Pontifícia Universidade Católica
do Rio de Janeiro

Renato Deris Prado

Visualização de Rótulos em Objetos de Modelos Massivos em Tempo Real

DISSERTAÇÃO DE MESTRADO

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Orientador: Alberto Barbosa Raposo

Rio de Janeiro
Abril de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Renato Deris Prado

Visualização de Rótulos em Objetos de Modelos Massivos em Tempo Real

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Prof. Alberto Barbosa Raposo
Orientador
Departamento de Informática – PUC-Rio

Prof. Waldemar Celes Filho
Departamento de Informática – PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Dr. Ismael Humberto Ferreira dos Santos
Petrobras

Prof. José Eugênio Leal
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 05 de abril de 2013

Renato Deris Prado

Graduado em Engenharia da Computação na Pontifícia Universidade Católica do Rio de Janeiro, atualmente trabalha no TecGraf, instituto de pesquisa associado à PUC-Rio, tendo como área de concentração Computação Gráfica.

Ficha Catalográfica

Prado, Renato Deris

Visualização de rótulos em objetos de modelos massivos em tempo real / Renato Deris Prado ; orientador: Alberto Barbosa Raposo. – 2013.

57 f: il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2013.

Inclui bibliografia

1. Informática – Teses. 2. Computação Gráfica. 3. Programação em GPU. 4. Visualização em tempo real. I. Raposo, Alberto Barbosa. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

A PUC-Rio pela bolsa de mestrado.

Ao orientador Alberto.

Ao Tecgraf, ao grupo CAE e a todos do projeto Environ, onde trabalho.

Ao professor Waldemar, pela grande ajuda.

A Fabrício, pelas revisões no texto.

A Mariana, a família e amigos.

Resumo

Prado, Renato Deris; Raposo, Alberto Barbosa. **Visualização de Rótulos em Objetos de Modelos Massivos em Tempo Real.** Rio de Janeiro, 2013. 57p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Rótulos virtuais são utilizados em aplicações de computação gráfica para representar informações textuais dispostas sobre superfícies geométricas. Tais informações consistem em nomes, numerações, ou outros dados relevantes que precisem ser notados rapidamente quando um usuário examina os objetos da cena. Este trabalho tem como foco os chamados modelos massivos, como modelos CAD (*Computer Aided Design*) de refinarias de petróleo, os quais possuem um grande número de primitivas geométricas cujo *rendering* apresenta um alto custo computacional. Em grandes projetos de engenharia, é desejável a visualização imediata de informações específicas de cada objeto ou de partes do modelo, as quais, se exibidas por meio de técnicas convencionais de texturização podem extrapolar os recursos computacionais disponíveis. Nesta dissertação desenvolvemos uma forma de exibir, em tempo real, rótulos virtuais com informações distintas, nas superfícies de objetos de modelos massivos. A técnica é implementada inteiramente em GPU, não apresenta perda significativa de desempenho e possui um baixo gasto de memória. Os objetos de modelos CAD são o foco principal do trabalho, apesar de a solução poder ser utilizada em outros tipos de objetos desde que suas coordenadas de textura sejam corretamente ajustadas.

Palavras-chave

Rótulos virtuais; Rotulação de superfícies; Programação em GPU; Visualização em tempo real

Abstract

Prado, Renato Deris; Raposo, Alberto Barbosa (advisor). **Real-Time Label Visualization in Massive Models Objects.** Rio de Janeiro, 2013. 57p. MSc. Dissertation– Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Virtual Labels are used in computer graphics applications to represent textual information arranged on geometric surfaces. Such information consists of names, numbering, or other relevant data that need to be noticed quickly when a user scans the objects in the scene. This paper focuses on the so-called massive models, as CAD models (Computer Aided Design) of oil refineries, which have a large number of geometric primitives whose rendering presents a high computational cost. In large engineering projects, the immediate visualization of information specific to each object or parts of the model is desirable, which, if displayed by conventional texturing techniques can extrapolate the available computational resources. In this work we have developed a way to view, in real time, virtual labels with different information on the surfaces of objects in massive models. The technique is implemented entirely on the GPU, shows no significant loss of performance and low memory cost. CAD models objects are the main focus of the work, although the solution can be used in other types of objects once their texture coordinates are adjusted correctly.

Keywords

Virtual labels; Surface labeling; GPU programming; Real time visualization

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

Um dos principais objetivos dos setores de engenharia de grandes companhias, tais como automotivas e de óleo & gás, é disponibilizar softwares de engenharia para a gerência de seus projetos. Tais companhias buscam sistemas computacionais que, em adição ao simples acesso a bancos de dados contendo informações do projeto, sejam interativos, forneçam recursos para a visualização de modelos tridimensionais (3D) com um nível suficiente de realismo e atuem como ferramentas que agilizem o trabalho dos profissionais envolvidos, ao permitirem que cada vez mais tarefas possam ser automatizadas.

Muitos dos modelos de engenharia utilizados nesses casos são provenientes de arquivos CAD (*Computer Aided Design*), que representam modelos para visualização científica, como prédios, veículos, entre outros [1]. Esses modelos são chamados massivos quando apresentam grande complexidade, como uma refinaria de petróleo constituída por milhões ou bilhões de objetos [2]. Um dos desafios na construção de sistemas de visualização em tempo real é o desenvolvimento de técnicas que permitam o *rendering* desse grande volume de dados com qualidade gráfica e realismo aceitáveis, mesmo em máquinas convencionais (que não necessariamente apresentam grande poder computacional).

Os arquivos CAD utilizados neste trabalho descrevem modelos compostos por objetos de diferentes geometrias, as quais podem representar superfícies arredondadas (esferas, cones, cilindros e toros), superfícies planas (caixas e pirâmides) ou malhas arbitrárias de triângulos. Para objetos de superfícies arredondadas e planas, existem informações específicas associadas, como raio e altura de um cilindro, enquanto para as outras malhas, essas informações são coordenadas de vértices e normais. Em conjunto, esses objetos podem formar, por exemplo, uma grande refinaria ou plataforma de petróleo. A aplicação é responsável por ler esses arquivos e mostrar os objetos na tela corretamente.

O custo computacional necessário para o *rendering* dos modelos CAD de alta complexidade pode comprometer a possibilidade de serem usados em um ambiente de realidade virtual. Um gargalo bastante comum em aplicações de computação gráfica são as geometrias a serem visualizadas, o que é relacionado à quantidade de vértices que constituem o modelo. Tal problema pode estar relacionado às limitações impostas pela largura de banda da interface entre CPU e GPU [3]. A Figura 1 mostra parte de uma refinaria de petróleo, sendo visualizada no software Environ [4], [5], na qual é possível notar uma grande concentração de objetos de diferentes formas e tamanhos.

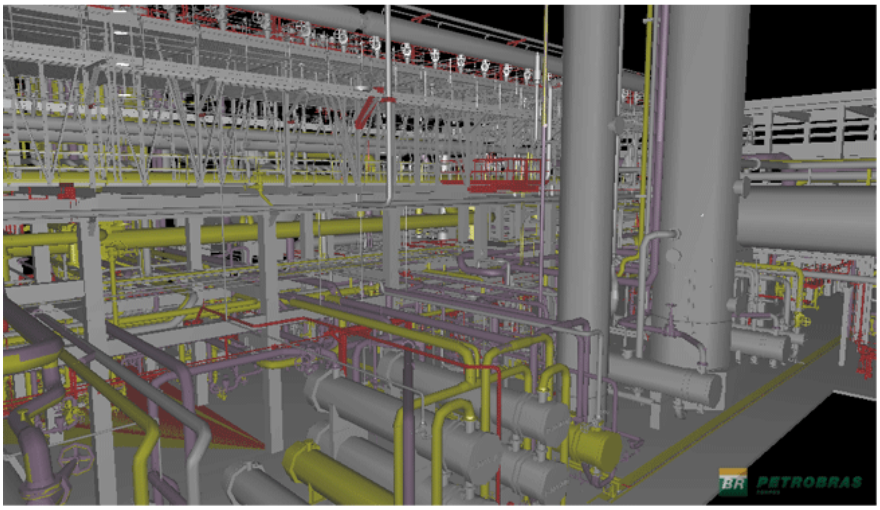


Figura 1: Refinaria de petróleo visualizada no software Environ.

Em construções criadas pela indústria petrolífera, como refinarias e plataformas de petróleo, existem estruturas que contém rótulos aplicados a suas superfícies, que exibem os nomes dessas estruturas ou outras informações sobre elas, com o objetivo de ajudar os profissionais a identificá-los facilmente nos trabalhos de campo, conforme ilustrado na Figura 2. Da mesma forma que esses rótulos ajudam os engenheiros a identificar as estruturas reais, rótulos virtuais podem ajudar os usuários a identificar objetos em softwares de visualização em tempo real.

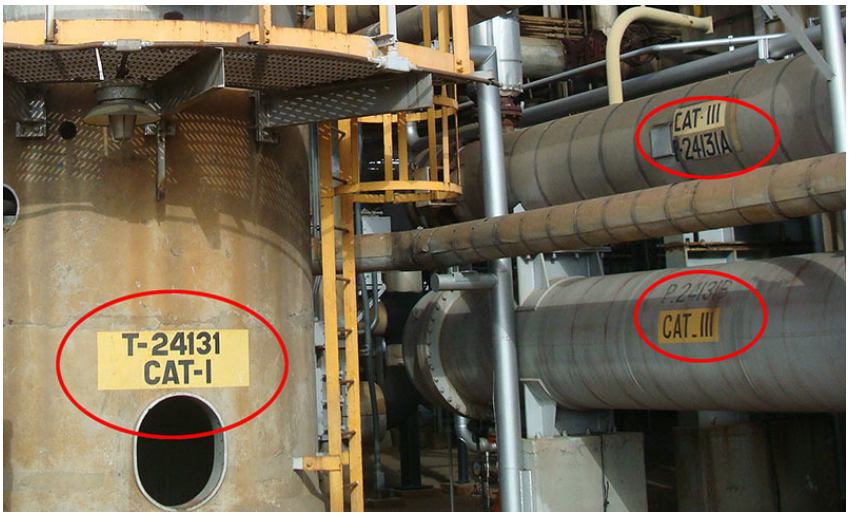


Figura 2: Rótulos de objetos em uma refinaria de petróleo.

A exibição de rótulos nas superfícies dos objetos virtuais tem como principal objetivo ajudar o usuário da aplicação a identificar rapidamente quais desses objetos estão sendo visualizados, além de fornecer auxílio quanto à orientação no cenário virtual. Algo comum em softwares de visualização de modelos CAD é que exista uma lista que mostre os nomes de todos os objetos presentes na cena Com isso, para descobrir o nome de um objeto, o usuário precisa clicar sobre ele e verificar naquela lista o nome que é selecionado, em um processo mais lento do que usar os rótulos. Outra melhoria que pode ser obtida com o uso de rótulos virtuais é a exibição não só de nomes dos objetos, mas também de outras informações visuais em forma de texto, como a identificação de objetos em manutenção ou qualquer outro tipo de informação cuja visualização imediata seja interessante.

No entanto, existem alguns desafios para a visualização de rótulos virtuais, um deles é relacionado à existência de um grande número de objetos com nomes distintos em um modelo CAD. Para exemplificar esse problema, se uma técnica simples de texturização fosse utilizada, poderia ser construída uma textura diferente para cada objeto (devido a cada objeto possuir um nome diferente). Tal abordagem implica em um gasto de memória que pode exceder a capacidade da placa de vídeo. Para contornar esse problema, torna-se necessária a elaboração de estratégias para manter em memória de vídeo (VRAM) um limite de texturas por quadro. Também é possível que nem todas as texturas possam ser guardadas na memória principal ao mesmo tempo e, dessa forma, precisem ser reconstruídas

para formar as palavras de um rótulo em tempo real, o que pode prejudicar o desempenho da aplicação.

Mesmo em um cenário onde é possível armazenar uma textura diferente para cada objeto, existe ainda a necessidade de realizar múltiplas trocas de contexto de textura a cada quadro, o que pode se tornar o gargalo da aplicação de computação gráfica [6]. Outro grande desafio é o posicionamento de rótulos em geometrias de diferentes formatos e tamanhos, sendo necessário que os rótulos virtuais sejam posicionados automaticamente pela aplicação em todos os casos. Além disso, a distância de um objeto em relação à câmera pode causar *aliasing* no rótulo quando ele é ampliado ou reduzido.

Esses problemas (gasto excessivo de memória de vídeo, muitas trocas de contexto de textura por quadro, posicionamento de rótulos em diferentes superfícies e *aliasing*) dão margem à exploração de técnicas para a exibição de rótulos virtuais em modelos massivos visualizados em tempo real. Como a visualização de modelos massivos por si só já apresenta um alto custo computacional, é necessária uma solução que não piore o desempenho e que gaste o mínimo de memória possível.

A proposta desta dissertação, portanto, é apresentar uma técnica para exibir rótulos virtuais em modelos massivos, a qual foi implementada inteiramente em GPU. Tal técnica não causa perda de desempenho significativa independente do número de objetos que exibem rótulos e consome pouca memória de vídeo com rótulos codificados e armazenados em um *buffer* no *hardware* gráfico. A técnica se baseia em um algoritmo de duas passadas, em que a primeira realiza o *rendering* da cena e armazena informações de coordenadas de textura em buffers fora da tela, enquanto a segunda utiliza as informações desse buffer para mapear os rótulos nos objetos virtuais, calculando a cor final de cada pixel da tela individualmente por meio da amostragem de uma textura que contém todos os caracteres.

O Capítulo 2 apresenta alguns trabalhos relacionados a problemas de *rendering* de texturas contendo informações textuais. O Capítulo 3 descreve a técnica criada neste trabalho para o *rendering* de rótulos em GPU. O Capítulo 4 mostra como posicionar os rótulos virtuais em cilindros, caixas, esferas e “dishes” (uma geometria existente nos modelos CAD que será apresentada nesse mesmo capítulo). O Capítulo 5 fala sobre a *engine* de modelos CAD construída para este

trabalho e mostra os rótulos sendo utilizados em modelos CAD reais. O Capítulo 6 mostra um estudo sobre *aliasing* nos rótulos e fala sobre uma possível solução para esse problema, enquanto o Capítulo 7 mostra todos os resultados alcançados. O Capítulo 8, por fim, apresenta as conclusões e propostas de trabalhos futuros.

2
Trabalhos relacionados

Existem alguns poucos trabalhos que abordam diretamente o problema de visualização em tempo real de texto em modelos massivos. Este capítulo apresenta resumidamente esses trabalhos e em seguida mostra alguns outros, que tratam de problemas mais gerais de *rendering* de texto (como posicionamento, gasto de memória e *aliasing*) e os relaciona com algumas questões se fossem aplicados para o caso dos rótulos virtuais.

O software comercial Aveva Review [7] consegue exibir rótulos nas superfícies dos objetos de modelos CAD (Figura 3), os quais possuem nomes únicos. No entanto, somente cilindros e caixas exibem rótulos e nem todos eles mostram seus rótulos ao mesmo tempo – existe uma estratégia para selecionar quais objetos exibem seus nomes a cada quadro, aparentemente os com maior tamanho na tela tem prioridade. À medida que a câmera se move é possível notar o texto aparecendo e desaparecendo dos objetos.

Figura 3: Modelo CAD com rótulos no software Aveva Review

Algo que pode levar uma aplicação a limitar o número de objetos que exibem rótulos ao mesmo tempo é o grande número de trocas de contexto de textura que podem ser necessárias a cada quadro – devido a cada objeto possuir um nome diferente. Para resolver o problema de múltiplas trocas de contexto de

textura, a empresa NVIDIA [6], explica o conceito de "atlas de textura" mostrando os benefícios de agrupar imagens em uma única textura chamada de *atlas* e usar coordenadas de textura para acessar o sub-retângulo relevante desse *atlas*. No presente trabalho é utilizado um *atlas* que guarda todos os caracteres necessários, e os rótulos precisam ser construídos a partir dele.

Prado, Raposo e Soares [8] abordam no artigo "Autotag" o mesmo assunto deste trabalho (exibição de texto em objetos de modelos massivos), e utiliza um *atlas* que armazena todos os caracteres necessários. Para criar os rótulos (ou *tags* como são chamados nesse caso), são construídas texturas a partir desse atlas contendo o nome de cada objeto, o que é feito na primeira vez que o arquivo CAD é lido e os nomes identificados. Com isso, é realizado um mapeamento das texturas em cilindros e caixas de tamanhos fixos.

Porém, em "Autotag", não existe gerência de memória RAM e nem de memória de vídeo, todas as texturas são criadas e se mantém em memória enquanto a aplicação está ativa, o que é inviável para modelos massivos. Devido à existência de uma textura diferente por objeto, o problema de troca de contexto de texturas existe, afetando bastante o desempenho da aplicação de teste. Esse problema é contornado com um limite de objetos que exibem rótulos por quadro – um número arbitrário de objetos que está a uma determinada distância da câmera.

Existem outros trabalhos que não tratam diretamente de modelos massivos, mas abordam problemas ligados ao *rendering* de texto. Lefebvre, Hornus e Neyret [9] tentam resolver o problema de mapeamento de texturas em geometrias complexas. Para isso utilizam o conceito de "Texture Sprites", objetos que guardam pequenas texturas, e que juntos podem formar uma textura maior. Para cada objeto que possui *sprites*, é utilizada uma estrutura similar a uma *octree* que pode ser mapeada em uma textura 3D em GPU e que guarda informações de cada *sprite* como posição, tamanho e *id* da textura. O trabalho não mostra exemplos que utilizem texto diretamente, mas seria possível que *sprites* guardassem letras e que juntos formassem palavras.

Qin, McCool e Kaplan [10] apresentam uma forma de realizar o *rendering* de texto em objetos tridimensionais representando texto de forma vetorial para que caracteres apresentem muito pouco serrilhamento quando magnificados. Utilizam uma estratégia similar a de "Texture Sprites" [9] para posicionar as palavras nos objetos 3D. Porém, utilizam uma *quadtree* em CPU para guardar

uma tabela de caracteres, com resoluções diferentes dependendo de uma complexidade estipulada para cada caractere, e uma "tabela de sprites" para organizar palavras. A tabela é codificada no *hardware* gráfico como uma textura, de forma que possa ser amostrada diretamente em GPU. Assim como em "Texture Sprites", a técnica é mostrada em um único objeto por vez.

Cipriano e Gleicher [11] se concentram no posicionamento de texto em superfícies com curvatura acentuada e até mesmo com buracos. Para isso, criam malhas, chamadas de "andaimes", para guardar as letras e depois as posicionam de forma que flutuem sobre os objetos. Um único *atlas* é utilizado para guardar todas as letras e as coordenadas de textura dos "andaimes" são configuradas para que sejam obtidas as letras relevantes. Um argumento contra essa solução é a introdução de novas geometrias na cena, o que pode não ser desejável em conjunto com a visualização de modelos massivos, no caso de todos os objetos precisarem exibir algum texto.

O uso de texturas virtuais como uma solução *out-of-core*, para texturas que ocupam um tamanho muito grande e não podem ser armazenadas inteiramente em RAM ou VRAM, foi explicado por Barret [12] e Mittring [13]. Uma "textura virtual" é armazenada no HD e dividida em partes de mesmo tamanho chamadas de páginas. Somente as páginas necessárias pela API gráfica são transferidas para memória e armazenadas em uma "textura física", como é chamada por Barret. Para mapear as páginas virtuais em páginas físicas é construída uma tabela de páginas. *Mipmaps* virtuais e uma tabela de páginas por *mipmap* virtual são utilizados para resolver o problema de *aliasing* de textura.

No caso de rótulos virtuais, eles precisariam ser construídos como texturas em pré-processamento e organizados em uma textura virtual, o que poderia resolver o problema de memória. A dificuldade se encontra em como organizar a textura virtual pois os nomes dos objetos variam de tamanho e seria preciso que o tamanho de página (que é igual para todas as páginas), fosse suficiente para conter o maior dos nomes. Isso implicaria em páginas com espaço inútil. Outra forma de organizar a textura virtual seria com um nome podendo ocupar mais de uma página e objetos, portanto, precisando de mais de uma página para exibirem seus nomes. A solução utilizada neste trabalho seguiu outro caminho criando somente os rótulos necessários (que irão aparecer na tela) a cada quadro e gastando pouca memória para guardá-los, como será mostrado no Capítulo 3.

3
Aplicação de Rótulos em GPU

Este capítulo explica o procedimento para exibição de rótulos em um grande número de objetos e escrita de palavras sobre as geometrias em tempo real. Para resolver essas questões, foi desenvolvido um algoritmo de duas passadas em GPU, sobre o qual será dada uma visão geral nos próximos parágrafos. Nas seções subsequentes, serão apresentados detalhes da implementação do algoritmo, realizada utilizando a API OpenGL 2.1 [14] e OpenGL Shading Language 1.2 [15], junto a breves considerações sobre o seu desempenho.

A primeira passada do algoritmo realiza o *rendering* da cena para um *buffer* de cores fora da tela, o qual é consultado na segunda passada para o cálculo da cor final dos *pixels*. Além das informações de cor, a adição de rótulos à cena depende do acesso a um outro conjunto de dados na segunda etapa, como as coordenadas de textura por *pixel* e um identificador de qual rótulo deve ser exibido sobre a geometria à qual o *pixel* pertence. Portanto, é necessário que a primeira passada preencha um segundo *buffer* com essas informações adicionais, as quais são obtidas na primeira passada da seguinte forma: as coordenadas de textura por *pixel* são geradas pela GPU a partir da interpolação das coordenadas de textura definidas no programa de vértices; o identificador do rótulo é informado pela aplicação antes do desenho de cada objeto e é armazenado para cada *pixel* em GPU.

Além das informações preenchidas na primeira passada, existem outros recursos que precisam estar disponíveis na etapa posterior. Para calcular a cor final dos *pixels*, é preciso saber se eles devem exibir parte de algum rótulo e que informação textual esse rótulo contém (como “Box 1”, “Box 2”, “Box 3”, etc.). Por esse motivo, antes do *rendering*, o conteúdo dos rótulos foi armazenado em uma textura que é consultada como uma tabela relacionando o índice do rótulo ao seu texto. Também precisa estar disponível um *atlas* que contém todos os caracteres necessários para que, em tempo real, sejam desenhadas palavras sobre as superfícies geométricas.

3.1 Codificação de Rótulos em GPU

A informação textual de todos os rótulos precisa estar em GPU para que os *pixels* possam ter suas cores finais calculadas corretamente. A aplicação precisa, portanto, organizar os textos em uma estrutura de dados que possa ser consultada em GPU e, para este propósito, foi escolhida uma textura 2D. A quantidade de *texels* que uma textura 2D pode armazenar depende do hardware gráfico, mas em geral há espaço para bastante informação – em uma placa gráfica NVIDIA 9600GT, por exemplo, é possível armazenar 8192 x 8192 *texels*, sendo que cada *pixel* contém 4 *bytes* para as componentes de cor RGBA, totalizando em 256 MB de espaço. Como cada caractere ASCII ocupa um *byte*, mais de 256 milhões de caracteres podem ser armazenados.

É claro que, considerando que os modelos a serem visualizados são massivos e suas geometrias podem consumir bastante memória de vídeo, pode não ser desejável gastar tanto com os rótulos. No entanto, se cada rótulo tiver 20 letras e forem necessários dois milhões de rótulos, o gasto de memória de vídeo será de aproximadamente 38 MB, o que não é alto considerando a grande quantidade de informações. Com a escolha de usar uma textura 2D, a aplicação deve preenchê-la organizando os caracteres em seu espaço de memória e em seguida deve guardá-la em GPU para que possa ser amostrada na segunda passada de *rendering*.

A aplicação divide a textura em partes iguais para conter o texto dos rótulos (código ASCII de cada caractere). Isso significa que o tamanho máximo dos rótulos deve ser determinado e, independente da quantidade de caracteres que eles possuem, todos gastam a mesma quantidade de memória. Assim, é possível que um objeto guarde o índice que aponta para onde está o seu texto na textura 2D, o qual é utilizado pela GPU para localizar o rótulo correspondente.

3.2 Atlas de Caracteres

Para que esteja disponível em GPU a imagem dos caracteres que podem formar o texto dos rótulos, todos os caracteres necessários foram agrupados em um único *atlas* de textura, como pode ser visto na Figura 4. Essa textura pode ter qualquer tamanho desde que o espaço (largura e altura), ocupado por todos os caracteres seja o mesmo e que a aplicação os conheça. Um dos *atlas* utilizados no

trabalho possui 512 x 512 *texels*, cada caractere ocupa 32 x 32 *texels*, logo, é possível tratar essa textura como uma matriz de caracteres de 16 linhas por 16 colunas.

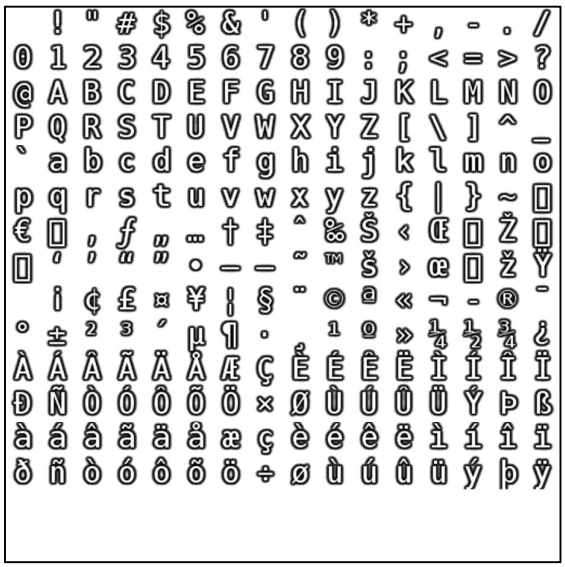


Figura 4: Atlas com todos os caracteres.

Os caracteres foram dispostos horizontalmente seguindo a mesma ordem da codificação ASCII. Dessa forma, é possível encontrar na imagem a posição de um caractere, por sua linha e coluna iniciais. Para isto, no entanto, é necessário definir o índice desse caractere em relação ao primeiro caractere da textura, no caso o espaço em branco (*whitespace*), de valor 32 na tabela ASCII. A determinação da linha l e da coluna c de um determinado caractere de código ASCII i pode ser realizada de acordo com as equações 3-1 a 3-3:

$$i = c_{des} - c_{whitespace} = c_{des} - 32 \tag{3-1}$$

$$c = i \bmod n_{col} \tag{3-2}$$

$$l = \text{floor}\left(\frac{i}{n_{col}}\right) \tag{3-3}$$

onde c_{des} é o índice do caractere em relação ao primeiro caractere do atlas, no caso o espaço, e n_{col} o número de colunas da textura utilizada.

Para encontrar o caractere ‘w’ (de código ASCII 119) no atlas mostrado na Figura 4, por exemplo, obtém-se o índice 87 através da equação 3-1, o qual, utilizado diretamente nas equações 3-2 e 3-3, resulta na linha 5 e coluna 7, o que pode ser confirmado a partir de uma busca visual na figura, tendo em vista que a primeira linha e coluna têm índice 0. Porém, esses calculos são realizados somente

na segunda passada de *rendering*, em GPU, quando é preciso saber a posição de um caractere (linha e coluna) em coordenadas de textura.

Sabendo que o espaço de textura varia de 0 a 1 em ambas as dimensões e sua origem é no canto esquerdo inferior do atlas, com o cálculo das equações 3-1 a 3-3 tendo sido realizado previamente, é simples encontrar esses valores como mostram as equações 3-4 a 3-7:

$$charWidth_{atlas} = \frac{1}{n_{col}} \tag{3-4}$$

$$charHeight_{atlas} = \frac{1}{n_{lin}} \tag{3-5}$$

$$s_{char} = c * charWidth_{atlas} \tag{3-6}$$

$$t_{char} = 1 - (l * charHeight_{atlas}) \tag{3-7}$$

onde $charWidth_{atlas}$ e $charHeight_{atlas}$ são, respectivamente, a largura e a altura do caractere no espaço da textura, *c* e *l* são os resultados das equações 3-2 e 3-3. As coordenadas de textura que representam a coluna e a linha do caractere são, respectivamente, s_{char} e t_{char}.

Alguns valores como o número de caracteres por linha (equação 3-1) e o tamanho do caractere no espaço da textura (equações 3-4 e 3-5), podem ser calculados pela aplicação uma única vez e guardados em GPU para consulta. Outros cálculos, como os de encontrar a linha e a coluna do caractere, precisam ser realizados por *pixel* após ter sido descoberto a qual rótulo e caractere o *pixel* faz parte.

3.3 Primeira etapa do algoritmo de rendering

Disponíveis em GPU o atlas e as informações textuais dos rótulos, pode ser iniciado o *rendering* da cena 3D, que, conforme exposto na introdução deste capítulo, é um processo feito em duas etapas. Na primeira, antes dos comandos de desenho, a aplicação informa à GPU o identificador do rótulo que deve ser exibido sobre cada objeto (índice na textura de informações textuais) e a quantidade de caracteres desse rótulo.

Em seguida, o *vertex shader* gera as coordenadas de textura por vértice e o *fragment shader* escreve informações em dois *buffers* fora da tela – em um processo chamado *off-screen rendering* ou *render to texture*. Neste trabalho, esses

buffers são texturas anexadas a um *Frame Buffer Object* (FBO). Como pode ser visto na Figura 5, no primeiro *buffer* são escritas as informações de cor da cena, enquanto no segundo são escritas as coordenadas de textura interpoladas por fragmento (campos *s* e *t*), o identificador do rótulo que deve sobrepor o fragmento (campo *id*) e a quantidade de caracteres do rótulo (campo n_{chars}).

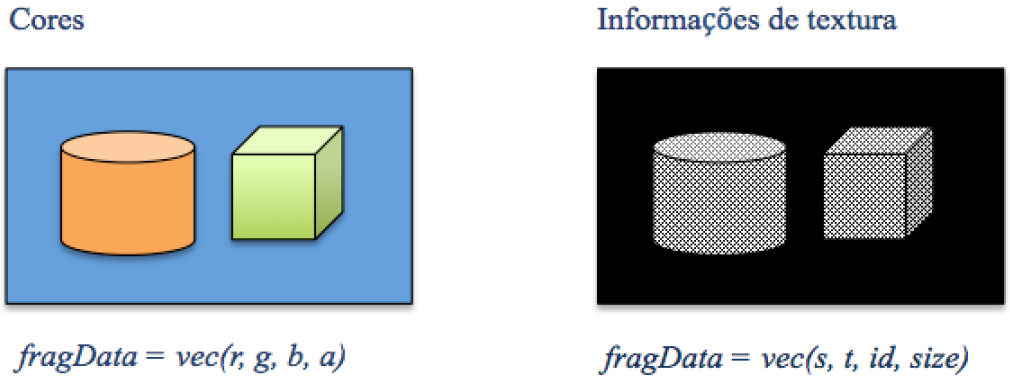


Figura 5: *Buffers* preenchidos na primeira passada de *rendering*.

As texturas do FBO utilizadas na primeira passada precisam ter a mesma resolução da janela da aplicação gráfica e, dessa forma, toda vez que é feito um redimensionamento da janela, seus tamanhos precisam ser alterados. Esses *buffers* guardam informações por fragmento que são usadas na segunda passada para que os rótulos sejam acrescentados à cena.

3.4 Segunda etapa do algoritmo de rendering

Consultando o atlas de caracteres, a textura que guarda as informações textuais dos rótulos e os *buffers* com informações por fragmento de cor e textura (preenchidos na primeira passada), a segunda etapa de *rendering* pode finalmente escrever texto na cena calculando para cada fragmento se ele faz parte de algum rótulo. O posicionamento de rótulos nos objetos depende da definição prévia de coordenadas de textura para as geometrias, na primeira passada – a segunda etapa considera que as coordenadas de textura foram definidas anteriormente com valores de zero a um, ou, de zero a algum valor maior que um quando se deseja repetir o texto ao longo do objeto.

No início desta etapa, a aplicação seleciona o *frame buffer* padrão, usado para mostrar a cena na tela, manda a GPU desenhar um *quad* do tamanho da tela e o

vertex shader gera coordenadas de textura para os vértices desse *quad*. O *fragment shader* consulta todas as informações criadas nas fases anteriores e, para cada fragmento do *quad*, calcula sua cor final, resultando em uma cena com geometrias e rótulos escritos sobre elas.

O *fragment shader* possui automaticamente as coordenadas de textura interpoladas dos fragmentos do *quad* e, utilizando essas coordenadas, amostra o *buffer* de cores e o *buffer* de informações de textura gerados na passada anterior (Figura 6) – *buffers* que possuem o mesmo tamanho da tela. A cor e as coordenadas de textura extraídas dos *buffers* passam a ser a nova cor e as novas coordenadas de textura dos fragmentos do *quad* e, com isso, o *quad* pode ser colorido para exibir a cena final. Quando um fragmento não possui um rótulo, como um fragmento que deve mostrar apenas a cor de fundo, ele recebe na passada anterior um identificador (índice) de rótulo igual a zero e coordenadas de textura iguais a zero – os índices dos rótulos começam a contar a partir de um.

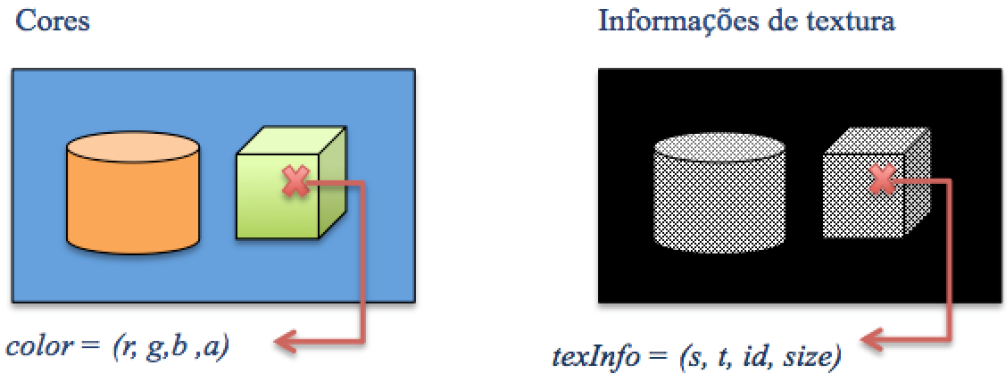


Figura 6: Consulta aos buffers preenchidos na primeira passada.

Depois de definir novas informações de cor e de textura para um fragmento, o *fragment shader* realiza cálculos para acrescentar os rótulos à cena. A seguir, é explicado em etapas esse procedimento para cada fragmento, lembrando que *s* e *t* são as novas coordenadas de textura dos fragmentos do *quad*, *id* é o identificador do rótulo na textura de informações textuais e n_{chars} é a quantidade de caracteres do rótulo.

Passo 1:

Inicialmente é preciso considerar que a geometria está dividida em colunas e que cada coluna possui espaço para exibir um caractere. Este passo serve para descobrir em que coluna o fragmento corrente está contido e, com isso, é possível

saber qual caractere do rótulo deve ser exibido nessa coluna. Por exemplo, a coluna que deve ser encontrada, para o fragmento indicado na Figura 7, é 2, e o caractere 'M' possui índice 2 na palavra "NAME". Essa informação é necessária para o próximo passo, quando será encontrado o código ASCII do caractere que deve ser exibido em cada coluna.

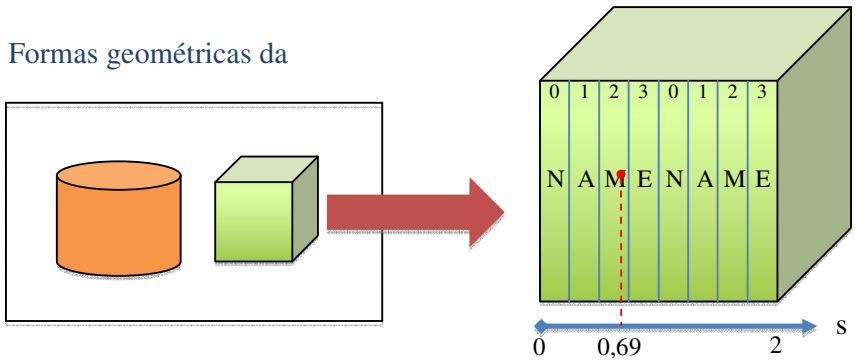


Figura 7: Um fragmento cuja coluna possui índice 2.

A segunda passada de *rendering* considera que o texto deve ser escrito paralelo ao eixo *s* (espaço da textura). Portanto, para encontrar a coluna do fragmento é preciso multiplicar sua coordenada de textura *s* pelo número de caracteres do rótulo. Porém, *s* (assim como t) pode não ser um valor entre 0 e 1 – é possível que as coordenadas de textura do objeto tenham sido definidas, na passada anterior, com o intuito de repetir o rótulo várias vezes sobre a geometria, como na Figura 7– e, nesses casos é preciso normalizar *s* para estar entre 0 e 1. As equações 3-8 e 3-9 mostram como encontrar a coluna (ou índice do caractere dentro de seu rótulo):

$$s_{normalized} = s - floor(s) \quad (3\text{-}8)$$

$$t_{normalized} = t - floor(t) \quad (3\text{-}9)$$

$$i_{char} = floor(s_{normalized} * n_{chars}) \quad (3\text{-}10)$$

onde $s_{normalized}$ e $t_{normalized}$ (que será utilizado mais a frente na equação 3-19), são as coordenadas de textura s e t normalizadas pra um intervalo de 0 a 1 e i_{char} é o índice do caractere dentro de seu rótulo.

Passo 2:

Encontrado o índice do caractere, é preciso descobrir o código ASCII desse caractere. Como foi explicado na seção 3.2, existe uma textura que armazena os códigos ASCII dos caracteres de todos os rótulos da cena e, portanto, é preciso

encontrar dentro dessa textura o *texel* que contém o código do caractere procurado. Para encontrar esse *texel* é preciso utilizar o índice do caractere encontrado no passo anterior e o identificador do rótulo do fragmento corrente.

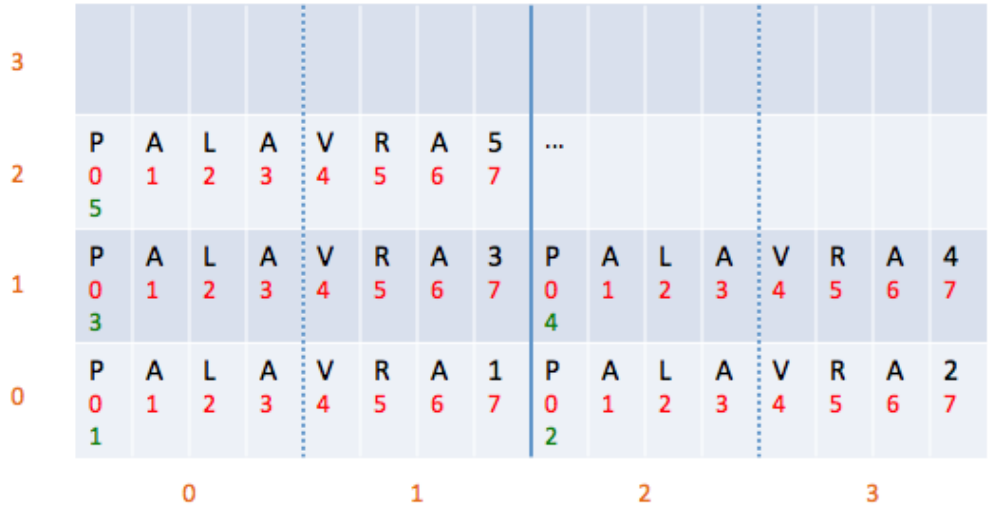


Figura 8: Armazenamento dos códigos ASCII dos caracteres em uma textura 2D.

A Figura 8 mostra um exemplo de como os caracteres são armazenados na textura de informações textuais que, nesse caso, gasta dois *texels* para armazenar cada rótulo – como um *texel* possui quatro componentes de cor, em dois *texels* é possível armazenar oito caracteres. Os valores em verde são os identificadores dos rótulos e em vermelho são os índices de cada caractere dentro de seus rótulos (índice encontrado no passo 1). Os valores em laranja mostram que a textura possui quatro *texels* na horizontal e quatro na vertical e ao mesmo tempo representam um índice para cada *texel* ao longo de *s* e *t*. O primeiro passo nessa etapa é encontrar esses índices (valores laranja na Figura 8), como mostram as equações 3-11 e 3-12:

$$is = \left((id - 1) \bmod namesPerRow\right) * texelsPerName + \text{floor}(i_{char}/4) \qquad (3\text{-}11)$$

$$it = \text{floor}\left(\frac{id-1}{namesPerRow}\right) \qquad (3\text{-}12)$$

onde is e it são os índices (e não coordenadas de textura) do *texel* nas dimensões *s* e *t* respectivamente, $namesPerRow$ a quantidade de nomes que cabem em uma linha da textura e $texelsPerName$ o número de *texels* que é usado para guardar cada nome (valores constantes calculados em CPU dependendo de como a textura foi organizada). O identificador do rótulo, em verde na Figura 8, é representado por id (subtraído de um pois os *ids* começam a partir de um e não de zero), e i_{char} é o índice do caractere, encontrado no passo anterior.

Por exemplo, utilizando uma textura como a da Figura 8, queremos encontrar o caractere localizado na posição 7 (valor em vermelho na Figura 8), do rótulo identificado como o de número 4 (valor em verde). Nesse caso, $namesPerRow$ é 2, $texelsPerName$ é também 2, o rótulo do fragmento corrente possui *id* igual a 4 e o valor encontrado no Passo 1 para i_{char} foi 7, com as equações 3-11 e 3-12 seria encontrado 3 para i_s e 1 para i_t. Utilizando os índices é preciso encontrar as coordenadas de textura do *texel* para que a textura possa ser amostrada:

$$s_{temp} = \frac{i_s+0.5}{w} \tag{3-13}$$

$$t_{temp} = \frac{i_t+0.5}{h} \tag{3-14}$$

onde w e h são respectivamente a largura e altura da textura 2D, em *texels*. O objetivo das equações 3-13 e 3-14 é encontrar as coordenadas de textura cujos valores representam o centro do *texel* procurado, para que com um filtro de "vizinho mais próximo" como o GL_NEAREST[1], ele possa ser recuperado da textura 2D[2]. A Figura 9 mostra uma textura organizada da mesma forma que na Figura 8, quando w é 4, e mostra o valor de s_{temp} que seria encontrado com a equação 3-13 para i_s igual a 3.

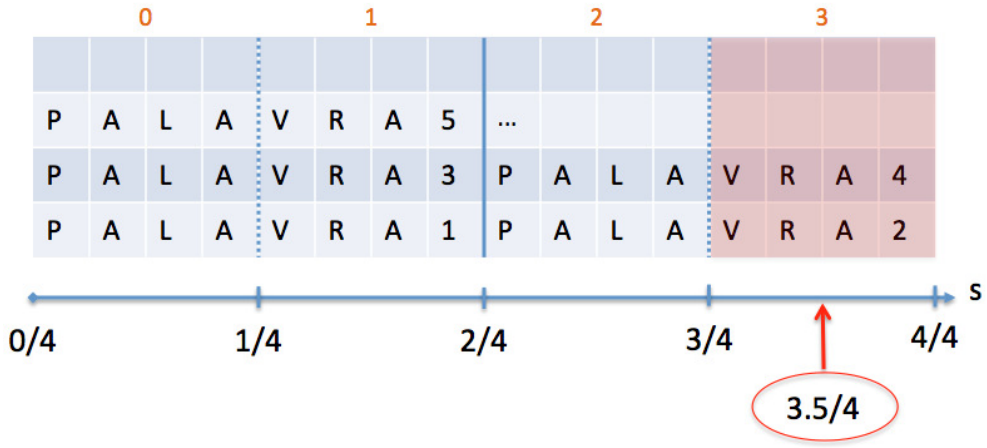


Figura 9: Cálculo de coordenada de textura *s* com a equação 3-13.

Com s_{temp} e t_{temp} é possível amostrar a textura e obter o ***texel***, que possui 4 componentes RGBA e que contém o caractere procurado em uma dessas

[1] Filtro que obtém o elemento de textura mais próximo da coordenada de textura especificada.
[2] Foram constatadas outras formas amostrar um *texel* a partir seus índices com versões mais novas da API gráfica, porém, com GLSL 1.2, não foram encontradas funções já prontas tal finalidade.

componentes. Para, finalmente, achar o caractere é possível realizar o cálculo mostrado na equação 3-15:

$$char = \boldsymbol{texel}[i_{char}\ mod\ 4] \tag{3-15}$$

sendo $i_{char}\ mod\ 4$ o índice do caractere que pode ir de 0 a 3 devido as componentes RGBA e *char*, o código ASCII do caractere procurado. No exemplo mostrado nas figuras anteriores, onde o *id* do rótulo é 4 e $i_{char} = 7$, encontraremos o código ASCII do caractere '4'.

Passo 3:

Conhecendo o código do caractere é preciso encontrar sua representação gráfica no atlas de textura que contém todos os caracteres (seção 3.3). Utilizando as equações 3-1 a 3-7 é possível encontrar, em coordenadas de textura, a linha e coluna iniciais da imagem do caractere procurado. Porém, isso não é suficiente, pois como pode ser visto na Figura 10, o fragmento pode ser representado por qualquer posição dentro da imagem do atlas.

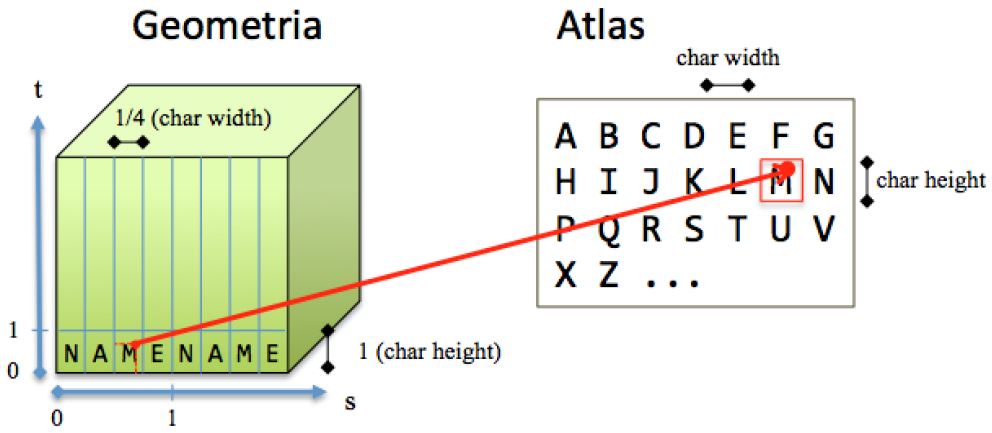


Figura 10: À esquerda um fragmento e seu *offset* em relação ao início de uma coluna e de uma linha e à direita o *texel* associado.

É preciso, portanto, calcular o *offset* do fragmento na geometria, em coordenadas de textura, como mostra a Figura 10 e as equações 3-16 a 3-18:

$$charWidth_{geom} = \frac{1}{n_{chars}} \tag{3-16}$$

$$sOffset_{geom} = s_{normalized} - (i_{char} * charWidth_{geom}) \tag{3-17}$$

$$tOffset_{geom} = t_{normalized} \tag{3-18}$$

onde $charWidth_{geom}$ é a dimensão horizontal ocupada por cada caractere no espaço de textura da geometria. Os valores de $s_{normalized}$, $t_{normalized}$ e i_{char} foram calculados previamente com as equações 3-8 a 3-10 e n_{chars} representa a quantidade de caracteres do rótulo de qual o fragmento faz parte.

Com o *offset* do fragmento na geometria é preciso calcular o *offset* relacionado no atlas:

$$r = \frac{charWidth_{atlas}}{charWidth_{geom}} \tag{3-19}$$

$$sOffset_{atlas} = sOffset_{geom} * r \tag{3-20}$$

$$tOffset_{atlas} = tOffset_{geom} * charHeight_{atlas} \tag{3-21}$$

onde $charWidth_{atlas}$ e $charHeight_{atlas}$ são os resultados das equações 3-4 e 3-5 respectivamente e representam a largura e a altura de cada caractere do atlas no espaço de textura. A variável $charWidth_{geom}$ é o resultado da equação 3-16 e r é a razão entre a dimensão horizontal dos caracteres, no *atlas* e na geometria. A razão entre as dimensões verticais é $charHeight_{atlas}$ pois a altura do caractere na geometria é sempre 1 como mostra a Figura 10.

O *offset*, encontrado com as equações 3-19 a 3-21, somado à posição inicial do caractere no atlas, resulta na coordenada de textura para amostrar o atlas na posição correta, como mostram as equações 3-22 e 3-23.

$$s_{final} = s_{char} + sOffset_{atlas} \tag{3-22}$$

$$t_{final} = t_{char} - tOffset_{atlas} \tag{3-23}$$

onde s_{char} e t_{char} representam a posição inicial do caractere e precisam ser calculados com as equações 3-1 a 3-7.

Passo 4:

Com a cor obtida do atlas e a cor do objeto (que está guardada no *buffer* de cores) é possível calcular a cor final do fragmento tratando o rótulo como um decalque.

Com a técnica apresentada, é possível notar que independente do número de objetos carregados pela aplicação, somente para os *pixels* da tela é preciso realizar cálculos relativos aos rótulos, o que justifica o uso da técnica para modelos massivos. Na segunda passada são realizadas algumas equações com operações simples e alguns acessos a texturas e, em relação a esses acessos, o algoritmo se

beneficia dos recursos de hardware para texturização, pois os fragmentos próximos uns dos outros, que são processados em paralelo, na maioria dos casos, precisam acessar partes próximas das texturas. A técnica de exibição de rótulos não demonstrou perda significativa de desempenho para modelos massivos e os resultados serão exibidos em detalhes no Capítulo 7. O próximo capítulo discute como posicionar os rótulos da melhor forma em objetos de modelos CAD e, com imagens, apresenta os primeiros resultados visuais.

4 Posicionamento dos Rótulos

Até então foi explicada a técnica de *rendering* de duas passadas para exibir objetos com rótulos. No entanto, essa técnica considera que as coordenadas de textura devem ser definidas para as geometrias durante a primeira passada. Neste capítulo será explicado o procedimento para posicionar rótulos sobre algumas superfícies de modelos CAD, com o objetivo de fornecer para o usuário uma boa forma de visualizar o texto, independente do ângulo de visão utilizado. Este capítulo tem também o objetivo de mostrar que, com a definição correta de coordenadas de textura, o algoritmo de exibição dos rótulos funciona para objetos de diferentes formatos e tamanhos.

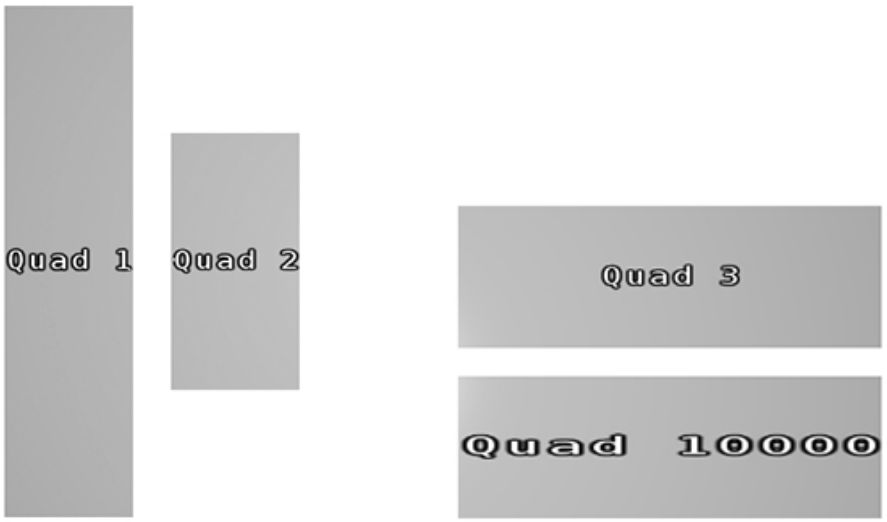
Primeiramente, a técnica foi testada em geometrias simples, como *quads* – pois foi fácil estender a ideia para outras geometrias –, cujas coordenadas de textura foram geradas no programa de vértices da primeira passada. Tais *quads* possuem arestas unitárias e seus centros localizados na origem, então a determinação das coordenadas de textura pode ser realizada apenas por meio da adição de 0.5 à posição de cada vértice nos eixos x e y. O resultado dessa operação é mostrado na Figura 11, onde é possível perceber que praticamente todo o espaço da geometria é ocupado com letras, o que, dependendo do tamanho horizontal ou vertical do objeto, pode apresentar um aspecto esticado do texto. Em uma cena com grande quantidade de objetos contendo rótulos, o excesso de espaço ocupado pelas informações textuais pode prejudicar a visualização dos modelos.

Figura 11: Quads com rótulos ocupando uma área grande.

Uma forma do rótulo ocupar menos espaço é ajustar as coordenadas de textura dos vértices para que apenas determinada área do objeto possua coordenadas que variem de 0 a 1. Com isso, os fragmentos cujas coordenadas de textura estão fora desse intervalo são identificados no *fragment shader* da primeira passada e associados a um identificador de rótulo igual a zero – o que faz a segunda passada calcular a cor final desses fragmentos sem o mapeamento dos rótulos.

O cálculo da área para exibir texto no centro do objeto deve levar em conta que *quads* possuem tamanhos diferentes e rótulos com quantidade de caracteres variável. Neste trabalho, a centralização vertical do texto é realizada automaticamente por meio da aplicação de uma escala nas coordenadas de textura no eixo *t* dependendo do tamanho vertical do *quad* seguida do deslocamento da região de intervalo 0 a 1 para a área central da face. A centralização horizontal, por sua vez, também deve levar em consideração a quantidade de caracteres do rótulo, dado que rótulos com um grande número de caracteres ocupam um maior espaço horizontal.

Como exemplo, se é encontrado para um *quad* que a escala a ser aplicada às coordenadas de textura *t* dos vértices é 3, então, as coordenadas que são 0 e 1 (para os vértices inferiores e superiores respectivamente), passam a ser de 0 e 3, e para centralizar o rótulo, elas devem ser alteradas para irem de -1 a 2. A Figura 12 (a) mostra a centralização vertical em dois objetos de tamanhos diferentes, enquanto a Figura 12 (b) mostra a acomodação de dois rótulos contendo quantidades de caracteres diferentes em dois *quads* de mesmas dimensões.



(a) (b)

Figura 12: Posicionamento vertical (a) e horizontal (b) de rótulos.

No entanto, se a escala a ser aplicada às coordenadas de textura é calculada separadamente para cada dimensão, quanto maior a geometria, maior é a desproporção entre o tamanho do objeto e o tamanho do rótulo – se um *quad* possui dimensões 1 x 1, ou 100 x 100 a área ocupada pelo rótulo é a mesma, mas poderia ser maior no segundo caso. Logo, a razão entre as dimensões do objeto deve ser considerada para uma melhor exibição dos rótulos. Se a razão entre largura e altura é 1, então uma escala de 1 deve ser aplicada as coordenadas em *s e t*. Porém, se a largura é 100 vezes maior que a altura, a escala que deve ser aplicada em *s* é em torno de 100 vezes maior do que a que deve ser aplicada em *t.*

Para exemplificar melhor esse processo, considere um *quad* que possui largura 5 e altura 2, que o rótulo a ser aplicado nesse objeto possui 4 caracteres e que foi notado que em um *quad* unitário o número ideal de caracteres, para que um rótulo não fique comprimido ou esticado horizontalmente, é 6, como mostra a Figura 13 (a). Com isso, se no *quad* unitário cabem 6 caracteres, em um *quad* de largura 5 cabem 30 caracteres. Como o rótulo do objeto possui 4 caracteres, basta dividir 30 por 4 o que resulta em 7 vezes que o rótulo cabe horizontalmente no objeto, como pode ser visto na Figura 13 (b) – esse valor seria a escala aplicada as coordenadas de textura *s* dos vértices do *quad*. Porém, queremos considerar a razão entre as dimensões do objeto, e como a largura é 5 e a altura é 2, então a razão entre as dimensões é aproximadamente 2. Portanto, para encontrar o número de caracteres ideal que cabem no objeto, é preciso multiplicar a razão encontrada por 6

o que resulta em 12 caracteres, resultado mostrado na Figura 13 (c). Como o rótulo possui 4 caracteres, a escala a ser aplicada em *s* é 3 e em *t* é 1 pois a altura é menor do que a largura.

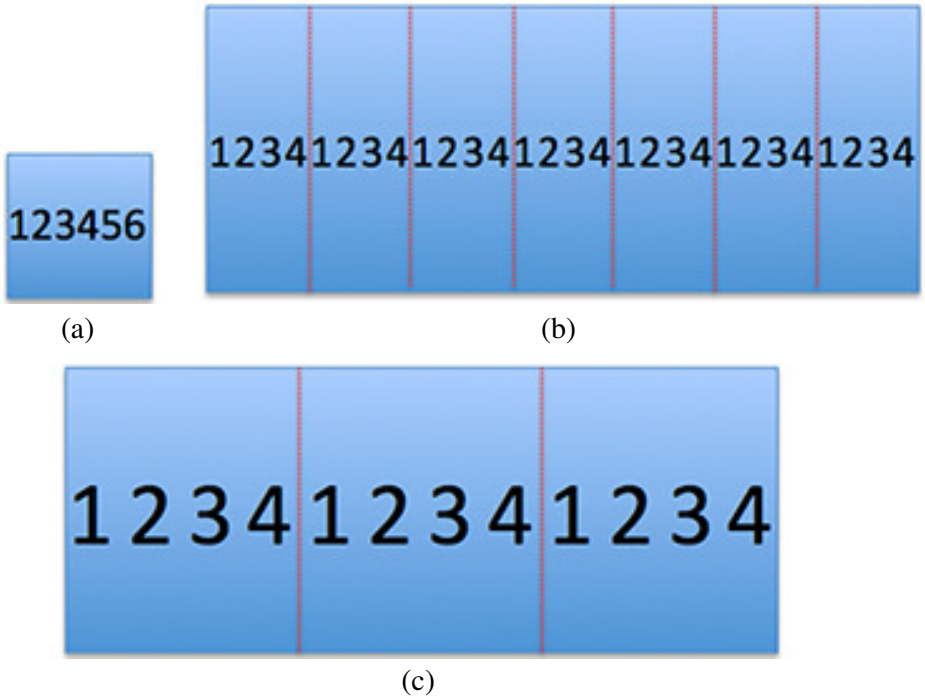


Figura 13: Quantidade de caracteres ideal para um *quad* unitário (a); *quad* de 5 x 2 com rótulo de 4 caracteres e cálculo de repetições baseado em suas dimensões separadamente (b); *quad* de 5 x 2 com rótulo de 4 caracteres e cálculo de repetições baseado em uma razão entre suas dimensões.

Ao levar em conta a razão entre as dimensões do objeto, o rótulo ocupa uma maior área quanto maior for o tamanho do objeto e, com isso, é sempre possível enxergar o texto independente de quão grande é o objeto. Além disso, algo a mais a ser considerado é que se um *quad* possui, por exemplo, a largura muito maior do que a altura, dependendo da posição que a câmera aponte para o objeto o rótulo não será visualizado caso ele esteja centralizado – há uma grande área do objeto sem texto – e, portanto, exibir o rótulo várias vezes ao longo do objeto é uma boa opção. Para repetir o texto, o cálculo é praticamente o mesmo explicado nos parágrafos anteriores: se for encontrado que um bom número de repetições ao longo de *s* é 15, então as coordenadas de textura precisam variar de 0 a 15 – pois essa é a forma esperada na segunda passada para repetir texto – e não mais de -7 a 7 para exibir o texto no centro do objeto.

Na repetição de texto ao longo do eixo *s*, para espaçar melhor os rótulos, foram utilizados caracteres de espaço em branco antes e depois de cada palavra. Para espaçar os rótulos verticalmente não é possível, neste trabalho, adicionar caracteres para separar o texto verticalmente e, com isso, os rótulos ficariam colados uns nos outros. Uma forma de espaçar o texto, sem ter que refinar a malha do *quad* e definir coordenadas de textura para novos vértices, é calcular normalmente o número de repetições ao longo do eixo *t* e no *fragment shader* da primeira passada, determinar os fragmentos que terão rótulos mapeados a eles ou não (definindo um identificador de rótulo igual a zero no último caso). Como exemplo, para intercalar as áreas que exibem rótulos, o *fragment shader* pode configurar um identificador de rótulo igual a zero para os fragmentos que tenham coordenadas de textura *t* cuja parte inteira seja par.

Por fim, no caso dos *quads*, foi observado que para uma melhor visualização dos rótulos, que não polua excessivamente a tela com informações textuais e, considerando modelos CAD que possuem um grande número de objetos, decidiu-se que seria melhor repetir o texto ao longo de *s* mas não de *t*. O resultado final pode ser visto na Figura 14.

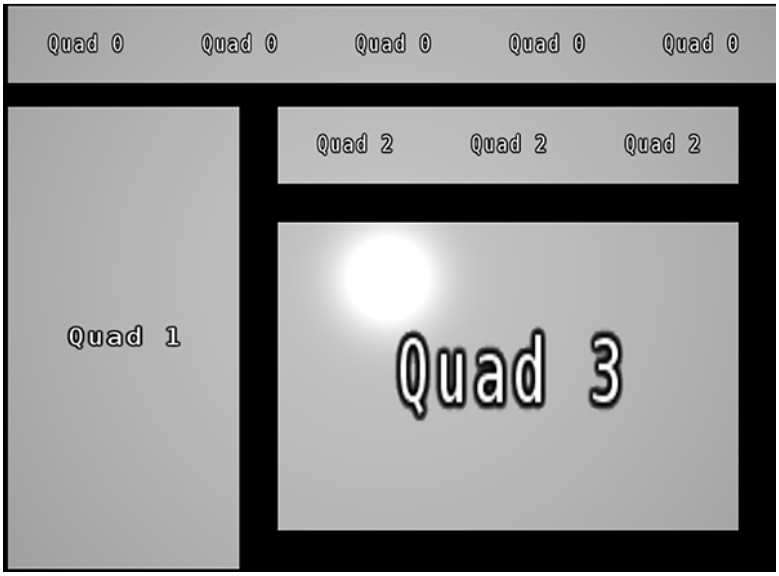


Figura 14: *Quads* de tamanhos variados e posicionamento automático de rótulos.

Outras primitivas que tiveram suas coordenadas de textura ajustadas para exibir rótulos foram caixas (cubos), cilindros, esferas e *dishes* (cuja forma geométrica será explicada mais a frente). Todas essas primitivas tiveram suas coordenadas de textura geradas no *vertex shader* da primeira passada. No caso da

caixa, os cálculos são praticamente os mesmos do *quad*, porém, é preciso definir coordenadas de textura diferentes para cada uma de suas faces.

Para cilindros, as coordenadas de textura dos vértices foram organizadas de forma que o eixo *s* do espaço de textura fosse paralelo ao eixo y do espaço do objeto – eixo da altura do cilindro – e o eixo *t* fosse paralelo ao eixo x. Com isso, o texto se repete várias vezes ao longo de *s* dependendo da altura do objeto e, em torno do cilindro, foi definido que o texto deve se repetir apenas quatro vezes. O cálculo para definir as escalas que precisam ser aplicadas às coordenadas de textura, dependendo do tamanho do cilindro e da quantidade de caracteres dos rótulos, foi todo baseado nos cálculos apresentados para os *quads.*

No caso de esferas, o texto é exibido apenas uma vez ao longo de *t*, no centro do objeto, e pode se repetir várias vezes ao longo de *s* dependendo do diâmetro da esfera.

As primitivas chamadas de *dishes* contidas em modelos CAD possuem o formato de meio elipsoide e com informações de raio e altura é possível parametrizar sua superfície. A Figura 15 mostra cubos, cilindros, esferas e *dishes* com rótulos.

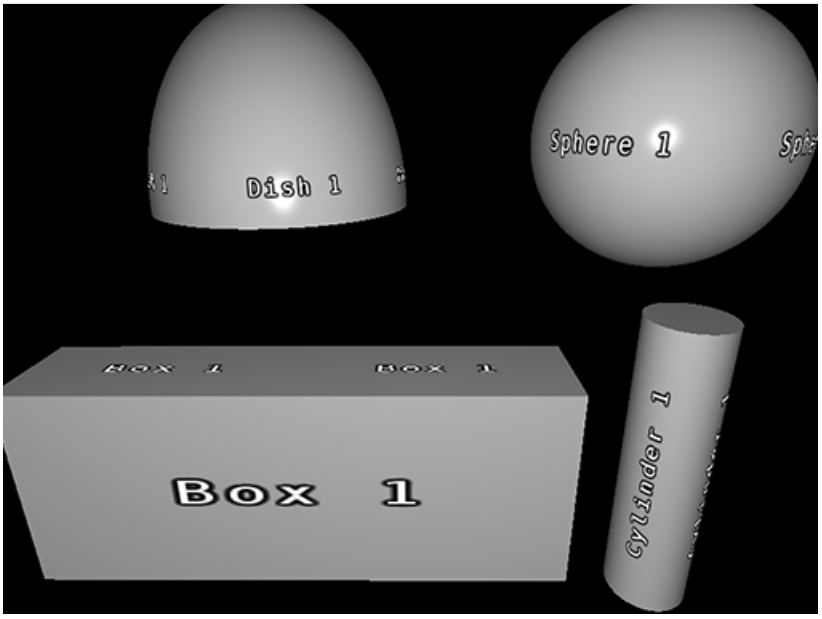


Figura 15: Diferentes primitivas com rótulos.

As coordenadas de textura dos objetos podem ser determinadas de várias formas e neste capítulo foi mostrado o modo que tal procedimento foi realizado neste trabalho, para algumas primitivas encontradas em modelos CAD. Além disso,

foi mostrado que com coordenadas de textura definidas corretamente, que estejam de acordo com o esperado pela segunda passada, o algoritmo de *rendering* dos rótulos funciona corretamente para objetos de diferentes tamanhos e formatos.

O posicionamento de rótulos em objetos como cones e pirâmides pode ser um pouco mais complexo devido às diferentes dimensões entre as bases desses objetos. Com isso, as coordenadas de textura devem ser calculadas de forma a compensar essa diferença, para que o texto não seja esticado proporcionalmente à diferença de tamanho entre as bases. Apesar deste cálculo não ter sido feito até o presente momento, com a técnica explicada neste trabalho é possível posicionar rótulos em mais primitivas do que nos trabalhos relacionados – que permitem a exibição de rótulos em caixas e cilindros apenas. Porém, caixas e cilindros são, de fato, os objetos em maior número nos modelos CAD utilizados.

5 Rótulos em Modelos CAD Reais

A exibição de texto com a técnica deste trabalho pode ser utilizada para diferentes propósitos, mas a motivação de criá-la foi a exibição de rótulos em objetos de modelos massivos e, mais especificamente, de modelos CAD. Este capítulo mostra, portanto, a aplicação da técnica proposta nesse tipo de modelo, explicando brevemente a *engine* gráfica que foi utilizada para o *rendering* de modelos CAD e apresentando os rótulos aplicados a modelos CAD reais.

Os modelos utilizados baseiam-se em arquivos que contém informações de cada objeto a ser desenhado. Dentre esses objetos existem malhas de triângulos descritas pela posição de seus vértices e normais, mas também existem primitivas descritas apenas por parâmetros como raio, altura, dimensões e matrizes de translação, rotação e escala. A *engine* precisa saber interpretá-los para criar as formas geométricas e, neste trabalho, apenas as malhas não são tratadas pela *engine*.

Os objetos tratados possuem uma superfície curva como esferas, cilindros, cones, "dishes" e toros ou uma superfície "flat" como caixas e pirâmides. Para o *rendering* dos objetos de superfície curva foi utilizada uma mesma malha base de triângulos que é moldada no *vertex shader* para dar forma às diferentes superfícies geométricas. No caso de caixas e pirâmides também foi utilizada uma malha base que é moldada no programa de vértices. Tal técnica é conhecida como instanciação de primitivas e, com ela, todas as primitivas gastam a mesma quantidade de memória de vídeo independente da quantidade de objetos (malhas seriam exceções, mas não as estamos tratando).

Os parâmetros dos objetos são passados para a GPU através de coordenadas de textura, pois essa foi a forma mais rápida encontrada, considerando as limitações de OpenGL 2.1 e GLSL 1.2. Além dessa técnica de instanciação, nenhuma outra otimização foi utilizada na *engine*, mas com ela é possível o *rendering* de um grande número de objetos a taxas interativas, como será mostrado no Capítulo 7. Além de exibir modelos CAD, é possível montar qualquer cena com os objetos citados no parágrafo anterior.

A Figura 16 e a Figura 17 mostram rótulos aplicados a objetos de um modelo CAD de refinaria de petróleo. Como pode ser visto, é possível exibir rótulos em uma grande quantidade de objetos de modelos CAD e, também, que a maioria das geometrias são caixas ou cilindros. Algo que também pode ser notado é que a diferença de tamanho dos objetos na tela pode ser muito grande, pois alguns deles possuem um tamanho pequeno ou estão muito distantes da câmera e, nesses casos, fica difícil enxergar o texto que está escrito. Logo, será discutido no próximo capítulo como melhorar o efeito de *aliasing* existente.

Figura 16: Rótulos em um modelo CAD de refinaria de petróleo.

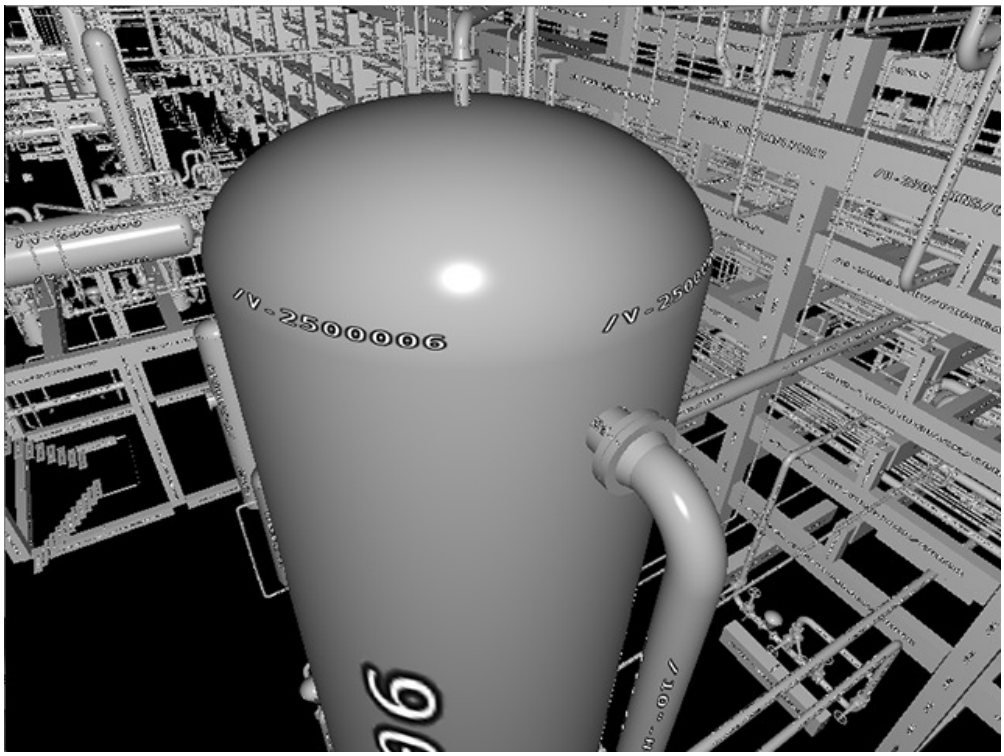


Figura 17: Mais um *screenshot* de rótulos em modelo CAD.

6 Aliasing e Melhoria da Qualidade Visual

Em uma cena 3D como as que são utilizadas nesse trabalho existem objetos de todos os tamanhos sendo exibidos na tela ao mesmo tempo. Com isso, alguns deles possuem imagens (de rótulos) maiores do que outros. O problema disso é que com um único atlas de caracteres, de mesmo tamanho, é possível imaginar que a imagem dos caracteres precisará ser ampliada ou reduzida dependendo do tamanho que cada objeto possui na tela.

O problema de ampliação de textura ocorre quando a área de um *texel* passa a corresponder a vários *pixels* da tela. Por exemplo, quando um quadrado com uma textura é projetado na tela e possui o mesmo tamanho da textura, a textura aparece praticamente igual à da imagem original. Porém, quando o quadrado é ampliado na tela e passa a possuir um tamanho maior do que a textura, um conjunto de *pixels* relativos ao quadrado que antes utilizavam cada um, um *texel*, passam a utilizar um mesmo *texel* da textura.

O efeito visual causado pela magnificação depende do tipo de amostragem e filtragem utilizados. Os filtros mais comuns para esses casos são o de "vizinho mais próximo", que seleciona o *texel* mais próximo da coordenada de textura especificada, e a interpolação bi-linear, que acha os quatro *texels* vizinhos e realiza uma interpolação linear em duas dimensões para achar um valor combinado para o *pixel*. Enquanto o primeiro pode gerar um resultado visual com bastante *aliasing*, o segundo reduz o *aliasing* gerando um melhor resultado visual, mas ainda com algum possível efeito de borrado [16].

O problema de redução de textura ocorre quando a área de um fragmento corresponde à área de vários *texels*. Quando se dá a redução da textura, também podem ser utilizados os filtros de "vizinho mais próximo" e a interpolação bi-linear. Porém, nesse caso, ambos os filtros causam *aliasing* e a interpolação bi-linear é apenas um pouco melhor do que o filtro de "vizinho mais próximo", pois a interpolação combina quatro *texels* ao invés de utilizar apenas um, e quando um fragmento é influenciado por mais de quatro *texels,* o filtro logo falha. O método mais comum de *antialiasing* de texturas é chamado de *mipmapping,* quando é

utilizada uma coleção de texturas de resoluções diferentes variando desde o tamanho da textura principal até uma textura de 1x1 *texels*. O fator de redução no tamanho de uma textura para outra é $\log_2(n)$ e, dessa forma, é como se todas as texturas juntas formassem uma pirâmide, daí o nome pirâmide de *mipmap*

Para cada grupo de fragmentos determinado pela API gráfica é escolhido o nível da pirâmide que cause menos redução de textura. Quando esse método é utilizado o *aliasing* é bastante reduzido, mas, um dos efeitos indesejáveis pode ser um efeito de borrado excessivo [16].

Como já mencionado, no caso da exibição de rótulos em uma cena com objetos tridimensionais que variam de tamanho e formato, os dois problemas (ampliação e redução) podem ocorrer – existem objetos que vão aparecer muito grandes na tela e outros muito pequenos, e ambos precisam utilizar a textura que contém a imagem dos caracteres

O uso de *mipmapping* pode ser realizado junto à técnica apresentada neste trabalho utilizando-se funções do OpenGL com cálculos extras para que o nível correto de *mipmapping* seja escolhido. O OpenGL fornece as funções *dFdx* e *dFdy*, que podem ser utilizadas no *fragment shader* e que recebem, nesse caso, como parâmetro a coordenada de textura de cada fragmento. O resultado dessas funções é razão entre a taxa de variação de coordenadas de textura da imagem a ser aplicada e a taxa de variação de coordenadas de textura do objeto que irá receber a imagem. A variação de coordenadas de textura é medida no intervalo [0,0] a [1,1]. Normalmente, para amostrar o nível correto da pirâmide, bastaria passar os resultados de *dFdx* e *dFdy* como parâmetros para a função do OpenGL *textureGrad* o que retornaria o dado final desejado.

Porém, não queremos aplicar a um objeto o atlas de caracteres inteiro, cujo espaço de textura vai de [0,0] a [1,1], mas sim uma quantidade variável de letras. Dessa forma, a razão entre as taxas de variação de coordenadas de textura precisa ser corrigida. Por exemplo, se uma textura possui 100 *texels* horizontalmente e seu espaço de textura em *s* varia de 0 a 1, a variação de coordenada de textura de um *texel* para seu vizinho é de 0.01. Porém, se queremos aplicar apenas metade dessa textura a um objeto precisamos imaginar que a textura tem metade do tamanho e, então, a taxa de variação de coordenadas de textura passa a ser 1/50 ou 0.02, o que irá interferir na razão entre a variação de coordenadas de textura da imagem e do objeto.

O resultado de *dFdx*, *fx*, é um vetor bidimensional de componentes *s* e *t*, e o resultado de *dFdy, fy*, também é um vetor bidimensional de componentes *s* e *t*. Cada uma dessas componentes precisou sofrer correções como mostram os seguintes cálculos:

$$fx = dFdx(texCoord) \tag{3-24}$$

$$fy = dFdy(texCoord) \tag{3-25}$$

$$fx'.s = fx.\mathrm{s} * \frac{n_{chars}}{n_{col}} \tag{3-26}$$

$$fx'.t = fx.\mathrm{t} * \frac{1}{n_{col}} \tag{3-27}$$

$$fy'.s = fy.\mathrm{s} * \frac{n_{chars}}{n_{col}} \tag{3-28}$$

$$fy'.t = fy.\mathrm{t} * \frac{1}{n_{col}} \tag{3-29}$$

Utilizando-se os vetores resultantes *fx'* e *fy'* como parâmetros da função *textureGrad* obtém-se a cor final amostrada do nível correto de *mipmapping*. Para essa escolha manual do *mipmap* foi preciso utilizar GLSL 1.3, já que isso não é possível na versão 1.2 que foi utilizada para todo o resto do código. O uso das funções *dFdx* e *dFdy* precisa ser realizado na primeira passada de *rendering* quando o OpenGL está desenhando os objetos da cena e pode, portanto, obter a taxa de variação de coordenadas de textura no objeto. Como o cálculo da cor final é feito na segunda passada, foi necessário o uso de um terceiro *buffer* (textura) para transportar essas informações entre as duas passadas. Como a implementação relativa a *mipmapping* só foi concluída no final do trabalho, somente então tivemos um forte argumento para realizar tudo em uma passada de *rendering*, o que será deixado como trabalho futuro.

As texturas para os níveis da pirâmide foram geradas automaticamente e manualmente (sem aplicação de escala nas imagens). Gerando os níveis de *mipmapping* automaticamente, foi possível notar alguns artefatos, devido possivelmente a que em texturas de níveis altos as letras invadiam o espaço umas das outras. Gerando as texturas manualmente e corrigindo esses defeitos das imagens, o resultado ficou melhor e sem artefatos. Porém, foi possível notar que a partir de 64 x 64 *texels*, os caracteres da imagem já eram pouco legíveis – como o atlas continha 16 x 16 caracteres, sobrava apenas quatro 4 x 4 *texels* para cada caractere, o que começou a se tornar inviável para representar um caractere. Para

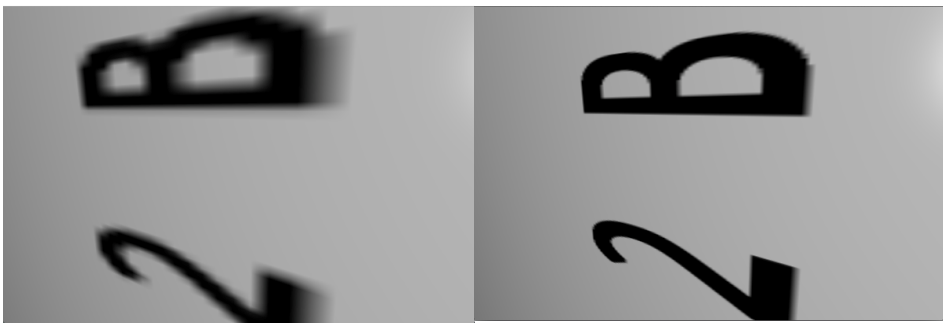
níveis acima desse o problema piorava, ficando cada vez mais difícil de representar bem as letras.

Com os atlas gerados automaticamente pelo OpenGL foi possível utilizar um filtro tri-linear para a redução de textura, porém quando os mapas foram gerados manualmente foi utilizado *mipmapping* com um filtro bi-linear (que não utiliza mais de um nível da pirâmide). Como dito antes, o resultado foi melhor com mapas gerados manualmente.

O atlas apresentado no Capítulo 3 utilizava letras brancas com bordas pretas, o que pode ser bom para destacar os rótulos em superfícies de diferentes cores. Mas, pode não ser desejável o uso de letras com bordas em todos os casos, até porque, foi constatado que elas apresentam mais *aliasing*, principalmente quando a câmera está em movimento. Portanto, também foi utilizado um atlas com letras sólidas e pretas sendo simples modificar essa cor no *fragment shader* da segunda passada quando a textura é amostrada.

A Figura 18 mostra uma maior definição do texto quando ele é ampliado com o uso de uma textura de 2048 x 2048 para a base da pirâmide de *mipmapping* em contraste com a textura que estava sendo usada sem *mipmapping* de 512 x 512.

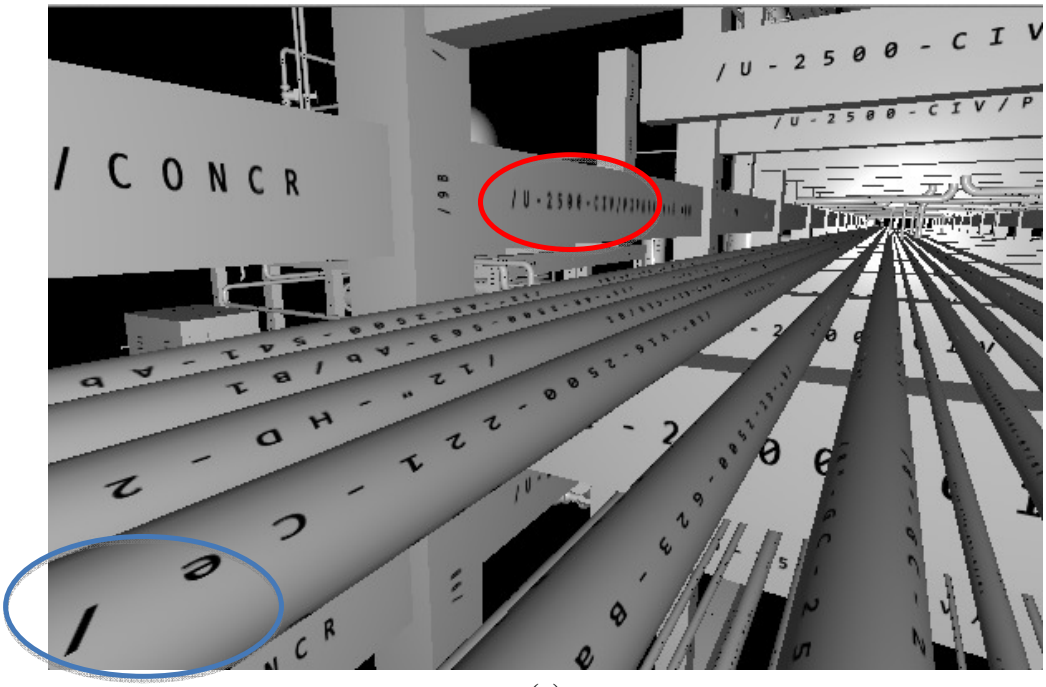
A Figura 19 compara a qualidade do texto com e sem o uso de *mipmapping* em uma cena com um modelo CAD real, e é possível perceber que com *mipmapping* os objetos mais próximos exibem texto com uma maior definição e os mais distantes apresentam um resultado melhor e mais suave. Na Figura 20 é possível ver modelos reais com rótulos coloridos e, para a criação dessas cenas, foi utilizado um mesmo atlas com letras de cor preta sendo que essa cor foi modificada em GPU.



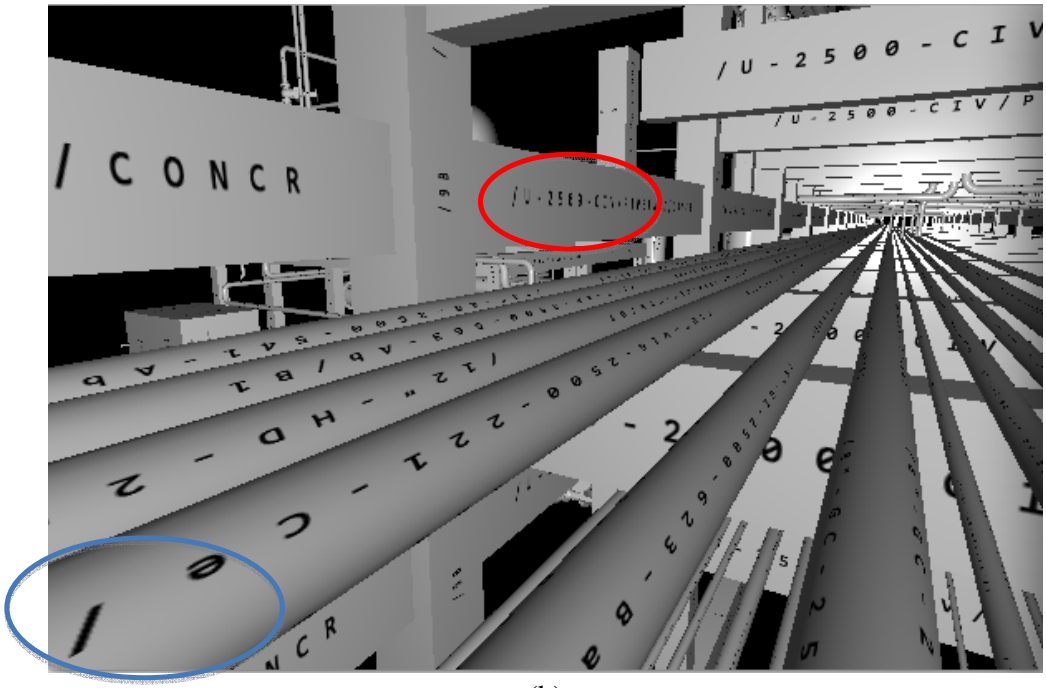
(a) (b)

Figura 18: Texto magnificado e uso de um atlas de caracteres de 512 x 512, sem *mipmapping* (a); e texto magnificado com *mipmapping* cuja base da pirâmide é uma atlas de 2048 x 2048 *texels* (b).

O uso de *mipmapping* é vantajoso, pois com ele é possível utilizar texturas de mais alta resolução e, com isso, melhorar o problema de magnificação de textura sem piorar o problema de redução. Mas, como os níveis mais altos da pirâmide não representam bem as imagens dos caracteres, da forma que o atlas foi organizado, para objetos muito pequenos na tela o resultado do texto não fica muito melhor. Portanto, para melhorar a qualidade visual dos rótulos, em um trabalho futuro pode ser feita uma investigação mais extensa sobre esse assunto. Com *mipmapping*, o gasto de memória aumenta devido ao uso de texturas de maior resolução.



(a)



(b)

Figura 19: Comparação de cena exibindo modelo CAD, com (a) e sem (b) o uso de *mipmapping*. O círculo em vermelho destaca *pixels* afastados da tela e o círculo em azul, *pixels* bem próximos.

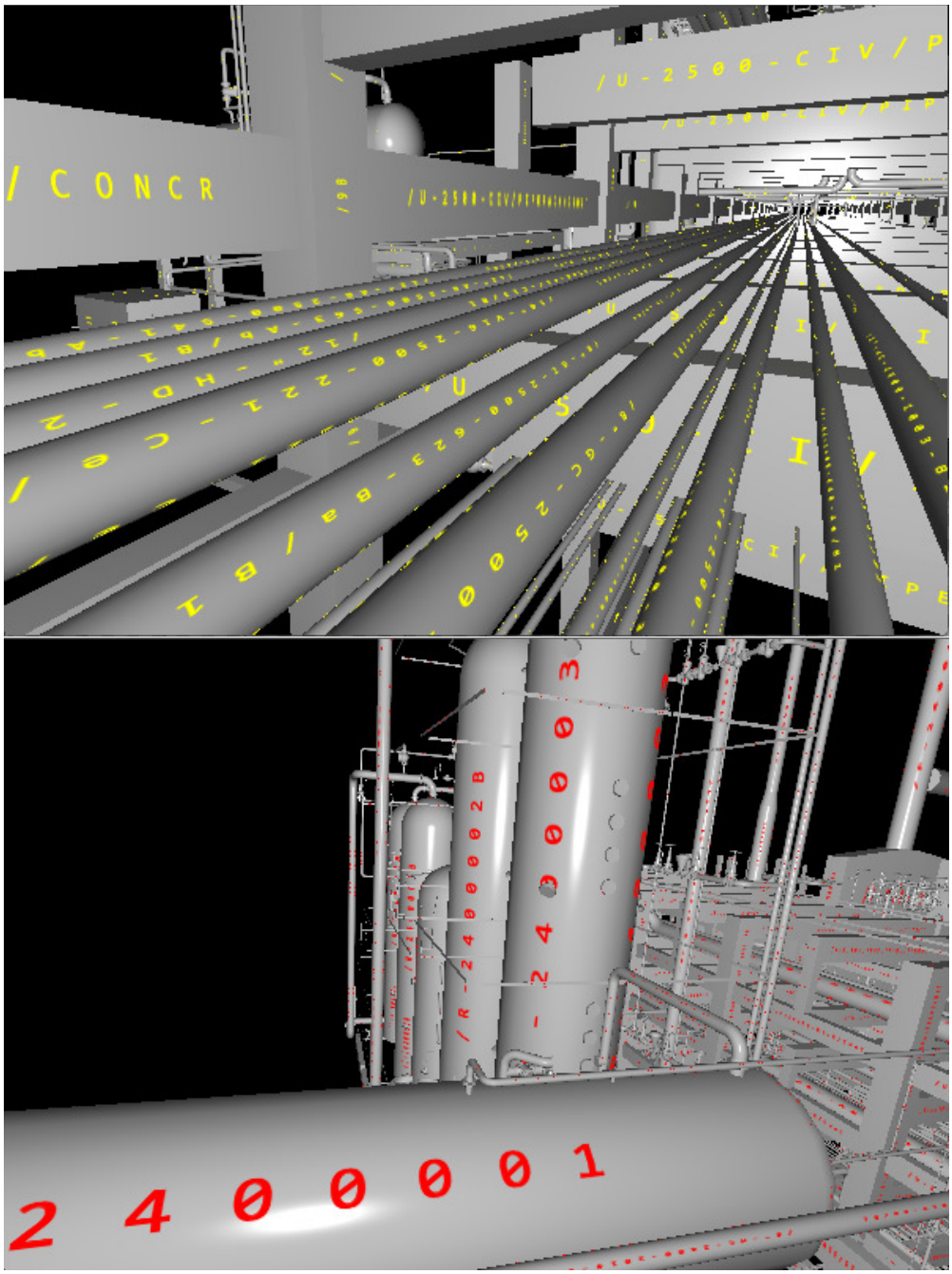


Figura 20: Modelos CAD com rótulos coloridos.

7 Resultados

A técnica apresentada neste trabalho, para exibição de rótulos em superfícies geométricas, utiliza duas passadas, sendo que a primeira não acrescenta muita complexidade à *engine* gráfica de modelos CAD – na primeira passada, são adicionados cálculos apenas para gerar coordenadas de textura e o *fragment shader* escrever informações em dois *buffers* ao invés de um só como seria se não estivesse realizando duas passadas. Já na segunda passada, são realizados vários cálculos por *pixel*, além de alguns acessos a diferentes texturas. Portanto, a segunda passada poderia prejudicar o desempenho, principalmente quando a aplicação utilizasse resoluções altas. Para verificar essas questões, foram realizados os testes descritos a seguir.

Inicialmente, os objetos utilizados para os testes foram *quads* com coordenadas de textura geradas da forma mais simples possível, como mostrado na Figura 11 do Capítulo 4, de forma que os nomes ocupassem praticamente toda a área dos objetos e, consequentemente, para todos os *pixels* dos *quads* seria necessário realizar cálculos relativos a exibição de rótulos. Os rótulos dos objetos possuíam tamanhos e caracteres diferentes. Para os primeiros testes foi utilizado um computador com sistema operacional Windows 7 64 bits, processador Intel® Core I7™ 870 de 2.93 GHz, 8GB de memória RAM e placa gráfica GeForce GTX 580.

O objetivo do primeiro teste era verificar como a exibição de rótulos afetaria o desempenho da aplicação gráfica em uma cena com resoluções diferentes e com um número não muito grande de objetos – quantidade de objetos cujo *rendering* a *engine* consegue realizar com um FPS bastante alto. Para isso, foi criada uma mesma cena com e sem rótulos, sendo que no segundo caso o algoritmo realiza apenas uma passada e não utiliza código para exibir texto, nem mesmo a geração de coordenadas de textura para as geometrias. Neste teste ainda não foi usado o *mipmapping*. Os objetos gerados possuíam posição, orientação e tamanho aleatórios a cada vez que a cena era executada, porém, eram sempre posicionados dentro de uma mesma caixa envolvente (como mostra a Figura 21).

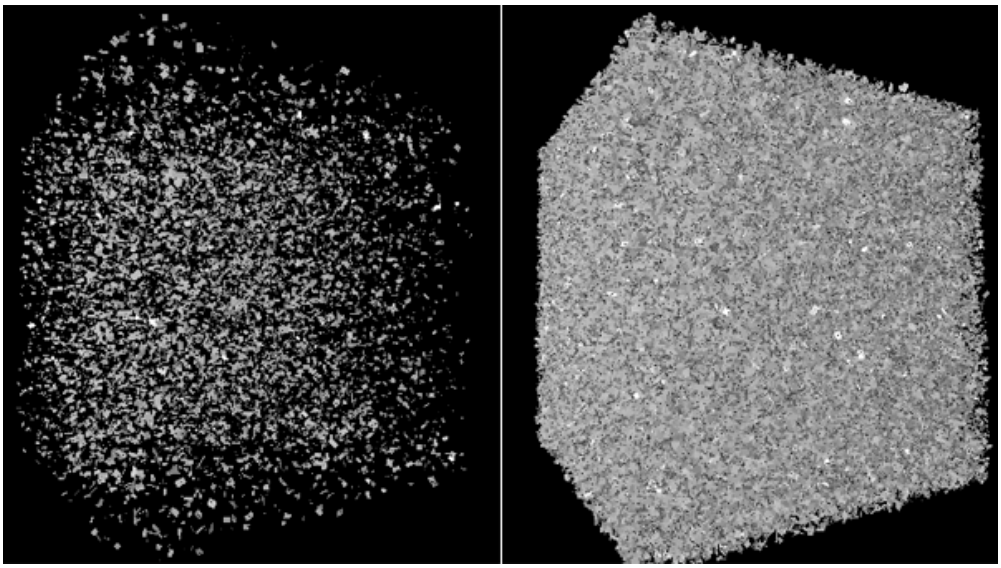


Figura 21: Exemplo de como a cena foi organizada para os testes, vista de longe, com vinte mil e um milhão de *quads*.

A Tabela 1 apresenta os resultados da média de FPS obtidos para uma navegação pelas cenas. Com a Tabela 1 é possível realizar duas análises diferentes, sendo a primeira uma comparação de desempenho ao aumentar a resolução para uma cena sem rótulos e para uma cena com rótulos. A segunda análise compara a taxa de quadros por segundo utilizando-se a mesma resolução, o que pode ser visto na última coluna da tabela. A primeira análise mostra que para dez mil objetos, o aumento de resolução (de 600 x 400 para 1920 x 1200), causou uma queda em torno de 10 FPS no caso de uma cena sem rótulos e de 13 FPS para uma cena com rótulos. Logo, pode ser concluído que independente do uso de rótulos há uma queda de desempenho relacionada a uma maior resolução, e é próxima em ambos os casos, sendo pouco maior para os objetos com texto. Com cinquenta mil objetos a queda devido ao aumento da resolução foi menor, de aproximadamente 2 FPS com ou sem rótulos.

Objetos	Resolução	FPS (Sem Rótulos)	FPS (Com Rótulos)	Diferença
10.000	600 x 400	494,8	472,2	22,6
	1920 x 1200	484,5	459,5	25,0
50.000	600 x 400	102,4	102,9	-0,5
	1920 x 1200	100,5	100,6	-0,1

Tabela 1: Desempenho com dez mil e cinquenta mil objetos.

Fazendo a análise para uma mesma resolução os resultados se encontram na coluna última coluna da Tabela 1, e com dez mil objetos e resolução de 600 x 400,

houve uma queda de aproximadamente 23 FPS para a cena com rótulos, e com resolução de 1920 x 1200 a queda foi de 25 FPS – valores que parecem altos, porém, a taxa de quadros por segundo resultante permaneceu acima de 450 FPS, no pior caso, o que é mais que suficiente para um visualizador em tempo real. Já para cinquenta mil objetos, com as duas resoluções não foi constatada perda de desempenho. Nesses casos, a taxa de quadros por segundo resultante foi até minimamente maior com rótulos, 0.5 FPS maior para a resolução de 600 x 400, o que a princípio seria algo inesperado. Mas isso pode ser explicado por se tratar de uma média de FPS e pelo fato das cenas terem sido geradas aleatoriamente. A diferença mostra que não houve perda significativa de desempenho no *rendering* com a inclusão dos rótulos.

O segundo teste teve o objetivo de verificar o desempenho da *engine* gráfica com a exibição de rótulos em uma cena com um grande número de objetos, o que é o foco principal do trabalho. Como visto no teste anterior, quando o número de objetos aumentou para cinquenta mil, não houve queda significativa de desempenho, e tal comportamento se manteve com ainda mais objetos, como pode ser visto na Tabela 2. Para quinhentos mil, um milhão e um milhão e duzentos mil objetos, não houve perda de desempenho considerável com a exibição de rótulos em ambas as resoluções testadas.

Objetos	Resolução	FPS (Sem Rótulos)	FPS (Com Rótulos)	Diferença
500.000	600 x 400	10,6	10,5	0,1
	1920 x 1200	10,7	9,4	1,3
1.000.000	600 x 400	5,4	5,5	-0,1
	1920 x 1200	5,5	5,3	0,2
1.200.000	600 x 400	4,7	4,6	0,1
	1920 x 1200	4,7	4,5	0,2

Tabela 2: Desempenho com quinhentos, um milhão e um milhão e duzentos mil objetos.

O terceiro teste foi realizado com modelos CAD reais utilizando-se todas as primitivas possíveis existentes no modelo, com exceção das malhas de triângulos, sendo que cilindros, caixas, esferas e "dishes" exibiam rótulos. As coordenadas de textura dos objetos foram definidas como explicado no Capítulo 4, sendo geradas automaticamente dependendo do tamanho dos objetos e da quantidade de caracteres dos rótulos. Os resultados se encontram na Tabela 3 e na Figura 22 e mostram que também não houve perda de desempenho com o uso dos rótulos.

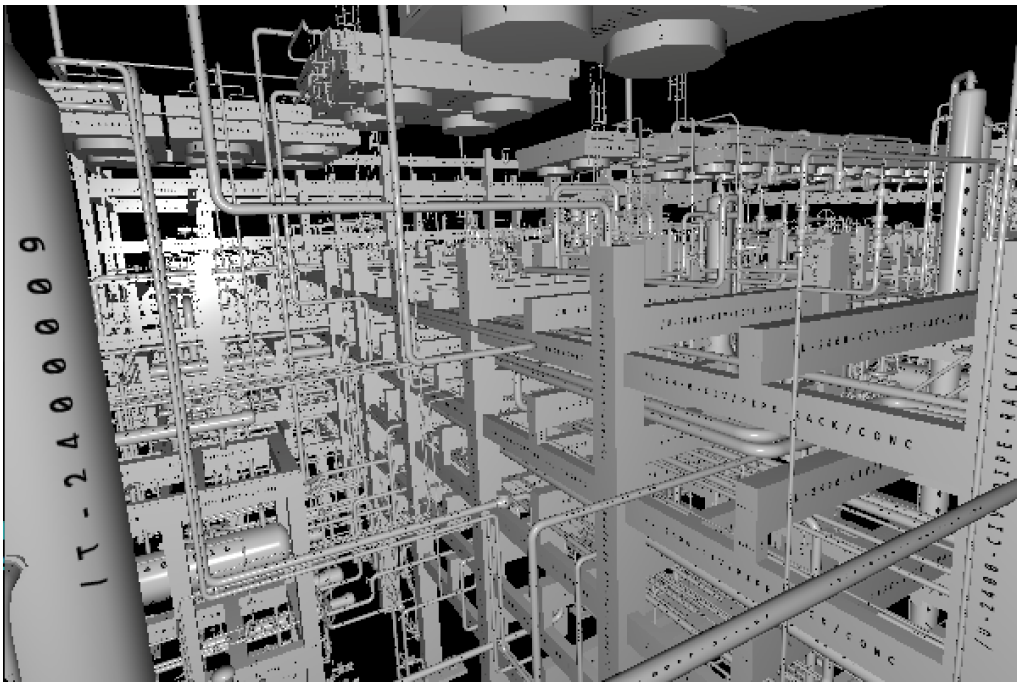


Figura 22: Modelo CAD utilizado para testes, sem *mipmapping*.

Objetos	Resolução	FPS (Sem Rótulos)	FPS (Com Rótulos)	Diferença
300.000	600 x 400	16,6	16,6	0,0
	1920 x 1200	15,0	15,0	0,0

Tabela 3: Desempenho com modelo CAD real.

O quarto teste teve o objetivo de medir o desempenho com o uso de *mipmapping* e, nesse caso, foi sempre utilizada a resolução de 1920x1200. Os resultados são apresentados na Tabela 4 e, como pode ser visto, houve uma pequena queda de desempenho, cerca de 6 FPS, no caso de dez mil objetos, a qual diminuiu conforme o número de objetos aumentou. Para quinhentos mil objetos o desempenho com rótulos foi até maior, o que pode ter ocorrido pelo fato desse valor ser na verdade uma média de várias medições durante uma navegação pela cena.

Objetos	FPS (Com rótulos e sem *mipmapping*)	FPS (Com rótulos e com *mipmapping*)	Diferença
10.000	459,5	453	6,5
50.000	100,6	97	3,6
500.000	9,4	10,5	-1,1
1.000.000	5,3	5,24	0,06
1.200.000	4,5	4,48	0,02

Tabela 4: Comparação de desempenho com e sem *mipmapping*, com resolução de 1920x1200.

Por último, foram feitos testes de desempenho com *quads*, como os apresentados na Tabela 1 e na Tabela 2, mas em uma máquina menos potente, com sistema operacional Windows 7 32 bits, processador Intel® Core(TM)2 Duo E4500 de 2.2GHz, com 2GB de memória RAM e placa de vídeo GeForce 9600GT. Os resultados se encontram na Tabela 5, que mostra não ter havido queda significativa de desempenho com o uso dos rótulos, exceto para dez mil objetos, quando houve uma queda maior do que com o primeiro computador, mas ainda assim a taxa de quadros por segundo continuou acima do necessário para um visualizador em tempo real.

Objetos	Resolução	FPS (Sem Rótulos)	FPS (Com Rótulos)	Diferença
10.000	600 x 400	573,0	513,0	60,0
	1920 x 1080	291,5	178,0	113,5
50.000	600 x 400	70,9	70,2	0,7
	1920 x 1080	68,0	69,6	-1,6
200.000	600 x 400	17,9	17,9	0,0
	1920 x 1080	17,7	17,4	0,3
500.000	600 x 400	7,3	7,3	0,0
	1920 x 1080	7,3	7,2	0,1

Tabela 5: Testes de desempenho de *quads* em placa GeForce 9600GT

8 Conclusões e Trabalhos Futuros

Nesta dissertação desenvolvemos um algoritmo em GPU para exibir, em tempo real, rótulos nas superfícies de formas geométricas de modelos massivos que apresenta bons resultados visuais e de desempenho. Um dos problemas apontados na introdução deste trabalho é que, ao lidar com modelos massivos, cada um de seus objetos poderia possuir um nome diferente e precisaria de um rótulo diferente e, se fosse utilizada uma textura por rótulo, seria necessário realizar múltiplas trocas de contexto de textura por quadro prejudicando bastante o desempenho da aplicação. No algoritmo desenvolvido neste trabalho somente os rótulos que precisam aparecer na tela são construídos a cada quadro, em GPU, e o mapeamento dos rótulos é feito para cada fragmento utilizando-se para consulta um atlas de caracteres e um *buffer* com os códigos ASCII necessários. Com isso, não há necessidade de construir e armazenar texturas com as imagens de cada rótulo em memória de vídeo antes do *rendering* dos objetos e, também, não é preciso lidar com problemas de troca de contexto de textura.

Como apontado também na introdução deste trabalho, se fosse utilizada uma textura para cada objeto, poderia haver um gasto de memória excessivo e tornaria-se necessária a elaboração de estratégias para manter em memória de vídeo um limite de texturas por quadro. Além de que seria possível que nem todas as texturas pudessem ser guardadas na memória principal ao mesmo tempo e, dessa forma, precisassem ser reconstruídas para formar as palavras de um rótulo em tempo real, o que pode prejudicar o desempenho da aplicação. Como o algoritmo desenvolvido cria as texturas necessárias a cada quadro em GPU, só existe a necessidade de armazenar o atlas de caracteres e o *buffer* com os códigos ASCII dos rótulos. Guardar códigos ASCII, que gastam 1 byte cada, gasta muito menos memória do que armazenar imagens dos rótulos. Como foi mostrado no Capítulo 3, se cada rótulo possuir 20 letras e for necessário dois milhões de rótulos, o gasto de memória de vídeo será de aproximadamente 38 MB.

O problema de posicionamento de texto em objetos de modelos CAD foi resolvido para cilindros, caixas, “dishes” e esferas. Foi identificado que para o

usuário enxergar os rótulos independentemente de como a câmera aponte para um objeto, repetição de texto era uma boa opção. Seria possível que o posicionamento do rótulo no objeto fosse dependente da posição da câmera, porém, foi pensado que não seria bom que os rótulos ficassem se movendo a cada quadro, pois devido ao grande número de rótulos na cena isso poderia fazer o usuário se perder, além de poluir demais a cena. O posicionamento dos rótulos é dependente da definição de coordenadas de textura e, para os objetos tratados nesse caso (cilindros, caixas, esferas e "dishes"), elas são geradas de forma automática para se ajustarem corretamente a objetos de diferentes tamanhos. Existem outras primitivas nos modelos CAD utilizados, como cones, pirâmides, toros e malhas que não exibem rótulos devido a não terem sido geradas coordenadas de textura para eles, o que será tratado em trabalho futuro.

Para contornar o problema de serrilhamento dos rótulos foi utilizada a técnica de *mipmapping*, que apresentou melhorias na qualidade do texto visualizado, como mostrado no Capítulo 6. Foi identificado um problema na geração das texturas para a pirâmide de *mipmapping*, pois em seus níveis mais altos as imagens não representavam bem o atlas de caracteres, do jeito que ele foi organizado. Com isso, esse assunto também pode ser mais explorado em trabalhos futuros.

Como explicado nos capítulos anteriores, o algoritmo para a exibição de rótulos é de duas passadas, sendo que na primeira é feito o *rendering* da cena para um *buffer* de cores fora da tela, também são configuradas coordenadas de textura para os objetos e as informações por *pixel* necessárias para a exibição de rótulos são escritas em um segundo *buffer*. A segunda passada consulta esses buffers e calcula a cor final dos *pixels* resultando em uma cena com rótulos exibidos e mapeados corretamente sobre os objetos. Seria possível, porém, realizar isso tudo em uma passada e é preciso explicar porque foram utilizadas duas passadas.

O motivo de serem realizadas duas passadas ao invés de uma se deve a como o algoritmo foi desenvolvido. O cálculo para mapear o texto em um objeto é feito por *pixel* e não por vértice e sabia-se desde o início que, como não bastava definir coordenadas de textura para os vértices, seria necessário diversas operações por *pixel* para o mapeamento dos rótulos, e para realizar testes e verificar o funcionamento do algoritmo seria melhor realizar esses cálculos em CPU. Com isso, no início do desenvolvimento do algoritmo a primeira passada fazia o mesmo que faz hoje, escrevia informações em dois *buffers* fora da tela e a segunda, em

CPU, consultava esses *buffers* e em um processamento de imagens, fazia o mapeamento dos rótulos para cada *pixel* e gerava uma imagem final com objetos e seus rótulos que era transferida para o Frame Buffer.

Uma vez que foram acertados e validados os cálculos, passá-los de CPU para a GPU era algo simples, e isso foi feito, resultando no algoritmo de duas passadas explicado neste trabalho. Como os testes de desempenho iniciais foram bons, não houve a necessidade de juntar as duas passadas em uma só, e o resto do trabalho continuou a ser desenvolvido a partir daí. O possível uso de uma única passada não garante uma melhoria no desempenho, ou um desempenho semelhante ao alcançado, e o *fragment shader* da primeira passada, que realiza cálculos de iluminação por fragmento, teria que realizar, também, todos os cálculos para os rótulos. Mas, é algo que poderia ser realizado, pois todas as informações necessárias para a exibição de rótulos poderiam estar disponíveis na primeira passada. Uma vantagem de uma passada apenas é que não seria necessário que informações fossem transferidas entre as duas etapas com o uso dos *buffers* fora da tela e, portanto, seria gasto menos memória. Fica como trabalho futuro a implementação da técnica em uma única passada e a avaliação comparativa em relação ao processo com duas passadas.

Outro problema identificado no caso de modelos CAD é a orientação dos rótulos nos objetos. Independente de como os objetos 3D foram modelados, se possuem rotações, ou não, os rótulos nunca deveriam ser visualizados de cabeça pra baixo. Tais rotações precisam ser identificadas e compensadas para que o texto apareça posicionado corretamente. No caso da *engine* gráfica utilizada, a rotação de todos os objetos é conhecida antes da geração de suas coordenadas de textura e, portanto, esse problema poderia ser resolvido em trabalhos futuros.

Ainda como trabalho futuro, existe a possibilidade de estudar como escrever rótulos com mais de uma linha. Também deve ser explorada a aplicação da técnica desenvolvida fora do domínio de CAD, uma vez que ela pode atender outros domínios de aplicação.

9
Referências

[1] A. B. Raposo, E. T. L. Courseil, G. N. Wagner e I. H. F. G. M. Santos, "Towards the Use of CAD Models in VR Applications," *ACM International Conference on Virtual-Reality Continuum and its Applications in Industry - VRCAI,* pp. 67-74, 2006.

[2] S.-e. Yoon, E. Gobbetti, D. Kasik e D. Manocha, Real-Time Massive Model Rendering (Synthesis Lectures on Computer Graphics and Animation), Morgan & Claypool Pubs., 2008.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. Stone e J. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, pp. 879-899, 2008.

[4] I. H. F. Santos, A. Raposo, L. P. Soares, E. T. L. Courseil, G. Wagner, P. Santos, R. Toledo e M. Gattass, "EnViron: An Integrated VR Tool for Engineering Projects," *Proceedings of the 12th International Conference on CSCW in Design,* vol. II, pp. 721-726, 2008.

[5] A. Raposo, I. Santos, L. Soares, G. Wagner, E. Corseuil e M. Gattass, "Environ: Integrating VR and CAD in Engineering Projects," *IEEE Computer Graphics & Applications,* vol. 29, pp. 91-95, 2009.

[6] NVIDIA Corporation, "Improve Batching Using Texture Atlases," 2004. [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/tools/files/Texture_Atlas_Whitepaper.pdf. [Acesso em março 2013].

[7] A. R. "Aveva Home Page," 2010. [Online]. Available: http://www.aveva.com. [Acesso em abril 2010].

[8] R. D. Prado, A. B. Raposo and L. P. Soares, "Autotag: Applying Tags to Virtual Objects in Real-Time Visualization Systems," *VIII Workshop de Realidade Virtual e Aumentada - WRVA,* 2011.

[9] S. Lefebvre, S. Hornus e F. Neyret, "Texture Sprites: Texture Elements Splatted on Surfaces," *ACM Symposium on Interactive 3D Graphics,* 2005.

[10] Z. Qin, M. D. McCool e C. S. Kaplan, “Real-Time Texture-Mapped Vector Glyphs,” *Proceedings of the 2006 symposium on interactive 3D graphics and games - I3D,* pp. 125-132, 2006.

[11] G. Cipriano e M. Gleicher, “Text Scaffolds for Effective Surface Labeling,” *IEEE Trans. Visualization and Computer Graphics,* vol. 14(6), pp. 1675-1682, Novembro 2008.

[12] S. Barret, “Sparse Virtual Textures,” 2008. [Online]. Available: http://silverspaceship.com/src/svt. [Acesso em 2013].

[13] M. Mittring, “Advanced Virtual Texture Topics,” *Proceedings of SIGGRAPH 2008,* pp. 23-51, 2008.

[14] OpenGL 2.1, “OpenGL 2.1 Reference Pages,” 2006. [Online]. Available: http://www.opengl.org/sdk/docs/man2/. [Acesso em 2013].

[15] GLSL 1.2, “The OpenGL® Shading Language 1.2,” 2006. [Online]. Available: http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf. [Acesso em 2013].

[16] T. Akenine-Moller, E. Haines e N. Hoffman, Real-Time Rendering, 3rd Edition ed., 2008.