UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Breno Mattos de Paula

Wagner Luiz Oliveira dos Santos

Ferramenta para reconstrução de cenas 3D utilizando imagens.

Niterói
2010

UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Breno Mattos de Paula

Wagner Luiz Oliveira dos Santos

Ferramenta para reconstrução de cenas 3D utilizando imagens.

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel.

Área de Concentração: Computação Gráfica.

Orientador: Profº. Dr. Anselmo Antunes Montenegro

Niterói
2010

Ferramenta para reconstrução de cenas 3D utilizando imagens.

Breno Mattos de Paula

Wagner Luiz Oliveira dos Santos

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel.

Área de Concentração: Computação Gráfica.

Aprovado em Fevereiro de 2010.

BANCA EXAMINADORA

_____

Prof^{o}. Dr. Anselmo Antunes Montenegro - Orientador
UFF

_____

Prof^{o}.Dr. Esteban Walter Gonzalez Clua

_____

Msc. Luciene Cristina Soares Motta
UFF

Niterói
2010

AGRADECIMENTOS

A Deus por sempre me guiar pelos caminhos corretos e ter me proporcionado o prazer de aprender em uma faculdade de alto nível como a UFF.

Aos meus familiares, por terem me incentivado a chegar até a faculdade, mesmo diante de tantas dificuldades e sacrifícios, que é financiar um ensino de qualidade.

A todos os meus amigos e parceiros da turma 204.31, com os quais compartilhei as dificuldades do dia a dia, e com os quais sempre pude contar para me ajudar nos momentos mais complicados.

Ao Professor Anselmo, por ter confiado em nós, na responsabilidade de trabalhar em um projeto tão importante e tão mágico como esse, compartilhando seu conhecimento e sempre disposto a ajudar.

A todos os professores da universidade, por sempre estarem disposto a ajudar, com um ensino de alta qualidade, e conhecimentos tão importantes, mesmo diante das dificuldades que é lecionar em uma faculdade pública.

Ao meu parceiro de projeto e amigo Wagner, que além de compartilhar as dificuldades do projeto, foi um parceiro importante para tudo se desenvolver.

Breno Mattos de Paula

Agradeço primeiramente a Deus por me dar toda a condição necessária para a execução deste trabalho de final de curso, ciente que sem Ele, eu não conseguiria vencer mais uma etapa da minha vida.

A meus pais, a minha eterna gratidão e admiração por terem se privado de muitas coisas em sua vida para me dar uma educação de qualidade.

Ao Professor Anselmo Antunes Montenegro, por ter confiado em nós, um projeto que teve uma importância muito grande em sua carreira acadêmica e por ter nos apoiado durante todo o desenvolvimento do mesmo.

A banca examinadora Esteban Walter Gonzalez Clua e Luciene Cristina Soares Motta por aceitarem fazer parte deste projeto.

A todos os amigos que fiz na turma 204.31, que me ajudaram muito no decorrer do curso de ciência computação. Em especial ao Matheus Bersot Siqueira Barros e Rafael Machado Alves que me ajudaram em diversas etapas importantes deste trabalho.

A meu parceiro e amigo Breno Mattos de Paula com quem fiz diversos trabalhos e sempre obtive sucesso graças a sua competência e dedicação.

E a todos que ajudaram direta e indiretamente para execução deste projeto.

Wagner Luiz Oliveira dos Santos.

“O segredo da criatividade é saber como esconder as fontes.”

Albert Einstein

RESUMO

A computação gráfica é uma área da ciência da computação na qual técnica e criatividade são combinadas para produzir resultados impressionantes. Muitos dos avanços alcançados na computação gráfica podem ser aplicados em diferentes áreas como medicina, engenharia, ciências sociais e etc. Por exemplo, o uso de modelos 3D e cenas virtuais é cada vez mais comum em processos que visam facilitar o diagnóstico em exames médicos, a construção de prédios mais modernos, e a especificação de rotas de trânsito dos transportes públicos e etc.

Muitas das atividades que envolvem a computação gráfica são praticamente impossíveis de serem realizadas sem a presença de um artista especialista em modelagem virtual. Por este mesmo motivo, muitos estudos têm sido feitos como objetivo de facilitar as tarefas executadas por artistas e desenvolvedores de games.

O trabalho proposto tem como objetivo investigar formas de se criar modelos robustos e com desempenho aceitável, que possam ser adquiridos de referências reais, através de imagens obtidas de câmeras fotográficas comuns, da forma mais automatizada possível, com técnicas e implementações simples de serem entendidas, e com exemplos que mostram ser possível tal solução.

Neste trabalho foi implementado um sistema capaz de demonstrar o uso da técnica de reconstrução volumétrica a partir de imagens. O sistema atualmente se restringe a imagens sintéticas. No entanto, é completamente viável o uso de imagens do mundo real, desde que previamente calibradas. Foram realizados experimentos que mostram a viabilidade e flexibilidade do uso desta técnica, como um mecanismo para a reconstrução de objetos 3D.

Palavras-chave: <Voxel, Voxel Coloring, Opengl, Java, MVC, Programação paralela, Modelos 3D, Foto-consistência>

ABSTRACT

Computer graphics is an area that requires the combination of creativity and technique to accomplish effective, robust and impressive results. Many of the advances achieved are applied to areas such as medicine, engineering, social sciences, arts and others.

The use of 3D models and virtual scenes are increasingly used to facilitate medical examinations, construct modern buildings, study traffic routes in public transport design and develop game scenarios. In all these cases, the skills of an artist expert in virtual modeling play a fundamental role in the overall process. Hence, many of the developments nowadays aim to make easier the life of artists and game developers.

This work presents ways to create robust models with acceptable performance, which can be acquired from real world references, from images extracted from real cameras and using techniques and implementation codes that are simple to understand, presenting examples that can show that the solution is possible.

In this work we implemented a system that demonstrates the use of the technique of volumetric reconstruction from images. The system currently is limited to synthetic images; however it is possible to use images from the real world when previously calibrated. Experiments were carried out to show the viability and flexibility of its use as a mechanism for the reconstruction of 3D objects.

Keywords : <Voxel, Voxel Coloring, Opengl, Java, MVC, Parallel programming, 3D Models, Image photo-consistence>

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AWT	Abstract Window Toolkit
JFC	Java Foundation Classes
JOGL	Java OpenGL
MVC	Model-view-controller
XML	Extensible Markup Language

1 -Introdução

O mercado de games está em amplo crescimento e, no ramo de entretenimento, já pode ser considerado o maior. Impulsionado por esse crescimento, várias tecnologias foram desenvolvidas, tanto em software como em hardware. São exemplos desse desenvolvimento as modernas GPUs, softwares de edição de imagem e realidade virtual e modeladores de objetos 3D extremamente sofisticados.

Apesar do avanço tecnológico, ainda existem problemas a serem resolvidos na área de *games*, já que os jogadores estão cada vez mais dispostos a terem uma maior experiência de imersão, interagindo com modelos virtuais que se aproximam da realidade. Conseqüentemente, com um mercado tão propício, é preciso encontrar formas de melhorar a qualidade da construção dos modelos, barateando custos de produção e melhorando o processamento dos games, com soluções inteligentes[4].

Nos estudos acadêmicos na área de medicina, a computação gráfica é amplamente utilizada como meio de pesquisa. Através de simulações virtuais do funcionamento do corpo humano, investigações mais precisas podem ser feitas, sem a necessidade de cobaias ou corpos humanos reais.

Um sistema gráfico que pode ser muito útil para a medicina é um que seja capaz de realizar reconstruções 3D de órgãos. Nesse software, seria possível reconstruir um modelo virtual a partir de um órgão doente e, por exemplo, compará-lo com um sadio, possibilitando a realização de um diagnóstico mais detalhado por parte do profissional que examina o resultado. Exames desse tipo já existem no mercado[1], mas a melhoria da qualidade dos exames é muito importante.

No ramo da saúde, ainda que existam sistemas avançados para diagnósticos de pacientes utilizando computação gráfica, é importante a busca por uma melhoria do desempenho dessas aplicações e da qualidade dos modelos reconstruídos, para que os profissionais da área possam encontrar a forma mais eficiente de tratar um paciente.

Considerando todos estes aspectos, questiona-se se é possível desenvolver uma solução para reconstrução de objetos 3D, com qualidade compatível com os sistemas em hardware de alto custo disponíveis atualmente, e capaz de desempenhar sua tarefa em tempo computacional aceitável, para as configurações das máquinas atuais. Também é

importante considerar o aspecto de automação, de tal forma que o próprio computador possa realizar o processo de reconstrução, com a mínima intervenção humana possível.

Com essa finalidade, o presente trabalho se propõe a investigar e implementar uma solução para a reconstrução de objetos 3D, apresentando uma abordagem de modelagem a partir de imagens, que podem ser extraídas tanto de referencias reais, através de câmeras fotográficas, quanto de virtuais, neste último caso, fornecendo uma alternativa para conversão entre diferentes formas de representação de modelos, como, por exemplo, de malhas para modelos volumétricos.

A abordagem explicitada justifica-se pelo fato de que, com várias imagens de uma mesma referência, retiradas em angulações diferentes, é possível captar todas as informações possíveis de cor, posição e delimitações da forma de um objeto. Logo, por que não utilizar essas informações, para criar uma escultura baseada nessas imagens de referência ?

Com uma meta traçada de construir modelos de qualidade, com tempo otimizado, de forma automática, com mínima intervenção humana, o presente projeto apresenta soluções e ferramentas importantes para utilização e interação, objetivando sempre baixar custos nos projetos onde necessitam de construção de cenas 3D, onde artistas gastam muito tempo para construir cenas utilizando de softwares presentes no mercado. Outra meta é apresentar solução para melhorar o desempenho no momento da reconstrução, em uma tentativa de alcançar uma reconstrução em tempo real.

Deve-se citar também, que a ferramenta desenvolvida é uma plataforma de testes e pesquisas, aberta ao público, tendo como requisito básico a estruturação do código, com a finalidade de fácil acoplamento de outros recursos. Algumas idéias são apresentadas no final do documento, no capítulo de conclusão no sub-tópico trabalhos futuros.

Apesar de o método ter como alvo a reconstrução de objetos do mundo real, neste trabalho, os experimentos restringiram-se a reconstrução de cenas a partir de imagens obtidas de cenas tridimensionais sintéticas, já que o objetivo primordial é mostrar a viabilidade da técnica. O tratamento de imagens reais pode ser facilmente realizado, bastando para isso alimentar o sistema com imagens previamente calibradas. Com as hipóteses e objetivos traçados, o presente documento apresenta-se organizado nos seguintes capítulos:

- Capítulo 2 – Voxel Coloring.
  É descrito de forma teórica, o método usado na solução do problema de reconstrução de cenas 3D a partir de imagens.

- Capítulo 3 – Ferramenta.
  É descrita em detalhes uma solução para o problema de reconstrução 3D, apresentando uma ferramenta própria, explicitando as tecnologias utilizadas e um passo a passo de como utilizá-la.

- Capítulo 4 – Implementação.
  É abordado todo o processo de implementação, apresentando toda a estrutura de classes, padrões de projetos, estrutura de dados e uma solução para paralelização do algoritmo em CPUs *multicore*.

- Capítulo 5 – Resultados
  São apresentados os testes realizados e os resultados obtidos pela ferramenta, apresentando dados comparativos de imagens e gráficos gerados nos experimentos.

- Capítulo 6- Conclusão
  São expostas as conclusões finais sobre os resultados obtidos e as expectativas para o trabalho, mostrando idéias para possíveis trabalhos futuros.

2 -Voxel coloring

2.1 - Introdução

O método que será abordado nessa seção, tem como objetivo reconstruir uma cena por meio de uma representação volumétrica do espaço. Massone [11] foi um dos primeiros a propor uma técnica que tinha como resultado final uma casca visual, que era descrito por um conjunto de voxels.

Segundo o trabalho em [12], pode-se agrupar os métodos de reconstrução de cena através de reconstrução volumétrica, em dois grandes grupos:

- Métodos baseados em silhuetas.

- Métodos baseados em foto-consistência.

Os métodos que são baseados em silhuetas abrangem as técnicas de reconstrução, que tem como base, a aproximação da forma em função do "contorno" do modelo. Por sua vez, os baseados em foto-consistência abrangem técnicas para reconstruir objetos com base nas informações fotométricas dos mesmos.

O *Voxel Coloring* encontra-se no grupo de métodos baseados em foto-consistência. Esse método foi proposto por Seitz e Dyer[15], tratando o problema de reconstrução de cenas através de imagens com uma técnica de coloração de voxels. Nas seções seguintes, encontram-se definidos os principais conceitos utilizados no trabalho, é apresentado o problema de coloração de *voxels*, caracterizadas as restrições do problema, especificado o algoritmo de visibilidade e, por último, descrito como é feito o cálculo da coloração de voxels em si.

2.2 -Voxel

"O voxel é o menor elemento de unidade de volume. O equivalente 3D do pixel." Voxelogic [6].

O voxel possui uma posição no espaço tridimensional, de modo análogo ao pixel em um espaço bidimensional, acrescentando a coordenada de *z*, para que sua posição fique completamente caracterizada. Outra característica que um voxel possui é ter uma cor única

e absoluta. Essa representação é muito importante para o método proposto, já que toda reconstrução é baseada no modelo de representação por voxels.

2.3 - Definição do problema

Para definir o problema, considera-se que a reconstrução no mundo virtual esteja toda contida em uma caixa envolvente invisível, com tamanho suficiente para conter o modelo gerado. Esse invólucro pode ser subdividido em um conjunto de voxels, de tamanho arbitrário, de tal forma que todo voxel tenha uma posição e ocupe um espaço único, bem definido, na caixa envolvente, além de possuir uma única cor. A noção acima descrita pode ser definida formalmente conforme o trabalho [12]:

"Primeiramente, considera-se que a cena tridimensional *S*, a ser reconstruída, esteja contida em um subconjunto fechado $U \subset R^3$, o qual é representado através de um conjunto de voxels *V*, onde cada voxel $v \in V$ ocupa um volume homogêneo do espaço e possui uma única cor. Identificamos *S* com sua representação discreta induzida por *V*, com a exigência de que todo voxel $v \in S$ seja completamente opaco. Além disso, utilizamos a notação *cor* (*v*, *S*) para representar a cor de um voxel *v* em uma cena *S*."

Com um conjunto de imagens I_i geradas por várias câmeras em posições e orientações distintas, obtêm-se uma gama de pixels para cada foto. As informações das cores em imagens distintas, correspondentes a projeção de um mesmo voxel, que deve ser visível, são utilizadas para se determinar se tal voxel existe ou não na caixa envolvente, segundo um processo que será explicado a seguir. Caso um voxel seja sinalizado como pertencente ao espaço que contém o modelo, então a cor a ele atribuída será dada por uma função das cores dos pixels nos quais se projeta.

A partir das definições e explicações acima, apresenta-se o problema de *voxel coloring*, que consiste em obter uma rotulação

$$cor(v,S) = \begin{cases} c, v \in S \\ c_t, v \notin S \end{cases},$$

onde $c = f\text{c}(CP_i(v,I_i))$ é uma função das cores nas projeções $CP_i(v)$ de *v* nas imagens I_i, $0 \leq i \leq n$, nas quais *v* encontra-se visível, e c_t uma cor transparente, a qual indica que o voxel não faz parte do modelo.

A questão sobre como definir quando um voxel *v* pertence à cena *S* é extremamente delicada e, na verdade, o cerne do problema.

O método de *voxel coloring* trata desta questão através do conceito de *foto-consistência*, a qual é definida por uma relação entre os voxels e as imagens de entrada. A relação de foto-consistência $f_{cn}(v,I_1,I_2,...,I_n)$, estabelece que um voxel *v* é considerado foto-consistente, em relação a um conjunto de imagens $I_1,...,I_n$, $0 \leq i \leq n$, e, portanto, pertencente ao modelo gerado quando, ao se atribuir a *v* uma cor apropriada, for possível reproduzir as cores em cada uma das imagens em que sua projeção encontra-se visível. (Figura 1).

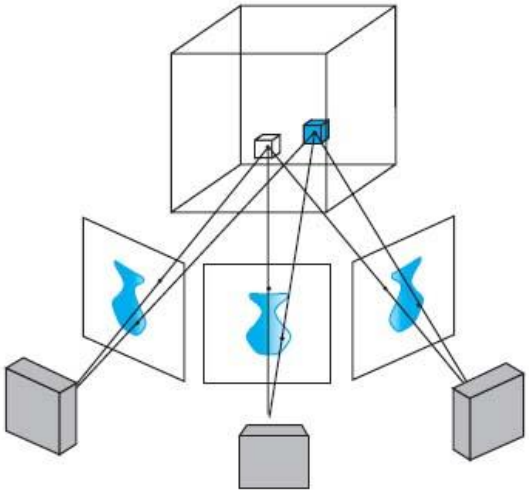


Figura 1: Mapeamento de voxels

Para simplificar o método, e tornar factível a especificação de uma função capaz de determinar a foto-consistência dos elementos no espaço de reconstrução, os criadores do método *voxel coloring* impuseram algumas restrições, as quais são descritas na próxima seção.

2.3.1 -Restrições

No desenvolvimento de um método, com base em um conjunto de princípios, é fundamental identificar em que situações tais princípios são válidos para que o método possa produzir resultados corretos, ao mesmo tempo em que possa ser possível detectar as em que situações ele não é apropriado.

A idéia de reconstrução baseada em foto-consistência propõe que um conjunto de pontos em um espaço seja classificado segundo uma medida de foto-consistência, com o

objetivo de detectar quais destes pontos fazem parte dos objetos gráficos que se projetam em um conjunto de imagens.

Analisando precisamente tal afirmação, é possível identificar naturalmente alguns aspectos que devem ser considerados no desenvolvimento do método. O primeiro deles diz respeito ao modelo de iluminação, utilizado para descrever a forma como as cores são geradas na superfície dos objetos. O segundo aspecto importante é o da representação do espaço de reconstrução, o qual é idealmente contínuo, no universo matemático, mas que precisa ser discretizado em um conjunto de voxels, de forma a tornar o processo computacionalmente viável.

Ambos os aspectos têm implicações diretas um no outro. Como os voxels possuem uma única cor, deve-se considerar um modelo de iluminação no qual os elementos do espaço emitem energia radiante igual em todas as direções. Isto leva a primeira restrição:

Na reconstrução das cenas através de coloração de voxels considera-se, por hipótese, que os objetos sejam representados por superfícies que possam ser aproximadas pelo modelo lambertiano.

Tomando como referência o artigo escrito por Rinaldo[13], para a revista "Lumiere Online", especializada em fotografia, uma superfície lambertiana é definida da seguinte forma: "*Uma superfície difusora perfeita, é aquela que emite ou reflete o fluxo luminoso de tal forma que a luminância é a mesma em qualquer ângulo de visão. Essa superfície é chamada de Lambertiana*."

Retornando a questão da representação discreta, chega-se a segunda restrição:

Para que a forma geométrica obtida seja um mapeamento correto do conjunto de imagens de entrada, o tamanho da projeção dos voxels deve corresponder a aproximadamente o tamanho de um pixel.

Na verdade, a segunda restrição pode ser relaxada, se não for necessário obter um modelo que reproduza as imagens na resolução original, mas em uma resolução compatível com uma sub-amostragem das mesmas. Deve-se ter em mente, entretanto, que a relação entre o número de voxels e a resolução de imagens pode levar a sérios problemas de *aliasing*. Em [12] este problema é considerado, todavia, este aspecto está fora do escopo do presente trabalho.

Finalmente, uma questão importantíssima ainda não foi considerada. Na avaliação de foto-consistência de um voxel, se este não for visível em uma das imagens, então as cores na sua projeção, em tal imagem, não devem impor nenhuma restrição a sua foto-consistência. Sem considerar este fato, o método será incapaz de produzir modelos que façam sentido.

2.3.2 -O problema da visibilidade

A definição desse problema e a solução apresentada são muito importantes para a obtenção de modelos corretos do ponto de vista da foto-consistência, e que sejam apenas formados pelos voxels consistentes que realmente são vistos por alguma das câmeras.

O grande dilema é que não é possível determinar uma função de visibilidade sobre um modelo que não é conhecido, isto é, só é possível determinar a visibilidade de um voxel se forem determinados os voxels que fazem parte do modelo, o que é exatamente o problema que se deseja resolver.

Uma forma de resolver esta questão é definir uma relação de prioridade sobre a coleção de voxels, no sentido de que a avaliação da foto-consistência dos voxels contidos na caixa envolvente comece pelos voxels que estão mais próximos do conjunto de câmeras, indo em direção aos mais distantes.

Em [12] é descrita a solução para o problema:

“Através de uma ordenação dos elementos de *V* em relação ao sistema de câmeras, podemos percorrer os voxels no sentido dos mais próximos às câmeras para os mais distantes. Assim, a visibilidade dos voxels mais próximos as câmeras sempre fica determinada antes da visibilidade dos voxels mais distantes. Com base neste artifício, não corremos o risco de que a visibilidade de um voxel específico venha a ser modificada posteriormente por uma operação de remoção de voxels, o que invalidaria as decisões já tomadas.”

A técnica apresentada para solução do problema foi explicitado por Seitz [15], e tem como objetivo criar um algoritmo que, com apenas um passo de varredura, seja possível solucionar o problema da visibilidade e reconstrução de modelos através de *voxel coloring*.

A implementação do método é feita através de mapas de visibilidade, sendo que cada câmera possui seu próprio mapa. No começo, todos os campos da matriz de

visibilidade são inicializados como invisíveis e, a partir daí, com a varredura do conjunto de voxels, partindo de fatias mais próximas do conjunto de câmeras, os voxels foto-consistentes mais próximos são atribuídos na matriz como visíveis (Figura 2).

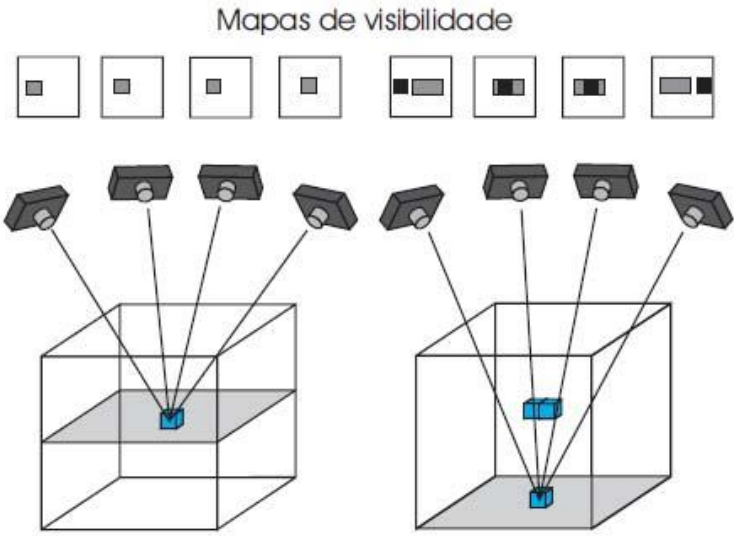


Figura 2: mapa de visibilidade.

Observe que este método somente funcionará corretamente se o espaço de reconstrução estiver fora do *fecho convexo* das câmeras, o que permite a determinação de uma ordenação dos elementos. A solução para uma configuração arbitrária requer outro algoritmo, denominado *Space Carving*, proposto por Kutulakos e Seitz [8], e que consiste de um processo de múltiplas varreduras, uma para cada sentido nos eixos cartesianos, até que o modelo convirja.

2.3.3 - Cálculo da coloração de voxels

Primeiramente, para especificarmos o processo de geração da coloração dos voxels, é preciso saber distinguir quando um voxel é *foto-consistente*, *não-foto-consis*tente e *indefinido*.

Na determinação da foto-consistência, é muito útil poder distinguir os pixels que fazem parte do objeto dos que fazem parte do fundo homogêneo. No momento em que um voxel corrente for mapeado em uma região não pertencente ao fundo da imagem (*background*), este será considerado como um candidato a ser rotulado como foto-consistente, precisando ser avaliado pela função de foto-consistência. Se a função avaliá-lo

como foto-consistente então ele será considerado como um elemento que faz parte do objeto, caso contrário será removido do modelo, através da atribuição de uma cor transparente. Por sua vez, um voxel que se projeta no fundo da cena é considerado automaticamente como não-foto-consistente e, por conseguinte, removido da cena a ser reconstruída.

Nem sempre, tem-se a informação sobre a segmentação da imagem em objetos da cena e fundo. No entanto, isto não é necessário ao funcionamento do método, pois se trata de apenas uma informação auxiliar.

Elementos do espaço de reconstrução são classificados como indefinidos, quando forem mapeados fora das câmeras utilizadas. Não há como atribuir uma cor para um voxel que não tem mapeamento no modelo real.

O processo de avaliação da foto-consistência determina quem será considerado consistente ou não. Esse processo possui duas etapas, que são: percorrer as câmeras da cena e mapear o voxels presentes, levando em consideração a visibilidade do voxel; e avaliar a foto-consistência do voxel em questão.

Para que seja possível desenvolver um método baseado em foto-consistência é necessário tornar precisa a noção de foto-consistência de um voxel. Em outras palavras, é preciso determinar uma forma de estimá-la quantitativamente.

A experiência do dia-a-dia indica que um ponto sobre uma superfície de um objeto aparece com cores ligeiramentes distintas, em função da variação de iluminação do ambiente, das propriedades de reflectância da superfície, isto é, da forma como ela reflete em certa direção a energia radiante incidente, e das próprias características dos sensores que registram a informação luminosa.

Por este motivo, não se deve esperar que um ponto da superfície produza exatamente as mesmas cores em cada uma das imagens, mas que estas sejam coerentes, ou melhor, consistentes, segundo um modelo estatístico.

Um modelo razoável é considerar que as cores vistas nas projeções de um voxel v em um conjunto de imagens I_i, sejam representadas, respectivamente, por um conjunto de variáveis aleatórias X_i independentes com distribuição normal $N(\mu_i, \sigma^2)$, com mesma variância e médias possivelmente distintas.

Para que as cores na projeção de um voxel, correspondam a realizações da cor de um voxel que pertença ao modelo, precisamos testar se $u_0 = u_1 = ... = u_n$, o que significa que as cores são apenas realizações de variáveis aleatórias de mesma natureza.

Isto pode ser feito através de um teste de hipótese sobre a igualdade das médias, onde:

H_0:(hipótese nula): as médias são idênticas ($u_0 = u_1 = ... = u_n$).

H_1:(hipótese alternativa): as médias são distintas.

A confirmação de H_0 significa que as cores são consistentes e o voxel pertencente ao modelo. Por outro lado, quando ela é refutada, tem-se que a hipótese alternativa é confirmada, o que significa que o voxel não faz parte do objeto.

Para realizar o teste de hipótese acima é utilizada a estatística $Z = \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{\sigma} \right)^2 = \frac{ns^2}{\sigma}$, onde x_i são as cores das amostras e $\bar{x}$ a média das cores das amostras. Um resultado bem conhecido é o de que Z possui distribuição χ^2, por ser uma soma dos quadrados de variáveis aleatórias independentes com distribuição normal.

Intuitivamente, a estatística mede o quanto o valor das cores nas amostras se afasta da média amostral. Se a hipótese nula for verdadeira, o valor de Z, para uma dada amostra de cores, não deve se afastar excessivamente da distribuição amostral da estatística. Precisamos então checar se o valor de Z é probabilisticamente aceitável para aceitar ou refutar a hipótese nula.

Para isto, basta verificar se o valor de Z está dentro de uma região crítica $RC = [0, \chi_0^2]$ tal que $P(Z < \chi_0^2) = \alpha$, onde α é um *nível de significância* escolhido, comumente considerado igual a 0.05. Se $Z \in RC$ então aceitamos H_0, caso contrário a refutamos, ficando com a hipótese alternativa.

No sistema implementado, como foram consideradas apenas imagens sintéticas, não é possível avaliar a variância das distribuições das variáveis aleatórias correspondentes às cores. Por este motivo, foi realizado um teste mais simples, comparando-se a variância da amostra de cores com uma tolerância *tol*, pré-estabelecida, conforme ilustrado abaixo:

$$\text{Se} \left(\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right) > tol \text{ então } v.consistência \leftarrow V,$$
$$\text{senão } v.fotoconsistência \leftarrow F$$

Este cálculo, ainda que bastante rudimentar, permite detectar se as cores avaliadas, na projeção de um voxel nas imagens, são muito diferentes entre si. Um exemplo seria se a imagem em um pixel correspondente a um voxel *v* se projetasse com cor verde (r = 0, g = 1, b = 0, α = 1) e em outra imagem, o mesmo voxel se projetasse com cor vermelha (r=1, g=0, b=0, α = 1). Nesse caso, a média seria uma cor amarela (r= 0,5, g = 0,5, b = 0, α = 1), e o resultado levaria a remoção de tal voxel, uma vez que não há consistência entre as cores projetadas.

Após o teste de consistência, as cores são atribuídas aos voxels, respeitando o teste anterior.

Todos os voxels considerados indefinidos recebem uma coloração pré-definida. No sistema, por escolha arbitrária, a cor atribuída é a verde absoluta (r = 0, g = 1, b = 0, α = 1), tornando fácil a identificação de voxel indefinidos.

Um voxel considerado inconsistente deve receber o valor de cor transparente (r = 0, g = 0, g = 0, α = 0).

Por último, deve-se tratar do caso quando um voxel é considerado consistente. Esse caso é o mais difícil, pois são os voxels que são realmente importantes para o resultado final. Para esses elementos, é preciso tomar uma média de todas as cores, nas imagens onde o voxel é visível. O cálculo realizado para achar a cor final do voxel adotado é uma simples média aritmética das cores nas projeções nas imagens em que ele é visível, na figura 3 é exibido o algoritmo de verificação de consistência do voxel.

```
Boleano verificaConsistenteEDefineCorDoVoxel(Voxel voxel,Vetor vetorCameras,Cor corMedia){
        Para camera=vetorCameras[i] faça
        {
                Ponto ponto=mapeaPonto(camera,voxel);
                Se dentroDoLimiteDaFoto(ponto,camera) faça{
                        Se pontoNaoVisitado(ponto,camera) faça{
                                Se não corDeFundo(camera.getCor(ponto)) faça
                                        listaDeCores.adiciona(camera.getCor(ponto));
                                }Senão {
                                        retorna false;
                                }
                        }
                }
         }
        Se listaDeCores não vazia faça{
                Se desvioPadraoAceitavel(listaDeCores){
                    corMedia=mediaDeCor(listaDeCores);
                }Senão{
                        retorna false;
                }
        }Senão {
                retorna false;
        }
        retorna true;
}
```

Figura 3: Função de foto-consistência

3 -Implementação

Este capítulo aborda aspectos técnicos do projeto como a implementação do sistema, os padrões de projeto utilizados, a arquitetura do projeto, como foi implementada a geração de entrada de dados e o processo de reconstrução da cena. Serão abordadas também as principais tecnologias utilizadas para o desenvolvimento do software e as descrições de casos de uso importantes para interação do usuário com o sistema.

O conhecimento dos pontos citados acima auxilia consideravelmente o entendimento do sistema como um todo.

3.1 -Tecnologias utilizadas

- **Linguagem Java:**

  Java é uma linguagem de programação desenvolvida com os seguintes objetivos: criar uma linguagem orientada a objetos; tornar o código portátil e por isso é interpretada e eliminar práticas que afetam a robustez e segurança do código como aritmética de ponteiros e alocação e desalocação de memória[14].

  Muitos devem se perguntar, automaticamente, por que a linguagem Java foi escolhida para um projeto de computação gráfica? Essa pergunta é respondida facilmente quando posto em pauta o objetivo inicial e como o projeto foi sendo desenvolvido no decorrer do tempo. Quando o projeto foi especificado, inicialmente, o principal e talvez mais forte requisito fosse que o código tivesse um mínimo de organização e que ao mesmo tempo fosse legível para os pesquisadores que quiserem utilizar, modificar e adaptar a ferramenta conforme seus propósitos.

  Deve-se ter em mente que Java, hoje em dia, é uma linguagem bem difundida, de fácil aprendizado e leitura, e de fácil acesso, por se encontrarem disponíveis compiladores gratuitos e de muito boa qualidade na rede mundial de computadores. Além disso, Java facilita e incentiva a aplicação de técnicas de organização de código, ao mesmo tempo em que possui uma grande

comunidade de desenvolvedores, o que ajudou na obtenção de bibliotecas que facilitaram e tornaram ágil o processo de desenvolvimento.

A eficiência, apesar de ser um requisito fundamental em muitas aplicações em computação gráfica, em particular, aquelas que demandam resposta em tempo real, não foi considerada como um aspecto fundamental na ferramenta aqui proposta e desenvolvida. Por outro lado, nada impede que este requisito seja contemplado, através de pequenas adaptações no núcleo do sistema, que pode ser tanto implementado em *shaders*, o que transfere todo o esforço computacional para o hardware gráfico, como através de tecnologias ainda mais modernas como a CUDA (*Compute Unified Device Architecture*) [2] que já possui versões disponíveis para linguagem Java.

- **Opengl:**

Opengl é uma biblioteca gráfica que tem embutida em suas funções, algoritmos para criação de cenas e modelos 3D, cálculos de matrizes e manipulação das mesmas, além de recursos para exibição dos resultados em tela. As funcionalidades proporcionadas pela biblioteca foram importantes para o sistema, tanto para questões de exibição de resultados, quanto para a construção dos modelos sintéticos e no processo de reconstrução dos modelos.

- **JOGL:**

O projeto JOGL hospeda a versão de desenvolvimento do Java ™ Binding para a OpenGL ® API (JSR-231), e foi projetado para oferecer suporte a hardware de gráficos 3D para aplicações escritas em Java. JOGL fornece acesso total às APIs da especificação OpenGL 2.0, assim como quase todas as extensões do fornecedor, além de integrar-se com o AWT e Swing. É parte de um conjunto de tecnologias *open-source* iniciada pelo Game Technology Group da Sun Microsystems [3].

Uma das vantagens é pode tratar a OpenGL como se fosse orientada a objetos, pois para manipular a OpenGL e a GLU, é necessário instanciar objetos das classes GL e GLU.

- **Glut:**

  Glut é uma biblioteca fornecida junto do pacote da OpenGL, para criação de interfaces, como captura de eventos de *mouse* e teclado, cálculos de janelas na tela, matrizes de câmera e outras facilidades que estendem as funcionalidades da OpenGL. Ela foi utilizada como biblioteca auxiliar, facilitando muito os cálculos e reconstrução dos modelos. Vale lembrar que ela não foi utilizada para a construção da interface do sistema.

- **Compilador NetBeans:**

  Esse compilador foi escolhido por ser o de maior conhecimento entre os membros da equipe, além de ser gratuito. Nenhum outro motivo especial levou a escolha do mesmo. Como o código estará aberto para a comunidade científica, é aconselhável para quem desejar modificar e adaptar o sistema que utilize o compilador proposto.

- **Swing:**

  É um componente, que faz parte da JFC (Java Foundation Classes). Ela é uma API que tem o papel de fornecer uma interface gráfica para sistemas desenvolvidos em linguagem Java.

3.2 -Padrão de projeto

A idéia original de padrão de projeto (*design pattern*) surgiu com Christopher Alexander, que afirma: *"Cada padrão descreve um problema que ocorre repetidamente em nosso meio, e descreve uma forma de solução para esse problema, de tal forma que você pode usar esta solução um milhão de vezes, sem nunca fazê-lo da mesma maneira duas vezes"* [7].

O padrão utilizado neste projeto foi o *Model-view-controller* (*MVC*), que tem por objetivo separar a interface com o usuário, da parte lógica. Com isso é possível equipes desenvolverem em paralelo uma interface da aplicação e outra a parte lógica. O MVC pode ser considerado como um padrão composto, pois a sua implementação depende de outros padrões, por exemplo:

- A comunicação com uso de eventos utiliza do padrão Observer.
- O acesso ao controle depende do padrão Singleton.

O *Modelo-visão-controle* como o próprio nome diz, molda uma aplicação em três camadas (Figura 4). A primeira camada, o *Modelo*, representa os objetos ou dados do problema abordado. O *Modelo* representa um estado persistente da aplicação ou as informações do sistema. A *Visão*, a segunda camada do padrão, é utilizada como interface de visualização com o usuário, através da qual as informações do modelo estão dispostas, permitindo que o usuário defina as suas ações. O *Controlador*, por sua vez, implementa a interatividade, através do processamento das ações tomadas pelo usuário e atualização do modelo. Ele possui uma ligação bidirecional entre o *Modelo* e a *Visão* [10].

Abaixo são enumeradas algumas das principais vantagens de se utilizar o padrão MVC:

- Permite o gerenciamento de múltiplos visualizadores usando o mesmo modelo.
- É muito simples incluir novos clientes.
- É possível realizar o desenvolvimento em paralelo para o *modelo*, *visualizador* e *controle*, pois são independentes.

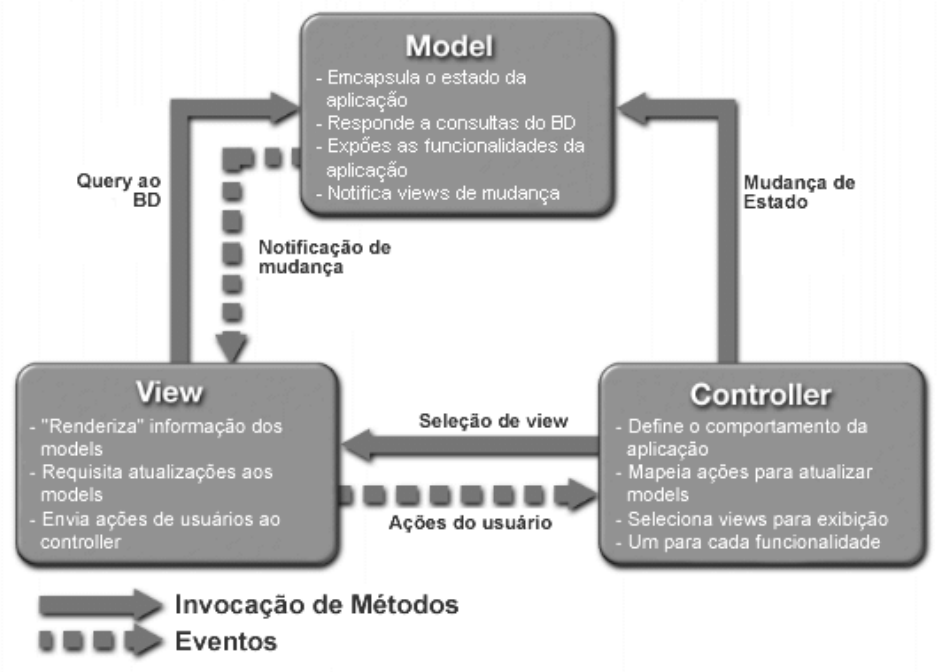


Figura 4: Estrutura do MVC. Figura extraída de [16].

Na implementação das classes de controle foi utilizado o padrão *Singleton*, no qual se deseja restringir o número máximo de instâncias de uma classe a somente uma. Caso

objetos diferentes (ex.: várias janelas) queiram usar uma instância da classe de controle, eles irão compartilhar somente a mesma instância da classe.

Para que isso funcione é necessário que a classe do padrão *Singleton* tenha um construtor privado, impedindo que ela seja instanciada diretamente, e que tenha um método estático que retorne sempre a mesma instância desta classe. Este método estático torna, automaticamente, o *Singleton* num objeto global.

A grande vantagem desta implementação no controle é que é possível centralizar as ações do controle a um único objeto. A figura 5 mostra a estrutura básica para que uma classe siga este padrão.

```
public class Controle{
        private static Controle  controle;
        private Controle(){}
        public static Controle get(){
                if(controle==null){
                        controle=new Controle();
                }
                return controle;
        }
}
```

Figura 5: Classe no padrão singleton.

Na aplicação também foi utilizado o padrão DAO, que é uma camada para persistência de dados. Toda a lógica de gravação e obtenção da informação fica nessa camada, não importando para o restante do sistema qual a estrutura dos dados persistentes, se é XML, XLS ou banco de dados, ou seja, a manipulação da informação é feita de forma transparente.

3.3 -Casos de uso

Os casos de uso abaixo, explicam as interações das duas partes mais importantes do sistema. Como a ferramenta é dividida em geração de dados de entrada e reconstrução do

modelo através dos dados de entrada, julgou-se mais propício a apresentação apenas as descrições da Figura x e Figura y, onde está presente o diagrama.

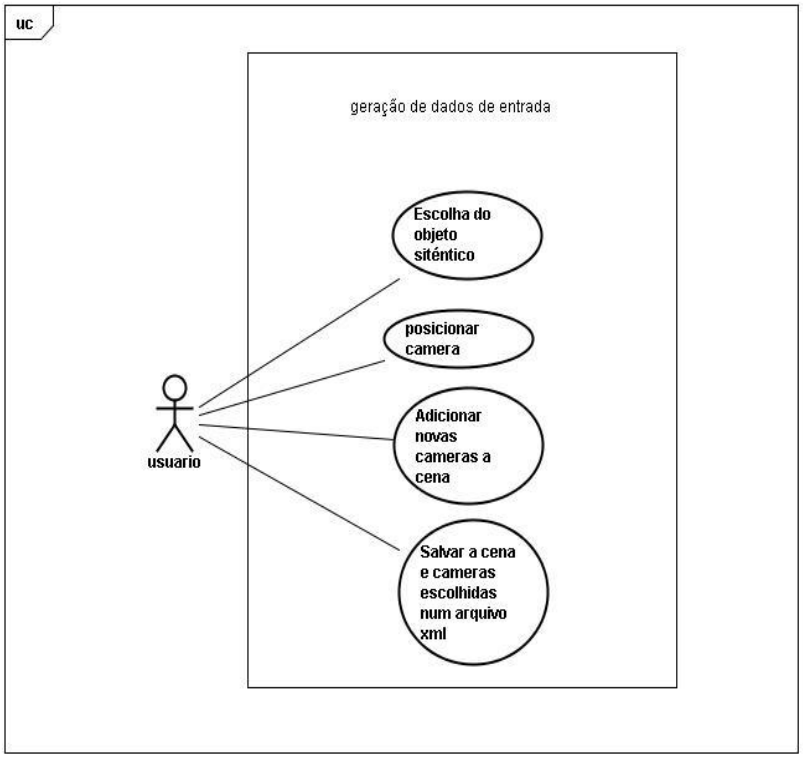


Figura 6: Geração de dados de entrada

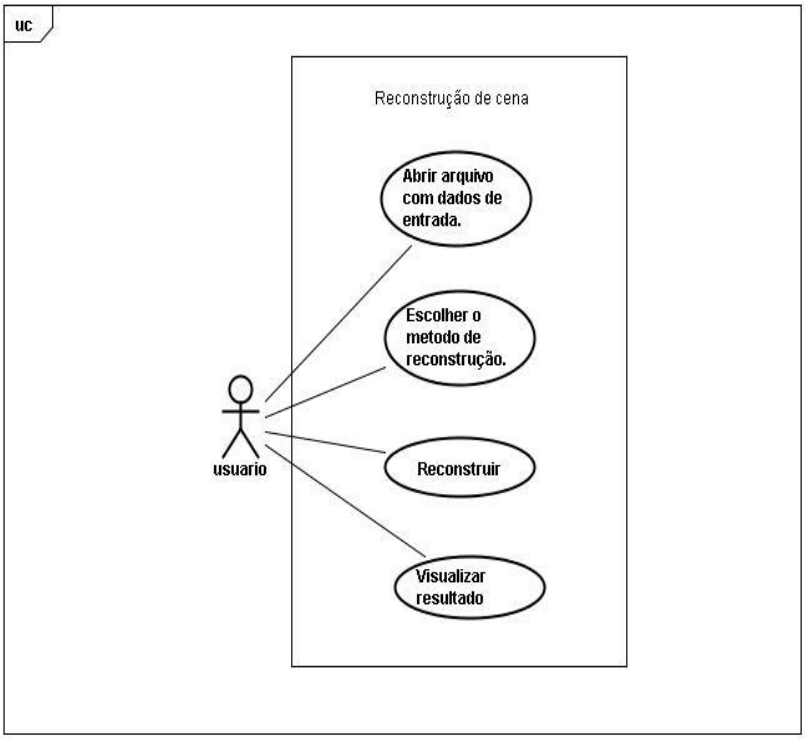


Figura 7: Reconstrução de cena

Gerando modelo (entrada de dados do sistema)

Gerar modelo é ação do sistema onde o usuário deseja gerar os dados que servirão de entrada, para posteriormente efetuar a reconstrução. Esses dados gerados contêm o conjunto de imagens geradas juntamente com um arquivo XML, contendo os dados das matrizes necessárias para a reconstrução.

Cenário principal: gerando modelo com sucesso.

1. O cliente escolhe um objeto 3D, a partir do qual ele vai gerar o modelo.
2. O sistema renderiza o objeto escolhido.
3. O cliente especifica os ângulos da câmera em torno do eixo X, Y, Z.
4. O sistema rotaciona o objeto.
5. O cliente adiciona uma câmera, e volta ao passo três, até que tenha pelo menos quatro câmeras adicionadas.
6. O cliente determina que o sistema salve o modelo.
7. O sistema salva todas as câmeras adicionadas, com informações das matrizes de projeção, *viewport* e *modelview*, juntamente com as imagens capturadas.

Reconstruindo cena

Reconstruir cena é a ação onde o usuário deseja, através dos dados de entrada, obter uma cena 3D gerada a partir do conjunto de imagens fornecidas e dos dados das matrizes contidas no arquivo XML. No final, o resultado obtido pode ser visualizado através da janela de exibição e com auxílio das ferramentas de rotação e aproximação (zoom) dos modelos.

Cenário principal: reconstruindo cena de um modelo existente.

1. O cliente deve escolher o modelo gerado.
2. O sistema valida o modelo (*extends - Validar modelo*).
3. O cliente define o método de reconstrução de cena.
4. O cliente manda iniciar o processo.
5. O sistema reconstrói a cena.

Cenário alternativo: reconstruindo cena de um modelo não existente.

1. O cliente deve gerar um modelo (*includes- Gerar modelo*).
2. Continua a partir do caso passo um do cenário principal.

Valida modelo

1. Sistema verifica se existe erro no arquivo do modelo.
2. Sistema informa ao cliente caso o arquivo se encontre corrompido.

3.4 -Diagrama de Classes

O diagrama de classes é um artefato muito importante, pois facilita o entendimento do sistema, além de permitir uma visão estática do mesmo, podendo ser elaborado de forma conceitual ou de implementação. Na figura 8, é apresentado um diagrama conceitual, para facilitar o entendimento do problema e mostrar como ele foi modelado em alto nível de abstração.

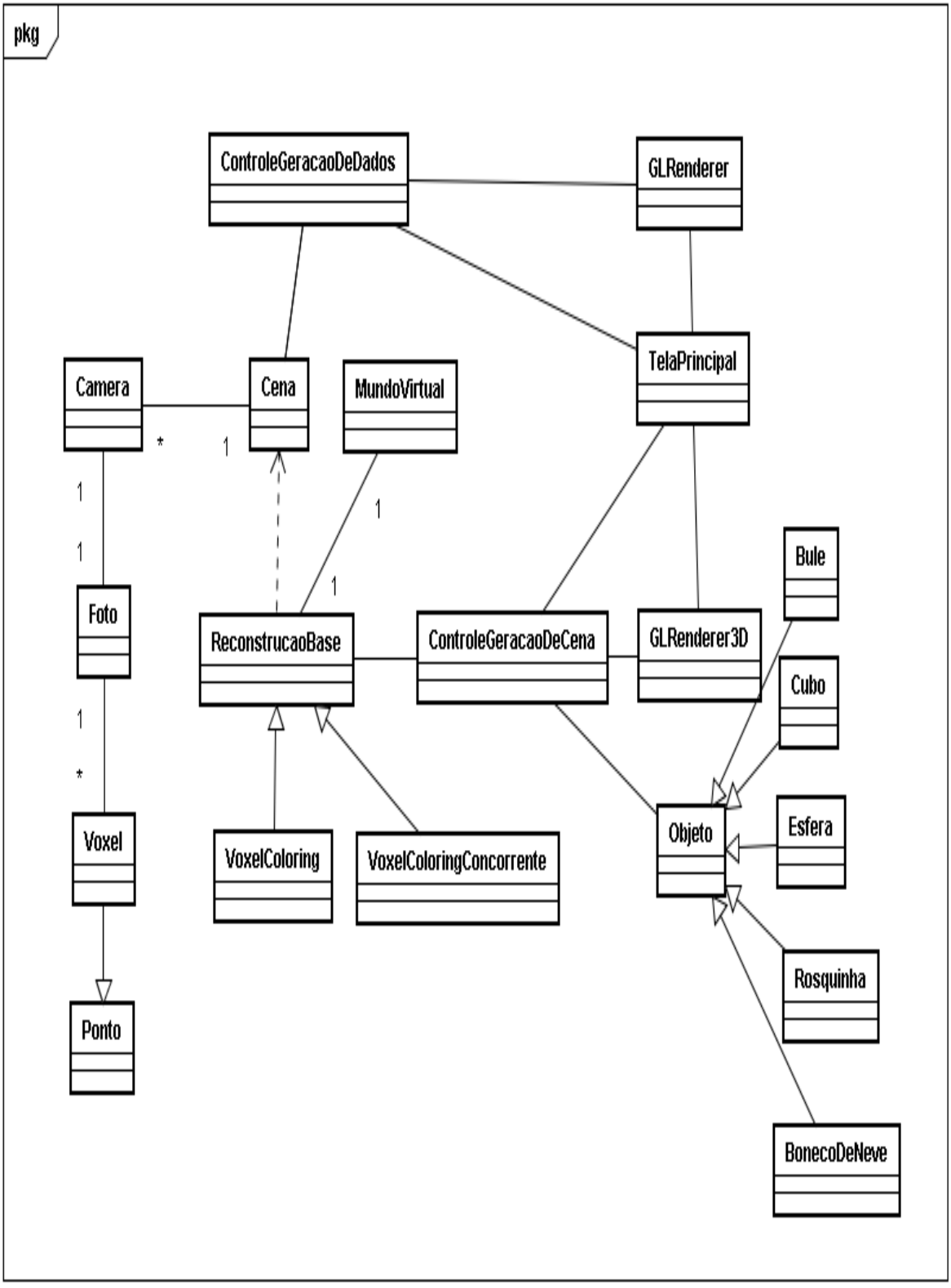


Figura 8: Diagrama de classes.

Com relação às classes envolvidas, podemos complementar o diagrama de classes com as seguintes informações:

Ponto: esta é uma classe do modelo que armazena basicamente os atributos *cor* e as coordenadas de *x* e *y*, além de disponibilizar os métodos *get*( ) e *set*( ).

Voxel: esta classe é uma especialização da classe *Ponto* e possui uma coordenada adicional para o eixo *z*. Esta classe é essencial para se descrever as estruturas tridimensionais.

Foto: classe que tem como função armazenar a imagem capturada pela câmera e também descrever os pixels que já foram visitados numa foto, através de suas instâncias que atuam como matrizes de visibilidade, durante o processo de reconstrução.

Câmera: representa a visão da cena de um determinado ponto de vista. Para representar este estado, a classe *Câmera* possui algumas matrizes sendo elas a de *viewport*, a *modelview* e a *projection*, além de possuir uma referência para uma instância da classe *Foto*.

Cena: representa uma coleção de vistas de ângulos diferentes da cena.

Mundo virtual: classe associada à *ReconstrucaoBase* formada por uma coleção de voxels consistentes, os quais irão ser renderizados formando o modelo da cena reconstruída.

ReconstrucaoBase: é uma classe abstrata para reconstrução de cena. Possui como único atributo o *mundoVirtual* e como método abstrato o *construir* (no qual é implementado o algoritmo *Voxel Coloring* ou *Space Carving*).

VoxelColoring: É uma especialização do *ReconstrucaoBase*, contendo o algoritmo básico do *Voxel Coloring*.

VoxelColoringConcorrente: classe análoga a *Voxel Coloring*, com a diferença de que ela utiliza *threads* no processo de reconstrução.

ControleGeracaoDeDados: controlador que faz a ligação entre a camada de modelo e a visão (tela de geração de entrada de dados). Tem como principais atributos o *modeloDeEntrada*, que é o objeto a ser desenhado, uma instância de *cena* e as variáveis para rotação *eixoX*, *eixoY* e *eixoZ*. Disponibilizar os métodos *addCamera* (*GL gl*), *salvaCenaEmXML* (*File fileXml*) *e desenha* (*GL gl, GLUT glut, GLAutoDrawable drawable*).

ControleGeracaoDeCena: este controlador tem como função ligar a *view* (de reconstrução de cena) ao modelo, transferindo parâmetros especificados pelo usuário para que seja executado o algoritmo de reconstrução.

GLRenderer, GLRederer3d: representam os componentes OpenGL sendo um para visualização da instância da classe *Objeto* e outro pra visualização da classe *MundoVirtual*, respectivamente. Dentre os principais métodos estão o *init* ( ), *reshape* ( ) e *display* ( ) para controlar a OpenGL.

TelaPrincipal: classe para a edição da camada de visão, através da qual haverá a interação com o usuário. Nela é possível criar e manipular qualquer componente da tela principal, sendo compostos de painéis *Swing*, canvas OpenGL, menus e abas, além de tratar eventos gerados pela ação do mouse e teclado.

Objeto: classe abstrata que representa um objeto sintético qualquer no componente do OpenGL. Seu único método implementado é o *desenha* (*GL gl, GLUT glut, GLAutoDrawable drawable*).

Bule, Cubo, Esfera, Rosquinha, Bonecodeneve: estas classes são especializações da classe objeto, com implementações próprias do método desenha, para auxiliar no processo de geração de dados sintéticos de teste.

3.5 -Entrada de dados

Para que o sistema de reconstrução de cena 3D funcione é necessário que exista uma cena válida, armazenada em memória secundária, que serve como entrada de dados. Nesta seção será explicado como foi implementada a sua geração.

Para que exista uma cena é preciso que se tenha um conjunto de câmeras e, para cada uma delas, haja uma foto e três matrizes (*viewport*, *modelview* e *projection*), que foram explicadas no capítulo 2.

Para a geração da cena o sistema segue alguns passos:

Módulo de importação de arquivos .obj :como uma alternativa a quantidade limitada de objetos disponíveis para reconstrução, foi elaborado um módulo separado que permite a importação de arquivos *.obj, que é um formato presente nos principais softwares de modelagem 3D como, por exemplo, Blender , 3d Studio Max e Maya, permitindo que o

software use como entrada objetos complexos, tanto em seu formato, quanto na variação de cores e textura. Este módulo foi criado a partir da biblioteca objimport.zip [5].

Captura de imagens da tela: O processo de captura de tela é fundamental, pois ele é um atributo importante da câmera. A partir dessa necessidade criou-se a classe *Imagem.java* que possui métodos para carregar, salvar, consultar pixels e o mais importante, disponibilizar o método para capturar uma imagem do canvas OpenGL. Este último foi criado a partir do conhecimento do método da biblioteca OpenGL:

```
gl.glReadPixels(0,0,width,height,gl.GL_RGBA,gl.GL_UNSIGNED_BY,framebys );
```

Figura 9: Código para captura de pixel da tela.

O glReadPixels lê os pixels do framebuffer de uma área definida pela matriz de *viewport*.

Captura das matrizes OpenGL: como na ferramenta proposta neste trabalho, o foco está na reconstrução de objetos sintéticos, e não objetos obtidos de fotos reais, é fundamental capturar as matrizes OpenGL, no momento da adição de câmeras, pois elas guardam os estado do mundo naquele exato momento em que foi tirada uma foto da cena sintética, fornecendo condições para que se possa fazer um mapeamento entre os voxels no espaço de reconstrução e as imagens, o processo de captura de matrizes é demonstrado na figura 10.

```
//instancia as matrizes
int matrizViewport[]=new  int[4];
Double matrizModelview[]=new Double[16],matrizProjecao[]=new double[16];
//captura as matrizes corrente do mundo OpenGL
gl.glGetIntegerv(gl.GL_VIEWPORT ,matrizViewport , 0);
gl.glGetDoublev( gl.GL_MODELVIEW_MATRIX ,matrizModelview , 0);
gl.glGetDoublev( gl.GL_PROJECTION _MATRIX ,matrizModelview , 0);
```

Figura 10: Código para obtenção das matrizes opengl.

Serialização das câmeras da cena: a serialização é feita em XML, facilitando, desta forma, o entendimento e organização dos dados. Outro fator importante, que levou a adoção de

XML foi à simplicidade de gerenciar a persistência com base na utilização da API Java, que já disponibiliza as classes XMLEncoder.java e XMLDecoder.Java para a serialização e desserialização de objetos, respectivamente. O arquivo XML segue o formato abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_16" class="java.beans.XMLDecoder">
 <object class="org.modelo.Cena">
  <void property="cameras">
   <void method="add">
    <object class="org.modelo.Camera">
     <void property="eixoX">90.0</void>
     <void property="eixoY"> 0.0 </void>
     <void property="eixoZ">0.0 </void>
     <void property="foto">
      <object class="org.modelo.Foto">
       <void property="filename">
C:\Users\Wagner\Documents\modelos_prontos\buledecima\screenShot_90.0_0.0_0.0.png
       </void>
      </object>
     </void>
     <void property="matrizModelview">
      <void index="0"> 1.0  </void>
      <void index="5"> 1.2167964413833943E-8</void>
      <void index="6"> 1.0</void>
      <void index="9"> -1.0</void>
      <void index="10">1.2167964413833943E-8</void>
      <void index="15">1.0</void>
     </void>
     <void property="matrizProjecao">
      <void index="0">0.0714285746216774</void>
      <void index="5">0.0714285746216774</void>
      <void index="10">-0.0714285746216774</void>
      <void index="12">-0.0</void>
      <void index="13">-0.0</void>
      <void index="14">-0.0</void>
      <void index="15">1.0</void>
     </void>
     <void property="matrizViewport">
      <void index="2">512</void>
      <void index="3">512</void>
     </void>
    </object>
   </void>
  </void>
 </object>
</java>
```

Figura 11: Estrutura do arquivo XML do projeto

Como vemos na figura 11, o arquivo XML correspondente a serialização de um objeto do tipo Cena, com todos os seus atributos, exceto as imagens associadas às câmeras, que são definidas como transientes, já que não faz sentido persistir uma imagem em XML, e sim somente guardar somente uma referência para ela.

3.6 -Reconstrução de cenas 3D

Nesta seção será abordado o processo de reconstrução de cenas 3D, suas fases e como elas foram implementadas. As fases que compõem o processo de reconstrução são:

1. Definição do mundo virtual: o primeiro passo, para reconstrução de cenas, é a definição do tamanho do mundo virtual (figura 12) e a quantidade de voxels em cada dimensão. O seu valor *default* é 32, ou seja, o tamanho do mundo é 32x32x32 voxels, mas esta quantidade pode ser escolhida pelo usuário do sistema, Ela determina a definição do objeto a ser construído: quanto maior o numero de voxels, maior a definição do modelo reconstruído.

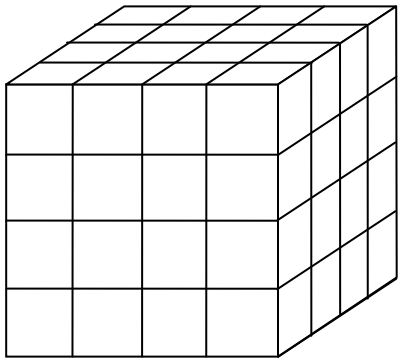


Figura 12: Mundo virtual definido com quatro voxels por aresta.

2. Ordem de acesso dos voxels: como o algoritmo de *voxel coloring* implementado nesse projeto trata a visibilidade, ou seja, só reconstrói os voxels visíveis as câmeras, isto é, somente a casca do objeto, é necessário que a reconstrução siga certa ordem para garantir a visibilidade.

   Considerando que as fotos de entradas foram tiradas de cima do objeto, como mostra a figura 13, é preciso que o processo de reconstrução percorra os voxels num plano horizontal, sendo que os planos horizontais superiores devem ser reconstruídos primeiro. Logo, a reconstrução de uma cena ocorre de cima para

baixo, nunca acessando um voxel de um plano qualquer antes que todos os voxels dos planos acima sejam classificados.

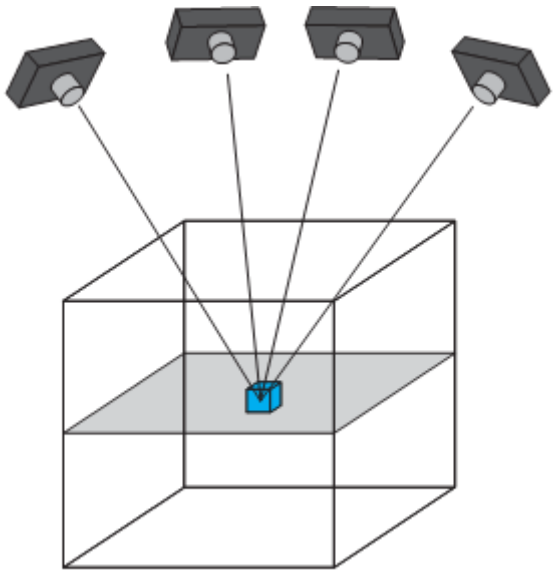


Figura 13: Posição das câmeras.

3. Mapeamento: O mapeamento dos objetos nas imagens na OpenGL pode ser feito de duas formas: obter as coordenadas no espaço da imagem a partir das coordenadas no espaço do objeto, ou o caminho inverso. Para fazer estas projeções a OpenGL disponibiliza duas funções: *gluProject* e *gluUnProject*. Nos algoritmos de reconstruções de cenas, mapeia-se cada voxel do mundo virtual em todas as fotos para determinação da foto-consistência. Este mapeamento é feito com o gluProject.

```
  public static boolean objetoParaTela(GLU glu,Voxel ponto,double[] modalView,
        double[] proj,int[] view,Voxel voxel)
  {

    int offSet=0;
    double posicaoDeTela[]=new double[3];
    boolean converteu=glu.gluProject(voxel.getX(),voxel.getY(),voxel.getZ()
          ,modalView,offSet,proj,offSet,view,offSet,posicaoDeTela,offSet);

    ponto.setX(posicaoDeTela[0]);
    ponto.setY(posicaoDeTela[1]);
    ponto.setZ(posicaoDeTela[2]);

    return converteu;
  }
```

Figura 14: Código para mapeamento do mundo virtual nas câmeras.

O trecho de código acima indica onde um voxel é mapeado em uma foto, sendo necessário fazer o mapeamento de cada voxel em todas as fotos.

4. Classificação do voxel: Um voxel pode ser classificado em foto-consistente, não-foto-consistente e indefinido. A princípio todos os voxels do mundo são indefinidos, e através do método *boolean isConsistente (Voxel voxel, Cena cena)*, verificamos se o voxel é não-foto-consistente ou foto-consistente. Na figura 15 indicamos o código desse método.

```
public boolean isConsistente(Voxel voxel, Cena cena) {
    Boolean consistente = true;
    List<Camera> cameras = cena.getCameras();
    List<Color> colors = new ArrayList<Color>();
    List<PontoFoto> listPontoFoto = new ArrayList<PontoFoto>();
    for (Camera camera : cameras) {
        Voxel ponto = new Voxel();
        Mapeamento.objetoParaTela(glu, ponto, camera.getMatrizModelview(),
camera.getMatrizProjecao(),
                camera.getMatrizViewport(), voxel);
        int x = (int) Math.round(ponto.getX());
        int y = (int) Math.round(ponto.getY());
        if (((0 <= x) && (x < largura)) &&
                ((0 <= y) && (y < altura)) &&
                ((-1 < ponto.getZ()) && (ponto.getZ() < 1))) {
            if (!camera.getFoto().getVisitado(x, y)) {
                int corRGB = camera.getFoto().getImagem().getRGB(x, y);
                Color color = new Color(corRGB);
                if (color.getRGB() != Color.BLACK.getRGB()) {
                    colors.add(color);
                    listPontoFoto.add(new PontoFoto(ponto, camera.getFoto()));
                } else {
                    return false;
                }
            }
```

Figura 15: Método para verificação de consistência de um voxel.

Um passo importante neste algoritmo é a verificação se o ponto mapeado é visível. Para isso, associada a cada foto existe uma matriz de voxels já visitados. Caso um ponto seja mapeado num voxel já visitado ele não é considerado visível. Após definido um voxel como visível, a cor em sua projeção é comparada com a cor de fundo, e se estas forem iguais, tal voxel é descartado imediatamente.

5. Calculo da coloração dos voxel: Após determinar um voxel como foto-consistente, ou seja, que ele deve aparecer na reconstrução, deve se fazer o cálculo da coloração desse voxel, porque um mesmo voxel da cena pode ser mapeado em várias

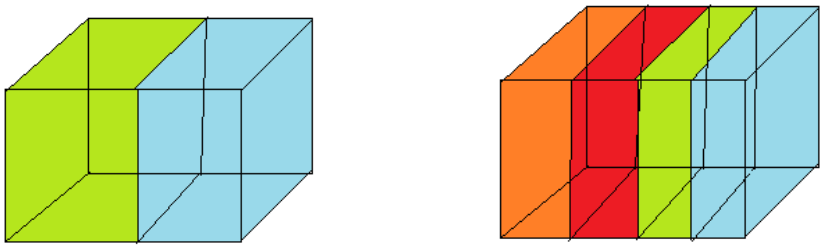
imagens, e em cada uma delas, com um tom ligeiramente diferente. Para resolver esse problema calculamos a média aritmética desses tons.

6. Calculo do desvio padrão: Como visto anteriormente um voxel pode ser visto por várias câmeras diferentes, mas suas cores não podem ser muito diferentes entre si. Por este motivo é calculado o desvio padrão da lista de cores que foram encontradas no mapeamento. Se o resultado for menor que um valor definido pelo usuário, este voxel é aceito, senão é taxado como inconsistente.

3.7 -Paralelização do algoritmo *Voxel Coloring*

O maior desafio para o problema de reconstrução de cenas a partir de imagens é reproduzir uma cena com boa qualidade em um tempo aceitável. Para reduzir o tempo de execução do algoritmo e utilizar os vários processadores dos computadores atuais, apresentaremos abaixo a paralelização do algoritmo *voxel coloring*.

A figura 16 mostra o fracionamento do mundo virtual para um conjunto de *threads*, sendo cada uma delas responsável pela reconstrução de parte do objeto. A divisão deve ocorrer através de cortes verticais, para que um bloco não fique totalmente dependente ao outro na questão da visibilidade.



.

Figura 16: divisão do mundo virtual para 2 e 4 theards respectivamente.

Como vimos na secção 3.6 existe uma ordem de acesso aos voxels, o que implica que uma cena tem que ser reconstruída da fatia mais próxima ao conjunto de câmeras para a mais distante, nunca acessando um voxel de um plano qualquer antes que todos os voxels

dos planos mais próximos sejam processados. Se considerada está hipótese o algoritmo de visibilidade funcionará sem maiores problemas.

Entretanto, no algoritmo *multi-thread* é necessário que as mesmas sejam sincronizadas para que problemas de visibilidade não ocorram.

Dado que temos um problema de sincronização, foi necessário utilizar de um mecanismo de proteção para manter certa ordem na execução do código. Para atender tal função escolhemos a *barreira*.

Uma barreira permite sincronizar várias *threads*[9] em uma linha de código específica. Ela é usada especialmente quando todas as *theards* precisam terminar uma etapa antes que as demais etapas prossigam.

Uma thread automaticamente entra em estado de espera quando atinge uma barreira. Quando a ultima *thread* alcança a sua barreira, todas elas recebem uma notificação para sair do estado de espera e ir para o estado pronto, para que possam seguir sua execução juntas. Exatamente isso que deve acontecer nesse sistema, toda vez que uma thread terminar um plano no eixo y, esta deve entrar em estado espera, até que todas as outras terminem também aquele plano e libere todas elas. Portanto a barreira cuida exatamente do nosso problema.

```
public void run() {
        …

        for (int y = tam; y > -tam; y--) {
                for (int x = -tam; x < tam; x++) {
                        for (int z = -tam + planoInicial; z < -tam + planoFinal; z++) {
                           classificaVoxel(mundoVirtual.getVoxel(x,y,z));

                        }
                }
                barreira.await();// ponto de parada
           }
        …
```

Figura 17: algoritmo de sincronização da theard.

A figura 17 ilustra o funcionamento da barreira, no qual existem três laços aninhados. Os dois internos percorrem os eixos *x* e z do mundo virtual, ou seja, varrem um plano horizontal, e o mais externo percorre o eixo *y*, se movimentando de cima para baixo.

Para que exista uma sincronização entre threads no nível do plano horizontal é colocada uma barreira antes da interação no eixo *y*.

4 - Ferramenta

4.1 -Introdução

Neste capítulo será mostrado o sistema desenvolvido (**Figura 18**), que tem como objetivo a geração de dados de entrada, que servirão como alimentação para o sistema de reconstrução, e para a visualização de resultados das cenas 3D.

Na seção 4.2, um resumo geral do ambiente de trabalho é exposto, através de um passo a passo da interação com o sistema, utilizando imagens ilustrativas, mostrando como reconstruir um modelo novo, incluindo desde a geração dos dados de entrada até a reconstrução da cena 3D total. Ao final, é apresentado um resumo geral das vantagens e desvantagens do sistema desenvolvido.

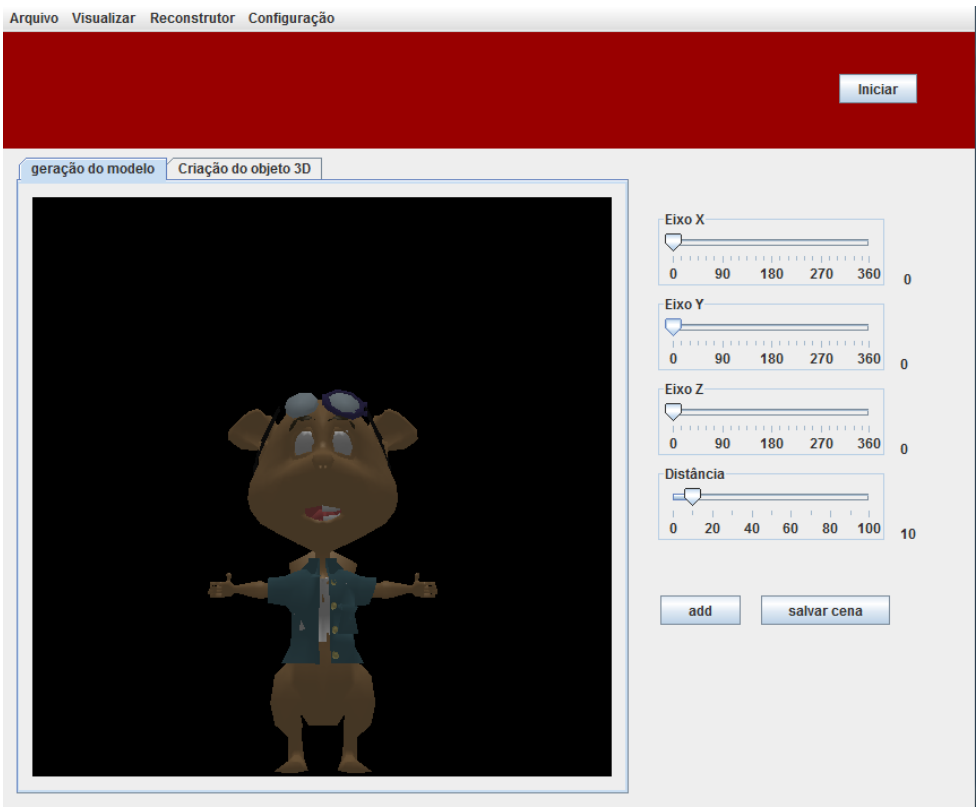


Figura 18: Tela principal da aplicação.

4.2 - Ambiente de trabalho

4.2.1 -Visão geral

A ferramenta aqui descrita contém uma interface baseada em janelas, menus e abas, que são elementos agradáveis e de conhecimento comum para qualquer usuário, por existirem muitos softwares no mercado que utilizam a mesma base de interação. Todos os eventos gerados pela interface têm como entrada o *mouse*, e foram programados com a ajuda do componente *Swing*, descrito na seção de tecnologias.

A figura 19 ilustra como a janela principal foi dividida em quatro áreas diferentes, correspondendo cada número a uma área importante da janela. Neste ponto, serão explicadas superficialmente as seções destacadas, deixando para outro tópico as funções mais importantes, as quais requerem uma descrição mais detalhada.

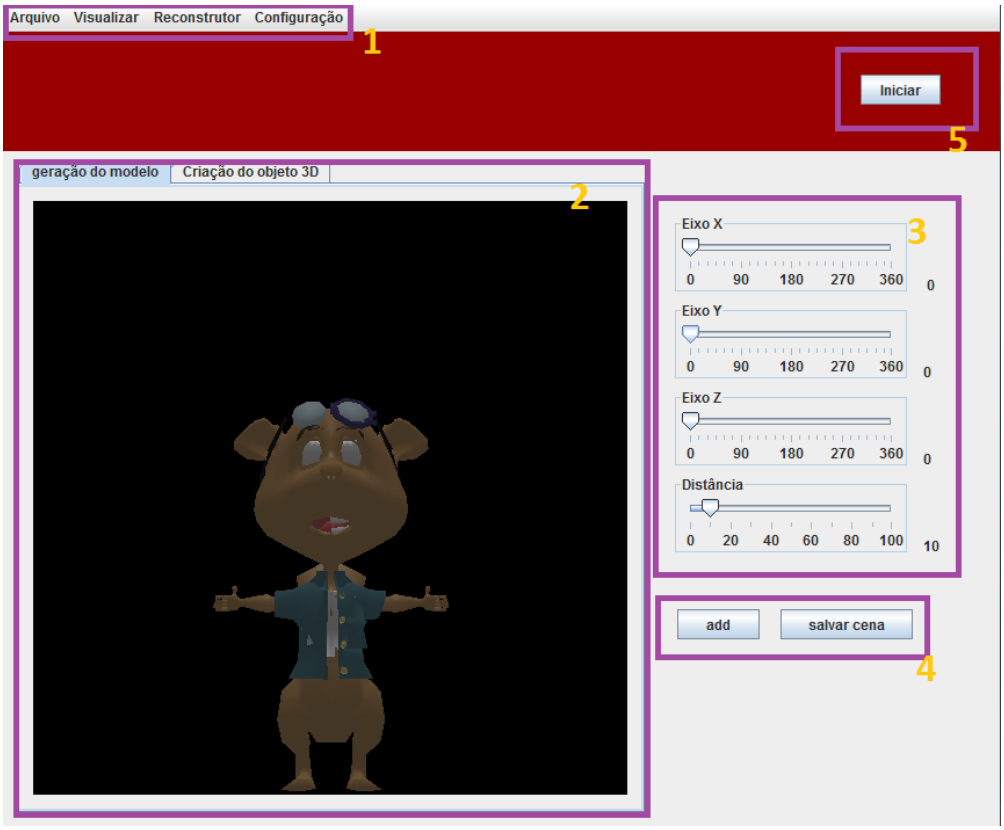


Figura 19: Visão geral do sistema.

O ambiente de trabalho, esta dividido em quatro partes:

1. Barra de menus - Permite o acesso aos comandos da ferramenta. Cada um dos menus pode ser expandido de modo a apresentar a opção correspondente e de desejo do usuário, organizando as ações mais freqüentes.
2. Abas para geração de modelo / criação de objetos 3D - Permite alternar o *software* entre as duas etapas mais importantes para utilização do sistema. A primeira etapa visa gerar a entrada para o sistema, construindo um modelo referencial usado para a extração das imagens de perspectivas diferentes. Na segunda aba, correspondente ao segundo passo do sistema, é feita a reconstrução do modelo, que após o resultado final, torna-se um visualizador para o resultado obtido.
3. Barra de rolagem - Permite a rotação do objeto 3D em torno do seu centro e a modifica a sua distância em relação à câmera, com o objetivo de melhorar a visualização do modelo, tanto na aba de geração de modelo, quanto na aba de criação de objetos 3D.
4. Botões de controle de cena - Nesta área são exibidos dois botões, um com a função de adição de novas câmeras (*add*) e outro para salvar as imagens e arquivos que serão a entrada do sistema (salvar cena).
5. Botão iniciar – Este botão é exibido apenas quando um conjunto de dados de entrada é aberto. Ao clicar nesse botão é exibida a tela de configuração de *voxels* e desvio padrão.

4.2.2 -Gerando entrada de dados

Para gerar a entrada de dados do sistema é preciso uma seqüência de passos importantes, que serão explicitados a seguir.

Primeiramente, deve ser escolhido o tipo de modelo que se deseja reconstruir, clicando-se em visualizar e selecionando-se uma das cinco opções disponíveis: cubo, esfera, bule, rosquinha ou boneco de neve.

Após a seleção, o modelo escolhido é apresentado na janela da aba de Geração de modelo, como ilustrado na figura 20.

Figura 20: Item de menu visualizar.

Na área à direita da área de visualização, estão os três slides descritos anteriormente. Com eles é possível girar, em qualquer ângulo e eixo, o objeto apresentado.

O objetivo dessa funcionalidade é dar a possibilidade de se gerar diferentes vistas, a partir das quais as fotografias sintéticas serão obtidas. Então, uma orientação deve ser escolhida e o botão "*add*" clicado, adicionando-se assim uma câmera na posição e orientação selecionada.

O passo dois deve ser repetido até que seja considerada apropriada a quantidade de vistas que serão utilizadas para a reconstrução do modelo e, em seguida, deve-se clicar no botão "salvar cena". Experimentos apontaram que muitas câmeras não fazem tanta diferença na reconstrução, todavia um número excessivamente pequeno não é suficiente, em muitos dos casos, para a geração de um modelo tão consistente. O ideal é uma seleção de 3 até 8 perspectivas, rotacionando a posição das adições em apenas um eixo, no sentido de cobrir 360º graus do objeto através das imagens capturadas (Figura 21).

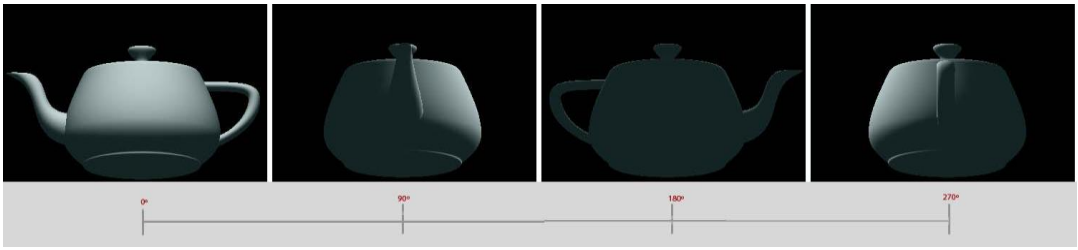


Figura 21: Item de menu visualizar.

Após apertar o botão salvar, uma janela de escolha de caminho será exibida. Selecione a pasta desejada e mande salvar. Os arquivos salvos na pasta contêm todas as fotos extraídas pelas câmeras adicionadas e mais um arquivo XML, contendo as informações das matrizes correspondentes à configuração das câmeras das imagens de referência.

Após a conclusão de todos os passos anteriores o conjunto de entrada de dados é criado, não sendo mais necessários os passos anteriores. Toda vez que for necessário reconstruir o modelo escolhido, basta abrir o arquivo XML junto com as fotos salvas.

4.2.3 - Utilizando o importador para gerar dados

O importador é uma opção para gerar reconstruções de modelos construídos em programa externos, como *3DStudio* e o *Blender*. As extensões dos arquivos importados devem estar obrigatoriamente em formato *obj* (Figura 22).

Clicando-se no menu Arquivo→Importar, uma janela da escolha de caminho de arquivo será aberta. Selecione o local onde se encontra modelo exportado no formato *obj*, e clique em abrir.

Figura 22: Menu Importar

Depois de concluída a localização do arquivo obj, e finalizado o processamento para importação, o objeto será exibido na aba geração de modelo, podendo ser visualizados com os *slides* localizados na esquerda da janela de visualização.

Para gerar os dados de entrada, são utilizados os mesmos passos do tópico anterior, e após a geração dos dados, não será mais necessária a etapa de importação para reconstruir o modelo.

4.2.4 - Reconstruindo cena 3D

Para reconstrução de um modelo, basta seguir os seguintes passos:

Passo 1 - Com o arquivo XML e o conjunto de imagens geradas anteriormente, juntos em uma mesma pasta, clique no menu “Arquivo” e em seguida “Abrir”(Figura 23).

Figura 23: Item de menu arquivo.

Passo 2 - Uma janela de escolha de caminho será apresentada, e então o caminho do arquivo XML deve ser selecionado e a opção “Abrir” clicada.

Passo 3 - Após escolhido o arquivo correto, um botão "Iniciar" aparecerá para o usuário clicar. Antes disso é preciso escolher o tipo de reconstrução que será feito, clicando no menu "Reconstrutor". A ferramenta apresenta três opções (Figura 24). São elas:

- *Voxel coloring*: utiliza o algoritmo de *voxel coloring* padrão sem nenhuma opção a mais.
- *Voxel Coloring Multi-Thread*: utiliza o algoritmo de *voxel* coloring com a vantagem de ser concorrente, gerando *threads* para resolver partes separadas da reconstrução.
- *Voxel Coloring* com Visibilidade: utiliza o algoritmo de *voxel coloring*, não concorrente, junto com o problema de visibilidade de *voxels*.

Figura 24: Item de menu reconstrutor.

Passo 4 - Após a escolha do método reconstrutor, deve-se clicar no botão "Iniciar" para começar a reconstrução. A ferramenta possui uma visualização em tempo real, que pode ser acompanhada na aba "Criação do objeto 3D".

É extremamente recomendado que, nessa etapa, apenas o sistema esteja executando no computador, pois este consome bastante do recurso de memória RAM e processamento.

Passo 5 - Concluída a reconstrução do modelo, é possível utilizar os *slides bars* para girar o modelo em torno dos eixos e verificar como ficou o resultado final.

4.2.5 - Menu configuração

Para facilitar na hora de testar e visualizar os objetos, foi criado um menu de configuração (Figura 25), que contém duas opções:

1. Desenhando a matriz de visibilidade: Essa opção permite que no momento da reconstrução de modelos com a opção de *voxel coloring* com visibilidade, a matriz de visibilidade seja impressa em forma de imagens em uma pasta separada. Isso facilita a depuração do sistema, caso voxels não tenham sido renderizados, devido a problemas no tratamento da visibilidade como, por exemplo, *aliasing* (problema de taxa de amostragem inferior as freqüências mais altas do sinal) nas matrizes de visibilidade.

2. Iluminação: Essa opção permite visualizar o modelo na aba de geração do modelo, com o efeito de iluminação ou sem o efeito de iluminação. Vale lembrar que, caso essa opção seja marcada, o reconstrutor levará em conta as cores da iluminação na reconstrução do resultado final.

Figura 25: Menu configuração

5 -Resultados

5.1 -Introdução

Neste capítulo será abordada a metodologia de teste utilizada para validar os resultados do algoritmo. Como o projeto trata de modelos e imagens, foi necessária a criação de uma metodologia que pudesse validar se o algoritmo estava realizando corretamente seu papel, através de estatísticas e comparações com os dados de entrada e seus respectivos modelos reconstruídos.

Em todos os testes, foi utilizado um ambiente com processador AMD Turion 64X2 de 2,00GHz, 3GB de memória RAM DDR2 de 533MHz, tendo como sistema operacional Windows 7 profissional.

Figuras dos dados de entrada e das reconstruções geradas através dessas informações são exibidas na seção dois do capítulo, com o objetivo de explicitar ao leitor os resultados visuais obtidos com o projeto.

Finalizando, na ultima seção, são feitas considerações finais sobre todos os testes e resultados obtidos pelo algoritmo de *Voxel Coloring* com e sem visibilidade. Além de dissertar sobre as metodologias de teste utilizadas.

5.2 -Metodologia de testes

Para atingir um teste satisfatório, para comparar os resultados gerados, foi escolhida uma metodologia qualitativa de cores e geometria das imagens retiradas a partir do modelo resultante.

Para o teste de cores, construimos um componente, que compara as cores dos pixels das imagens do dado de entrada com a cor dos pixels das imagens do modelo de saída, um a um, mapeando colorações nas mesmas posições. Após o processamento, o programa exibe algumas estatísticas que servem para a comparação de resultados.

No teste de geometria do objeto, o software compara as cores de fundo de imagem com as cores que são consideradas consistentes, formando uma silhueta de cor única para a área da imagem onde se encontra o modelo. Assim podem-se comparar formas geométricas com mais precisão. Após o processamento das imagens, dados estatísticos são

gerados para comparação das áreas brancas das imagens correspondentes de entrada com as extraídas do modelo resultante.

No teste de tempo de execução do algoritmo, consideramos apenas o tempo de processamento do algoritmo de reconstrução, sem levar em conta o as impressões em tela e nem o processo de renderização no OpenGl.

5.3 -Resultados gerados

Apresentando as figuras utilizadas como entrada no sistema, e as extraídas dos modelos finais, objetiva-se ilustrar ao leitor, as reconstruções realizadas através do algoritmo proposto, com um efeito comparativo visual. Para validar os resultados do sistema, são exibidos gráficos e tabelas comparativas de acertos.

Essa seção será subdividida em cada resultado obtido. Para cada um, foram utilizadas imagens de entrada extraídas por 8 câmeras rotacionadas em 45º graus no eixo *X*, posicionadas a 45º graus entre si no eixo *Y* e com o eixo *Z* sem rotação (0º graus).

5.3.1 -Resultado I

O exemplo a seguir, é a reconstrução do cubo com as faces com cores em gradiente. A figura 26 representa as imagens de entrada para o algoritmo e a figura 27, são imagens do mesmo ângulo do objeto após a reconstrução.

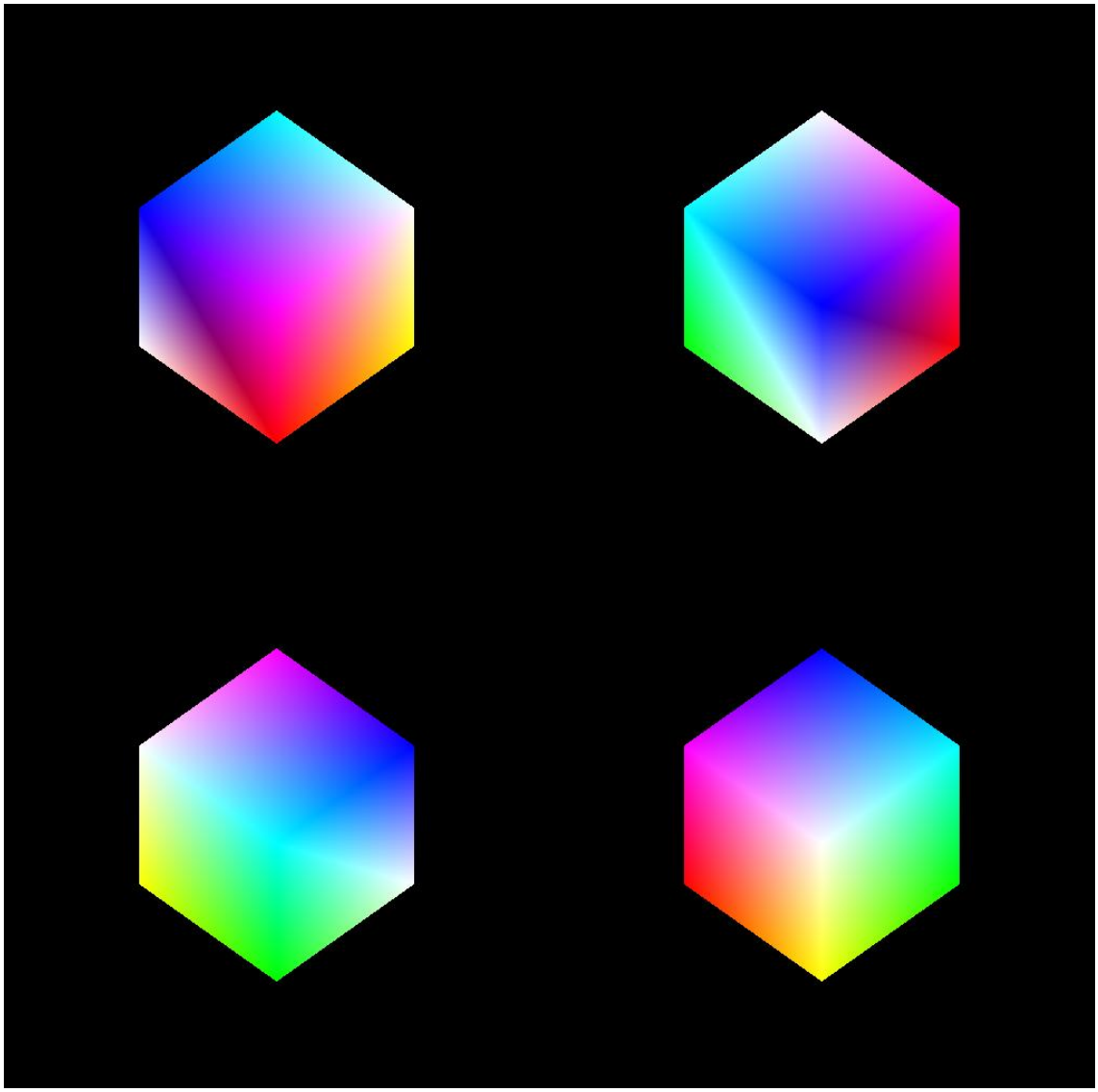


Figura 26: Resultado I - Conjunto de imagens de entrada

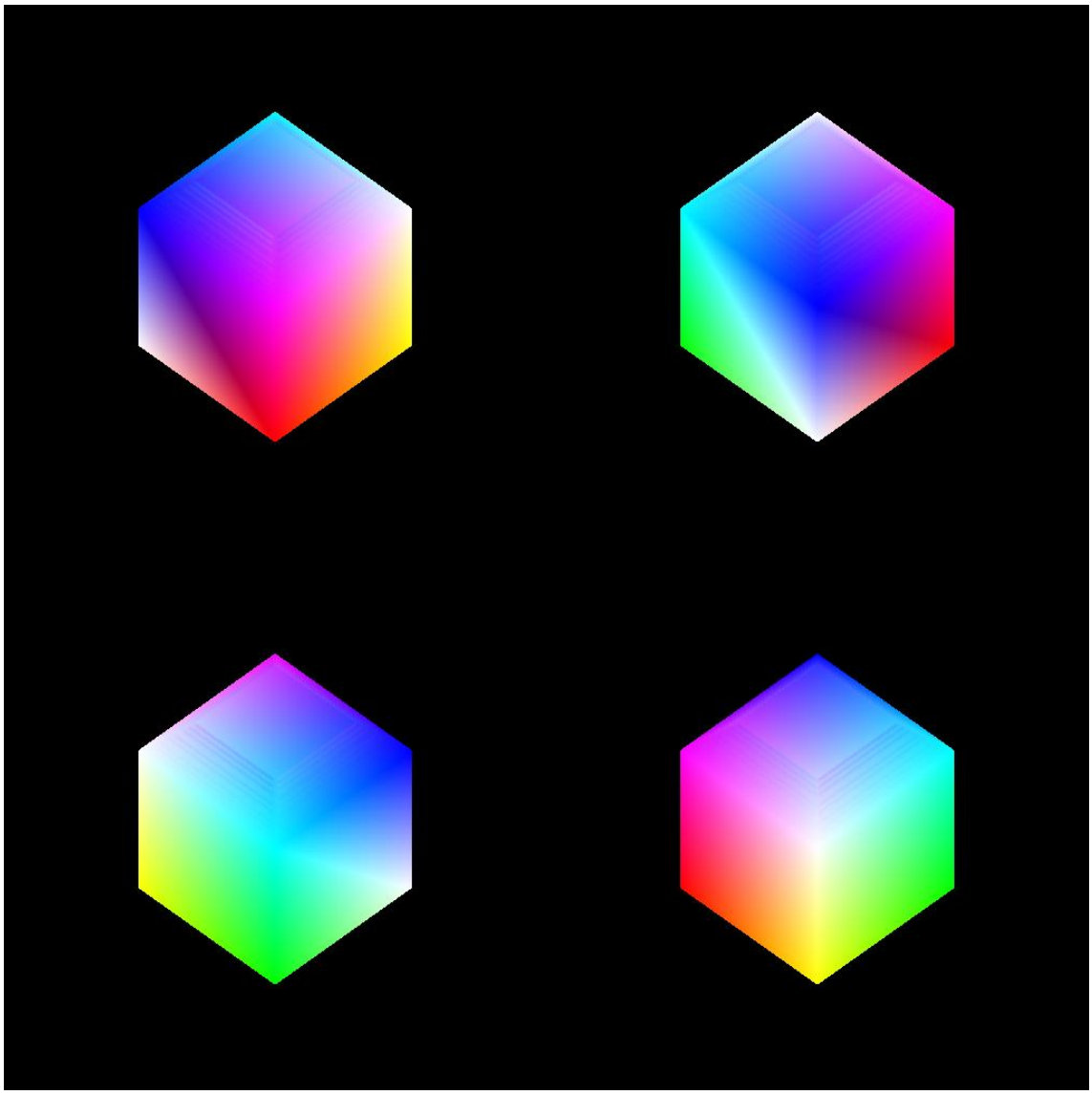


Figura 27: Resultados I - Conjunto de imagens da reconstrução gerada

Utilizando o programa de avaliação de cores, para validar o resultado de imagens correspondentes, foi construída a tabela 1:

Ângulo da imagem eixo Y	% de Cor correta	% de Geometria correta
0º	82,04	100,00
45º	78,24	99,67
90º	82,04	100,00
135º	78,25	99,67
180º	82,04	100,00

225º	78,25	99,67
270º	78,24	99,67
315º	82,04	100,00

Tabela 1: Resultado I- tabela de acertos

Fazendo uma média total do acerto de cores e comparando com a resolução de voxels escolhida para a reconstrução, o gráfico abaixo (figura 28) reflete a taxa de acerto de cores por resolução de voxels:

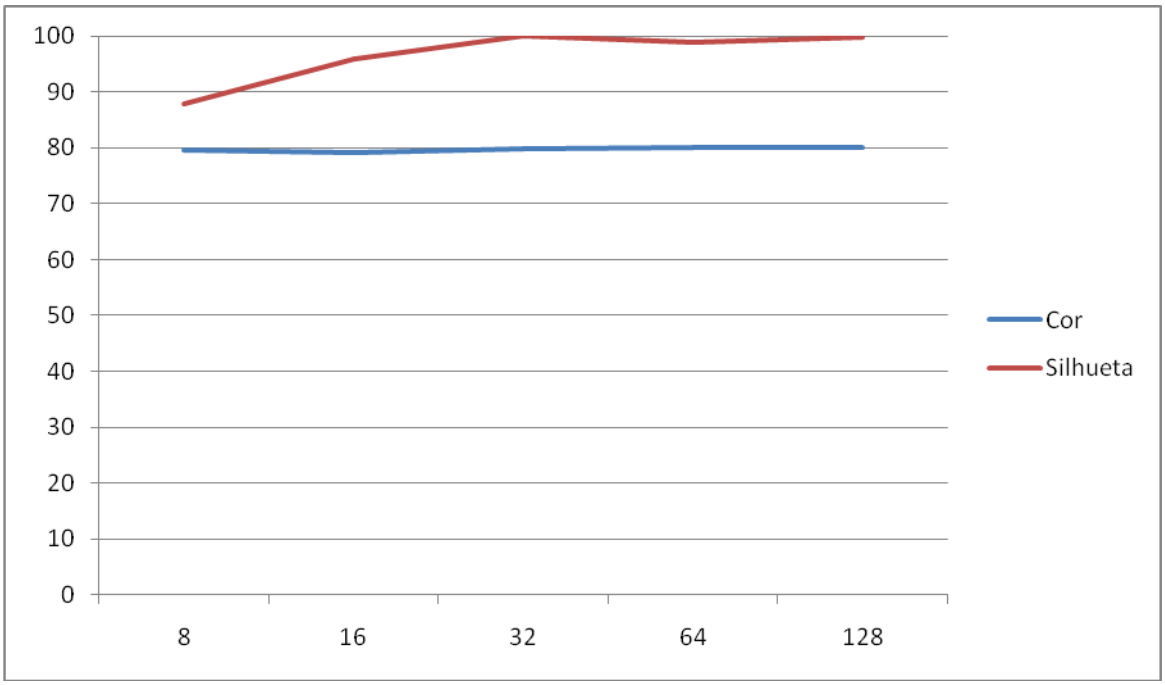


Figura 28: Resultado I - Gráfico Acerto x Quantidade de voxel por aresta.

De acordo com os resultados apresentados na tabela, o algoritmo não se comportou bem nas angulações onde são mostradas duas faces do cubo em gradiente. O motivo para esse problema ocorre pela da mistura de cores na região onde fica a borda do cubo, onde o algoritmo não consegue um resultado bom.

Analisando a taxa de acerto de cores e geometria, e comparando as imagens resultantes com as de entrada, concluímos que para esse exemplo, o sistema obteve um desempenho aceitável no quesito de cores e um desempenho excelente na geometria. Com esse exemplo podemos tirar a conclusão de que o *voxel coloring* trabalha bem quando não há uma variação tão intensa de cores.

5.3.2 -Resultado II

O exemplo a seguir, é a reconstrução do toróide de cor única com iluminação. A figura 29 representa um conjunto de imagens de entrada.

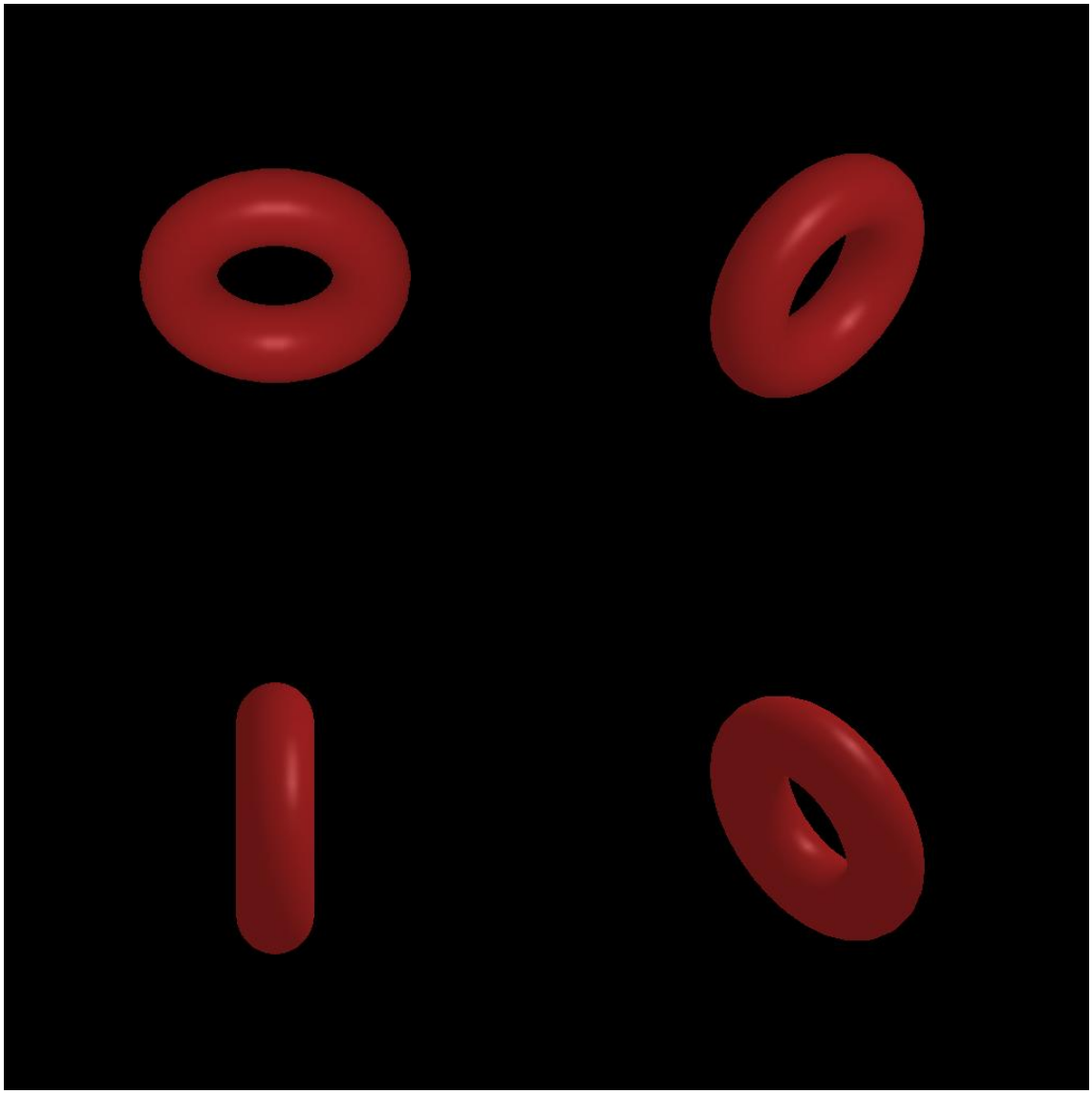


Figura 29: Resultado II - Conjunto de imagens de entrada

A figura 30 é um conjunto imagens extraídas do modelo após a reconstrução.

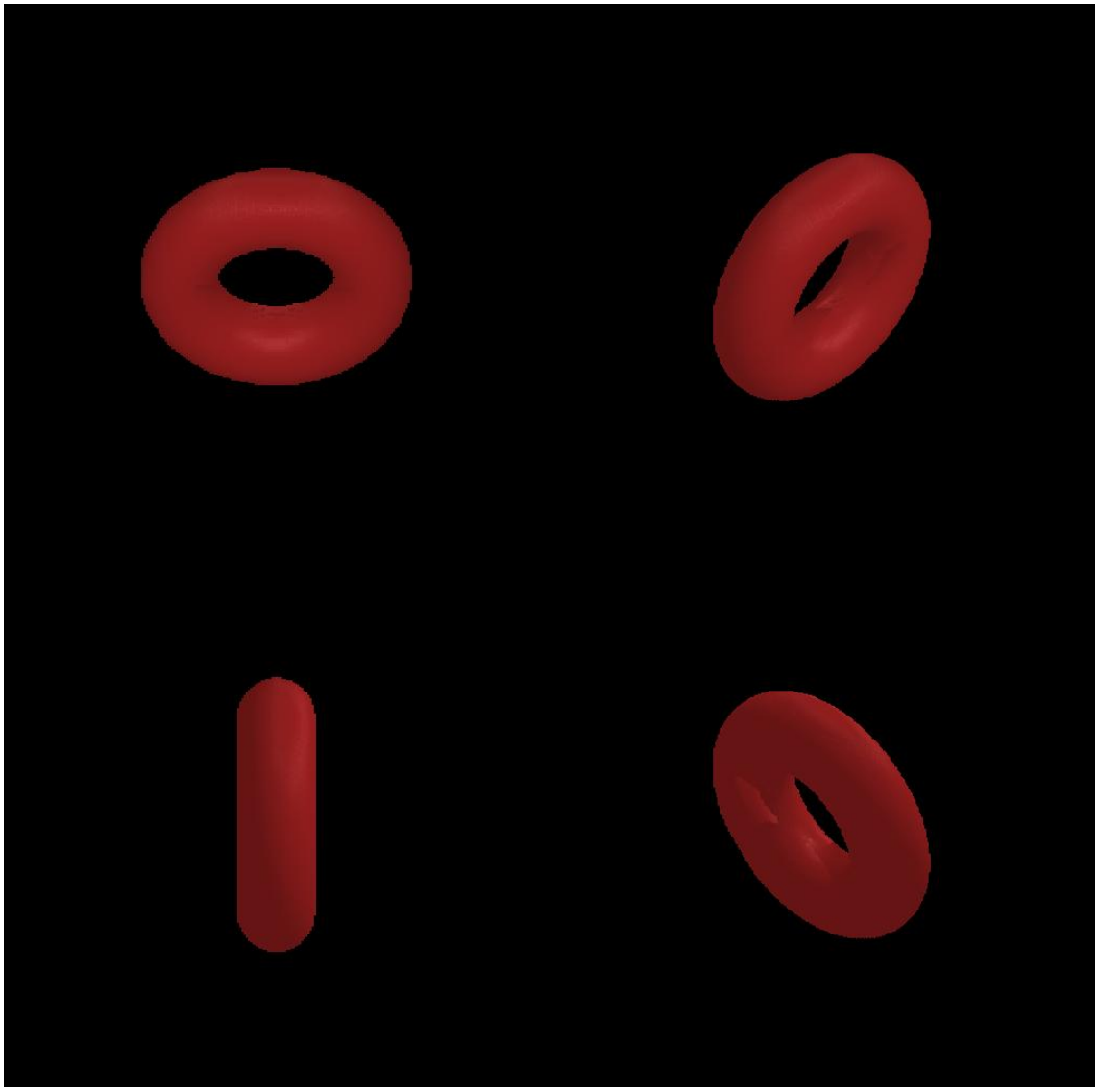


Figura 30: Resultados II - Conjunto de imagens da reconstrução gerada

Utilizando o programa de avaliação de cores, para validar o resultado de imagens correspondentes, foi construída a tabela 2:

Ângulo da imagem eixo Y	% de Cor correta	% de Geometria correta
0º	89,56	99,74
45º	95,23	100,00

90º	95,67	99,74
135º	95,30	100,00
180º	96,50	99,86
225º	90,85	100,00
270º	90,57	100,00
315º	96,14	99,86

Tabela 2: Resultado II- tabela de acertos

Fazendo uma média total do acerto de cores e comparando com a resolução de voxels escolhida para a reconstrução, o gráfico abaixo (Figura 31) reflete a taxa de acerto de cores por resolução de voxels:

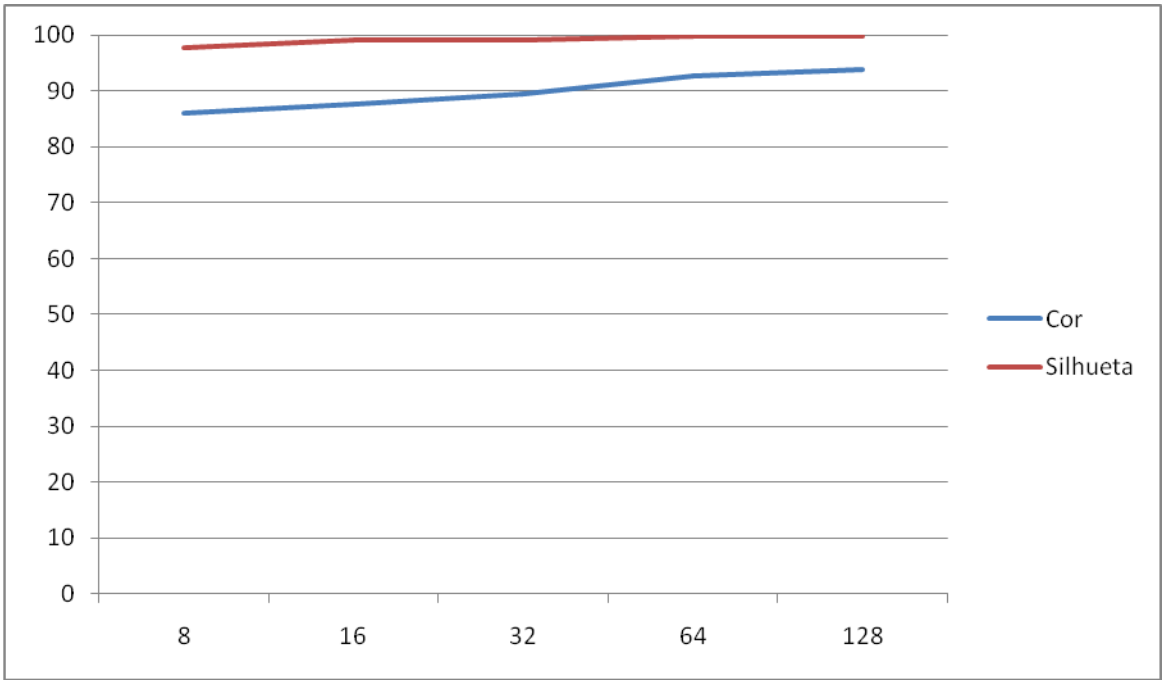


Figura 31: Resultado II - Gráfico Acerto x Quantidade de voxel por aresta.

De acordo com os resultados apresentados na tabela, o algoritmo não se comportou bem em alguns ângulos específicos. O motivo da ocorrência do problema é a iluminação. Esse erro é nítido quando visualizado as imagens de saída. Na área onde incide a iluminação o modelo fica manchado com uma cor meio amarelada.

Analisando a taxa de acerto de cores e geometria, e comparando as imagens resultantes com as de entrada, concluímos que para esse exemplo, o sistema obteve um desempenho excelente para geometria, e um desempenho bom para a comparação de cores.

5.3.3 -Resultado III

O exemplo a seguir, é a reconstrução de um esquilo, este é um modelo complexo que possui textura, para este exemplo foi utilizado à funcionalidade do importador de objetos. Algumas imagens de entrada retiradas são exibidas na figura 32:

Figura 32: Resultado III - Conjunto de imagens de entrada

A figura 33 contém imagens extraídas do modelo após a reconstrução:

Figura 33: Resultados III - Conjunto de imagens da reconstrução gerada

Utilizando o programa de avaliação de cores, para validar o resultado de imagens correspondentes, a tabela 3 foi construída:

Ângulo da imagem eixo Y	% de Cor correta	% de Geometria correta
0º	89,44	99,72
45º	89,78	99,97
90º	89,39	99,95

135º	89,52	99,86
180º	90,78	99,63
225º	89,45	99,83
270º	89,30	99,91
315º	89,72	99,92

Tabela 3: Resultado III- tabela de acertos

De acordo com os resultados apresentados na tabela, o algoritmo não se comportou bem nos resultados de cor. O motivo da ocorrência do problema é a alta complexidade do modelo utilizado. Eles possuem muitas bordas redondas, e acaba caindo no mesmo caso do cubo gradiente. Outro problema é a iluminação que também já foi abordado no caso do toróide.

Levando em consideração que este modelo é muito complexo e próximo ao nível de dificuldade de muitos objetos reais, concluímos que para esse exemplo, o sistema obteve um desempenho excelente para geometria, e um desempenho bom para a comparação de cores.

5.3.4 -Resultado IV

O exemplo a seguir é o segundo exemplo complexo que utilizou o importador de objetos. Algumas imagens de entrada retiradas são exibidas na figura 34.

Figura 34: Resultado IV - Conjunto de imagens de entrada

Imagens extraídas (figura 35) do modelo após a reconstrução em ângulos diferentes das apresentadas na figura 34.

Figura 35: Resultados IV - Conjunto de imagens da reconstrução gerada

Utilizando o programa de avaliação de cores, para validar o resultado de imagens correspondentes, a tabela 4 foi construída:

Ângulo da imagem eixo Y	% de Cor correta	% de Geometria correta
0º	94,29	99,30
45º	94,18	99,73
90º	94,94	99,09

135º	93,88	99,74
180º	93,69	99,12
225º	93,89	99,78
270º	94,94	99,13
315º	94,22	99,80

Tabela 4: Resultado IV- tabela de acertos

De acordo com os resultados apresentados na tabela, o algoritmo se comportou bem nos resultados de cor e silhueta, demonstrando que o algoritmo consegue reconstruir um modelo mais complexo com cores bem similares as imagens de entrada.

Analisando as imagens e percentual de acerto pode perceber que esse modelo obteve uma taxa de acerto de cores em torno de 94% e foi superior ao modelo do esquilo que ficou em torno de 89%, essa melhora se deve a baixa iluminação no modelo da mulher, o que diminui o brilho nas fotos.

5.4 -Resultado de tempo

A partir do exemplo acima 5.3.4 fizemos um comparativo levando em consideração o tempo relacionado o ao numero de threads.

O ambiente de teste foi composto de um computador com processador AMD Turion 64X2 de 2,00GHz, 3GB de memória RAM DDR2 de 533MHz, tendo como sistema operacional Windows 7 profissional.

256 X 256 X 256 voxels	
Numero de threads	Tempo (segundos)
1	21
2	15
4	15

Tabela 5: tabela de tempo para reconstruções baseada em 256 voxels por aresta.

A tabela 5 mostra a reconstrução de um modelo de 256 voxels por aresta, ou seja, 16.777.216 voxels. Nela podemos ver que utilizando mais de um thread houve uma diminuição de 28,57 % do tempo.

Um fator importante a se considerar é a utilização de um processador de dois núcleos, cada qual executando somente uma thread por vez.. Colocar um número de threads maior que de processadores não oferece nenhum ganho de desempenho. Portanto, executar duas ou quatro threads em um computador com processador de dois núcleos, não faz diferença no tempo do processo. Certamente em um computador com processador de quatro núcleos a execução das quatro threads seria mais rápida do que de duas threads, pois a relação processador thread estaria 1:1.

512 X 512 X512 voxels	
Numero de threads	Tempo (segundos)
1	152
2	111
4	111

Tabela 6: tabela de tempo para reconstruções baseada em 512 voxels por aresta.

A tabela 6 mostra a reconstrução de um modelo de 512 voxels por aresta, ou seja, 134.217.728 voxels. Nela podemos ver que utilizando mais de um *thread* houve uma diminuição de 26,97 % do tempo.

Pelo mesmo motivo da tabela 5, não houve nenhuma melhora de desempenho na variação de duas para quatro theards.

5.5 -Considerações

Com o teste proposto de qualidade de cor e geometria dos objetos, pode-se afirmar que o *voxel coloring* é um poderoso algoritmo de reconstrução de cenas 3D. Ele cumpre bem o papel de reconstruir objetos, tanto com formas simples, como visto nos resultados I e II, quanto nos mais complexos, como no modelo da mulher e do esquilo, obtendo um percentual acerto geométrico de aproximadamente cem porcento (100%) em todos os modelos. Deve-se tornar claro que o bom resultado geométrico deve-se muito ao fato de que as silhuetas dos objetos são consideradas na reconstrução. Em casos em que não se conhecem as silhuetas, deve-se esperar que a precisão geométrica seja reduzida consideravelmente.

No quesito cor, pode-se dizer que o algoritmo também produziu bons resultados. Nos resultados II e IV, as taxas de acerto foram acima de noventa porcento (90%). O pior caso para o algoritmo foi o resultado I, no qual o modelo é um cubo com gradiente, no qual todos os voxels possuem vizinhos com cores diferentes.

Mesmo com resultados bons no testes de contagem de cor e comparação de geometria, alguns problemas visuais são evidentes. No resultado III, ficou evidente que devido à iluminação do modelo, houve erros na reconstrução dos voxels onde incidia a luz. Isto é natural, considerando-se que estas regiões fogem ao comportamento lambertiano esperado.

Após a paralelização do algoritmo numa máquina com dois *cores*, tivemos uma diminuição acima de 25% no tempo de execução do algoritmo, mantendo a mesma qualidade de antes da utilização de thread.

6 -Conclusão

Na introdução do projeto, foram apresentadas algumas metas: Automação da construção de cenas 3D, melhoria de desempenho do algoritmo na tentativa de alcançar uma reconstrução em tempo real, ferramenta com facilidades para testes e novas pesquisas e um software passível de alteração sem muito esforço e de fácil entendimento.

Nos objetivos apresentados, quase todos foram atingidos com sucesso, tendo como pendência apenas um desempenho que pudesse ser viável uma reconstrução em tempo real. Mesmo assim a solução melhorou o tempo de reconstrução, utilizando as técnicas de paralelização através de threads e barreira, explicados no capítulo de implementação.

O algoritmo de *voxel coloring* trabalha bem na reconstrução de cenas 3D. Seu ponto forte está na fidelidade das geometrias "macroscópicas", ou seja, ele acerta na geometria do modelo como um todo, mas peca no detalhamento de alguns objetos. Pode-se dizer então, que o método é muito eficaz na reconstrução de modelos sem micro detalhes nas bordas. Nos resultados com objetos simples e até um pouco mais complexos, os teste apresentaram reconstrução de cem porcento (100%) em alguns casos e angulações.

Na classificação de cor, o método apresentou alguns problemas. Em objetos mais simples, as cores foram bem fiéis aos dados de entrada, acontecendo erros apenas em partes mais complexas, como nas bordas dos modelos, onde o algoritmo não sabe como agir e qual cor escolher quando existem cores muito distintas. Outro caso importante foi na reconstrução de modelos com muitas cores distintas, apesar de nesses casos o teste visual ser bem aceitável, o rendimento qualitativo foi bem inferior aos casos onde o gradiente de cores era menor.

A principal contribuição deste trabalho, além da própria ferramenta desenvolvida para visualização e teste do algoritmo, foi à paralelização do mesmo onde ouve uma redução de tempo considerável, tirando melhor proveito das máquinas atuais que possuem vários núcleos. Essa melhoria no algoritmo facilitará uma possível mudança para paralelização em GPU.

6.1 -Trabalhos Futuros

Como trabalhos futuros, existem muitas vertentes. Um trabalho futuro importante seria a utilização do sistema considerando câmeras reais para aquisição da geometria e textura de modelos reais.

Criando um módulo a parte ou até integrado ao sistema, uma ferramenta de refinamento manual é uma opção interessante para os modeladores, que poderiam dar mais qualidade ao resultado final.

Os resultados finais do sistema, ainda não são perfeitos em termos de qualidade. Um bom trabalho é encontrar uma solução para o sistema não errar em modelos mais complexos, com silhuetas mais elaboradas. Durante o projeto foram pensadas algumas soluções, como detecção dessas regiões e tratamento específico, e mais um passo de interação para retoques finais.

Outra possibilidade é a conversão do modelo volumétrico para malhas e a criação de um exportador para o Blender ou outro modelador 3D, onde o artista poderá dá seus últimos retoque no modelo e usar onde preferir depois.

Finalmente, um grande aumento no desempenho seria alcançado através da implementação do algoritmo em hardware gráfico possibilitando a reconstrução a partir de vídeos com frames calibrados ou até mesmo a reconstrução de objetos em tempo real.

Referências Bibliográficas

[1] Anúncio de exame de ultrasonografia 3d. disponível em: http://www.fetalmed.net/ultrassonografia-tridimensional-3d-4d.html. acessado em 25 de fevereiro de 2010.

[2] Cuda. disponível em: http://www.nvidia.com/object/cuda_home.html. acessado em 18 de janeiro de 2010.

[3] Java ™ binding para a opengl. diponível em: http://www.jogl.org. acessado em 10 novembro de 2009.

[4] Matéria sobre os altos custos de desenvolvimento de games. disponível em: http://gamereporter.uol.com.br/2010/01/13/custo-medio-de-desenvolvimento-de-um-game-pode-chegar-a-us-28-milhoes/ . acessado em 25 de fevereiro de 2010.

[5] Objimport: Importador de arquivo tipo *.obj. disponível em: http://www.pixelnerve.com/downloads/processing/objimport.zip. acessado em 25 de novembro de 2009.

[6] Voxel. disponível em: http://www.voxelogic.com/. acessado em 12 de novembro de 2009.

[7] Erich Gamma, Richard Helm, and John Johnson, Ralph e Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[8] Steven M. Kutulakos, Kiriakos N. e Seitz. A theory of shape by space carving. *Int. J. Comput. Vision*, 38(3):199–218, 2000.

[9] Lau C. Lung. Threads e concorrência em java. disponível em: http://www.inf.ufsc.br/ lau.lung. acessado em novembro de 2009.

[10] José C. Macoratti. Mvc-model-view-controller . disponível em: http://www.macoratti.net/08/06/asp_mvc1.htm. acessado em 11 de agosto de 2009.

[11] Morasso P. Zaccaria R. Massone, L. Shape from occluding contours. *Spie Conf. on Intelligent Robots and Computer Vision*, 521:114–120, 1985.

[12] Anselmo Montenegro. *Reconstrução de cenas a partir de imagens através de escultura do espaço por refinamento adaptativo.* PhD thesis, PUC-Rio, 2003.

[13] Rinaldo C. Pinto. Conhecendo mais conceitos e grandezas fotométricas. Revista Lumiere online (http://www.iee.usp.br/biblioteca/producao/2004/Artigos%20de%20Periodicos/rinaldo_aula3.pdf), Julho 2004. p. 1- 7.

[14] Carlos Ribeiro. *Apostila de Java Introdução*. Universidade Federal Fluminense, p. 5.

[15] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 1067, Washington, DC, USA, 1997. IEEE Computer Society.

[16] Tárcio Zemel. Estrutura do mvc. disponível em: http://codeigniterbrasil.com/passos-iniciais/mvc-model-view-controller/ . acessado em 11 de agosto de 2009.