOWN:
        switch (ev.key.keysym.sym){
        case  SDLK_ESCAPE:
        case SDLK_q:
            return 0;
        case SDLK_f:
            System::GL::ToggleFullScreen();
            break;
        case SDLK_PAGEUP:
            break;
        case SDLK_PAGEDOWN:
            break;
        case SDLK_UP:
                x += lx * fraction;
                z += lz * fraction;
                robotPose.tx += lx * fraction;
                robotPose.tz += lz * fraction;
                break;
        case SDLK_DOWN:
                x -= lx * fraction;
                z -= lz * fraction;
                robotPose.tx -= lx * fraction;
                robotPose.tz -= lz * fraction;
                break;
        case SDLK_LEFT:
            angleY += 0.01f;
            break;
        case SDLK_RIGHT:
            angleY -= 0.01f;
            break;
        case SDLK_w:
            angleX += 0.01f;
            break;

        case SDLK_s:
            angleX -= 0.01f;
            break;
        case SDLK_i:
            break;
        case SDLK_a:
```

```
                    break;
                case SDLK_x:
                    break;
                case SDLK_d:
                    break;
                case SDLK_p:
                    if (!plan) {
                        plan = true;
                    } else {
                        plan = false;
                    }
                    break;
                default:
                    std::cout << "key"<< std::endl;
                    break;
                }
                break;
            }

            if (plan) {
                engine.moveCam(0,390,-40, 0, 0, 0, 0, 0, 1);
            } else {

                lz = cos(angleY) * cos(angleX);
                lx = sin(angleY) * cos(angleX);
                ly = sin(angleX);

                engine.moveCam(x, 20.0f, z, x+lx, 20.0f+ly, z+lz,
0, 1, 0);

                meshRobot.setpose(robotPose);
            }

        }

        engine.drawScene();
        engine.updatePhysics();

    }

    return 0;
}
```

Bibliografia

[1] A. Staranowicz, Gian Luca Mariottini, "A survey and comparison of commercial and open-source robotic simulator software", In *Proc. Int. Conf. on Pervasive Technologies Related to Assistive Environments (PETRA)*, pp. 56:1-56:8, 2011.

[2] Brian Gerkey, Richard T. Vaughan and Andrew Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems", In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)* pp. 317-323, Coimbra, Portugal, Junho 30 - Julho 3, 2003.

[3] Gazebo, "http://gazebosim.org/", Março 2015

[4] Object-Oriented Graphics Rendering Engine, "http://www.ogre3d.org/", Março 2015

[5] Open Dynamics Engine, "http://www.ode.org/", Março 2015

[6] Robot Operating System, "http://www.ros.org/wiki/", Março 2015

[7] Modular OpenRobots Simulation Engine, "https://www.openrobots.org/wiki/morse/", Março 2015

[8] Blender, "http://www.blender.org/", Março 2015

[9] Bullet Physics Library, "http://bulletphysics.org/wordpress/", Março 2015

[10] Yet Another Robot Platform, "http://wiki.icub.org/yarpdoc/", Março 2015

[11] Pocolibs, "https://www.openrobots.org/wiki/pocolibs", Março 2015

[12] MOOS, "http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php/Main/HomePage", Março 2015

[13] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, C. Scrapper, "Usarsim: A robot simulator for research and education", In *Proceedings IEEE International Conference on Robotics and Automation (ICRA 2007)* pp. 1400-1405, Roma, Itália, Abril, 2007

[14] Robocup, "http://en.wikipedia.org/wiki/RoboCup", Março 2015

[15] Unreal Tournament, "http://www.unrealtournament.com", Março 2015

[16] MOAST, "http://sourceforge.net/projects/moast/", Março 2015

[17] MATLAB USARSim Toolbox, "http://robotics.mem.drexel.edu/USAR/", Março 2015

[18] Renderização 3D, "http://en.wikipedia.org/wiki/3D_rendering", Março 2015

[19] Unity, "http://unity3d.com/pt", Março 2015

[20] Binh-Son Le, Vy-Long Dang, Trong-Tu, "Swarm Robotics Simulation Using Unity", ICDV 2014, Novembro, 2014

[21] Jeff Craighead, Jenny Burke, Robin Murphy. “Using the Unity Game Engine to Develop SARGE: A Case Study”, In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008),* Setembro, 2008.

[22] U.H. Hernandez-Belmonte, V. Ayala-Ramirez, R.E. Sanchez-Yanez, "A mobile robot simulator using a game development engine", In *Proc. Robotics Summer Meeting ROSSUM 2011*, Xalapa, Mexico, Junho 27-28, 2011

[23] VirtualVEX, "https://sites.google.com/site/virtualvex/about", Março 2015

[24] Search and Rescue Game Environment, "http://sarge.sourceforge.net/page11/page11.html", Março 2015

[25] Botnavsim, "https://github.com/explosivose/botnavsim", Março 2015

[26] Excavator "http://www.fabrejean.net/projects/excavator/", Março 2015

[27] OpenGL, "https://www.opengl.org/", Março 2015

[28] OGL dev - Modern OpenGL Tutorials - "http://ogldev.atspace.co.uk/", Março 2015

[29] Tutorials for modern OpenGL (3.3+), "http://www.opengl-tutorial.org/", Março 2015

[30] COLLADA From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/COLLADA", Março 2015

[31] COLLADA site - "http://collada.org/", Março 2015

[32] Wavefront .obj file - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Wavefront_.obj_file", Março 2015

[33] John Kessenich, Dave Baldwin, Randi Rost, *The OpenGL Shading Language Version 3.30.6*, p3, Khronos Group, 2010

[34] OpenGL Shading Language From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/OpenGL_Shading_Language", Março 2015

[35] Robotics simulator - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Robotics_simulator", Março 2015

[36] Neto, P., Pires, J.N. , Moreira, A.P., "Robot path simulation: a low cost solution based on CAD ", In *IEEE Conference on Robotics Automation and Mechatronics (RAM) 2010*, pp. 333-338, Singapura, 28-30 June, 2010

[37] Robotics Simulation Softwares With 3D Modeling and Programming Support, "http://www.intorobotics.com/robotics-simulation-softwares-with-3d-modeling-and-programming-support/", Março 2015

[38] Extensible Markup Language, " http://www.w3.org/XML/", Março 2015

[39] Shader, "http://en.wikipedia.org/wiki/Shader/", Março 2015

[40] Cg, "http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html", Março 2015

[41] High Level Shading Language for DirectX, "https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx", Março 2015

[42] Ken Joy, Geometric Modeling Lectures, Institute for Data Analysis and Visualization Computer Science Department University of California, "http://graphics.cs.ucdavis.edu/~joy/GeometricModelingLectures/", Abril 2015

[43] Max K. Agoston , *Computer Graphics and Geometric Modelling: Implementation & Algorithms*, p156-188, Springer Science & Business Media, London, 2005

[44] João Brisson Lopes, Textos Computação Gráfica, "http://disciplinas.ist.utl.pt/leic-cg/textos/", Abril 2015

[45] Polygon mesh - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Polygon_mesh", Abril 2015

[46] Sungkil Lee, Jounghyun Kim, G., Seungmoon Choi, "Real-Time Tracking of Visually Attended Objects in Virtual Environments and Its Application to LOD", In *IEEE Transactions on Visualization and Computer Graphics*, pp. 6-19, Maio 2008

[47] Joaquim Madeira, Introdução à Modelação Geométrica usando Malhas Poligonais, "http://sweet.ua.pt/bss/aulas/2013/VI/CG_Malhas_Poligonais_JM.pdf", Abril 2015

[48] 3D Modelling - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/3D_modeling", Abril 2015

[49] Colin Smith, On Vertex-Vertex Meshes and Their Use in Geometric and Biological Modeling, "http://algorithmicbotany.org/papers/smithco.dis2006.pdf", pp.129-137, Abril 2015

[50] Triangle mesh - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Triangle_mesh", Abril 2015

[51] Triangle strip - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Triangle_strip", Abril 2015

[52] Vertex Buffer Object - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Vertex_Buffer_Object", Abril 2015

[53] Wavefront .OBJ - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/Wavefront_.obj_file", Abril 2015

[54] UV Mapping - From Wikipedia, the free encyclopedia, " http://en.wikipedia.org/wiki/UV_mapping", Abril 2015

[55] COLLADA - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/COLLADA", Abril 2015

[56] XML schema - From Wikipedia, the free encyclopedia, "http://en.wikipedia.org/wiki/XML_schema", Abril 2015

[57] COLLADA 1.5.0 Specification, "https://www.khronos.org/files/collada_spec_1_5.pdf", Abril 2015

[58] Triangle meshes, "http://gamma.cs.unc.edu/COMP770/LECTURES/11trimesh.pdf", Maio 2015

[59] Remi Arnaud, Mark C. Barnes, *Collada: Sailing the Gulf of 3d Digital Content Creation*, AK

P.ters, 2006, ISBN: 1568812876

[60] COLLADA 1.4.0 Specification,
"https://www.khronos.org/files/collada_spec_1_4.pdf", Maio 2015

[61] Skeletal Animation - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Skeletal_animation", Maio 2015

[62] Morph target animation - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Morph_target_animation", Maio 2015

[63] Boundary Representation, COLLADA, and STEP,
"http://www.openclblog.com/2012/04/boundary-representation-collada-and.html", Maio 2015

[64] Boundary Representation - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Boundary_representation", Maio 2015

[65] OpenGL Normal Vector Transformation,
"http://www.songho.ca/opengl/gl_normaltransform.html", Maio 2015

[66] Normal Mapping - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Normal_mapping", Maio 2015

[67] Viewing frustum - From Wikipedia, the free encyclopedia, "
http://en.wikipedia.org/wiki/Viewing_frustum", Maio 2015

[68] Field of view in video games - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Field_of_view_in_video_games", Maio 2015

[69] Display aspect ratio - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Display_aspect_ratio", Junho 2015

[70] Charge-coupled device - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Charge-coupled_device", Junho 2015

[71] Skeletal Animation - From Wikipedia, the free encyclopedia,
"http://en.wikipedia.org/wiki/Skeletal_animation", Junho 2015

[72] OpenGL:Tutorials:Basic Bones System,
"http://content.gpwiki.org/index.php/OpenGL:Tutorials:Basic_Bones_System", Junho 2015

[73] What is the difference between Concave and Convex?,

"http://www.rustycode.com/tutorials/convex.html", Junho 2015

[74] PhysX - From Wikipedia, the free encyclopedia,
"https://en.wikipedia.org/wiki/PhysX", Junho 2015

[75] Havok (software) - From Wikipedia, the free encyclopedia,
"https://en.wikipedia.org/wiki/Havok_(software)", Junho 2015

[76] Physics Abstraction Layer - From Wikipedia, the free encyclopedia,
"https://en.wikipedia.org/wiki/Physics_Abstraction_Layer", Junho 2015

[77] Momento de inércia - Origem: Wikipédia, a enciclopédia livre,
"https://pt.wikipedia.org/wiki/Momento_de_in%C3%A9rcia", Junho 2015

[78] Mike Bailey, Steve Cunningham, *Graphics Shaders Second Edition*, CRC Press, 2012

[79] Etay Meiri, OGL dev - Modern OpenGL Tutorials,
"http://ogldev.atspace.co.uk", Junho 2015

[80] Assimp, Open Asset Import Library, "http://assimp.sourceforge.net", Junho 2015