



**COPPE/UFRJ**

## PINTURA MULTIRRESOLUÇÃO DE OBJETOS 3D

José Ricardo Mello Viana

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro

Março de 2009

PINTURA MULTIRRESOLUÇÃO DE OBJETOS 3D

José Ricardo Mello Viana

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

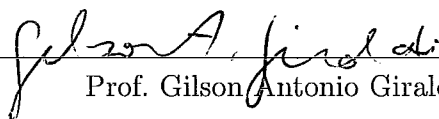
Aprovada por:



Prof. Claudio Esperança, Ph.D.



Prof. Antonio Alberto Fernandes de Oliveira, D.Sc.



Prof. Gilson Antonio Giraldo, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2009

Viana, José Ricardo Mello

Pintura Multirresolução de Objetos 3D/José Ricardo Mello Viana. – Rio de Janeiro: UFRJ/COPPE, 2009.

XI, 60 p.: il.; 29,7cm.

Orientador: Claudio Esperança

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2009.

Referências Bibliográficas: p. 57 – 60.

1. Pintura. 2. Multirresolução. 3. Programação em GPU. 4. Mapeamento UV. 5. Unwrapping. 6. Textura.  
I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

*À minha noiva e futura esposa,  
Nayane Ferreira da Ponte*

# Agradecimentos

Primeiramente a Deus, por permitir que eu tivesse a capacidade para concluir este trabalho.

À minha família, por todo o incentivo dado em toda minha trajetória de estudos, desde a alfabetização até a academia, principalmente nos momentos de fraqueza passados em solidão no Rio de Janeiro.

À minha noiva, Nayane Ferreira da Ponte, que sempre esteve a meu lado, tanto nos momentos complicados quanto nos momentos de alegria, me apoiando e incentivando na produção deste estudo.

Aos meus amigos e colegas de jornada, em especial Alvaro, Ricardo Marroquim, Yalmar, Alberto, Marcelo, Felipe, André, Carlos Eduardo, Saulo, e outros que não vem à mente agora, presença constante no Laboratório de Computação Gráfica e companheiros do dia-a-dia. Paulinho e Vieira, professores da Universidade Federal do Piauí que, mesmo distantes, foram importantes no desenvolvimento deste trabalho. Vinícius, Olivério e Fabrício que me acolheram todas as vezes que precisei retornar ao Rio de Janeiro para dar continuidade aos trabalhos.

Aos professores do Laboratório de Computação Gráfica da Universidade Federal do Rio de Janeiro, responsáveis pela minha formação nesta área de conhecimento. Em particular ao meu orientador, Claudio Esperança, pelas ideias que sempre deu, por sua sinceridade, por seu caráter e por seu apoio, especialmente no momento em que permitiu minha volta para Teresina, a fim de terminar este estudo perto de meus familiares.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico do Brasil (CNPq) pelo suporte financeiro, sem o qual seria praticamente impossível a conclusão deste mestrado em uma cidade tão distante de minha terra natal.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

## PINTURA MULTIRRESOLUÇÃO DE OBJETOS 3D

José Ricardo Mello Viana

Março/2009

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Nós propomos uma técnica para pintura de modelos 3D baseada na imagem renderizada do modelo e em um mapa 2D dos vetores normais do modelo. A geração de texturas e das coordenadas que mapeiam estas texturas sobre o modelo (mapeamento UV) são produzidas automaticamente e dependem apenas da resolução aparente da tela, sendo possível, assim, pintar tanto traços grossos quanto pequenos detalhes sem perdas, ou seja, a multirresolução é garantida. Grande parte do processamento é feito em placa gráfica (GPU - *Graphics Processing Unit*), o que melhora o desempenho do sistema. Os requisitos minimalistas para a pintura trazem consigo a possibilidade de a técnica ser facilmente aplicada sobre malhas triangulares e também sobre modelos baseados em pontos.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

## MULTIRESOLUTION PAINTING OF 3D OBJECTS

José Ricardo Mello Viana

March/2009

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

We propose a technique for painting 3D models based on a rendered image of the model and a 2D map of normal vectors. Texture generation and coordinate mapping (UV-mapping) is then produced on-the-fly and depend only of the apparent screen resolution, making it possible to paint in broad strokes or in small detail, i.e., the multiresolution is guaranteed. Much of the processing is done in GPU (Graphics Processing Unit), which improves system performance. The minimalistic requirements of this technique make it suitable for painting both triangle meshes and point-based models.

# Sumário

<b>Lista de Figuras</b>	<b>x</b>
<b>1 Introdução</b>	<b>1</b>
<b>2 Revisão Bibliográfica</b>	<b>4</b>
2.1 Pintura Digital . . . . .	4
2.2 Pintura de Objetos 3D . . . . .	6
2.3 Pintura de Vértices . . . . .	6
2.4 Mapeamento de texturas . . . . .	7
2.5 Atlas de textura . . . . .	9
2.6 <i>Unwrapping</i> adaptativo . . . . .	11
2.6.1 O Sistema Chameleon . . . . .	13
2.6.2 Pintando traços . . . . .	14
2.6.3 <i>Feedback</i> para traços de entrada . . . . .	16
2.6.4 Empacotamento . . . . .	16
2.6.5 Resultados do Chameleon . . . . .	18
2.6.6 Limitações do Chameleon . . . . .	18
2.7 Pintura com superfícies de subdivisão . . . . .	19
2.7.1 Representação da superfície . . . . .	20
2.7.2 Renderização da superfície . . . . .	21
2.7.3 Manipulação da superfície . . . . .	22
2.7.4 Pintura da superfície . . . . .	23
2.7.5 Deformação multirresolução . . . . .	24
2.7.6 Resultados desta abordagem . . . . .	24
2.8 Pintura interativa de modelos 3D baseados em pontos . . . . .	26



2.8.1	Descrição do sistema . . . . .	26
2.8.2	Modelo e dinâmica do pincel . . . . .	29
2.8.3	Transferência de tinta . . . . .	30
2.8.4	Amostragem dinâmica . . . . .	32
2.8.5	Implementação . . . . .	34
2.8.6	Resultados e conclusões desta implementação . . . . .	34
<b>3</b>	<b>Método Proposto</b>	<b>36</b>
3.1	Implementação desenvolvida . . . . .	38
3.2	Interface com o usuário . . . . .	39
3.3	<i>Buffer</i> de normais . . . . .	40
3.4	<i>Buffer</i> de cores . . . . .	42
3.5	<i>Buffer</i> de identificadores de pontos . . . . .	43
3.6	Pintura de Traços . . . . .	46
3.7	Criação de retalhos . . . . .	46
3.8	Associação de coordenadas de textura . . . . .	48
3.9	Gerenciamento de retalhos . . . . .	49
3.10	Empacotamento . . . . .	50
3.11	Aplicação com modelos baseados em pontos . . . . .	51
<b>4</b>	<b>Resultados e Discussão</b>	<b>52</b>
<b>5</b>	<b>Conclusões e Trabalhos Futuros</b>	<b>55</b>
	<b>Referências Bibliográficas</b>	<b>57</b>

# Lista de Figuras

2.1	Adobe Photoshop e Corel Draw: exemplos de editores 2D de imagens	5
2.2	Blender 3D e Autodesk 3D Studio Max: exemplos de programas de edição 3D . . . . .	6
2.3	Pintura de Vértices no Blender 3D . . . . .	7
2.4	Pintura com Mapeamento de Textura no Blender 3D . . . . .	8
2.5	Modelos 3D e o correspondente atlas de textura . . . . .	9
2.6	Modelos pintados à mão usando atlas gerado através dessa técnica . .	11
2.7	O sistema Chameleon . . . . .	12
2.8	Multirresolução alcançada pelo sistema . . . . .	13
2.9	<i>Overview</i> do processo de pintura do Chameleon . . . . .	15
2.10	Projeção de traços na superfície do modelo . . . . .	16
2.11	Algoritmo de empacotamento dos retalhos em um atlas de textura . .	17
2.12	Modelos pintados com o sistema Chameleon . . . . .	18
2.13	Representação da superfície para um tetraedro . . . . .	21
2.14	Performance geral do sistema . . . . .	25
2.15	Diferentes etapas da aplicação do algoritmo em um modelo de 2000 faces . . . . .	25
2.16	Interface com o usuário . . . . .	27
2.17	Transferência de tinta entre pincel e modelo . . . . .	28
2.18	Representação do pincel . . . . .	29
2.19	Deformação do pincel . . . . .	30
2.20	Divisão do pincel . . . . .	30
2.21	Passos executados durante um evento de pintura . . . . .	32
2.22	Estrutura de dados para representação da superfície . . . . .	33
2.23	Reamostragem da superfície do objeto . . . . .	33

2.24	Modelos pintados com o sistema baseado em pontos . . . . .	35
3.1	Representação esquemática da técnica proposta . . . . .	37
3.2	Tela Inicial do sistema implementado . . . . .	38
3.3	Diferentes formatos do cursor . . . . .	39
3.4	<i>Buffer</i> de normais . . . . .	41
3.5	<i>Buffer</i> de cores com efeito de iluminação apropriado . . . . .	43
3.6	Imagem renderizada do objeto em determinada posição e seu correspondente <i>buffer</i> de identificadores de pontos . . . . .	44
3.7	<i>Feedback</i> apropriado dos traços pintados no <i>buffer</i> de cores proporcionado pelo cálculo de iluminação utilizando o <i>buffer</i> de normais . . . . .	47
3.8	Fases para geração e associação de um novo retalho . . . . .	49
3.9	Gerenciamento de retalhos . . . . .	50
4.1	Modelos pintados com o sistema implementado . . . . .	53
4.2	Atlas de textura gerados pelo sistema . . . . .	54

# Capítulo 1

## Introdução

O principal objetivo deste trabalho é o desenvolvimento de um sistema de pintura de modelos 3D com mapeamento de texturas. Estas texturas são criadas e associadas dinamicamente com os vértices do objeto.

Durante a pintura, vários retalhos são gerados para representação das pinturas sobre o modelo. Por conseguinte, deve haver um sistema que gere esses retalhos de textura de maneira eficiente, com o objetivo de minimizar o gasto de memória. Isso faz com que sejam criadas texturas apenas para as áreas pintadas do modelo, e somente um retalho esteja associado com cada face do modelo em determinado momento.

Ao final, tem-se apenas um atlas de texturas, contendo todos os retalhos produzidos, que representará a totalidade da pintura efetuada durante uma sessão de pintura e sua correspondente associação com o modelo, chamado de mapeamento UV (*UV-mapping*). Dessa forma, o resultado deste sistema será o mesmo produto de sistemas de pinturas 3D tradicionais.

Grande parte dos sistemas de pintura 3D tradicionais seguem o paradigma chamado *UV-mapping*, onde uma textura 2D é mapeada sobre um objeto 3D para representar a pintura. Este paradigma requer um passo inicial fundamental onde será estabelecido o mapeamento entre a superfície e a textura 2D, passo chamado de *unwrapping*. Após isso, os primeiros sistemas possibilitavam ao usuário pintar a textura 2D utilizando várias ferramentas de pintura.

Com o avanço dos sistemas 3D, esse processo foi facilitado, ao passo que se permite ao usuário pintar diretamente na visão 3D e, automaticamente, a pintura

é reprojeta sobre o mapeamento definido previamente. Desta forma, é provida maior interatividade e uma realimentação mais precisa ao usuário.

Como sistemas deste tipo são bastante dependentes da qualidade e da resolução do mapeamento, o ideal é que tenha-se o mínimo de distorções possível [1]. Um bom mapeamento pode minimizar a extensão das costuras, que são as áreas de fronteira no mapeamento onde se passa áreas mapeadas com o modelo para áreas sem mapeamento, pois essas áreas são responsáveis pelas discontinuidades no mapeamento.

Outro problema é o fato de que esses métodos não se concentram apenas nas áreas que serão pintadas no modelo, pois toda a superfície é mapeada previamente independente de quais partes do modelo serão pintadas.

A resolução da textura também é um problema, pois, em sistemas desse tipo, ela é fixada no momento do estabelecimento do mapeamento, dificultando assim a pintura de pequenos detalhes em algumas partes do modelo sem aumentar o tamanho da textura ou alterar o mapeamento.

Neste trabalho são seguidas as ideias do sistema Chameleon [2]. Nele, as texturas e os mapeamentos são criados dinamicamente, em contraste aos modelos tradicionais onde são criados previamente. Somente são associadas à textura as áreas pintadas do objeto, evitando que seja desperdiçada memória de textura.

A pintura é efetuada diretamente sobre uma visualização do modelo. Usando um pincel virtual, o usuário realiza traços sobre o modelo, traços estes que são coletados em uma imagem auxiliar. Nesta etapa, o sistema simula a pintura sobre o modelo combinando as cores da imagem auxiliar e a imagem projetada do modelo usando um algoritmo de iluminação e composição implementado em *GPU* (*Graphics Processing Unit* - Unidade de Processamento Gráfico comumente chamada de placa de vídeo).

A segunda etapa se inicia assim que o usuário altera a projeção do modelo – usando uma rotação, por exemplo. Neste momento, a imagem auxiliar é processada gerando um retalho de textura. Os diversos retalhos de textura gerados pela repetição desse processo são dispostos de forma conveniente num atlas. O gerenciamento eficiente destes retalhos é um ponto importante do presente trabalho, já que longas sessões de pintura tendem a gerar um grande número de retalhos.

Outro importante ganho com a geração dinâmica de textura é a pintura multirresolução. Para cada evento de pintura, a resolução dos traços só dependerá da

resolução aparente da tela no momento específico da pintura deste traço. Diferentemente do modelo tradicional em que a resolução da textura é fixada no momento da criação e associação da textura ao modelo, que antecede a pintura. Dessa forma, torna-se possível pintar tanto grandes traços quanto pequenos detalhes no modelo.

Ao final da sessão de pintura, assim como em [2], efetua-se o empacotamento das texturas, ou seja, é criado um atlas de textura que irá conter os retalhos presentes ao final da sessão de pintura e recalcula-se as coordenadas de textura para cada retalho de acordo com sua posição no atlas.

Uma importante contribuição de nosso trabalho é o fato deste empregar texturas não só para atribuir cor a determinados *pixels* da tela. Como será visto na descrição do processo, texturas são usadas em diferentes momentos. Primeiramente no resultado, que consiste nas texturas que representam a pintura feita no sistema e que poderá ser salvo para ser usado em outros programas.

Além disso, texturas são usadas, também, como estruturas de dados para as computações efetuadas durante a execução do programa, a saber, o *buffer* de normais e o *buffer* de identificadores de pontos. Sendo usados, por exemplo, para fazer cálculos de iluminação, identificação da área do modelo, associação dos vértices do modelo com as texturas pintadas (*unwrapping*), entre outras. E, ainda, torna-se relativamente fácil usar essas estruturas para influenciar outras propriedades, no sentido de criar efeitos mais interessantes, tais como o *bump mapping*, que consiste em alterar as normais à superfície do modelo.

O restante deste trabalho está organizado da seguinte maneira: primeiramente serão descritos os trabalhos relacionados que serviram de base para este estudo. Logo em seguida é mostrado o sistema e são detalhadas cada uma de suas características, tais como a interface com o usuário, a pintura de traços, a criação e o gerenciamento das texturas e o empacotamento. Após isso são mostrados alguns resultados obtidos da aplicação do método. E, por fim, são apresentadas as conclusões e propostas de trabalhos futuros que podem ser desenvolvidos a partir deste estudo.

# Capítulo 2

## Revisão Bibliográfica

Nesta seção é feito um apanhado geral dos métodos desenvolvidos para pintura de modelos 3D. São abordados os conceitos de pintura digital, pintura 3D e métodos de pintura 3D como pintura de vértices, mapeamento de texturas e métodos automáticos de mapeamento de texturas. A partir deste levantamento pretende-se justificar a escolha do método de pintura de modelos 3D implementado.

### 2.1 Pintura Digital

Genericamente, pintar significa adicionar pigmentos em forma líquida a uma superfície a fim de torná-la colorida, atribuindo-lhe matizes, tons e texturas. A partir da introdução de ferramentas tecnológicas para criação de pinturas, pode-se expandir esse conceito para simplesmente a representação visual através de cores. Isto porque a cor é considerada como o elemento fundamental da pintura, como a base da imagem e, com o advento do computador e ferramentas computacionais de pintura, o substrato material de tintas e pigmentos não é necessário para a definição de pintura.

Com a introdução dos computadores, surge o conceito de arte digital (ou arte por computador), que é aquela produzida em ambiente gráfico computacional. Neste tipo de arte, tudo é feito de maneira virtual onde, ao contrário dos meios tradicionais, o trabalho é produzido por meios digitais e tem por objetivo dar vida virtual a objetos e mostrar que a arte não é feita somente à mão. Dentro deste tipo de arte enquadram-se várias categorias, tais como gravura digital, programas de modelagem

3D, edição de fotografias e imagens, animação, entre outros. A pintura digital surge, neste contexto de arte, com o objetivo de simular, digitalmente, aquelas pinturas feitas anteriormente à mão com pigmentos líquidos.

Dentro deste escopo de pintura digital, surgem os programas de edição 2D, onde o usuário dispõe de várias ferramentas para editar fotografias 2D virtuais na tela do computador (Figura 2.1). Programas como Microsoft Paint [3], KolourPaint [4], entre outros, são ferramentas simples de edição de figuras com poucas ferramentas disponíveis. Já aplicações como Adobe Photoshop [5], Corel Photo Paint [6] e Gimp [7] dispõem de uma grande variedade de ferramentas como aplicação de filtros, divisão em camadas, etc., para edição profissional de fotos e imagens. Por fim, programas como Corel Draw [6] e Adobe Illustrator [8] estão em outra categoria de programas, a de edição vetorial de imagens. Neles, elementos visuais são especificados através de curvas, polígonos e outros objetos geométricos especificados por coordenadas, o que possibilita cada parte ser aumentada ou diminuída sem perda de qualidade.

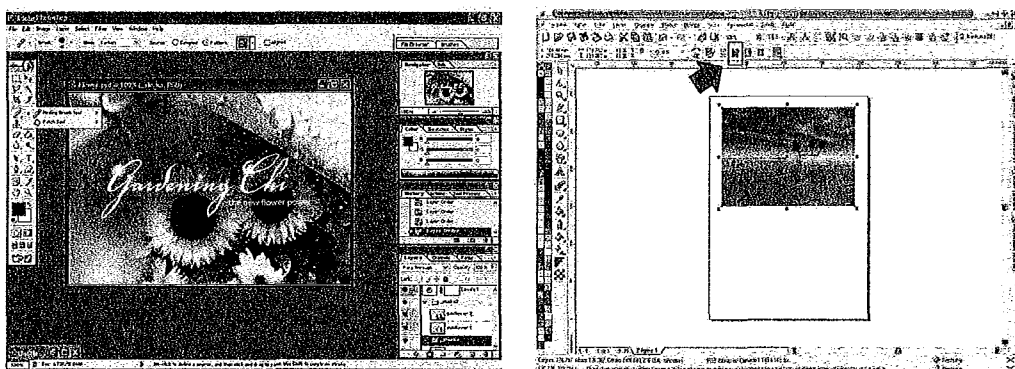


Figura 2.1: Adobe Photoshop e Corel Draw: exemplos de editores 2D de imagens

A partir daí, começaram a surgir programas de edição de objetos 3D. Nestes programas, é possível criar e manipular objetos em três dimensões das mais diversas formas (Figura 2.2). Aplicativos como Autodesk 3D Studio Max [9], Autodesk Maya [10] e Blender [11] possuem funcionalidades para modelar, renderizar, deformar, animar e criar jogos com modelos 3D. Com isso, também houve a introdução da funcionalidade de pintura de objetos 3D, com vários tipos distintos de implementação dos quais alguns serão discutidos a seguir.



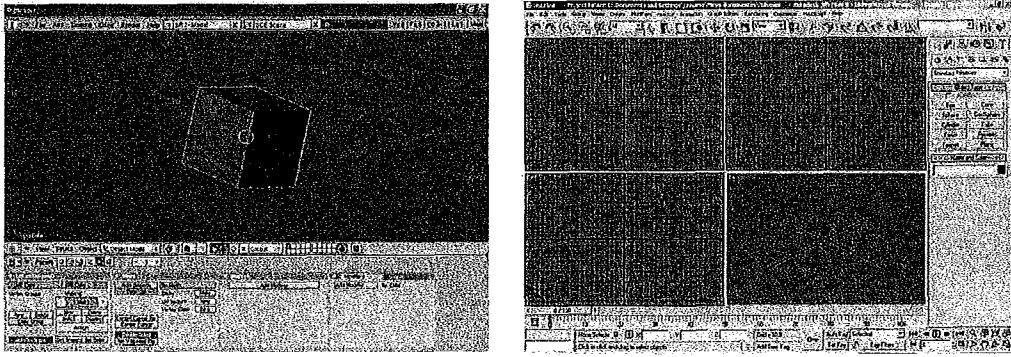


Figura 2.2: Blender 3D e Autodesk 3D Studio Max: exemplos de programas de edição 3D

## 2.2 Pintura de Objetos 3D

Para a pintura de objetos 3D existem várias ferramentas implementadas. Em aplicações comerciais, os tipos de implementação de pintura mais comuns são a pintura de vértices e o mapeamento de textura. Grande parte desses programas só utiliza modelos baseados em triângulos.

O assunto tem sido bastante investigado visando a obtenção de técnicas diferenciadas de pintura, seja para adicionar mais detalhes à pintura, facilitar o trabalho do usuário ou ainda aplicá-la a outros tipos de superfície 3D, como modelos baseados em pontos. Pode-se destacar entre essas técnicas o trabalho de Takeo Igarashi e Denis Cosgrove [2], que elaboraram um método onde a textura que será pintada e mapeada sobre a superfície do modelo 3D é gerada dinamicamente. Tobias Ritschell et al. [12] fizeram uma representação como superfície de subdivisão em *GPU* do modelo 3D para posteriormente pintá-lo com técnicas de manipulação desse tipo de superfície. Bart Adams et al. [13] desenvolveram uma interface com 6 graus de liberdade de um pincel virtual para, com ele, pintar objetos 3D baseados em pontos.

## 2.3 Pintura de Vértices

A pintura de vértices é a técnica mais simples de todas. Neste modelo de pintura, atribui-se a cada vértice uma cor que, além de indicar a tonalidade em que será renderizado o vértice, contribui para as cores em sua vizinhança, sendo esta interpolada com as cores de seus vértices vizinhos para determinar a cor da área ao redor

dele. Neste modelo de pintura não existem estruturas auxiliares, como texturas. Também não há a possibilidade de grande riqueza de detalhes, visto que a resolução depende exclusivamente da resolução dos polígonos da malha. A figura 2.3 mostra um exemplo deste modelo de pintura de objetos 3D.

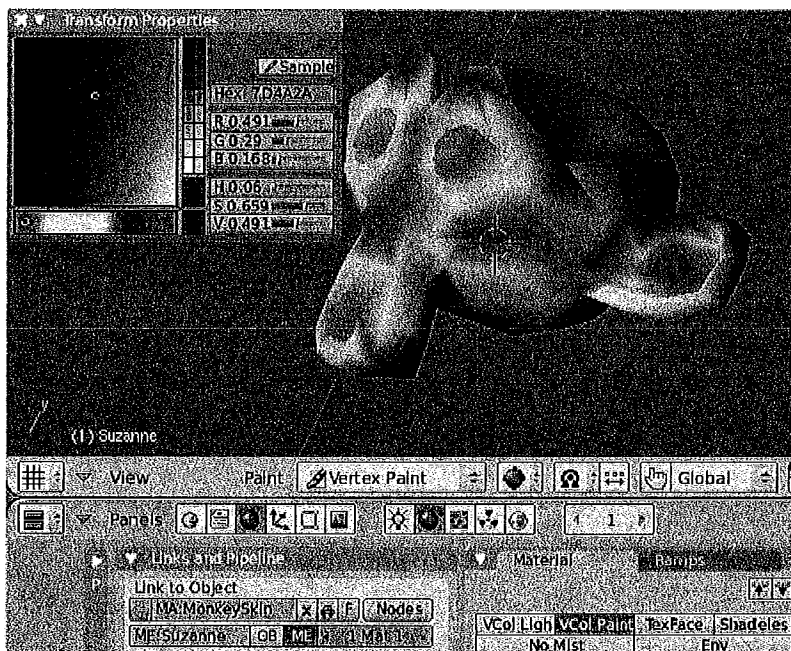


Figura 2.3: Pintura de Vértices no Blender 3D

## 2.4 Mapeamento de texturas

Em computação gráfica, texturas não são exatamente o que se costuma entender por textura no mundo real. No dia-a-dia, texturas são sulcos, saliências e outras irregularidades em superfícies. A noção de textura em computação gráfica é mais semelhante à definição de um adesivo, semelhante a um papel de parede liso que é aplicado no modelo tridimensional. As texturas são imagens 2D que podem ser associadas a materiais para representar características visuais de uma superfície. Com a utilização de texturas pode-se introduzir mais detalhes na pintura do que pintando apenas vértices. Foi com esse intuito que surgiu o conceito de mapeamento de texturas.

Mapeamento de texturas (*texture mapping*) é aplicar uma imagem 2D em um objeto 3D de modo a acrescentar características sem aumentar a complexidade da geometria [14]. Baseado nisso, muitos sistemas de pintura 3D seguem o chamado pa-

radigma do mapeamento UV (*UV-mapping*), onde, basicamente, são necessários dois passos para a realização da pintura: o primeiro é fazer a correspondência do objeto 3D com a textura. Logo em seguida, a textura 2D é pintada e automaticamente mapeada sobre o objeto 3D. Outros sistemas proporcionam uma interatividade maior com o usuário, possibilitando que a pintura seja feita diretamente sobre a visualização 3D do modelo e, a partir disso, os traços são re projetados sobre o modelo para em seguida serem associados à textura nos locais corretos. Um exemplo desse modelo de pintura pode ser visto na figura 2.4.

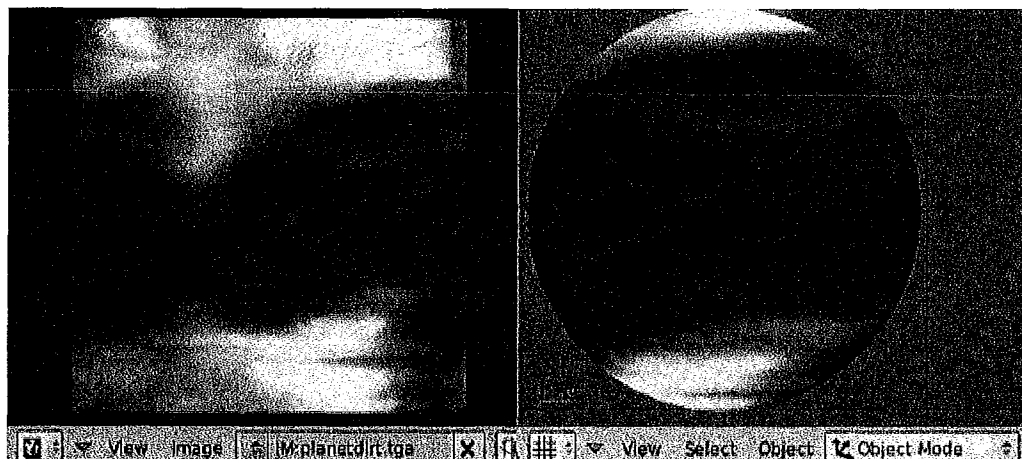


Figura 2.4: Pintura com Mapeamento de Textura no Blender 3D

Quem primeiro introduziu a ideia de pintura com texturas foram Hanharan e Haeberli [15]. Em seu trabalho, ao invés de criarem uma imagem 2D final para armazenar a pintura do objeto, é criado um objeto descritor que irá descrever as propriedades de materiais que compõem a superfície do objeto. Essas propriedades materiais são armazenadas como um conjunto de mapas de textura e irão interagir com a iluminação para criar a aparência do objeto.

O maior problema dessa abordagem é como fazer esse mapeamento. Como criar um bom mapeamento que reflita as características estéticas requeridas em determinada pintura, pois muitas vezes os traços ficam distorcidos pelo fato de cruzarem as costuras do mapeamento UV. Uma boa parametrização da superfície é aquela que minimiza essas distorções. Geralmente, em programas comerciais [16, 17, 18, 9, 11], essa parametrização é feita manualmente, o que torna esse processo bastante complicado. Artistas profissionais geralmente preferem esse método, pois assim podem atribuir especificamente uma textura a um objeto de modo a dar as características

estéticas imaginadas por ele. No entanto, este processo tende a ser demasiado custoso para um usuário comum. Uma alternativa é fazer este mapeamento de maneira automática (tal como V.A.M.P. [17]), só que para isso ainda é necessário configurar uma série de variáveis, tornando-se ainda complicado para o usuário comum.

## 2.5 Atlas de textura

Um atlas de textura é uma eficiente representação de cores para sistemas de pintura 3D. O modelo a ser texturizado é decomposto em partes homeomórficas a discos, cada parte é parametrizada e os retalhos resultantes são empacotados no espaço de textura.

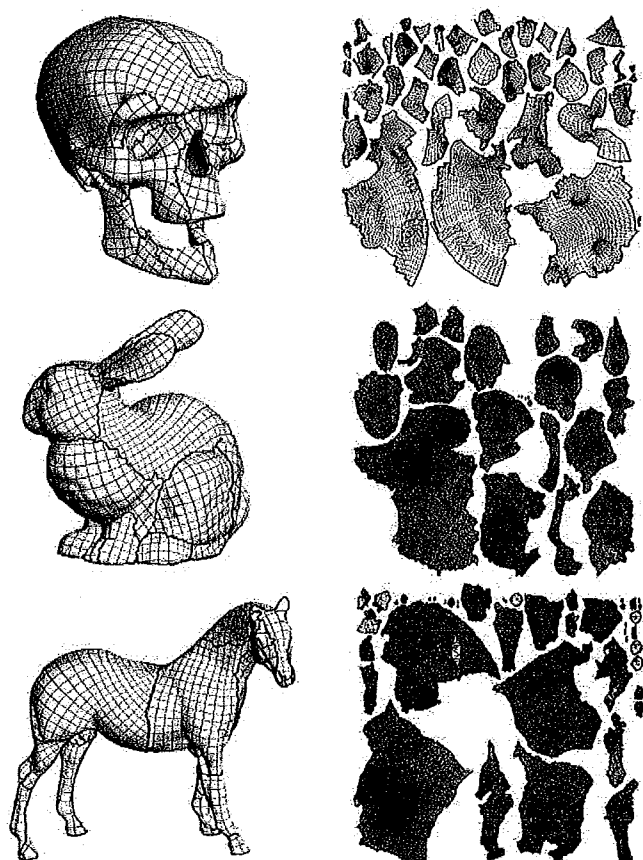


Figura 2.5: Modelos 3D e o correspondente atlas de textura

Os métodos existentes para geração de atlas de textura de superfícies trianguladas sofrem de algumas limitações. Requerem que seja gerado um grande número de pequenos retalhos com bordas simples. As discontinuidades entre retalhos causam artefatos, e fazem com que seja difícil pintar áreas largas com padrões regulares.

Lévy et al. [1] versam sobre um método para geração automática do atlas de texturas. O modelo a ser texturizado é dividido em um conjunto de partes homeomórficas a discos, os retalhos, e cada um deles possui sua própria parametrização. O atlas de textura pode ser representado facilmente em formatos de arquivo padrão e mostrado na tela através de *hardware* de mapeamento de textura padrão. Quando usado em sistemas de pintura 3D, o atlas de textura deverá satisfazer os seguintes requisitos:

- Os retalhos de borda devem ser escolhidos de modo a minimizar os artefatos nas texturas;
- A amostragem no espaço de textura deve ser tão uniforme quanto possível;
- O atlas deverá utilizar de maneira ótima o espaço de textura.

Sua principal contribuição foi a introdução dos mapas conformais usando mínimos quadrados (*LSCMs - Least Squares Conformal Maps*), um método de parametrização baseado em otimização com as seguintes propriedades:

- Seu critério minimiza deformações de ângulos e escalas não-uniformes. Isto pode ser obtido eficientemente por métodos clássicos de otimização, sem a necessidade de algoritmos complexos.
- O mínimo desse critério existe e é único. Portanto, na resolução do problema não é necessário lidar com mínimos locais.
- As bordas dos retalhos não necessitam de correções, como na maioria dos métodos existentes. Portanto, retalhos grandes com bordas arbitrárias podem ser parametrizados.
- A orientação dos triângulos é preservada, ou seja, não ocorrem *flips* de triângulos. Entretanto, sobreposições podem aparecer quando a borda da superfície se auto-intersecta no espaço de textura. Essas situações são detectadas automaticamente e os retalhos são subdivididos.
- O resultado é independente da resolução da malha. Este tipo de propriedade pode ser útil para reduzir desvios na textura quando parametrizando diferentes níveis de detalhes de um mesmo objeto.

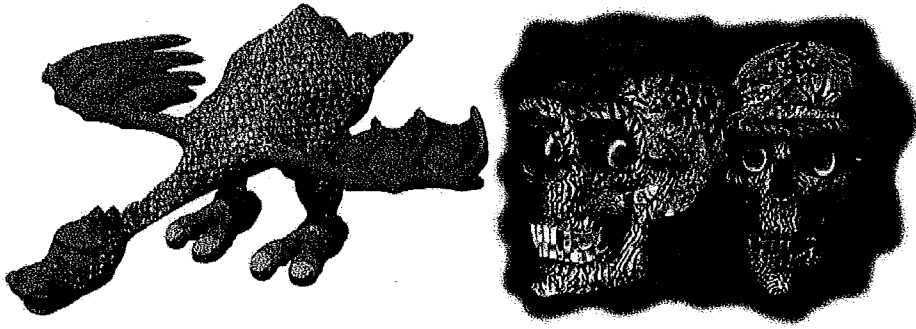


Figura 2.6: Modelos pintados à mão usando atlas gerado através dessa técnica

Alguns resultados da aplicação dessa técnica podem ser vistos na figura 2.5. Nota-se a presença de pequenos retalhos, a maioria dos quais correspondem à detalhes geométricos dos modelos (dentes, patas, etc.). Este não é um problema para a maioria dos sistemas de pintura, pois podem tratá-los adequadamente. Alguns exemplos de modelos texturizados podem ser vistos na figura 2.6.

## 2.6 *Unwrapping* adaptativo

Igarashi [2] apresenta um método para geração dinâmica e eficiente de texturas e o mapeamento UV associado em um sistema de pintura interativa de modelos 3D. Tipicamente, programas de pintura em textura de modelos 3D requerem que o usuário defina explicitamente o mapeamento UV da geometria 3D para a textura 2D antes de iniciar a pintura, sendo este mapeamento mantido imutável durante o processo de pintura. Entretanto, mapeamentos UV predefinidos podem causar distorções em localizações arbitrárias e desperdício de memória nas áreas não pintadas.

Para resolver estes problemas foi proposto um mecanismo de *unwrapping* adaptativo em que o sistema cria dinamicamente um mapeamento UV adaptado aos novos polígonos pintados durante o processo de pintura interativa. Ele elimina as distorções dos traços pintados e a textura resultante é mais compacta porque o sistema aloca espaço somente para os polígonos pintados. Adicionalmente, esta alocação dinâmica de textura possibilita ao usuário pintar suavemente em qualquer nível de escala (“*zoom*”). Os modelos pintados podem ser armazenados como modelos poligonais texturizados padrão. Para isso, foi implementado um sistema protótipo, chamado Chameleon (Figura 2.7). As experiências com usuários sugeriram que a

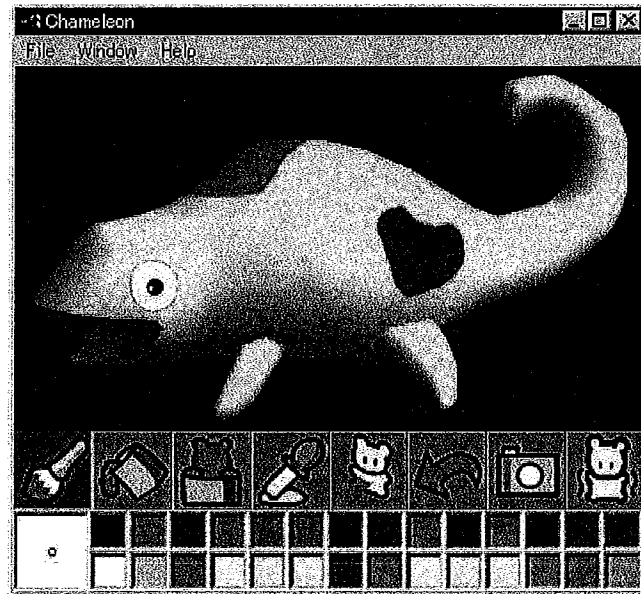


Figura 2.7: O sistema Chameleon

técnica é muito útil para pinturas simples por usuários casuais.

Em seu artigo, Igarashi tem por objetivo a criação e refinamento interativo de texturas pintadas à mão sobre um modelo poligonal 3D. Tradicionalmente, o usuário especifica previamente o mapeamento UV que associa o modelo 3D à uma textura 2D (processo chamado de *unwrapping*), e então pinta sobre a textura 2D usando várias ferramentas de pintura. O sistema reprojeta os traços pintados na visão 3D para a textura 2D de acordo com o mapeamento UV pré-definido e instantaneamente produz o resultado na visão 3D.

Entretanto, programas de pintura 3D tradicionais têm uma série de limitações. Primeiramente, especificar o mapeamento UV manualmente pode ser difícil e tedioso. O mapeamento geralmente demanda mais tempo do que a pintura em si. Um conjunto padrão de mapeamentos, como cilíndrico e esférico, são aproximações razoáveis para superfícies simples, porém falham para modelos com extrusões e concavidades. Diversos métodos de *unwrapping* automático avançados estão presentes em sistemas comerciais, mas continuam a exigir que o usuário ajuste diversos parâmetros que permitam criar um mapeamento personalizado para determinada pintura. Sem orientação explícita do usuário, o mapeamento gerado automaticamente pode colocar costuras em características visuais importantes e causar distorções.

A abordagem proposta consiste em realizar o *unwrapping* de forma progressiva (*on-the-fly*), durante a sessão interativa de pintura, ao invés de construir um mapeamento UV estático previamente. O sistema atribui uma nova textura e coordenadas UV para os polígonos pintados em cada operação de pintura. Isso melhora sistemas de pintura 3D interativos de várias maneiras. Os traços pintados não serão distorcidos. Alguns comportamentos úteis podem ser adicionados à pintura de traços, tal como a habilidade de pintar simultaneamente a superfície frontal e a traseira do objeto. Ademais, a textura resultante é compacta e robusta, pois só será associada com as áreas pintadas e a maioria das costuras está nas lacunas entre as regiões pintadas. Finalmente, a alocação dinâmica das texturas possibilita diferentes níveis de detalhes nas texturas sobre a superfície 3D (pintura multirresolução (Figura 2.8)).

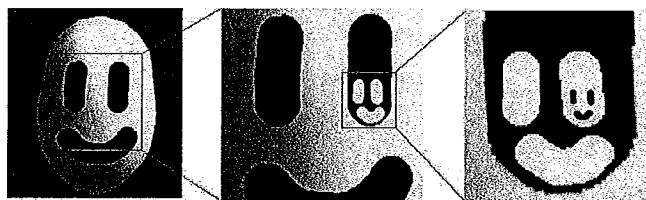


Figura 2.8: Multirresolução alcançada pelo sistema

### 2.6.1 O Sistema Chameleon

Basicamente, o sistema Chameleon trabalha da mesma forma que um programa de pintura 3D padrão, exceto pelo fato de não necessitar que o usuário especifique o mapeamento UV previamente. O usuário abre um modelo poligonal no sistema, especifica o tamanho do pincel e a cor e desenha traços diretamente sobre o modelo 3D.

O tamanho do pincel e seu formato permanecem constantes. Isto contrasta com os sistemas de pintura 3D tradicionais onde o tamanho e forma efetivas do pincel podem mudar na visualização. Adicionalmente, o pincel trabalha baseando-se na estrutura da geometria dada. Quando um pincel pinta traços próximos às arestas de borda a tinta não atravessa a aresta. As arestas de borda podem ser dadas pelo sistema de modelagem (por exemplo, Teddy [19] cria arestas de borda como resultado de operações de corte ou extrusão) ou explicitamente especificadas pelo usuário. Chameleon também possibilita pintar áreas parcialmente escondidas por



outros modelos.

A operação de escala (*zoom*) trabalha diferentemente de sistemas de pintura 2D ou 3D padrão onde a resolução pré-definida da textura limita a habilidade de representar detalhes. Com esses sistemas, quando o usuário aumenta a escala, os traços pintados exibem *aliasing* (serrilhado). No Chameleon, o usuário poderá pintar com suavidade constante mesmo em uma visão com escala extremamente aumentada, emprestando ao sistema características de multirresolução. Como resultado, o usuário pode adicionar quantos detalhes desejar para uma área específica do modelo.

Quando o usuário termina a pintura, o sistema armazena a textura e a geometria com as coordenadas UV atribuídas. A imagem (*bitmap*) resultante tem algumas boas características. As costuras estarão principalmente nos espaços entre características visuais importantes. Isto previne distorções ao reduzir, posteriormente, o tamanho do *bitmap*. O *bitmap* é atribuído principalmente para as áreas onde o usuário pintou traços e o restante da superfície não desperdiçará memória de textura. Adicionalmente, o sistema utiliza *bitmaps* com resoluções adequadas à complexidade visual de cada área. Assim, por exemplo, olhos tipicamente consomem uma grande área, enquanto que uma barriga colorida identicamente consome uma pequena área.

## 2.6.2 Pintando traços

A ideia essencial é construir o mapeamento UV e a textura correspondente progressivamente durante a operação de pintura, ao invés de usar um mapeamento UV pré-definido no processo de pintura. O sistema identifica os polígonos pintados durante o traçado e a eles atribui novas texturas e novas coordenadas UV. A Figura 2.9 mostra um panorama (*overview*) desse processo. Internamente, o sistema usa um modelo poligonal texturizado padrão e o renderiza usando um motor de renderização 3D padrão.

Quando o usuário carrega um modelo não pintado o sistema atribui as coordenadas UV (0.5, 0.5) e associa a todos os polígonos uma textura inicial branca. A textura inicial tem tamanho 1 e uma cor base especificada pelo usuário. Quando o usuário pinta traços, o sistema armazena estes traços como “traços de entrada” (*incoming strokes*) até o usuário rotacionar o modelo. O sistema representa os traços de entrada como traços 2D independentes na tela. Quando o usuário começa a rodar

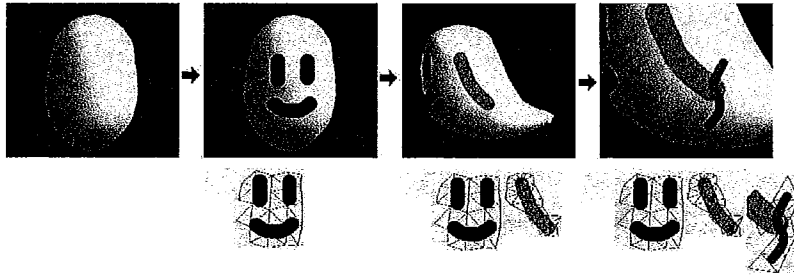


Figura 2.9: *Overview* do processo de pintura do Chameleon

o objeto, o sistema projeta os traços de entrada sobre a superfície do objeto. Esta ideia é similar a *projection paint* em [17], mas eles criam uma representação *bitmap* 2D dos traços de entrada e reprojeta cada pixel sobre a textura de acordo com um mapeamento UV pré-definido.

A projeção é feita pelo seguinte procedimento (Figura 2.10). Primeiramente, o sistema identifica os polígonos que serão pintados por aqueles traços (Figura 2.10(a)). O sistema procura por polígonos pintados começando pelo polígono onde cada traço inicia e, recursivamente, verifica a distância entre os polígonos adjacentes e o traço no espaço da tela. Se a distância for menor que o raio do traço, o polígono é identificado como pintado. Esta procura recursiva ao longo da superfície previne que um traço afete polígonos irrelevantes, e a procura termina nas arestas de borda. Segundo, o sistema gera uma nova textura e reprojeta todas as texturas existentes sobre os novos polígonos pintados (Figura 2.10(b)). O tamanho da nova textura é idêntico à caixa limitante (*bounding box*) dos polígonos pintados no espaço da tela. Terceiro, o sistema pinta os traços de entrada na nova textura usando um procedimento de desenho 2D padrão (Figura 2.10(c)). Finalmente, o sistema atualiza as coordenadas UV dos polígonos pintados e os associa com a nova textura (Figura 2.10d). Esta projeção é simples e rápida porque não requer operações complicadas ao nível de *pixel*. Todas as operações são feitas usando um motor de renderização padrão e um pincel de desenho 2D padrão.

A multirresolução é alcançada automaticamente nessa abordagem, pois a resolução de uma nova textura é determinada pela resolução aparente da tela. Entretanto, o tamanho dos polígonos dados impõe uma limitação. Quando o usuário aumenta muito a escala e os traços são muito menores que o tamanho dos polígonos pintados, o sistema desperdiça uma grande quantidade de espaço de textura já que

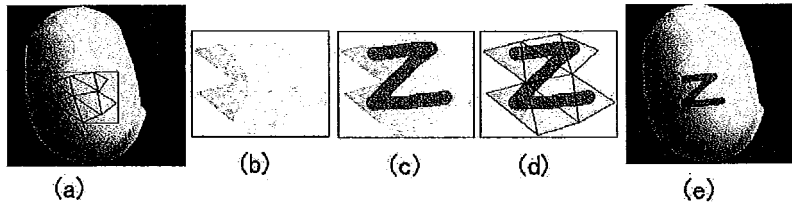


Figura 2.10: Projeção de traços na superfície do modelo

cada *bitmap* de textura deve ser maior que o polígono. Uma solução potencial é subdividir o polígono para gerar uma malha mais fina, mas isso não foi testado.

### 2.6.3 *Feedback* para traços de entrada

Quando o usuário finaliza a rotação, o sistema prepara uma textura temporária para os traços de entrada. O sistema associa todos os polígonos com essa textura e atribui a eles coordenadas UV temporárias. Ela é idêntica à visão 3D do modelo na tela, exceto pelo fato de ser renderizada usando um brilho muito alto para eliminar os efeitos de sombreamento. O sistema atribui coordenadas UV para fazer com que a imagem 3D resultante seja indistinguível da imagem renderizada usando as texturas originais.

O sistema então pinta os traços de entrada nesta textura temporária usando um pincel 2D padrão. Como resultado, os traços aparecem como traços iluminados na superfície do modelo, e nunca ficam fora do modelo na visão 3D. Quando o usuário inicia a rotação, o sistema descarta a textura temporária e o mapeamento UV temporário.

Um problema restante é que esta abordagem não provê uma realimentação visual adequada ao usuário quando esse pinta traços que cruzam arestas de borda ou atrás de outras partes. Para resolver este problema, o sistema gera múltiplas cópias da textura temporária e atribui uma cópia separada a cada parte das arestas de borda. Quando o usuário pinta um traço de entrada, o sistema identifica que parte será pintada e renderiza este traço na cópia apropriada da textura temporária.

### 2.6.4 Empacotamento

Quando o usuário finaliza a pintura e tenta salvar o resultado o sistema gera uma única textura (atlas de textura) pela montagem das texturas criadas durante a pin-

tura. Este processo é chamado de empacotamento (*packing*). O sistema ajusta as coordenadas UV para casar com a textura empacotada, e armazena o resultado em um formato de modelo texturizado padrão, fazendo com que o modelo completamente pintado possa ser lido por várias aplicações gráficas.

Uma etapa não trivial consiste em organizar as pequenas texturas (retalhos) dentro da textura final compactada. Para isso, foi usado um algoritmo de heurística simples, visto que o problema de empacotamento é NP-completo. Apesar deste algoritmo ser uma solução temporária, ele é fácil de implementar, rápido e produz bons resultados, que são suficientes para estes propósitos. A figura 2.11 mostra esse processo.

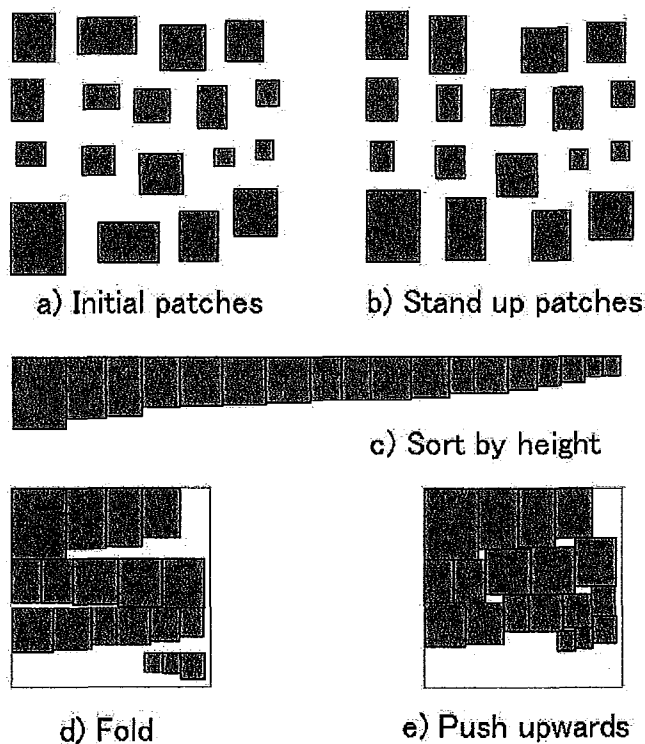


Figura 2.11: Algoritmo de empacotamento dos retalhos em um atlas de textura

Como um passo de pré-processamento, o sistema obtém a caixa envolvente (*bounding box*) de cada retalho, isto é, cada retalho é reduzido a um retângulo para fins de empacotamento. O sistema calcula a área total dos retalhos e define o tamanho alvo da textura resultante a partir daí, como sendo um pouco maior que a raiz quadrada dessa área total (multiplicada por 1,2). Após isso, o sistema procura os retalhos que tem sua altura maior que sua largura e os rotaciona 90 graus. Como resultado, todos os retalhos são mais altos do que grossos. Então esses retalhos são ordenados pela

sua altura e enfileirados, sendo esta fileira dividida como mostrado na figura 2.11 para preencher todo um quadrado de lado  $L$ . Finalmente, cada retalho é empurrado para cima até encostar-se a outro retalho de modo a minimizar os buracos.

### 2.6.5 Resultados do Chameleon

A figura 2.12 mostra alguns modelos pintados usando Chameleon. O usuário não demora mais que alguns minutos para pintá-la a partir do zero. A imagem é limpa e a textura é compacta e eficiente. O tempo de cálculo é irrisório durante a operação de pintura. Estes modelos também representam a classe alvo das pinturas do Chameleon. Somente uma pequena parte é tocada, a maior parte é coberta pela cor base.

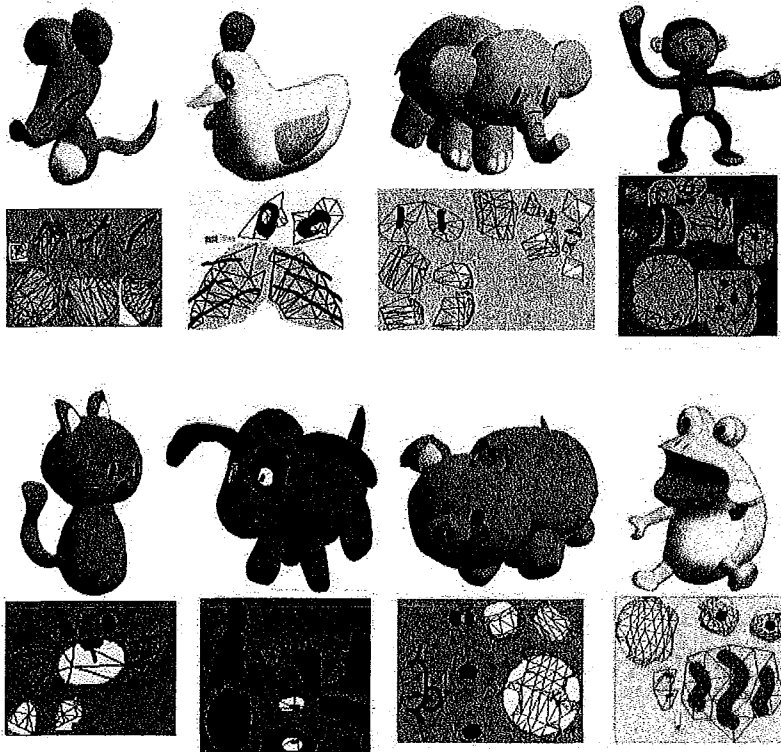


Figura 2.12: Modelos pintados com o sistema Chameleon

### 2.6.6 Limitações do Chameleon

O sistema Chameleon foi projetado especialmente para permitir a usuários casuais fazerem pinturas simples rapidamente. Este sistema não é indicado para usuários profissionais pintarem modelos altamente detalhados. Novos traços pintados afetam

a aparência das texturas previamente desenhadas, e a textura resultante não é necessariamente eficiente se toda a superfície é pintada. Costuras tornam-se visíveis quando o usuário cobre toda a superfície com padrões complexos, como madeira ou tijolos.

Adicionalmente a essa limitação fundamental, há vários problemas a serem corrigidos na implementação do sistema. O maior problema é com o preenchimento de regiões (*flood fill*). O sistema aplica um preenchimento 2D na textura correspondente a área que está sendo pintada, no entanto, isto pode produzir resultados inconsistente na visão 3D, pois a região a ser pintada pode aparecer em vários retalhos. É preciso estender o algoritmo do preenchimento para uma busca recursiva por retalhos adjacentes na superfície poligonal.

O segundo problema é que repetidas reprojeções de texturas existentes degradam gradualmente a qualidade da imagem. Uma possível solução é aplicar texturas em camadas à superfície com transparência. Quando o usuário pintar novos traços em uma área da superfície já pintada, o sistema pode converter novos traços em texturas 2D com transparência e adicioná-las à superfície sobre a textura já existente. As limitações dessa abordagem são a necessidade de um motor de renderização 3D especial com suporte a texturas em múltiplas camadas, e, por conseguinte, consumo excessivo de recursos quando o usuário pinta novos traços repetidas vezes.

Melhorias também podem ser feitas no algoritmo de empacotamento. É necessário um algoritmo que possa organizar os polígonos em um quadrado mais eficientemente recorrendo a outras técnicas de otimização. Adicionalmente, é necessário implementar um procedimento de escala dedicado para reduzir a textura empacotada enquanto toma-se cuidado com os retalhos de borda.

## 2.7 Pintura com superfícies de subdivisão

Adicionar geometria de alta resolução ou detalhes de cores a uma superfície dada é um importante problema na criação de conteúdo 3D. Interfaces intuitivas para esta funcionalidade são de especial interesse em aplicações artísticas de forma a dar suporte à criatividade dos *designers*. Várias aplicações usam a metáfora da pintura de malhas com este objetivo. Para modificar uma dada resolução da malha, o *designer*

simplesmente usa um pincel de tamanho ajustável para efetuar transformações diretamente na superfície do modelo. Essas transformações podem ser diversas como pintura de cores ou de textura, suavização local da malha, entalhe ou escultura.

Tobias Ritschell et. al. [12] propuseram a exploração do poder computacional do moderno *hardware* gráfico existente (*GPUs*) não somente para renderizar a superfície, mas também para a manipulação da superfície baseando-se no paradigma de pintura, onde eventualmente podem-se editar interativamente modelos com alguns milhões de triângulos.

A representação da superfície proposta por eles baseia-se em um atlas de imagens geométricas (*geometry images*). A estrutura regular das imagens geométricas permite processamento eficiente em *GPU*, como mostrado em [20]. Um vasto domínio de malhas pode ser considerado como entrada, visto que deverão, antes de tudo, passar pela subdivisão de Catmull-Clark, que naturalmente leva à representação com imagens geométricas.

### 2.7.1 Representação da superfície

O modelo de entrada é uma malha poligonal irregular de tamanho arbitrário, que após um passo da subdivisão de Catmull-Clark consiste somente de quadriláteros (Figura 2.13(1 e 2)). Após alguns ( $k$ ) passos de subdivisão adicionais cada um desses quadriláteros será refinado até ser formado de um retalho completamente regular de  $2^k \times 2^k$  quadriláteros, podendo ser, cada retalho, naturalmente representado como uma imagem geométrica em uma seção de tamanho  $2^k \times 2^k$  de uma textura em *GPU*.

Com todos os retalho compartilhando a mesma resolução, empacotá-los em um atlas de textura global é trivial. Os diferentes atributos da superfície são armazenados em texturas distintas na *GPU*. Isto porque enquanto para os atributos da geometria da superfície, como posições de vértices e vetores normais, é usada precisão de ponto flutuante, baixa precisão é suficiente para representar cor especular e difusa.

É importante notar que vértices ao longo da borda do retalho serão multiplicados, pois uma cópia será armazenada para cada retalho incidente, isto possibilitará fácil renderização e enumeração da vizinhança. Esta representação é redundante: vértices interiores são representados somente uma vez, mas vértices de uma aresta entre dois

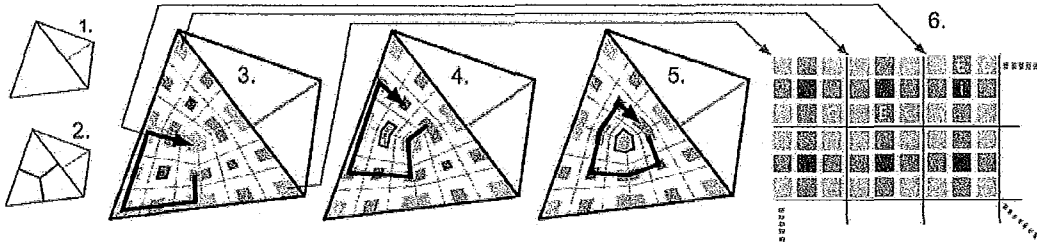


Figura 2.13: Representação da superfície para um tetraedro: (1) Superfície original a ser subdividida para gerar uma malha com quadriláteros somente (2). Após algumas subdivisões, os quadriláteros iniciais refinados são mapeados em quadriláteros no espaço de textura (6). Enumerando a vizinhança em anel de vértices interiores (3), vértices de aresta (4) e vértices de canto (5).

retalhos serão duplicados e vértices nos cantos dos retalhos serão armazenados  $n$  vezes, onde  $n$  é a valência do vértice (Figura 2.13).

Contudo, após uma série de subdivisões, a maioria dos vértices serão interiores. Por exemplo, no nível de subdivisão  $k = 6$  um retalho consiste de  $4^6 = 4096$  vértices, dos quais somente  $4\sqrt{4096} = 256 = 6.25\%$  são vértices de aresta e somente 4 vértices são de canto. Apesar desta representação impor sobrecarga de memória e de execução para vértices de arestas e de cantos, possibilita que os vértices interiores sejam processados pela *GPU* de forma mais eficiente.

## 2.7.2 Renderização da superfície

Para renderizar retalhos de  $2^k \times 2^k$  vértices, um quadrilátero unitário bidimensional  $[0, 1]^2$  é tesselado na resolução  $2^k \times 2^k$  e enviado para a *GPU*. Para cada vértice, sua posição dentro do quadrilátero unitário é usada como coordenada de textura e, a partir dela, é buscada na imagem geométrica correspondente sua posição 3D, vetor normal e atributos da superfície adicionais. Através disso, os triângulos dentro do quadrilátero unitário são mapeados para suas posições 3D correspondentes.

Renderizar a superfície consistindo de  $n$  retalhos, então, requer somente desenhar o quadrilátero unitário  $n$  vezes, uma para cada imagem geométrica. Percebe-se que é possível implementar um nível de detalhes adaptativo simplesmente renderizando o quadrilátero com uma resolução mais grosseira ( $l < k$ ). Isto seleciona um subconjunto de vértices, mas continua usando as normais de alta resolução da superfície,



garantindo assim um sombreamento suave.

### 2.7.3 Manipulação da superfície

Para modificar certo atributo da superfície (como cores ou posições de vértices), uma função que transforme tal atributo deve ser chamada para cada vértice. Em geral, essas transformações requerem acesso a um anel de vizinhos do vértice que se deseja alterar, com o objetivo de fazer a filtragem de cores ou de posições geométricas.

Neste contexto, transformar uma imagem geométrica  $S$  em  $S'$  significa que cada *texel* da imagem geométrica (isto é, cada vértice da malha) será transformado por um programa de fragmentos (*fragment shader*), que lê sua entrada da textura  $S$  e escreve o resultado em uma textura  $S'$ . Contudo, enquanto para vértices interiores regulares buscar a vizinhança é similar a uma filtragem simples da imagem, esses vizinhos não são regulares para vértices de arestas ou de cantos.

Com isso, armazenar e enumerar o anel de vizinhos é feito de forma diferenciada para cada um dos três tipos de vértices (Figura 2.13). Como vértices interiores, de aresta e de canto são tratados de forma diferente, eles devem ser enviados para a *GPU* em diferentes momentos.

Vértices interiores são armazenado somente uma vez e têm uma vizinhança regular. Seus 4 vizinhos podem ser acessados facilmente pelo incremento de um em suas coordenadas de textura em cada direção. Como os vértices interiores de um retalho  $2^k \times 2^k$  são regularmente arranjados como um quadrilátero no espaço de textura, eles podem ser enviados para o programa de fragmentos pela rasterização de um quadrilátero no espaço de imagem de tamanho  $(2^k - 2) \times (2^k - 2)$ , cada *pixel* dele então corresponde a um *texel* da imagem geométrica.

Vértices de aresta também têm 4 vizinhos, mas eles são divididos entre os dois retalhos que compartilham esta aresta. Para efetuar uma atualização em um retalho, a localização na textura do retalho oposto é passada para a *GPU* como uma constante, que então possibilita o acesso à outra metade da vizinhança.

Vértices de canto são armazenados para todos os retalho que compartilham aquele canto, cujo número é a valência do vértice. Para enumerar seus vizinhos, uma pequena textura é usada, onde são armazenadas as informações de vizinhança dos cantos. Cada linha da textura possui a valência do vértice e as coordenadas de

texturas dos vizinhos. Um programa de fragmentos que atualize um canto primeiro lê a valência  $v$  e, então, lê as  $v$  texturas dependentes para enumerar seus vizinhos. Como o número relativo de cantos é pequeno, este processamento adicional é desprezível. Esta representação restringe o uso adicional de computação e de memória a pequena porcentagem de vértices, sendo assim, grandes partes regulares podem ser processadas em *GPU* com grande eficiência.

#### 2.7.4 Pintura da superfície

Uma transformação de pintura requer várias sub-tarefas: busca do vértice que está sob o ponteiro do mouse, cálculo do peso da influência do pincel sobre os vértices, aplicação da transformação (ponderada) e atualização da malha. Cada um desses passos foi implementado em *GPU* conforme a descrição a seguir:

1. Buscar a localização do pincel, isto é, o vértice sob o cursor do mouse, é difícil, na medida em que a malha é deformada continuamente. Para tanto, a malha é renderizada em um segundo *buffer*, mas, ao invés de iluminar cada vértice, as coordenadas de textura são codificadas em canais de cores. A cor do pixel na posição do mouse, então, identifica o vértice mais próximo, isto é, o centro do pincel.
2. A região de influência do pincel é definida pelo cálculo de um fator de ponderação para cada vértice da malha a partir de sua distância ao centro do pincel, usando uma função de transferência linear ou gaussiana. Texturas podem ser usadas para modularizar esta ponderação. Esses cálculos são efetuados em um programa de fragmentos.
3. Outro programa de fragmentos detecta quais retalhos estão completamente fora da região de influência. Estes retalhos são descartados da transformação, o que evita cálculos desnecessários. Em particular, para pincéis de pequeno tamanho.
4. A transformação selecionada é aplicada a cada vértice, ponderada pela influência do pincel. O conjunto de transformações implementadas inclui pintura de cores, suavização de cores ou posições, escultura, extrusão e deformação multirresolução.

5. Os vetores normais a cada vértice são recalculados como normais da superfície limite do processo de subdivisão Catmull-Clark.
6. Opcionalmente, para deformações multirresolução, a reconstrução de detalhes é recalculada.

### 2.7.5 Deformação multirresolução

A metáfora de pintura proporciona uma deformação intuitiva da superfície geométrica, por exemplo, através de extrusão, escultura ou deformações por arraste. No entanto, quando a superfície é deformada em um nível de subdivisão grosseiro é necessário garantir que os detalhes geométricos em níveis de subdivisão maiores sejam preservados e transformados de maneira intuitiva.

Esta funcionalidade é provida pela chamada deformação multirresolução, ou multiescala. Uma vez que são usadas superfícies de subdivisão, são seguidas as ideias de [21]. A abordagem multirresolução provê operadores para subamostragem, subdivisão e codificação de detalhes.

Subamostragem mapeia a superfície para um nível mais baixo na hierarquia de subdivisão. Subamostragem do nível  $k$  para o nível  $l$ ,  $l < k$ , é implementada por um filtro gaussiano recursivo. No passo  $k - l$  de renderização os retalhos de tamanho  $2^k \times 2^k$  são sucessivamente reduzidos para o tamanho  $2^l \times 2^l$ .

A subdivisão do nível  $l$  para o nível  $k$ ,  $k > l$ , é implementada através de uma máscara Catmull-Clark em um programa de fragmentos. Aqui,  $k - l$  passos são usados para expandir retalhos de tamanho  $2^l \times 2^l$  para tamanho  $2^k \times 2^k$ .

Sempre que o usuário deforma a superfície base, detalhes de alta frequência são adicionados à superfície deformada subdividida, que finalmente produz a superfície de alta resolução deformada. Como todas as operações são feitas em *GPU*, deformações multirresolução de modelos complexos podem ser feitas com taxas de quadros interativas.

### 2.7.6 Resultados desta abordagem

A figura 2.14 mostra algumas estatísticas de performance deste algoritmo baseado em *GPU*. Com um tamanho de pincel médio é possível manipular e renderizar su-

Faces	1%	10%	50%	LoD	Full
100k	30 ms	30 ms	30 ms	16 ms	16 ms
600k	60 ms	80 ms	100 ms	50 ms	50 ms
2.3M	120 ms	220 ms	390 ms	75 ms	230 ms

Figura 2.14: Performance geral do sistema para diferentes tamanhos de pincel (1%, 10% e 50% da área da superfície), com malhas de diferentes resoluções, incluindo renderização com níveis de detalhes ou sem (*Full*) em resolução 800x600.

perfícies complexas com 2,3 milhões de vértices a 8,3 quadros por segundo numa placa de vídeo NVIDIA Geforce 7800. Neste tempo estão incluídos todos os passos de pintura descritos e renderização de alta qualidade com efeitos de iluminação. A máxima complexidade de superfície suportada está limitada pela memória disponível na *GPU*. Para 256 MB de memória na *GPU*, 2,3 milhões de vértices é o máximo possível. A figura 2.15 mostra diferentes etapas da aplicação deste algoritmo em um modelo com 2000 faces.

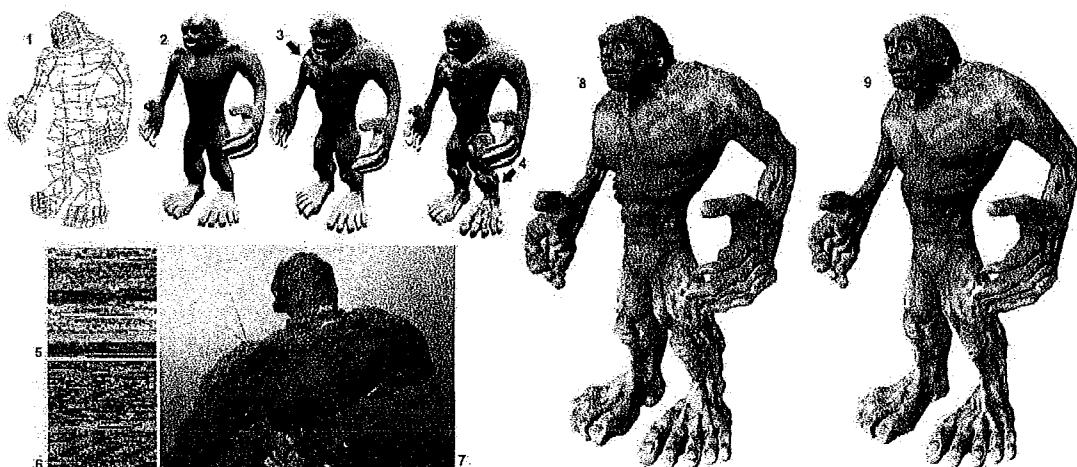


Figura 2.15: Malha com 2000 faces (1) subdividida ao nível 7 (2). Deformação orgânica (3, 4) é adicionada por pintura. A imagem geométrica resultante para esta malha (cores difusas em 5 e normais em 6). Canais de cores e de geometria podem ser pintados juntos com realimentação visual imediata de alta qualidade (7). Resultado final (8). Baixas tesselações usadas para renderização com níveis de detalhes (9).

## 2.8 Pintura interativa de modelos 3D baseados em pontos

Frequentemente, sistemas de pintura 3D trabalham com malhas poligonais para representar a geometria 3D. Pelo estabelecimento de algum mapeamento entre um parâmetro de domínio 2D e a superfície 3D, atributos de aparência resultantes das operações de pintura podem ser armazenados separadamente em mapas de textura. Após criados, esses mapas são reprojatados na superfície objeto.

Esta separação entre geometria e aparência tem diversas desvantagens inerentes: a parametrização necessária para mapear esses dois domínios conduz inevitavelmente a distorções que degradam a qualidade visual da pintura 3D. Além disso, a resolução uniforme do mapa de textura faz com que seja difícil lidar com diferentes níveis de detalhes na pintura. Muitas vezes, uma reamostragem força bruta é aplicada para acomodar traços de alta resolução.

Em seu trabalho, Bart Adams et al. [13] propõem uma solução para as limitações mencionadas para geometrias completamente baseada em pontos. Pela representação de ambos, objeto e pincel, como coleções de pontos é possível remover a separação entre aparência e geometria. Todos os atributos e parâmetros relevantes, como pigmentos de pintura, cores, posição espacial e normais, são armazenados juntamente com a amostra.

Baseado nessa representação foi implementado um protótipo para pintura 3D interativa. Esse sistema suporta uma variedade de efeitos de pintura, incluindo difusão de pintura, pintura dourada, cromada e mosaico e renderiza o objeto com alta qualidade. Para utilização intuitiva pelo usuário foi introduzido um dispositivo de entrada com 6 graus de liberdade e realimentação tátil mostrado na figura 2.16.

### 2.8.1 Descrição do sistema

**Interface com o usuário.** Foi desenvolvida uma interface com o usuário que permite manipular o pincel, misturar tintas, mover o objeto e aplicar as tintas de maneira intuitiva (Figura 2.16). O pincel virtual é posicionado usando um dispositivo de entrada com seis graus de liberdade que provê realimentação tátil ao usuário.

**Representação do objeto e reamostragem dinâmica.** Problemas como dis-



Figura 2.16: Interface com o usuário

torção de texturas e descontinuidades nos retalhos, que são frequentemente encontrados em modelos de pintura baseados em malhas poligonais, são evitados através do uso da representação baseada em pontos da superfície e uma estratégia de reamostragem dinâmica. A ideia fundamental é aumentar a amostragem localmente se necessário e diminuí-la sempre que possível. Por exemplo, se o artista pinta uma linha fina, o sistema deve aumentar a amostragem localmente. Se mais tarde o artista pinta sobre essa linha com um pincel mais grosso, o sistema diminui a amostragem das áreas afetadas sem perder nenhuma informação geométrica.

**Pincéis virtuais.** A modelagem dos pincéis virtuais também utiliza uma superfície baseada em pontos, acoplada a um esqueleto massa-mola. O esqueleto é usado para modelar a dinâmica do pincel e as amostras da superfície armazenam informações a respeito da tinta depositada. Este modelo flexível possibilita a introdução de diferentes tipos de pincéis com tamanhos e resoluções variáveis. Detecção de colisões entre os pincéis e os modelos 3D é possível com altas taxas.

Quando a visão da superfície do objeto é ampliada (*zoom*), o pincel é diminuído de tamanho. Como resultado, a densidade de amostragem do pincel aumenta em relação a densidade de amostragem do objeto. Isso possibilita ao artista pintar pequenos detalhes no objeto. Como ambos, objeto e pincel, são representados por amostragem de pontos, uma elegante implementação de transferência bidirecional de tinta é realizada.

**Modelo de tinta.** Baseado em [22], este sistema possui diferentes tipos de tinta, como aquarela, tinta a óleo, metálica e outras tintas reflexivas. Para implementar

essa variedade de tipos, a estrutura da tinta armazenada, além de cores, outros atributos, tais como coeficiente de difusão, refletividade, brilho e etc. Este modelo de tinta foi escolhido como base por possibilitar a transferência bidirecional de tinta, sendo computacionalmente barata.

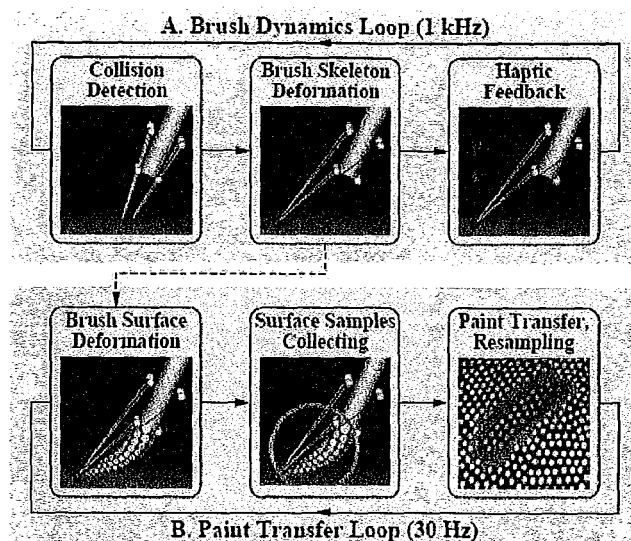


Figura 2.17: Linha de cima: laço de dinâmica do pincel. Quando uma colisão ocorre, a ponta do pincel é projetada sobre a superfície do objeto e o esqueleto do pincel é deformado de acordo com isso. A força resultante agindo sobre o cabo é enviada para o dispositivo tátil. Linha de baixo: laço de transferência de tinta. Os pontos do pincel são deformados de acordo com a deformação do esqueleto. Após coletar as amostras relevantes da superfície, a tinta é transferida entre o as amostras do pincel e da superfície.

**Desacoplamento da realimentação tátil.** Para garantir a frequência de atualização de 1KHz requerida pelo dispositivo tátil, o cálculo das forças foi desacoplado do resto da aplicação. Somente operações que sejam necessárias para simular o comportamento dinâmico do pincel, como a detecção de colisão e a deformação do esqueleto, são feitas no laço da dinâmica do pincel (Figura 2.17(A)). Todas as outras operações (mais custosas), como a deformação da superfície do pincel, transferência de tinta e reamostragem dinâmica são feitas no laço de transferência de tinta (Figura 2.17(B)), que tem velocidade de exibição 30 Hz.

## 2.8.2 Modelo e dinâmica do pincel

Para modelar o pincel virtual é necessária uma representação geométrica para sua superfície, bem como um modelo baseado em física para simular seu comportamento dinâmico. A representação da superfície do pincel também é baseada em pontos. Essas amostras são definidas em relação a um esqueleto massa-mola, que é usado para simular a dinâmica do pincel. As forças resultantes dessa dinâmica são enviadas diretamente para a interface tátil (Figura 2.18).

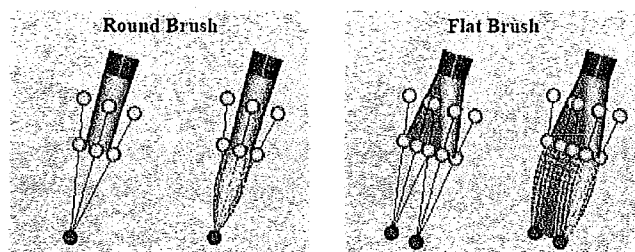


Figura 2.18: O pincel é representado como uma superfície baseada em pontos acompanhada de um esqueleto massa-mola. Esquerda: pincel arredondado consistindo de um esqueleto básico. Direita: pincel plano modelado usando duas pontas.

O esqueleto do pincel nunca deve penetrar no objeto. Por isso, realiza-se detecção de colisões para cada ponto do esqueleto massa-mola. São computadas as normais da superfície suavizada e a profundidade de penetração durante a detecção de colisão.

A força resultante pode ser calculada adicionando as forças exercidas por todas as molas do esqueleto do pincel. O torque resultante pode ser usado como realimentação tátil, se suportado. Quando o usuário aumenta a escala em alguma parte do objeto, as transformações retornadas pelo dispositivo tátil são reduzidas para permitir o controle dos movimentos mesmo em um pequeno campo de visão. As forças enviadas para o dispositivo tátil são aumentadas proporcionalmente para manter a ilusão de uma superfície dura.

As amostras que representam a superfície do pincel carregam atributos similares aos dos objetos, como posição, orientação, raio, volume de pintura por unidade de área e atributos de tinta. O pincel não deformado é modelado manualmente para lembrar um pincel real.

Para deformar a superfície do pincel, é aplicada uma combinação de *linear blend skinning* [23] e método de deformação de formas livres para geometria baseada em



pontos apresentado em [24] (Figura 2.19).

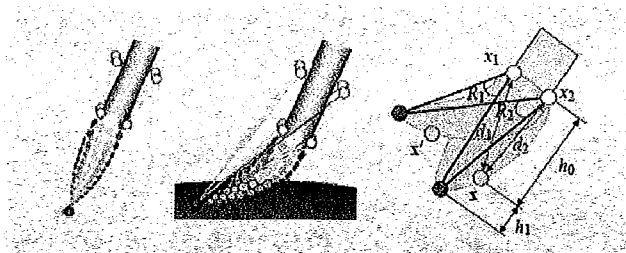


Figura 2.19: Esquerda: pincel não deformado. Meio e direita: quando o pincel é deformado, as novas posições dos pontos do pincel são calculadas das posições originais e da rotação das molas.

Quando um pincel com várias pontas se move sobre uma superfície muito curva, duas pontas podem ser lançadas para diferentes orientações da superfície (figura 2.20).

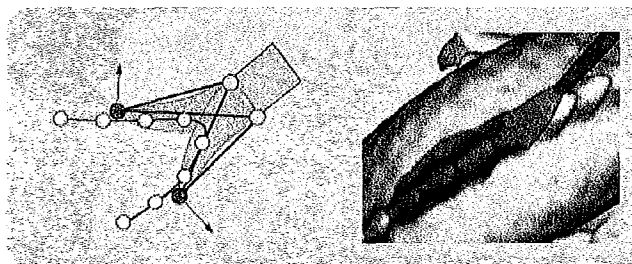


Figura 2.20: Esquerda: se dois ou mais esqueletos são restringidos a diferentes orientações da superfície, o pincel é dividido. Direita: divisão do pincel ao pintar as costas do dragão

Isto é detectado comparando as normais da superfície para cada uma das pontas. Se a orientação da superfície difere significativamente, isto é, o ângulo entre das normais é maior que 60 graus, o pincel é dividido e os pontos interiores do pincel são ativados para representar os dois volumes de pincelada. Desta forma, pode-se pintar superfícies altamente curvas como as costas do dragão (veja figura 2.20). A transferência de tinta é calculada separadamente para cada parte do pincel.

### 2.8.3 Transferência de tinta

Quando uma colisão entre o pincel e a superfície do objeto é detectada, tinta é transferida do pincel para a superfície e vice-versa. Os passos executados durante

eventos de pintura são os seguintes.

- A. Coleta de amostras da superfície.** No primeiro passo são coletados todos os pontos que são afetados pelo pincel. Os pontos do objeto estão armazenados em uma *kd-tree*, então, essa busca pode ser feita através de uma busca espacial à esfera limitante dos pontos do pincel (Figura 2.21(A)).
- B. Construção do *buffer* de tinta.** Após coletar os pontos relevantes, é construído o *buffer* de tintas em um plano definido pela normal média dos pontos coletados (Figura 2.21(B)). A posição do plano é arbitrária. As suas dimensões são escolhidas de modo que a esfera limitante do pincel esteja completamente dentro do *buffer*. Esse *buffer* de tintas é uma boa aproximação para a área da superfície que é tocada pelo pincel. Se a curvatura da região é muito alta, o pincel provavelmente será dividido. Nesse caso, são usados múltiplos *buffers* de tintas, um para cada ponta do pincel. Desta forma, as distorções são minimizadas quando pintando superfícies com alta curvatura.
- C. Projeção das amostras da superfície.** Uma implementação do método conhecido como *EWA splatting* [25] é usada para renderizar os pontos virados para frente no *buffer* de tintas (Figura 2.21(C)). Usando este algoritmo evita-se problemas de amostragem em todos os atributos. A projeção ortogonal simplifica o algoritmo de *splatting*.
- D. Projeção dos pontos do pincel.** Similarmente aos pontos do objeto, os pontos do pincel são projetados no *buffer* de tintas (Figura 2.21(D)). Somente fragmentos com profundidade maior que a profundidade já armazenada no *buffer* de tintas são escritos. Estes fragmentos representam partes do pincel que penetraram na superfície do objeto.
- E. Reprojção.** A transferência de tinta bidirecional é calculada usando o modelo proposto em [22] para determinar a cor resultante do *buffer* de tinta. Após calcular a transferência de tinta, os novos *pixels* pintados são reprojctados na superfície do objeto (Figura 2.21(E)). Se a taxa de amostragem do pincel for maior que a taxa de amostragem local do objeto, a superfície do objeto é reamostrada localmente para mais.

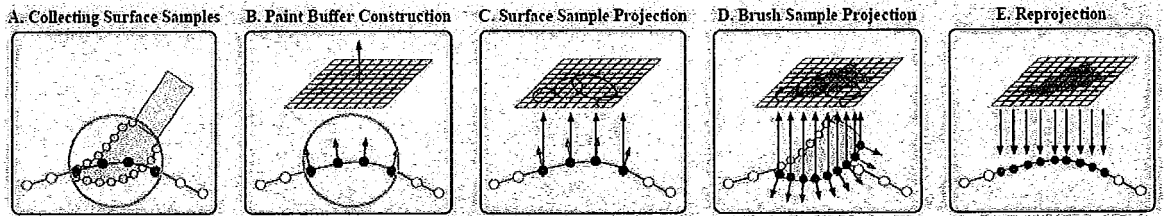


Figura 2.21: Diferentes passos executados durante um evento de pintura: A. Coleta dos pontos da superfície. B. Construção do *buffer* de tinta ortogonal à normal média da superfície. C. *EWA splatting* dos pontos coletados da superfície. D. *EWA splatting* dos pontos coletados do pincel. E. Reprojecção dos *pixels* do *buffer* de tinta sobre os pontos da superfície.

### 2.8.4 Amostragem dinâmica

Para preservar detalhes que são potencialmente criados com um pincel de alta resolução em um objeto de amostragem em baixa densidade, a superfície do objeto deve ser reamostrada para mais para poder acomodar os detalhes. Inversamente, se não existem detalhes que justifiquem uma densidade alta, a superfície do objeto necessita de uma reamostragem para menos para remover informações redundantes. Estes operadores de reamostragem dinâmica são baseados no pressuposto que a superfície original do objeto tem amostragem adequada.

A reamostragem para mais ou para menos é facilitada por uma estrutura de dados em dois níveis (Figura 2.22). Os pontos originais do modelos são armazenados em uma *kd-tree* estática. Eles nunca serão removidos, de maneira que se possa restaurar a geometria original. Entretanto, durante as reamostragens eles podem ser marcados como “mortos” para não serem renderizados. Estes pontos terão filhos, isto é, novos pontos substituindo ou complementando o pai. Filhos são associados a um único pai, que deve ser o ponto mais próximo na *kd-tree*. Filhos são armazenados em listas anexadas ao pai e são instantaneamente deletadas quando marcadas como “mortas”. Isso evita problemas de atualização na *kd-tree*, o que é muito custoso durante a pintura interativa.

**Reamostragem para mais.** O operador de reamostragem para mais será chamado sempre que o pincel pinta com menor densidade que a superfície. Ele aumenta, localmente, a quantidade de pontos na área em que mais detalhes sejam necessários

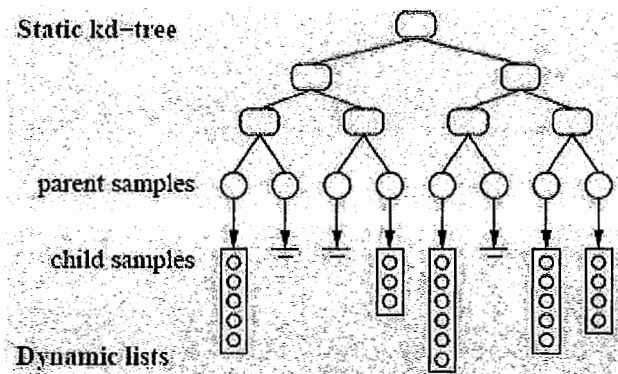


Figura 2.22: Estrutura de dados em dois níveis. Topo: os pontos da superfície original são armazenados em uma *kd-tree* estática. Baixo: quando novos pontos são adicionados, eles são armazenados em listas anexadas à *kd-tree* no ponto do objeto mais próximo.

(Figura 2.23B). Para encontrar a área afetada, o *buffer* de tinta é analisado.

**Reamostragem para menos.** A reprojeção do *buffer* de tinta pode resultar em pontos com os mesmos atributos de aparência. Isto geralmente acontece quando se pinta sobre detalhes finos com um pincel grosso. Para remover esses detalhes desnecessários, é aplicado um operador simples de reamostragem para menos (Figura 2.23C). Para cada ponto pai, são calculados os desvios de atributos de seus filhos. Quando esse desvio é inferior a um limite, os pontos filhos são removidos, o pai é ressuscitado, se necessário, e seus atributos serão a média de seus filhos.

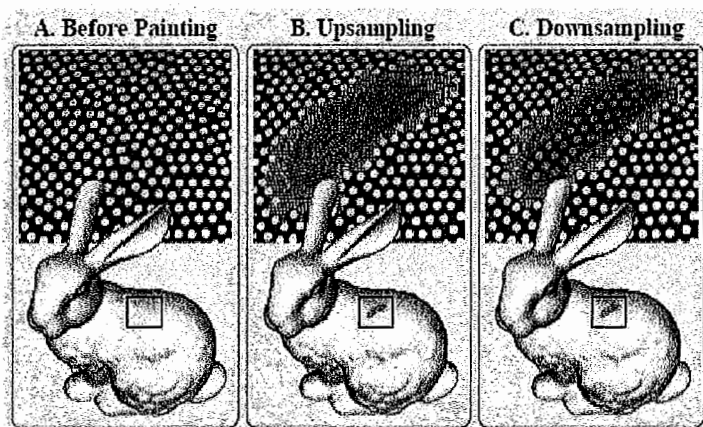


Figura 2.23: A densidade de amostragem é localmente adaptada para representar fielmente os detalhes. Esquerda: amostragem padrão. Meio: reamostragem para mais para representar o detalhe. Direita: reamostragem para menos.

### 2.8.5 Implementação

Renderizações de alta qualidade dos objetos pintados são geradas com uma implementação em *software* do *EWA splatting* [25]. Cada pintura afeta somente uma área local da superfície. Então, pode-se ter altas taxas de quadros somente atualizando a imagem já renderizada, eliminando os pontos que foram mortos e adicionando os novos pontos ou pontos ressuscitados.

Quando o objeto é rodado ou transladado, o sistema muda para uma implementação em *hardware* do *EWA splatting* por motivos de desempenho. A implementação em *software* é usada durante a pintura porque a implementação em *hardware* sofre de artefatos de quantização que ocorrem quando a imagem renderizada é atualizada localmente.

### 2.8.6 Resultados e conclusões desta implementação

A figura 2.24 mostra alguns resultados da aplicação deste modelo de pintura de objetos 3D. Este sistema provê pincéis virtuais, vários tipos de tinta e uma interface de usuário intuitiva. Para superar problemas de parametrização existentes nas aplicações de pintura, foi usado um sistema de amostragem por pontos para representar a geometria tanto da superfície dos objetos quanto dos pincéis.

O modelo de transferência de tinta aproxima localmente a superfície do objeto por um ou mais planos, manipulando também superfícies altamente curvas sem distorções. A reamostragem dinâmica da superfície possibilita a aplicação de arbitrários níveis de detalhes.

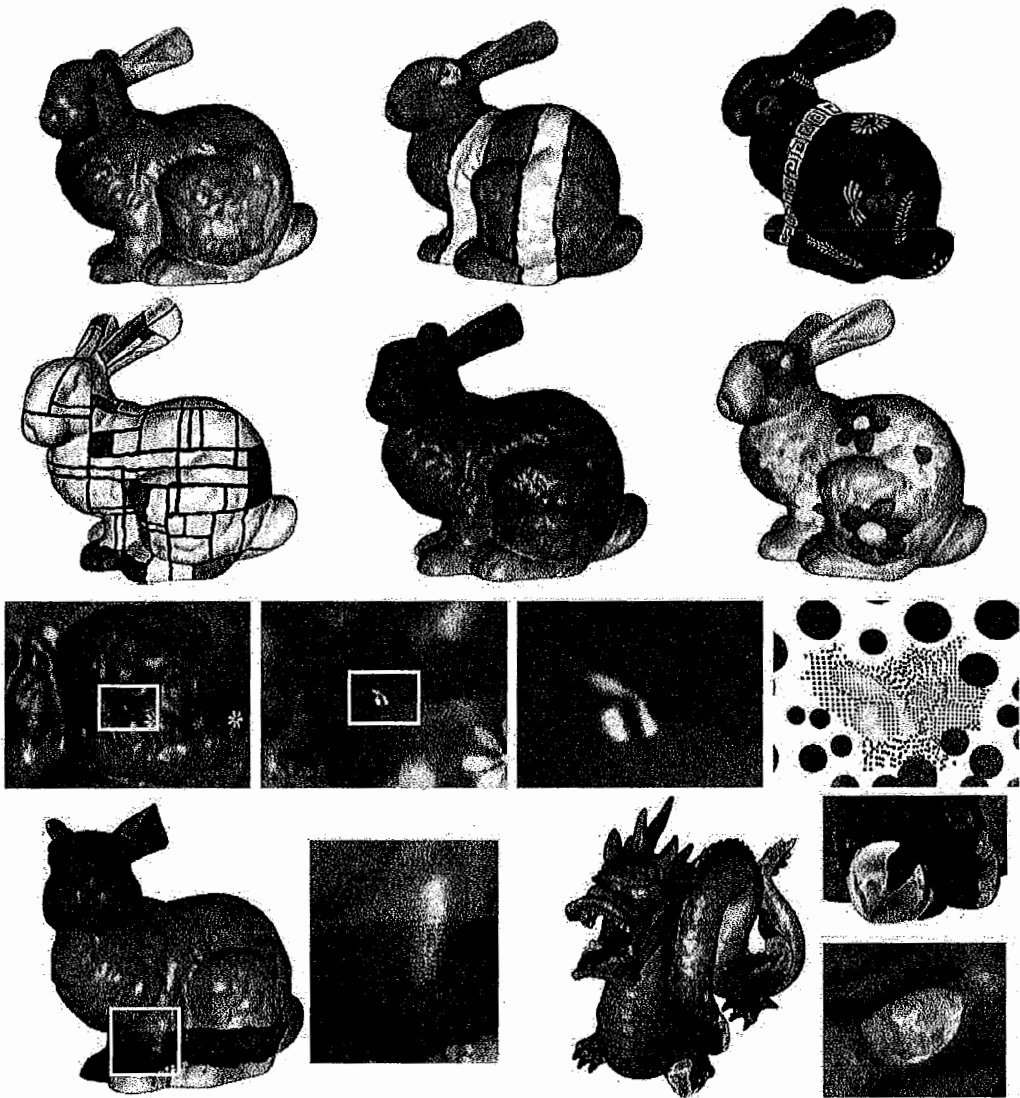


Figura 2.24: Modelos pintados com o sistema baseado em pontos

# Capítulo 3

## Método Proposto

Com o objetivo de desenvolver um sistema de pintura 3D de fácil utilização por parte do usuário, foi elaborada uma técnica de pintura de modelos 3D baseada em mapeamento de texturas. Esta técnica, baseada no trabalho de Igarashi e Cosgrove [2], não requer nenhum passo prévio para que a pintura possa ser feita, ou seja, diferentemente dos sistemas tradicionais, não é preciso especificar o mapeamento da textura com o modelo antes de iniciar o processo de pintura.

A técnica aqui proposta pode ser aplicada tanto a modelos poligonais quanto a modelos baseados em pontos. Dessa forma, vê-se que a técnica é abrangente e robusta, pois somente com algumas pequenas modificações pode-se aplicá-la a esses dois tipos de superfície.

A figura 3.1 apresenta, de maneira esquemática, uma visão geral da técnica. Basicamente a técnica possui três momentos distintos. No início, quando o usuário executa o sistema, a ideia é proporcionar a ele ferramentas que possibilitem uma interação intuitiva com o objeto. Para isso existe o cursor 3D que move-se livremente sobre a superfície do modelo. Este cursor, através de uma estrutura interna que representa as normais do objeto, se adapta perfeitamente à superfície do modelo, dando realmente a sensação de que o cursor passeia por sobre o modelo (Figura 3.1(a)).

O segundo momento é o da pintura. Através do cursor disponibilizado o usuário pode pintar sobre o modelo. A pintura deve parecer real, apesar do fato de, nesse momento, não serem geradas texturas. Os traços que o usuário pinta são apenas salvos em estruturas temporárias que, apenas mais tarde, se tornarão

realmente texturas com os correspondentes mapeamentos sobre o modelo. O aspecto real é conseguido através de cálculos de iluminação efetuados com as normais do objeto e os traços atualmente pintados. Dessa forma o usuário pinta e tem uma realimentação visual imediata e adequada (Figura 3.1(b)).

O terceiro momento é a geração das texturas. Quando o usuário efetuar uma rotação, translação ou escala no objeto, o sistema usará o conjunto de traços que foram pintados como entrada para criação de um retalho de textura. Para isso o sistema deverá usar outra estrutura interna através da qual se é capaz de identificar os vértices do objeto através a partir dos *pixels* da tela. A partir disso o sistema associa os pontos com o retalho gerado e limpa a estrutura que salva os traços para que o processo possa ser reiniciado (Figura 3.1(c)).

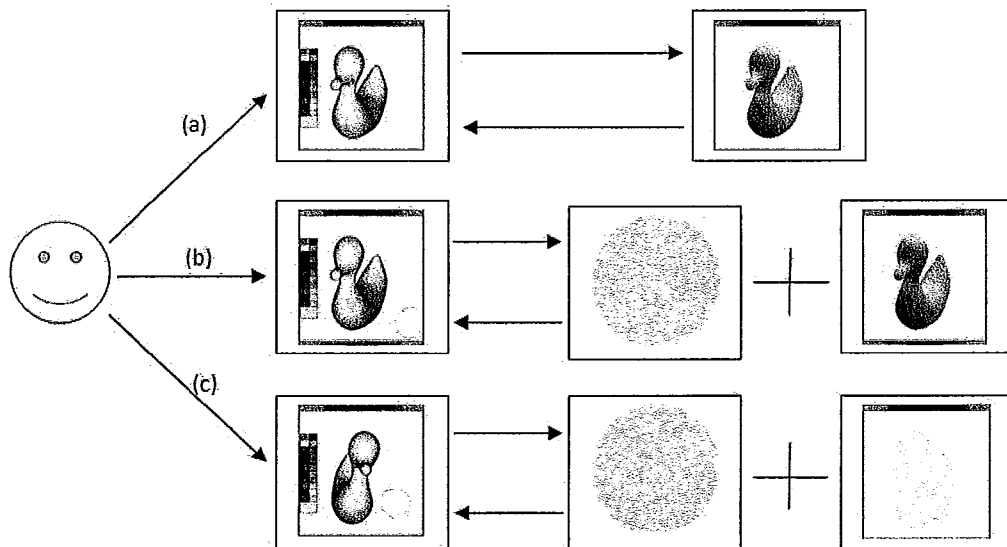


Figura 3.1: Representação esquemática da técnica proposta: (a) Cursor se adapta a curvatura do modelo baseado nas normais do objeto; (b) Ao pintar traços sobre o modelo, esses traços são salvos em uma estrutura temporária e, novamente através das normais do objeto, é produzida uma realimentação visual adequada; (c) Ao fazer rotações, por exemplo, os traços salvos serão usados como entrada para gerar um retalho, em conjunto com uma estrutura que identifica os pontos do modelo a partir dos *pixels* da tela, de modo a identificar quais pontos foram pintados para que sejam associados ao retalho

Nesse instante, volta-se ao primeiro momento identificado na técnica, onde o usuário ainda não pintou traços e dispõe do cursor para isso. Com isso, fecha-se



o ciclo e tem-se um retalho gerado com os correspondentes vértices associados à ele. Isto acontece interativamente enquanto o usuário estiver em uma sessão de pintura. Inúmeros retalhos podem ser gerados de acordo com as pinturas efetuadas.

É importante notar que esta técnica possibilita a multirresolução, que significa que o usuário poderá pintar diferentes níveis de detalhes sem perda de qualidade. Ou seja, o usuário pode pintar traços mais grossos ou detalhes mais finos que o sistema poderá tratá-los, pois a resolução de um retalho depende exclusivamente da resolução aparente da tela.

### 3.1 Implementação desenvolvida

O sistema desenvolvido trabalha similarmente a qualquer outro sistema de pintura 3D (Figura 3.2). O modelo 3D é passado para o sistema e este então o renderiza. A partir daí o usuário seleciona uma entre as ferramentas de pintura disponíveis passando a pintar diretamente sobre o modelo, com cores, padrões de cores ou efeitos como *bump mapping*.

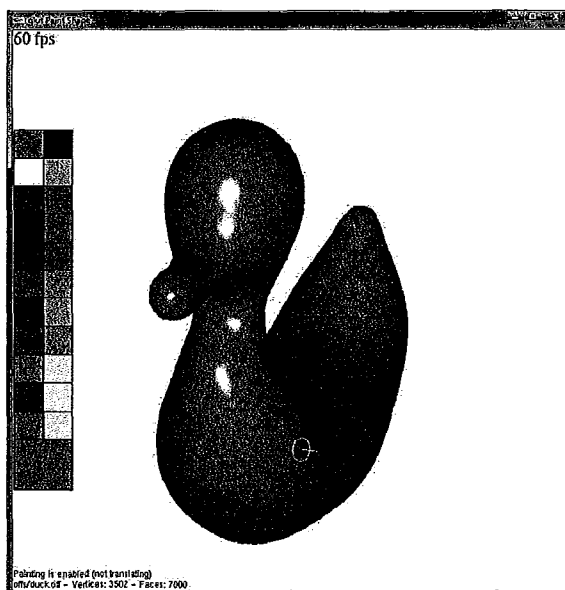


Figura 3.2: Tela Inicial do sistema implementado

A ferramenta utilizada para manipular o modelo é um cursor 3D que se adapta à curvatura do mesmo. Por meio desse cursor é possível pintar sobre a superfície do modelo, ou alterar sua visualização através de translações (*panning*), rotações ou escalas (*zoom*). O formato desse cursor é poligonal e pode variar conforme o usuário

necessite, indo desde o formato triangular (3 lados) até icosagonal (20 lados), onde seus vértices são distribuídos uniformemente sobre uma circunferência, sendo, assim, polígonos regulares. Diferentes formatos de cursor podem ser vistos na figura 3.3.

O usuário pode ainda aumentar ou diminuir o raio dessa circunferência, obtendo assim cursores maiores ou menores, sendo possível cobrir áreas maiores do modelos ou pintar pequenos detalhes sobre a superfície conforme seja necessário.

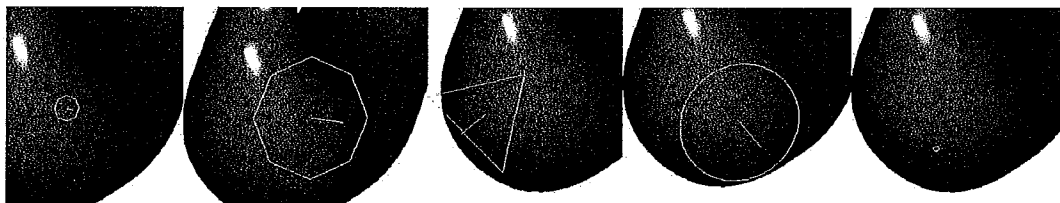


Figura 3.3: Diferentes formatos do cursor

A localização do cursor 3D é dada pela posição do *mouse*. Por conseguinte, é através da movimentação do mouse que são capturadas as movimentações do cursor. Isto é utilizado tanto para pintar o modelo como para rotacioná-lo, e ainda efetuar operações de translação e *zoom*.

Ao clicar e arrastar o *mouse* sobre a superfície do modelo, a cor do *cursor* é transferida para o modelo. Dessa forma, o modelo 3D é pintado gradativamente.

Ao final, é possível exportar, por meio de um modelo poligonal texturizado padrão, o que foi feito durante a sessão de pintura. Sendo assim, o usuário pode utilizar a pintura feita nesse sistema em outros sistemas de pintura 3D existentes que sejam compatíveis com esse padrão.

## 3.2 Interface com o usuário

Conforme dito anteriormente, ao ser iniciado, o sistema apresenta ao usuário a tela mostrada na figura 3.2. Nesta tela, pode ser visto o modelo 3D escolhido pelo usuário, o cursor 3D e a paleta de cores, responsável pela mudança do padrão de pintura do cursor.

Na paleta de cores, disponível do lado esquerdo da interface, o usuário pode escolher entre diversas cores, ou ainda optar por padrões de cores pré-definidos como listrado horizontal, listrado vertical, etc. e, ainda, aplicar efeitos como *bump*

*mapping*, de acordo com a cor selecionada.

Através da utilização do *mouse* estão disponíveis a operação de pintura, a operação de rotação, a translação e o zoom do modelo. Quando o botão direito do *mouse* é pressionado e o *mouse* é arrastado, o modelo é rotacionado seguindo a direção de arraste do *mouse*.

Com o botão do meio é possível usar a operação de escala. Ao clicar e arrastar para cima ou para baixo o tamanho da visualização do modelo será modificado. Ao arrastar para cima da posição de clique a visualização será diminuída (*zoom out*) e ao arrastar para baixo desta posição a visualização será ampliada (*zoom in*).

Com o botão esquerdo do *mouse* estão disponíveis duas operações distintas, que se diferenciam pela habilitação ou não do “modo de pintura”, através de um botão do teclado. Quando o modo de pintura está habilitado, o botão esquerdo do *mouse* servirá para pintar o modelo. Através do clique e arraste do *mouse* são criados traços, com as propriedades de tinta escolhidas, sobre a superfície do modelo.

Se o modo de pintura não está habilitado, o botão esquerdo do *mouse* servirá para transladar o modelo. Da mesma forma que as operações anteriores, o programa captura o deslocamento do *mouse* feito com este botão pressionado e o utiliza para transladar o modelo.

O cursor 3D, ao ser movimentado sobre o modelo, será transformado de modo a obter a mesma curvatura do modelo, dando a impressão que o cursor “passeia” por sobre o modelo. Esta possibilita que a tarefa de projeção dos traços pintados, típica de programas de pintura 3D tradicionais, seja eliminada, pois, desta forma, o cursor já está projetado sobre o modelo e os traços assumirão essa mesma projeção.

### 3.3 *Buffer* de normais

Esta “pré-projeção” do cursor sobre o modelo só é possível devido a existência de um *buffer* especial, chamado de *buffer* de normais. Este *buffer* é constituído das normais do modelo 3D no ângulo de visualização corrente. As normais do objeto 3D são interpoladas de modo que este *buffer* representará uma normal por *pixel* da tela, e não somente as normais originais do modelo. Ele é construído e mantido sempre em *GPU*, o que torna o processo bastante rápido.

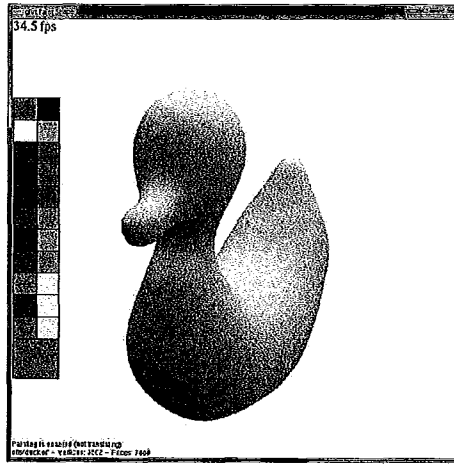


Figura 3.4: *Buffer* de normais

Para a construção desse *buffer* é efetuado o seguinte procedimento: o objeto 3D é enviado normalmente para ser renderizado pela placa de vídeo. Um processador de vértices (*vertex shader*) implementado, que é executado na *GPU*, atribui à componente de cor dos vértices enviados a normal do vértice. Para esta atribuição deve haver uma pequena adaptação dos valores, pois a faixa de valores de cada componente das normais é  $[-1, 1]$ , enquanto cores são representadas no intervalo  $[0, 1]$ . Sendo assim a atribuição feita é a seguinte:

$$corVertice = (normalVertice + (1.0, 1.0, 1.0)) \times 0.5$$

Desta maneira pode-se representar as normais do modelo no sistema de cores. Da mesma forma, a transformação inversa é necessária quando da utilização do *buffer* de normais para outros cálculos. Essa transformação mapeará a cor, que representa a normal, no intervalo  $[0, 1]$ , para a normal propriamente dita, no intervalo  $[-1, 1]$ . O cálculo feito para isso é o seguinte:

$$normalPixel = (corPixel \times 2.0) - (1.0, 1.0, 1.0)$$

Com isto, tem-se uma representação por *pixel* das normais do modelo, o que possibilita efeitos de iluminação bem mais precisos do que utilizando apenas as normais dos vértices. É através disso, também, que é possível inclinar o cursor para se adaptar à curvatura do modelo em qualquer ponto da tela, pois como tem-se a normal em cada *pixel* da tela, pode-se calcular para aquele *pixel*, em específico,

a matriz de rotação que transformará o cursor para se conformar exatamente à curvatura do modelo 3D naquela posição.

O *buffer* de normais tem ainda outra utilidade. Através da utilização da quarta componente de cor de cada elemento deste *buffer*, tem-se um mecanismo simples e direto para detecção de pertinência, ou não, de cada *pixel* da tela ao modelo.

Em outras palavras, antes do início do processo de criação do *buffer* de normais, é feita uma limpeza do *buffer*, atribuindo a todos os *pixels* do modelo a cor (1.0, 1.0, 1.0, 0.0), ou seja, com a quarta componente sendo 0.0 (zero). Na criação do *buffer*, para cada *pixel* selecionado para representar uma normal, é atribuído 1.0 (um) à sua quarta componente de cor.

Sendo assim, fica fácil saber se um *pixel* da tela pertence ou não ao modelo. Basta testar a quarta componente da normal daquele *pixel*, que, obviamente, pode ser acessada através do *buffer* de normais. Se a componente for 0.0 (zero), o *pixel* não pertence ao modelo e, assim, não entrará em cálculos de pintura ou iluminação. Se a componente for 1.0 (um), o *pixel* pertence ao modelo e deve fazer parte de todos os cálculos.

### 3.4 *Buffer* de cores

Quando o usuário pinta sobre o modelo, gera novos traços que servirão para criação de um novo retalho. No entanto, este retalho não é gerado no momento em que os traços são pintados. Ao invés disso, um *buffer* temporário é criado e, nele, são salvos todos os traços até que uma operação de rotação, translação ou escala seja efetuada.

Este *buffer* também é salvo como uma imagem em *GPU* e é atualizado sempre que uma operação de desenho for solicitada. Sua construção é feita de maneira direta apenas renderizando a área fechada formada pelo desenho do cursor, eliminando as áreas que estão fora do modelo com o auxílio do *buffer* de normais.

No momento em que um novo *buffer* de cores é criado, este não possui nenhuma informação de cor. À medida que vai sendo pintado, informações de cores vão sendo adicionadas.

Esta informação, entretanto, é insuficiente para a renderização do modelo com

os efeitos corretos de iluminação. Para tanto, novamente é utilizado o *buffer* de normais onde, através de um programa de fragmentos (*fragment shader*) em *GPU*, são acessados os valores desse *buffer* em cada *pixel* da tela para o cálculo correto da iluminação no momento do desenho do *buffer* de cores temporário. A figura 3.5 mostra como esse cálculo produz uma realimentação visual adequada.

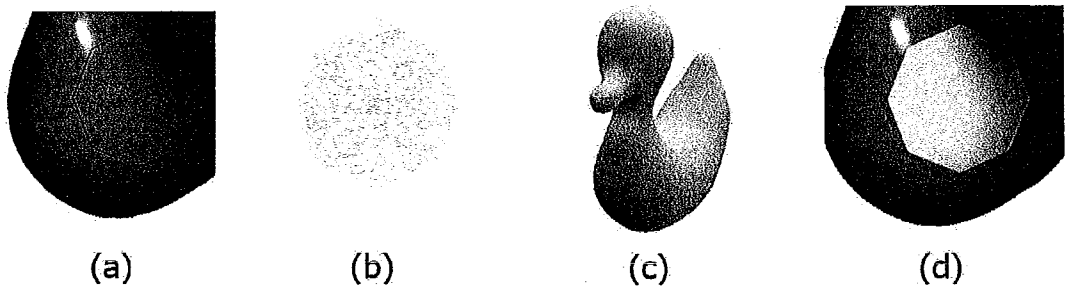


Figura 3.5: Desenho de um traço no *buffer* de cores: (a) cursor é posicionado, (b) retalho gerada para essa posição e cursor, (c) *buffer* de normais para o cálculo da iluminação, (d) desenho do retalho sobre o objeto 3D com *feedback* apropriado

Para a geração do retalho, no momento em que operações de rotação, translação ou *zoom* são efetuados, calcula-se, primeiramente, a caixa limitante (*bounding box*) dos traços existentes no *buffer* de cores. Com isso, diminui-se a área onde são necessários cálculos para a associação com o modelo.

Além disso, é necessário um procedimento para inserção dos traços que anteriormente já estavam pintados sobre o modelo na área desta caixa limitante. Desta forma, evita-se a perda de detalhes e também, no decurso da sessão de pintura, pode-se eliminar alguns retalhos que ficam obsoletos, ou seja, sem nenhuma face ou ponto do modelo associada a ele.

O próximo passo é verificar quais pontos pertencem à área da caixa limitante do *buffer* de cores, realizando tanto a associação das novas faces que ainda não foram pintadas, quanto a detecção de faces associadas a pinturas obsoletas para também associá-las à pintura que está sendo feita naquele momento.

### 3.5 *Buffer* de identificadores de pontos

Para que a localização dos pontos que estão na área da caixa limitante do *buffer* de cores possa ser possível, foi desenvolvido um mecanismo de identificação de quais

pontos correspondem a quais *pixels* da tela na visualização corrente do modelo. Ele é implementado através de um *buffer* de identificadores de pontos.

Este *buffer* faz a correspondência de cada vértice do modelo com os *pixels* da tela, de maneira a mostrar apenas os pontos visíveis. Em outras palavras, para determinada visão do objeto 3D, o *buffer* de identificadores de pontos apresentará os vértices do modelo que estão dentro da janela do programa e que não estão sendo ocluídos por nenhuma face do objeto. Um exemplo pode ser visto na figura 3.6.

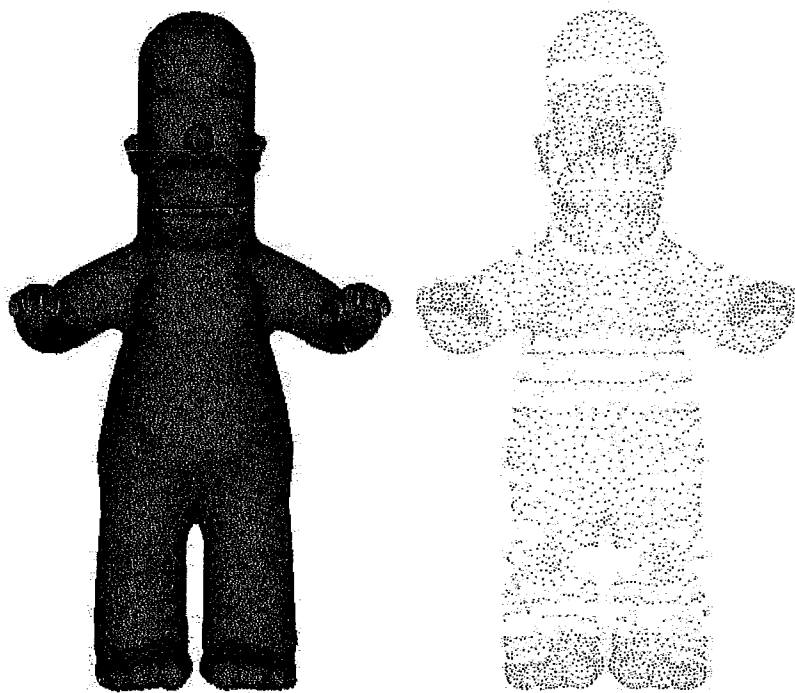


Figura 3.6: Imagem renderizada do objeto em determinada posição e seu correspondente *buffer* de identificadores de pontos

A construção deste *buffer* é feita em dois passos. Primeiramente renderiza-se o modelo 3D com a cor branca e sem iluminação. Esta renderização é usada como *buffer* de profundidade no próximo passo. Neste passo, são renderizados somente os vértices do modelo, obviamente, com as mesmas posições do primeiro passo. Sendo assim, o *buffer* de profundidade só permitirá a visibilidade dos vértices que estejam virados para frente, exatamente o objetivo proposto para este *buffer*.

Para que seja possível a identificação dos vértices para cada *pixel* da tela, são enviados para a *GPU* os identificadores dos vértices do modelo. Mais especificamente, na tupla que representa um vértice, a quarta componente é preenchida com a posição do vértice na estrutura de dados que armazena os vértices. Dessa forma é

possível identificar unicamente o vértice e acessá-lo facilmente.

No entanto, vem a tona novamente aqui o mesmo problema encontrado no *buffer* de normais. Como são renderizadas somente cores, deve haver uma codificação desses identificadores de pontos. Da mesma forma, deve haver a decodificação desse valor, em *CPU*, para a correta manipulação dos pontos.

A codificação é feita em um programa de fragmentos (*fragment shader*) onde, para cada vértice, é realizada a separação do valor do seu identificador em três canais de cores da renderização. O último canal de cor é responsável pela identificação se aquele *pixel* representa ou não um vértice do modelo (1.0 se sim, e 0.0 se não). A separação é feita da seguinte forma:

$$\begin{aligned}a &\leftarrow \text{floor}(id/100.0) \\b &\leftarrow \text{floor}((id - (a \times 100.0))/10.0) \\c &\leftarrow id - (a \times 100.0) - (b \times 10.0) \\corFragmento &= (a/100.0, b/10.0, c/10.0)\end{aligned}$$

Desta forma, obtém-se a separação dos algarismos do identificador. Ou seja, para o identificador 597, por exemplo, serão separados os algarismos 5, 9 e 7 e será renderizada a cor (0.05, 0.9, 0.7). Como constantemente modelos 3D têm mais que 1000 vértices, faz-se necessária a divisão da primeira componente por 100 para que não sejam perdidos identificadores com valores maiores que isso. Por exemplo, para o identificador 2437 será renderizada a cor (0.24, 0.3, 0.7).

Para decodificação desse valor e, por conseguinte, identificação de qual vértice está associado, efetua-se o seguinte cálculo:

$$id = \text{round}(cor[0] \times 100) \times 100 + \text{round}(cor[1] \times 10) \times 10 + \text{round}(cor[2] \times 10)$$

Com isso, reconstrói-se o valor do identificador do vértice que foi repassado pela *GPU*. Os arredondamentos são necessários devido à perda de precisão numérica dos dados na passagem da *GPU* para a *CPU*. Dessa forma, para a cor (0.24, 0.3, 0.7), tem-se reconstruído o valor 2437, identificador original do vértice.

As ferramentas acima explicitadas, são integradas de maneira que possa-se efetuar a pintura de traços, a geração de retalhos e a identificação dos pontos pintados,



a associação das coordenadas de textura de cada vértice identificado, e, por fim, o gerenciamento dos retalhos gerados. Cada um desses passos é explicado a seguir.

## 3.6 Pintura de Traços

A pintura de traços é feita quando o programa está em modo de pintura. Para desenhar um traço, posiciona-se o *mouse* no local desejado, clica-se com o botão esquerdo e arrasta-se o *mouse* para a nova posição desejada. À medida que o mouse é deslocado, o *buffer* de cores é preenchido com a representação do formato do cursor de acordo com as propriedades de cor escolhidas pelo usuário na paleta de cores.

Se o usuário ultrapassar as fronteiras do objeto 3D durante a pintura, o sistema, através do *buffer* de normais, identifica que a área não deverá ser pintada e nada faz até o momento em que o cursor retorne para a área pertencente ao modelo 3D.

Uma realimentação visual apropriada, como mostrado na figura 3.7, é propiciada ao usuário, de maneira que o resultado imediato da pintura de traços é mostrado na tela através do cálculo da iluminação dos *pixels* afetados. Isto é possível devido ao acesso ao *buffer* de normais e às propriedades da pintura, descritas pela quarta componente de cor em cada *pixel* da tela no *buffer* de cores.

Como o cursor se ajusta automaticamente à curvatura do modelo, não é preciso re-projetar o cursor para gerar os traços, visto que o cursor inclinado para se ajustar a essa curvatura já representa o que seria a projeção do mesmo sobre a superfície, desfazendo, assim, a necessidade do cálculo de projeção.

Quando o usuário executa uma operação de rotação, translação ou escala, o sistema encerra a pintura de traços no *buffer* de cores atual. Este *buffer* servirá de entrada para a criação do retalho e a correspondente associação de coordenadas de textura será efetuada. O *buffer* de cores é limpo e reinicia-se o processo de pintura de traços.

## 3.7 Criação de retalhos

Nesse momento, o *buffer* de cores é usado para criar um novo retalho. O processo para criação de um retalho segue os seguintes passos:

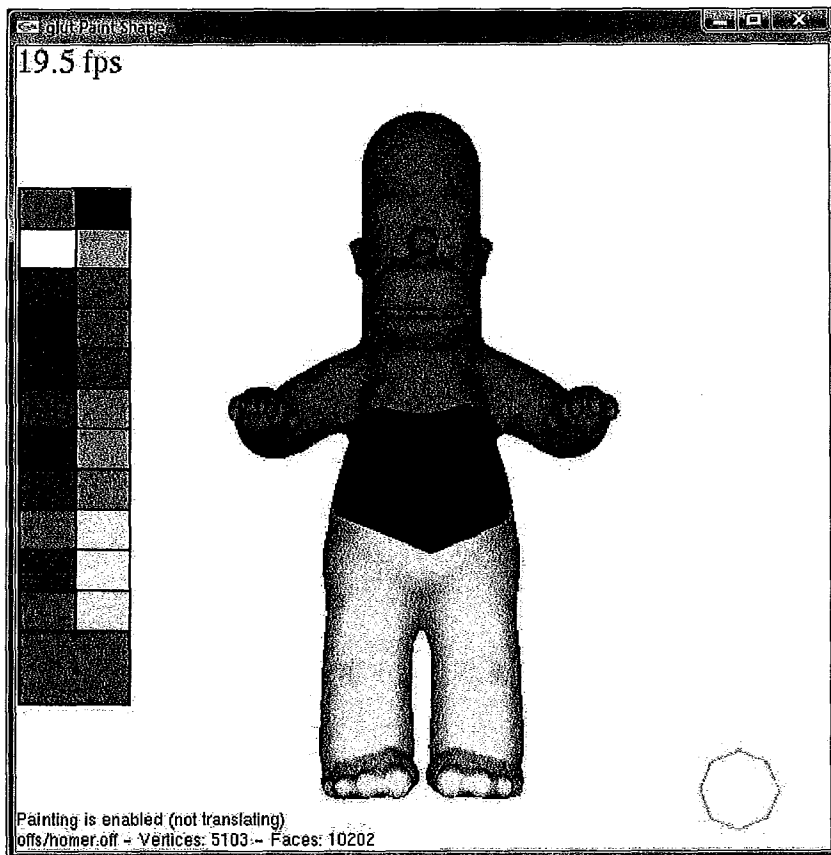


Figura 3.7: *Feedback* apropriado dos traços pintados no *buffer* de cores proporcionado pelo cálculo de iluminação utilizando o *buffer* de normais

- Calcula-se a caixa limitante B (*bounding box*) dos traços, ou seja, o menor retângulo que contém todos os traços pintados no *buffer* de cores.
- Expande-se a caixa limitante B em uma margem de segurança para garantir que detalhes não sejam perdidos nas bordas do retalho, no momento de sua associação com o modelo 3D. Essa margem de segurança corresponde ao tamanho máximo de aresta do objeto, um valor que garante a captura dos vértices necessários na coleta de pontos.
- Renderiza-se as partes já pintadas do modelo naquela área, para que o retalho possua as partes já pintadas previamente e, possivelmente, retalhos obsoletos sejam eliminados.
- Renderiza-se os novos traços pintados, que estão salvos no *buffer* de cores. Nesse momento, o que for sobreposto pelos novos traços será eliminado.

- Gera-se o retalho através da leitura da *GPU* do que foi renderizado.

Ao final desse processo tem-se o retalho contendo, além dos novos traços que foram pintados na área de sua caixa limitante, os traços que haviam sido pintados previamente na mesma área. Dessa forma, são preservados os detalhes que foram pintados previamente e surge a possibilidade de eliminar retalhos que ficarem obsoletos, ou seja, completamente sobrepostos.

Para que essa eliminação seja possível, deve-se associar os vértices ao novo retalho e, ao mesmo tempo, desassociá-los de outros retalhos que, eventualmente, possam estar associados. Ao fim, os retalhos que não estiverem associados a nenhum vértice podem ser excluídos.

### 3.8 Associação de coordenadas de textura

Com o retalho criado, o próximo passo é associar os vértices do modelo que pertencem à caixa limitante B expandida. Isso é possível através do *buffer* de identificadores de pontos.

Para que um ponto possa ser associado a um novo conjunto de coordenadas de textura ele deve satisfazer as seguintes condições:

- (a) deve ser visível na visão corrente do modelo, e
- (b) deve estar dentro da caixa limitante B.

As coordenadas de textura de cada vértice são associadas baseando-se, também, na caixa limitante B. Conforme a posição de cada vértice, o sistema associará a coordenada de textura que o acomode na caixa limitante B. Essas coordenadas de textura são utilizadas para renderização do retalho no local apropriado do modelo 3D.

O processo de geração do retalho, cálculo da caixa limitante, expansão da caixa limitante, captura de vértices, associação dos vértices com o modelo e renderização do modelo com o novo retalho pode ser visto na figura 3.8.

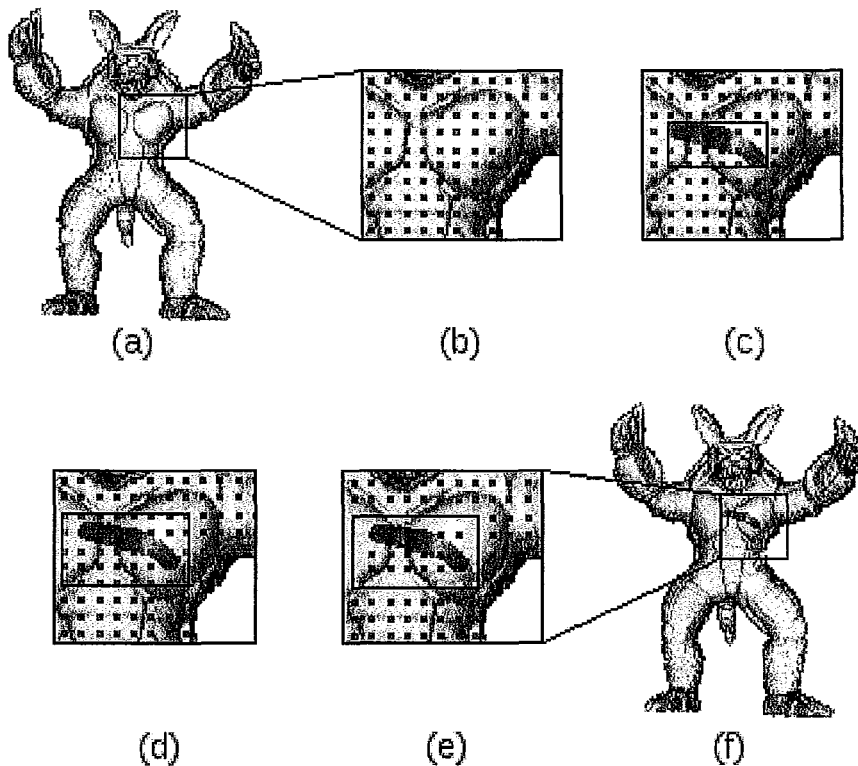


Figura 3.8: Fases para geração e associação de um novo retalho: (a) modelo de entrada para desenho dos traços, (b) pontos da área afetada, (c) caixa limitante (*bounding box*) do novo traço, (d) caixa limitante expandida para conter todos os vértices do modelo relacionados com o traço, (e) pontos críticos (verdes) possivelmente associados com mais de uma textura, (f) modelo com o novo traço

### 3.9 Gerenciamento de retalhos

Para cada novo retalho criado, associa-se a ele as faces do modelo que são cobertas por ele. Esse processo é feito de maneira simples, visto que, através do *buffer* de identificadores de pontos são identificados os vértices afetados pela pintura. Para fazer a correspondência com as faces, basta verificar dentre as faces formadas por esse conjunto de vértices, quais delas possuem todos os seus vértices marcados como afetados pela pintura.

Cada face possui uma indicação de qual retalho está associado com ela. Cada retalho possui um contador de quantas faces estão associadas a ela. Isto propicia um mecanismo simples de verificação de qual retalho pode ser excluído.

No momento em que faces são associadas a novos retalhos, verifica-se se a face

já está associada a algum retalho através da indicação citada anteriormente. Caso positivo, seu indicador de retalho é alterado para o retalho corrente e o número que indica a quantidade de faces associadas ao retalho anterior é decrementado de 1 (um).

Quando o contador de algum retalho chega a 0 (zero), é o instante em que esse retalho deve ser eliminado da memória, pois já não possui representatividade no sistema, ou seja, nenhuma face está associada a ela. Este processo pode ser visualizado na figura 3.9.

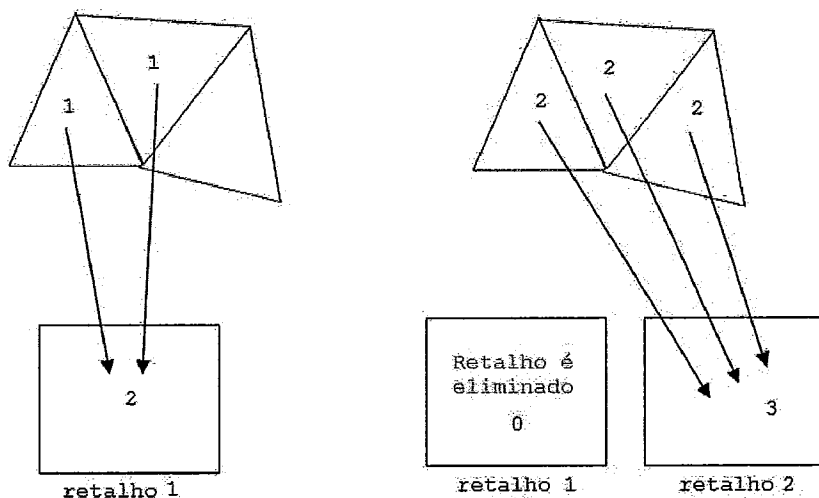


Figura 3.9: Gerenciamento de retalhos: cada face indica o retalho ao qual está mapeada e cada retalho possui um contador de quantas faces estão associadas à ela. Quando uma face é realocada para um novo retalho, o retalho anterior tem seu contador decrementado. Quando esse contador chega a zero o retalho é eliminado.

## 3.10 Empacotamento

Ao final do processo o usuário deve poder exportar seu trabalho para poder usá-lo em outro programa de pintura 3D, ou simplesmente para salvá-lo. Para isto, foi implementado um modelo de empacotamento de retalhos semelhante ao de Igarashi [2], veja seção 2.6.4.

Toma-se como entrada todos os retalhos gerados durante o processo de pintura e gera-se um atlas de textura. Cada textura é copiada para o atlas e as coordenadas UV serão recalculadas para casar com a nova posição da textura.

É importante salientar qual será o tamanho total do atlas de textura: o sistema soma a área total dos retalhos e calcula sua raiz quadrada ( $L$ ), o tamanho do atlas será um pouco maior que esse valor (multiplicado por 1.2). Mais detalhes sobre o processo de empacotamento estão presentes na seção 2.6.4.

### 3.11 Aplicação com modelos baseados em pontos

Devido aos requisitos mínimos para a aplicação deste método, pode-se utilizá-lo tanto em malhas triangulares, como vem sendo demonstrado até agora, quanto em modelos baseado em pontos. Isto é possível porque, nesses dois tipos de modelos, pode-se construir os *buffers* que dão suporte a todo o processo de pintura.

A diferença reside no tratamento da unidade básica de associação que, como no caso de malhas triangulares são os triângulos das faces, para modelos baseados em pontos serão os próprios pontos do modelo.

A utilização de modelos baseados em pontos traz, inclusive, facilidades na implementação da associação do modelo com os retalhos. Como a unidade básica são pontos, não há necessidade de verificar faces, associa-se diretamente cada vértice que for detectado como pintado.

Foi implementado um protótipo usando a renderização baseada em pontos de Marroquim et. al. [26]. Com ele percebeu-se que a técnica funciona perfeitamente também com modelos baseados em pontos.

# Capítulo 4

## Resultados e Discussão

A Figura 4.1 mostra alguns modelos pintados com o sistema implementado. Os modelos puderam ser pintados rapidamente, em apenas alguns minutos, por usuário inexperientes nesse tipo de sistema, mas experientes na utilização de computadores. Isto foi verificado através de testes com um grupo de cinco usuários sem previa orientação de como utilizar o sistema. Para cada um foram dados dois modelos e o tempo que precisassem para pintá-los. Constatou-se com os depoimentos dos usuários que o sistema possui interface amigável e fácil de manipular, visto que, nos primeiros minutos os usuários se ambientaram com o sistema e conseguiram efetuar facilmente suas pinturas.

As texturas geradas geralmente foram compactas e eficientes, pois não desperdiçam espaço. Na figura 4.2 tem-se alguns atlas produzidos pelo sistema. Como essas pinturas não foram realizadas por artistas profissionais, percebeu-se que apenas pequenas partes do modelo são pintadas, permanecendo grande parte dos mesmos não pintada. Com isto, nota-se que se fosse usado um sistema de pintura modelos 3D tradicional, as texturas geradas teriam provavelmente ocupado espaço de memória em demasia.

A máquina usada para efetuar essas pinturas foi um Intel Core2Quad Q6600 2.6GHz com 2GB de RAM e placa de vídeo NVIDIA Geforce 8600 GT. Durante a utilização do sistema percebeu-se que a taxa de quadros por segundo ficou constante em 60 quadros por segundo. Os modelos testados estão descritos na tabela a seguir com os respectivos números de vértices e faces e as respectivas taxas de quadros por segundo média na utilização do sistema.

modelo	vértices	faces	quadros por segundo
cactus.off	1494	2924	60
duck.off	3502	7000	60
homer.off	5103	10202	60
cheb.off	6669	13334	60

Pode-se perceber que a utilização da *GPU* em diversas operações do sistema, como construção dos *buffers* de normais, de cores e de identificadores de pontos, pintura de traços e desenho e manipulação do cursor, possibilitam que o sistema possua um desempenho satisfatório no *hardware* testado.

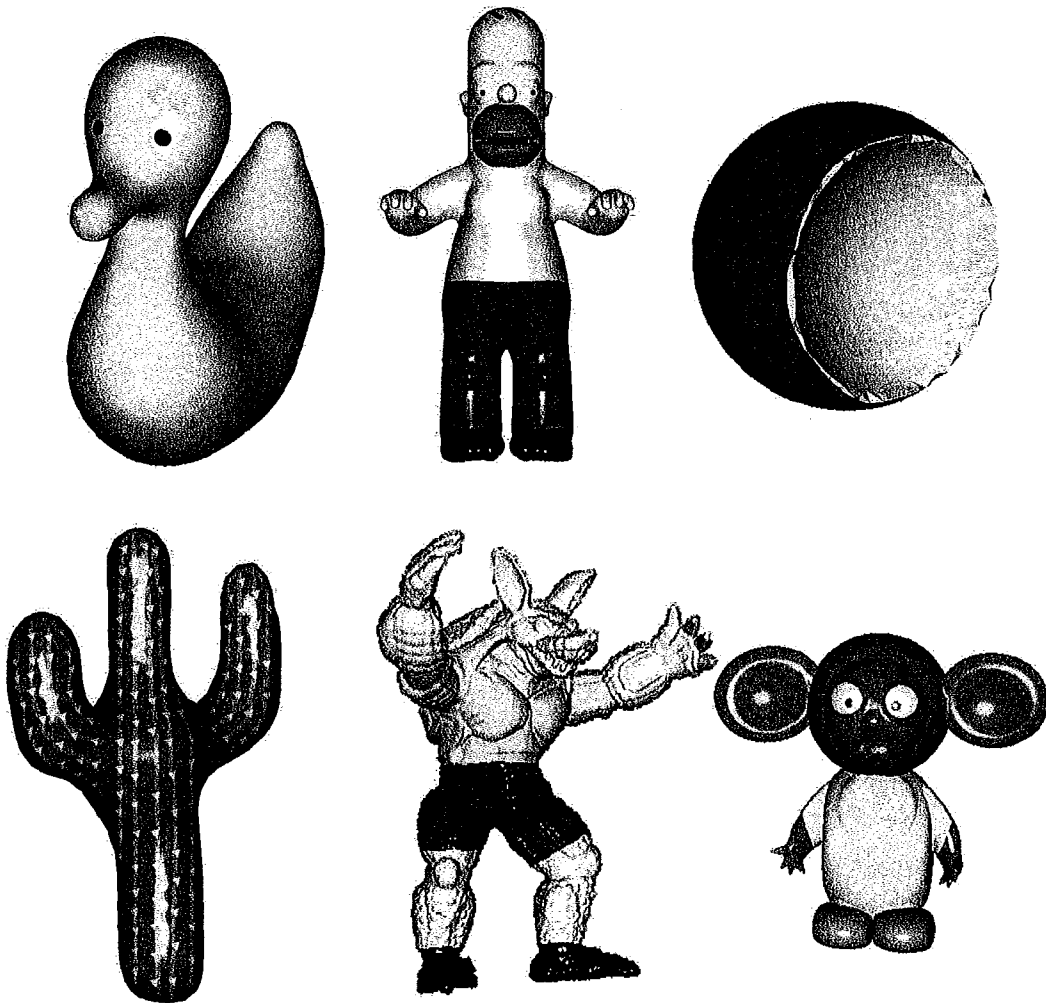


Figura 4.1: Modelos pintados com o sistema implementado.

No entanto, este desempenho é bastante dependente da *GPU* em que o sistema está sendo executado. Em *GPUs* com menor poder de processamento que a usada nos testes, o sistema tem um acentuado decréscimo em desempenho. Em particular,



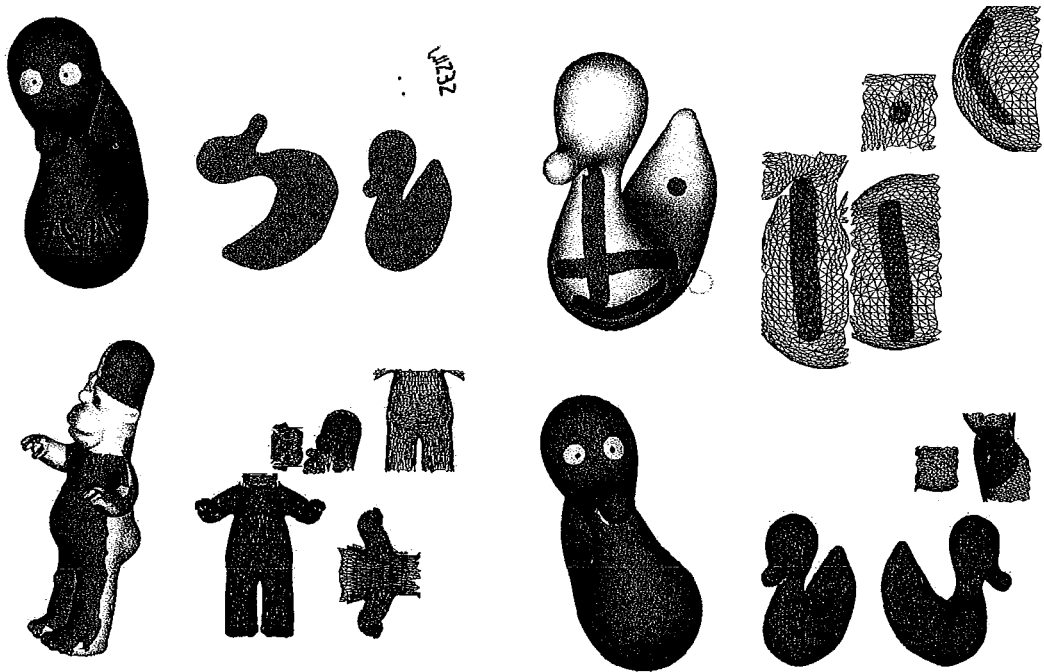


Figura 4.2: Atlas de textura gerados pelo sistema

na pintura do modelo *homer.off* usando a placa Intel GM965 X3100 observou-se uma taxa de 12 a 15 fps e na pintura do mesmo modelo usando a placa NVIDIA Geforce 6200 observou-se taxas de 8 a 10 fps. *GPUs* mais antigas, como Geforce 5500 ou Intel 945, não são suportadas pelo sistema, que requer facilidades de placas mais modernas, a saber, suportar programas de fragmento e outras funcionalidades específicas do *OpenGL 2.0*.

# Capítulo 5

## Conclusões e Trabalhos Futuros

Foi apresentada uma técnica para pintura de modelos 3D com criação automática de texturas e mapeamentos UV. As texturas criadas são compactas e eficientes na medida em que representam somente a área pintada dos objetos, diferentemente dos modelos de pintura tradicionais que mapeiam todo o modelo independente de quais áreas estejam pintadas. Ademais, o sistema dispensa a necessidade da definição de um mapeamento UV previamente.

Isso faz com que o usuário possa pintar em qualquer resolução sem perda de qualidade, ou seja, o sistema provê facilidades para pintura multirresolução. Sendo assim, o usuário pode pintar modelos sob qualquer ângulo e sujeitos a quaisquer fatores de escala. O sistema foi desenvolvido especialmente para usuários comuns, sendo de menos valia para artistas e *designers* profissionais.

A geração e o gerenciamento das texturas em *GPU* tornam o programa bastante eficiente. Como as texturas são sempre mantidas em *GPU*, seu gerenciamento torna-se mais simples. Este gerenciamento de texturas permite que a alocação de memória para o programa seja mínima, visto que somente existirão texturas para as áreas pintadas e, se uma área já pintada for pintada novamente, a textura anterior é expurgada, dando lugar à nova textura.

Como são necessários apenas um *buffer* de normais e um *buffer* de pontos para que o sistema possa ser implementado, torna-se fácil aplicar a técnica tanto para modelos baseados em pontos quanto para modelos baseados em malhas poligonais. Para isto basta alterar a unidade base de associação do sistema, de faces, no caso de malhas poligonais, para vértices, no caso de modelos baseados em pontos.

A utilização do *buffer* de normais permite ainda que o cursor seja projetado automaticamente conforme a curvatura do modelo, e isso elimina a necessidade de re-projeção das texturas pintadas antes de serem associadas ao modelo.

Como esses *buffers* também são mantidos em *GPU*, torna-se simples acessá-los para desenhar as texturas, desenhar o cursor, cálculos de iluminação e verificações de pertinência ao modelo. Antecipa-se ainda que seja relativamente fácil usá-los para influenciar outras propriedades como as próprias normais do modelo, criando efeitos mais interessantes, como *bump mapping*.

Como trabalhos futuros pode-se usar a informação passada pelo cursor ao fazer a pintura para fazer outro tipo de processamento como deformações no modelo, simplificações, *diffusion* e/ou *reaction diffusion* [27].

Outra vertente é implementar, além das cores e padrões de cores comuns e do *bump mapping* simples, efeitos mais sofisticados como *displacement mapping* [28] e um *bump mapping* mais elaborado [29, 30], por exemplo.

Pode-se também alterar a geometria do modelo inserindo novos pontos próximos à borda de cada textura, evitando a necessidade de se expandir a caixa limitante dos traços para poder abranger todos o polígonos necessários para que eles possam ser desenhados.

Mais um objetivo a ser atacado pode ser melhorar o gerenciamento de texturas, utilizando estruturas de dados mais avançadas no momento da associação de cada textura com seu conjunto de polígonos, pois no momento usa-se apenas uma lista comum.

# Referências Bibliográficas

- [1] LÉVY, B., PETITJEAN, S., RAY, N., et al., “Least squares conformal maps for automatic texture atlas generation”, *ACM Trans. Graph.*, v. 21, n. 3, pp. 362–371, 2002.
- [2] IGARASHI, T., COSGROVE, D., “Adaptive unwrapping for interactive texture painting”. In: *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 209–216, ACM: New York, NY, USA, 2001.
- [3] PAINT, “Microsoft”, <http://www.microsoft.com>, 2009.
- [4] KOLOURPAINT, “A free, easy-to-use paint program for KDE”, <http://kolourpaint.sourceforge.net>, 2008.
- [5] PHOTOSHOP, “Adobe”, <http://www.adobe.com>, 2009.
- [6] CORELDRAW, “Corel Corporation”, <http://www.corel.com.br>, 2009.
- [7] GIMP, “GNU Image Manipulation Program”, <http://www.gimp.org>, 2009.
- [8] ILLUSTRATOR, “Adobe”, <http://www.adobe.com>, 2009.
- [9] 3DSTUDIOMAX, “Autodesk”, <http://www.autodesk.com>, 2009.
- [10] MAYA, “Autodesk”, <http://www.autodesk.com>, 2009.
- [11] BLENDER3D, “Blender Foundation”, <http://www.blender.org>, 2009.
- [12] RITSCHER, T., BOTSCH, M., MÜLLER, S., “Multiresolution GPU Mesh Painting”. In: *Eurographics 2006 Short Papers*, pp. 17–20, 2006.
- [13] ADAMS, B., WICKE, M., DUTRÉ, P., et al., “Interactive 3D Painting on Point-Sampled Objects”. In: *Eurographics Symposium on Point-Based Graphics 2004*, Zurich, Switzerland, June 2-4 2004.

- [14] HECKBERT, P. S., “Survey of texture mapping”, *IEEE Comput. Graph. Appl.*, v. 6, n. 11, pp. 56–67, 1986.
- [15] HANRAHAN, P., HAEBERLI, P., “Direct WYSIWYG painting and texturing on 3D shapes”, *SIGGRAPH Comput. Graph.*, v. 24, n. 4, pp. 215–223, 1990.
- [16] AMAZON3DPAINT, “Interactive Effects, Inc.” <http://www.ifx.com>, 2008.
- [17] DEEPPAINT3D, “Right Hemisphere Ltd.” <http://www.righthemisphere.com>, 2008.
- [18] PAINTER3D, “Metacreations”, <http://www.metacreations.com>, 2008.
- [19] IGARASHI, T., MATSUOKA, S., TANAKA, H., “Teddy: a sketching interface for 3D freeform design”. In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, p. 21, ACM: New York, NY, USA, 2007.
- [20] LOSASSO, F., HOPPE, H., SCHAEFER, S., et al., “Smooth geometry images”. In: *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pp. 138–145, Eurographics Association: Aire-la-Ville, Switzerland, Switzerland, 2003.
- [21] ZORIN, D., SCHRÖDER, P., SWELDENS, W., “Interactive multiresolution mesh editing”. In: *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 259–268, ACM Press/Addison-Wesley Publishing Co.: New York, NY, USA, 1997.
- [22] BAXTER, B., SCHEIB, V., LIN, M. C., et al., “DAB: interactive haptic painting with 3D virtual brushes”. In: *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 461–468, ACM: New York, NY, USA, 2001.
- [23] LEWIS, J. P., CORDNER, M., FONG, N., “Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation”. In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 165–172, ACM Press/Addison-Wesley Publishing Co.: New York, NY, USA, 2000.

- [24] PAULY, M., KEISER, R., KOBBELT, L. P., et al., “Shape modeling with point-sampled geometry”, *ACM Trans. Graph.*, v. 22, n. 3, pp. 641–650, 2003.
- [25] ZWICKER, M., PFISTER, H., VAN BAAR, J., et al., “Surface splatting”. In: *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 371–378, ACM: New York, NY, USA, 2001.
- [26] MARROQUIM, R., KRAUS, M., CAVALCANTI, P. R., “Efficient Point-Based Rendering Using Image Reconstruction”. In: *Symposium on Point-Based Graphics 2007, Prague-Czech Republic*, September 2007.
- [27] TURK, G., “Generating textures on arbitrary surfaces using reaction-diffusion”, *SIGGRAPH Comput. Graph.*, v. 25, n. 4, pp. 289–298, 1991.
- [28] HIRCHE, J., EHLERT, A., GUTHE, S., et al., “Hardware accelerated per-pixel displacement mapping”. In: *GI '04: Proceedings of Graphics Interface 2004*, pp. 153–158, Canadian Human-Computer Communications Society: School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004.
- [29] KAWAI, N., “Bump mapping onto real objects”. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, p. 12, ACM: New York, NY, USA, 2005.
- [30] WANG, J., SUN, J., “Real-time bump mapped texture shading based-on hardware acceleration”. In: *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pp. 206–209, ACM: New York, NY, USA, 2004.
- [31] IGARASHI, T., MATSUOKA, S., TANAKA, H., “Teddy: a sketching interface for 3D freeform design”. In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, p. 21, ACM: New York, NY, USA, 2007.
- [32] CONWAY, M., AUDIA, S., BURNETTE, T., et al., “Alice: lessons learned from building a 3D system for novices”. In: *CHI '00: Proceedings of the*

*SIGCHI conference on Human factors in computing systems*, pp. 486–493, ACM: New York, NY, USA, 2000.

- [33] VIANA, J. R. M., ESPERANÇA, C., MARROQUIM, R., “3D Texture Paint of Point Models”. In: *Proceedings of the XXI SIBGRAPI*, Brazilian Symposium on Computer Graphics and Image Processing: Campo Grande, MS, Brasil, 2008.