5
Implementação

Hoje em dia, a qualidade visual das aplicações digitais vem crescendo, apresentando imagens cada vez mais realistas e impressionantes. Essa tendência também se aplica para personagens virtuais criados principalmente para os jogos de computador e videogames.

O *VisionLab*, Laboratório de Visualização, TV/Cinema Digital, Jogos e Efeitos Especiais da PUC, está desenvolvendo um motor de jogos (*engine* de *games*) 3D, chamado *Mob3D*. Como todo *engine*, esse está sendo desenvolvido de forma genérica, permitindo assim a criação de diferentes tipos de jogos tridimensionais. Para atender os mais variados tipos de desenvolvedores e usuários, dentro do projeto do motor, há a necessidade da existência de um módulo para simulação de personagens, sendo esse o principal motivo do estudo das técnicas apresentadas até então nessa dissertação.

Nesse capítulo é apresentada uma breve visão do *Mob3D* e da contextualização dentro do *engine* do módulo de renderização de pele humana criado. Também são apresentados detalhes de implementação desse módulo desde seu ambiente de desenvolvimento até os *inputs* e *outputs* gerados pelos algoritmos estudados no capítulo 4.

5.1.
Visão Geral do Mob3D

O módulo de renderização de pele humana desenvolvido com base nas técnicas apresentadas no capítulo 4, embora seja genérico, foi concebido com o intuito de funcionar dentro do sistema *Mob3D*. O *Mob3D* é um motor de jogos 3D para *Windows* desenvolvido em C++ com base na API do *Direct3D* e o SDK do *DirectX*. Dentro do *Mob3D*, existe um conjunto de módulos para aplicações em tempo real os quais utilizam a linguagem HLSL para programação em placa gráfica. O módulo de geração de pele humana é um aglomerado de *shaders* que está acoplado a esse conjunto de módulos e permite a utilização de qualquer uma

das técnicas estudadas anteriormente no desenvolvimento de novas aplicações. Sendo assim, a escolha da técnica fica de acordo com a necessidade/desejo do usuário/desenvolvedor. A Figura 25 ilustra a arquitetura geral do *Mob3D* assim como o contexto onde o módulo de renderização de pele humana está inserido.

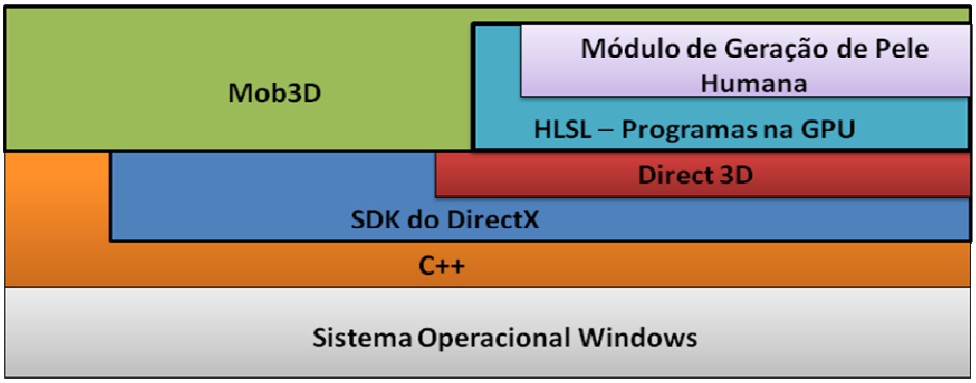


Figura 25 – Arquitetura geral do *Mob3D* e o contexto do Módulo de Geração de Pele Humana.

5.2. Ambiente de Desenvolvimento e Arquitetura da Solução

As técnicas estudadas no capítulo 4 estão projetadas para serem incorporadas em aplicações em tempo real que utilizam a GPU. Conforme descrito na seção 5.1, a linguagem de programação em placa gráfica utilizada pelo *Mob3D* é a HLSL. O desenvolvimento de códigos que executam na placa gráfica do computador não é uma tarefa simples, pois as ferramentas com esse intuito são escassas. Além disso, essas ferramentas nem sempre possuem atributos que permitem verificar os resultados obtidos com o código desenvolvido, dificultando ainda mais o processo de desenvolvimento.

Embora haja poucas, existem ferramentas no mercado que permitem o desenvolvimento de *shaders* para a GPU. Nessa dissertação foi utilizada o *RenderMonkey* da AMD. Esse software permite que os *shaders* desenvolvidos possam ser criados, atrelados e testados sem a necessidade da criação de um código principal (*core)* que os executem (no estilo de uma aplicação C++). Com isso, apenas os *shaders* necessários para a criação do módulo de renderização de pele humana foram criados, sem a necessidade de código adicional.

5.2.1. Organização da Solução

O objetivo dessa dissertação é criar uma série de *shaders* para serem utilizados dentro do *Mob3D*. Porém, para alcançar esse intuito, uma solução mais completa, com múltiplos módulos, foi gerada para obter uma formulação mais ampla do problema e para que os testes pudessem ser realizados de forma mais granular e estruturada. Com base nisso e na ferramenta de desenvolvimento utilizada, o projeto está organizado da seguinte maneira:

- *Workspace*: onde ficam armazenadas as variáveis globais como posição da luz, cor da luz, texturas, malhas 3D, etc. Essas variáveis são parâmetros dos *shaders* e automaticamente são reconhecidas pelos mesmos no *RenderMonkey*, por estarem localizadas em um ambiente global. Quando o módulo estiver inserido no *Mob3D*, essas variáveis devem ser passadas como parâmetro para os *shaders* pelo programa principal.

- Módulos: conjunto de *shaders* com um objetivo. Os seguintes módulos foram criados:

  - *Specular*, para verificação do algoritmo de reflexão especular;
  - *Original Texture*, para testar a leitura da malha 3D e do mapa de cor;
  - *Beckmann Texture*, para averiguar o cálculo da distribuição de Beckmann;
  - *Irradiance Texture*, para testar a geração do mapa de irradiância;
  - *Shadow Map*, para verificar o mapa de sombra criado;
  - *TSM*, para testar a geração do mapa de sombra translúcido modificado;
  - *Stretch Texture*, para gerar o mapa de correção da distorção causada pelo *unwrap* da malha;
  - *Convolution*, para verificar o efeito das convoluções sobre a textura de irradiância;
  - *Main Module*, que é o módulo com o código completo de geração de pele humana a ser acoplado ao motor de jogos *Mob3D*.

Como pode ser verificado, os módulos criados têm como objetivo verificar de forma mais granular a corretude de pequenas etapas de cada uma das técnicas

de renderização de pele humana estudadas. Sendo assim, eles possuem interligação entre si, principalmente com o módulo principal. De forma a reduzir ou até evitar repetição de código e permitir o reuso das principais funcionalidades dos módulos, foram criados arquivos com os principais trechos de código no ambiente externo à ferramenta de desenvolvimento e organizados em pastas de acordo com as suas funções. Essa centralização permite que uma correção em um módulo devido a um teste realizado afete todos os módulos que necessitem daquela funcionalidade. A Figura 26 demonstra a forma de organização adotada.

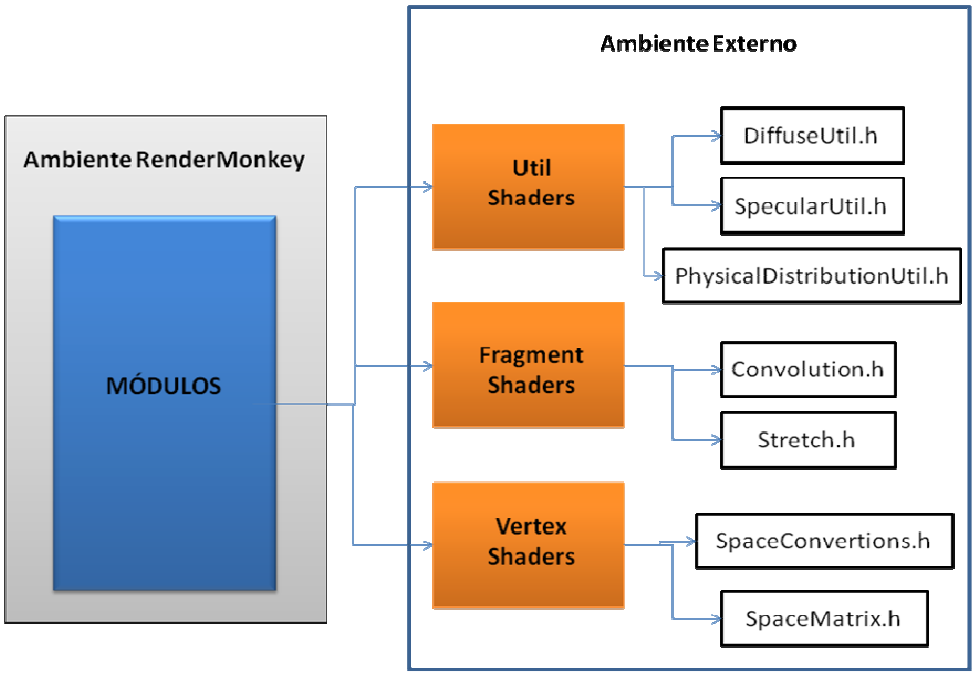


Figura 26 – Organização dos arquivos, de acordo com suas funções, fora do *RenderMonkey* e com os trechos de código mais importantes.

5.2.2. Shaders

A GPU é utilizada para melhorar o desempenho das aplicações gráficas, principalmente daquelas que devem executar em tempo real. A quantidade de processadores internos que possui permite à GPU que muitos dados sejam processados de forma paralela com um único comando.

As instruções processadas de forma simultânea pela GPU seguem o *pipeline* gráfico padrão, o qual é dividido basicamente em duas partes: geometrias e fragmentos. A parte que processa as geometrias é responsável por realizar

transformações de espaço, iluminar a cena, entre outras atividades, enquanto que a parte responsável pelos fragmentos realiza tarefas como combinação de pixels e mapeamento de textura.

As linguagens *shaders* têm como objetivo substituir partes do processamento padrão do *pipeline* gráfico por códigos próprios, os quais geralmente são mais otimizados ou disponibilizam técnicas mais avançadas do que as existentes no *pipeline* padrão. Tais códigos funcionam diretamente na placa gráfica e usufruem das mesmas vantagens de processamento paralelo que a GPU disponibiliza. Sendo assim, os códigos *shaders* são divididos em dois tipos: *vertex shader* e *fragment* (ou *pixel*) *shader* (de fato os *shaders* possuem um terceiro tipo chamado *geometry shader*, o qual não foi utilizado para a solução dessa dissertação). O *vertex shader* é responsável pelo processamento dos vértices das geometrias e o *pixel shader*, pelo processamento dos fragmentos. Conforme mencionado anteriormente, a linguagem de *shader* utilizada nesse projeto é a HLSL (*ShaderModel 3.0*).

Muitas soluções desenvolvidas para aplicações gráficas não são geradas com apenas uma passagem dos vértices e texturas pelo *pipeline* gráfico. Tais soluções utilizam múltiplas passadas (passos), cada uma responsável por uma determinada ação para obter os resultados desejados. Essa metodologia também se aplica para o desenvolvimento do módulo de renderização da pele humana com base nas técnicas estudadas do capítulo 4, conforme pode ser resumidamente visto nas Figuras 4 e 12. Cada parte do fluxograma dessas figuras representa basicamente um passo no *pipeline* gráfico.

Um passo é composto por um conjunto de variáveis locais, um *vertex shader* e um *fragment shader*. Dessa forma, temos que o módulo principal para geração de pele humana é estruturado de acordo com a Figura 27.

Figura 27 – Organização do módulo de geração de pele humana com vários passos.

As Tabelas 3 e 4 mostram a quantidade de passos necessários para a execução das técnicas de Gosselin et al. (2004a) e d'Eon e Luebke (2007), respectivamente. Além disso, é apresentado em que contexto várias etapas de suas técnicas são realizadas.

Tabela 3 – A tabela mostra todos os passos para realização da técnica de Gosselin et al. (2004a). Além disso, é exibido em que passo e em qual *shader* cada etapa da técnica deve ser aplicada.

Passo	Objetivo	*Vertex Shader*	*Pixel Shader*
1 – *Shadow Map*	Criação do mapa de sombra.	Coloca a luz como ponto de vista do objeto para renderizar os pontos iluminados. Passa a distância *z/w* para o *pixel shader* (seção 4.1.1).	Escreve a distância no mapa de sombra (seção 4.1.1)
2 – *Pre Pass 1* (opcional)	Otimização da técnica.	Projeta uma malha retangular para mapeamento e	Verifica se o pixel do mapa de cor é válido. Caso não

		processamento da textura de entrada no *fragment shader*.	seja, escreve *no z-buffer* um valor diferente de *0.0* para evitar futuro processamento (seção 5.3.1).
3 – *Pre Pass 2* (opcional)	Otimização da técnica.	*Unwrap* da malha (seção 4.1.2) para verificar a qual pixel o vértice processado equivale. Esse pixel é ignorado nos passos seguintes caso seu vértice não se encontre dentro do *frustum* de visão e não seja visto pelo observador (5.3.1).	Nada a realizar.
4 - *Irradiance*	Uso do espaço da textura para iluminar a malha.	*Unwrap* da malha (seção 4.1.2). Também são passados para o *pixel shader* os vetores para o cálculo de iluminação.	Cálculo de iluminação difusa com sombra (seções 4.1.1 e 4.1.2). Gera o *Light Map* com o *kernel size* armazenado no canal alfa (vide seção 5.3).
5 à 7 - *Blur*	Reproduzir o *subsurface scattering* da pele.	Projeta uma malha retangular para mapeamento e processamento da textura de entrada no *fragment shader*.	Processo de convolução da textura de irradiância por um filtro de *Poisson* (seção 4.1.3).

8 – *Final Rendering*	Gera a imagem final simulando a pele humana.	Simples passagem da malha e vetores para processamento no *fragment shader*.	Aplicação da textura final, após as convoluções, na malha, adição de cor e da reflexão especular (seções 4.1.4 e 4.1.5).

Tabela 4 - A tabela mostra os passos para realização da técnica de d'Eon e Luebke (2007). Além disso, é exibido em que passo e em qual *shader* cada etapa da técnica descrita na seção 4.2 deve ser aplicada.

Passo	Objetivo	*Vertex Shader*	*Pixel Shader*
1 – *TSM Modified*	Criação do mapa de sombra translúcida modificado.	Coloca a luz como ponto de vista do objeto para renderizar os pontos iluminados. Passa a distância *z/w* e as coordenadas de textura para o *pixel shader* (seções 4.1.1 e 4.2.7).	Escreve a distância e as coordenadas de textura no mapa de sombra (seções 4.1.1 e 4.2.7).
2 – *Pre Pass 1* (opcional)	Otimização da técnica.	Projeta uma malha retangular para mapeamento e processamento da textura de entrada no *fragment shader*.	Verifica se o pixel do mapa de cor é válido. Caso não seja, escreve *no z-buffer* um valor diferente de *0.0* para evitar futuro processamento (seção 5.3.1).
3 – *Pre Pass 2* (opcional)	Otimização da técnica.	*Unwrap* da malha (seção 4.1.2) para verificar a qual pixel	Nada a realizar.

		o vértice processado equivale. Esse pixel é ignorado nos passos seguintes caso seu vértice não se encontre dentro do *frustum* de visão e não seja visto pelo observador (5.3.1).	
4 - *Irradiance*	Uso do espaço da textura para iluminar a malha.	*Unwrap* da malha (seção 4.1.2 e 4.2.2). Também são passados para o *pixel shader* os vetores para o cálculo de iluminação.	Cálculo de iluminação difusa com sombra, parte da simulação de não uniformidades da pele e da conservação de energia (seções 4.1.1, 4.1.2, 4.2.2, 4.2.5 e 4.2.6). Além disso, é armazenada a distância corrigida no canal alfa do mapa de irradiância gerado (seção 4.2.7).
5, 7, 9, 11 e 13 - *ConvolutionU*	Reproduzir o *subsurface scattering* da pele.	Projeta uma malha retangular para mapeamento e processamento da textura de entrada no *fragment shader*.	Processo de convolução da textura de irradiância na direção *U* para cada uma das gaussianas que formam o perfil de difusão. Utiliza

			o *Stretch Map* para corrigir o raio do *blur* (seções 4.2.3 e 4.2.4).
6, 8, 10, 12 e 14 – *ConvolutionUV*	Reproduzir o *subsurface scattering* da pele.	Projeta uma malha retangular para mapeamento e processamento da textura de entrada no *fragment shader*.	Processo de convolução da textura de irradiância na direção *V* para cada uma das gaussianas que formam o perfil de difusão. Utiliza o *Stretch Map* para corrigir o raio do *blur* (seções 4.2.3 e 4.2.4).
15 – Final Rendering	Gera a imagem final simulando a pele humana	Simples passagem da malha e de vetores para processamento no *fragment shader*.	Combinação linear das texturas convoluídas para gerar o perfil de difusão junto com parte do cálculo de conservação de energia e da simulação de não uniformidades na pele. Também é realizada a simulação do espalhamento global em regiões finas da pele e adicionada a

			reflexão especular à malha (seções 4.2.5, 4.2.6, 4.2.7, 4.2.8 e 4.1.4).

5.2.3. Entradas e Saídas

Nessa seção são apresentadas todas as entradas e saídas dos algoritmos estudados no capítulo 4. As Figuras 28 e 29 ilustram de forma geral os principais dados iniciais e os resultados obtidos do módulo de renderização da pele humana para as técnicas de Gosselin et al. (2004a) e d'Eon e Luebke (2007). As Tabelas 5 e 6 descrevem detalhadamente a maioria das entradas e saídas de cada passo dos algoritmos.

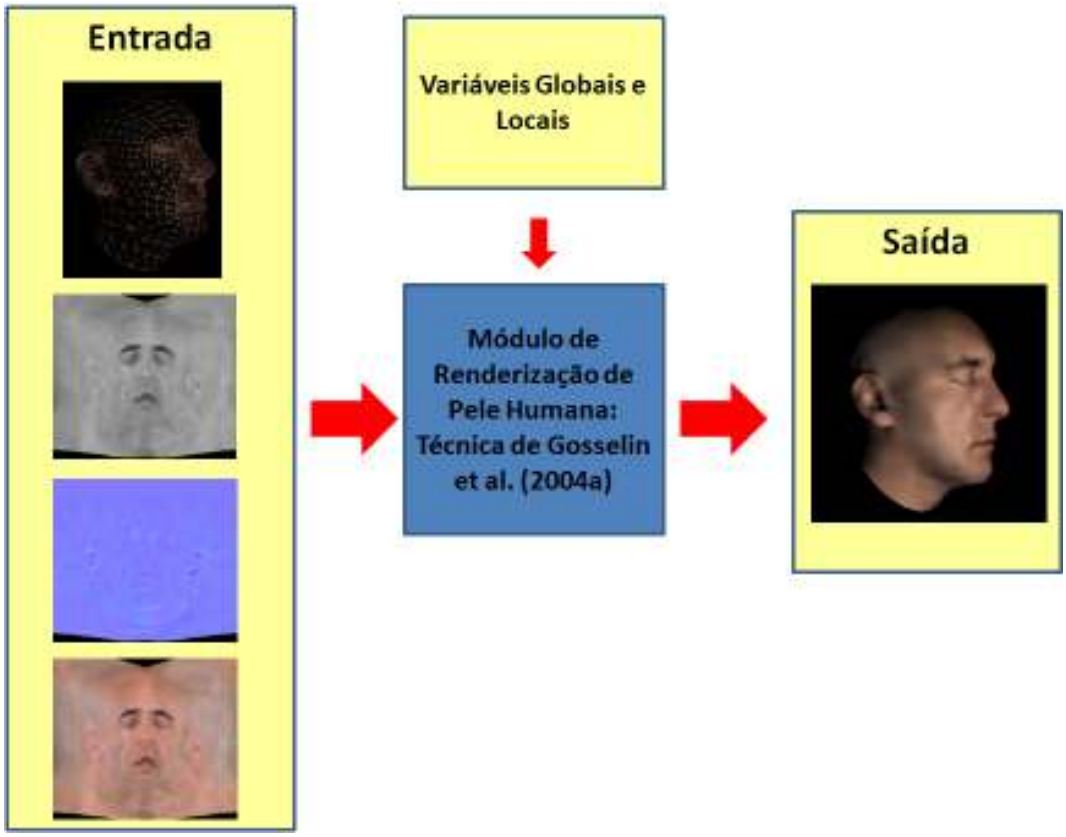


Figura 28 – Entradas e saídas do algoritmo de Gosselin et al. (2004a). A malha tridimensional com suas texturas de alta definição para a cor, normal e *kernel size* (ver seção 5.3) é introduzida no módulo de renderização de pele humana o qual processa os dados de entrada em conjunto com uma série de variáveis para reproduzir a pele humana. As variáveis globais envolvem cor da luz, posição da luz, tolerância de erro de

distância no *Shadow Map*, entre outras. Já as variáveis locais englobam basicamente matrizes de conversão de base.

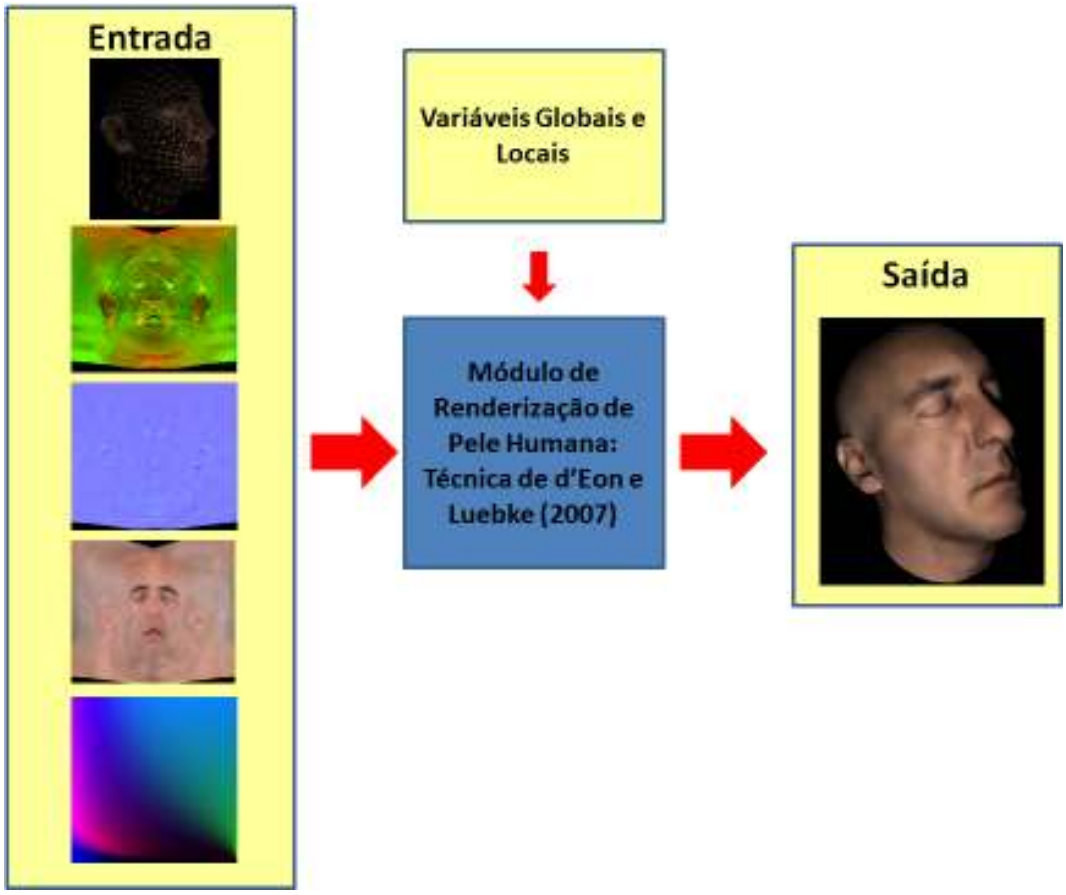


Figura 29 - Entradas e saídas do algoritmo de d'Eon e Luebke (2007). A malha tridimensional com suas texturas de alta definição para a cor e normal é introduzida no módulo de renderização de pele humana o qual processa os dados de entrada em conjunto com várias variáveis para reproduzir a pele humana. Além disso, são passadas as texturas adicionais *Stretch Map* e de atenuação de energia (ver seção 5.3). As variáveis globais envolvem cor da luz, posição da luz, tolerância de erro de distância no *Shadow Map*, pesos das gaussianas, desvio padrão das gaussianas, entre outras. Já as variáveis locais englobam basicamente matrizes de conversão de base.

Tabela 5 – Entradas e saídas de cada passo do algoritmo de Gosselin et al. (2004a).

Passo	Entrada	Saída
1 – *Shadow Map*	Malha tridimensional, posição da luz, parâmetros para montar a pirâmide de visão da luz (como os planos *near* e *far*), matrizes de conversão de base (*world*, *view*, *projection*, etc.).	Mapa de Sombra

2 – *Pre Pass 1*	Malha retangular para se enquadrar na tela e processar os pixels da textura de entrada, mapa de cor.	Nenhuma (o objetivo é apenas informar ao hardware quais pixels não devem ser processados posteriormente).
3 – *Pre Pass 2*	Malha tridimensional, posição do observador, matrizes de conversão de base, parâmetros da câmera (ângulo de abertura, comprimento, largura).	Nenhuma (o objetivo é apenas informar ao hardware quais pixels não devem ser processados posteriormente).
4 - *Irradiance*	Malha tridimensional, posição da luz, parâmetros para montar a pirâmide de visão da luz, matrizes de conversão de base, mapa de normal, mapa de sombra, *Kernel Size Map*, cor da luz, tolerância de erro da distância do *Shadow Map*.	Mapa de Irradiância
5 à 7 - *Blur*	Malha retangular para se enquadrar na tela e processar os pixels da textura de entrada, textura de irradiância ou textura convoluída do passo anterior.	Mapa de Irradiância convoluído pelo filtro de *Poisson*.
8 – *Final Rendering*	Malha tridimensional, posição da luz, posição do observador, matrizes de conversão de base, mapa de normal, mapa de cor, mapa de irradiância convoluído, cor da luz.	Imagem final.

Tabela 6 - Entradas e saídas de cada passo do algoritmo de d'Eon e Luebke (2007).

Passo	Entrada	Saída
1 – *TSM Modified*	Malha tridimensional, posição da luz, parâmetros para montar a pirâmide de visão da luz (como os planos *near* e *far*), matrizes de conversão de base (*world*, *view*, *projection*, etc.).	Mapa com a distância dos pontos iluminados mais as coordenadas de textura *U* e *V* desses pontos.
2 – *Pre Pass 1*	Malha retangular para se enquadrar na tela e processar os pixels da textura de entrada, mapa de cor.	Nenhuma (o objetivo é apenas informar ao hardware quais pixels não devem ser processados posteriormente).
3 – *Pre Pass 2*	Malha tridimensional, posição do observador, matrizes de conversão de base, parâmetros da câmera (ângulo de abertura, comprimento, largura).	Nenhuma (o objetivo é apenas informar ao hardware quais pixels não devem ser processados posteriormente).
4 - *Irradiance*	Malha tridimensional, posição da luz, parâmetros para montar a pirâmide de visão da luz, matrizes de conversão de base, mapa de cor, mapa de normal, textura de atenuação de energia, *TSM* modificado, cor da luz, tolerância de erro da distância do *Shadow Map, mix* de escala de cor para simulação de não uniformidades na pele.	Mapa de Irradiância

5, 7, 9, 11 e 13 - *ConvolutionU*	Malha retangular, *Stretch Map*, textura de irradiância ou textura convoluída do passo anterior, desvio padrão da gaussiana.	Mapa de Irradiância convoluído por uma gaussiana na direção *U*.
6, 8, 10, 12 e 14 – *ConvolutionUV*	Malha retangular, *Stretch Map*, textura de irradiância ou textura convoluída do passo anterior, desvio padrão da gaussiana.	Mapa de Irradiância convoluído por uma gaussiana na direção *V*.
15 – Final Rendering	Malha tridimensional, posição da luz, posição do observador, matrizes de conversão de base, parâmetros para montar a pirâmide de visão da luz (*near* e *far*), texturas de irradiância dos passos 4, 6, 8, 10, 12 e 14, mapa de normal, mapa de cor, textura de atenuação de energia, *TSM* modificado, pesos das gaussianas, cor da luz, tolerância de erro da distância do *Shadow Map, mix* de escala de cor.	Imagem final.

5.3. Detalhes de Implementação

Nessa seção são apresentados vários detalhes utilizados nessa dissertação para o desenvolvimento das duas técnicas estudadas no capítulo 4. Alguns desses detalhes são fornecidos pelos próprios autores das técnicas supracitadas, outros foram obtidos através de uma série de ajustes e testes durante a implementação para obter resultados satisfatórios.

Na equação (7) da seção 4.1.1 é mencionado que um valor de tolerância *tol* deve ser utilizado para amenizar os erros provocados nas sombras devido à falta de precisão e arredondamento no *buffer*. Nessa dissertação o valor utilizado para *tol* em ambas as técnicas é de *0.005*. Para as duas técnicas também é utilizado o mesmo valor de rugosidade $m = 0.3$, constante em toda a face, para resolver a

equação (15) (Donner e Jensen 2005 citado por d'Eon 2007b). A constante *mix = 0.4* é utilizada para resolver as equações (28) e (34), assim como a constante *c=0.1* é usada no cálculo da equação (27).

O filtro de Poisson e um filtro gaussiano (seções 4.1.3 e 4.2.3) são utilizados para criar as convoluções e aparência suave da pele nas técnicas de Gosselin et al. (2004a) e d'Eon e Luebke (2007). Ambos os filtros formam uma circunferência de atuação ao redor do pixel processado. O filtro de Poisson é bidimensional e os pixels vizinhos representando a distribuição são obtidos através de 12 pontos sugeridos por Gosselin et al. (2004a): *(-0.326212, -0.40581), (-0.840144, -0.07358), (-0.695914, 0.457137), (-0.203345, 0.620716), (0.96234, -0.194983), (0.473434, -0.480026), (0.519456, 0.767022), (0.185461, -0.893124), (0.507431, 0.064425), (0.89642, 0.412458), (-0.32194, -0.932615), (-0.791559, -0.59771)*. Para as gaussianas é utilizado o filtro unidimensional sugerido por d'Eon e Luebke (2007) (aplicado para ambas as dimensões *U* e *V*) com valores *(0.006), (0.061), (0.242), (0.383), (0.242), (0.061), (0.006)* o qual é escalado pelo desvio padrão de cada gaussiana que forma o perfil de difusão da luz na pele. Esses dois filtros são utilizados para resolver as equações (13) e (26).

Um problema que pode ocorrer ao usar o espaço da textura para iluminar os pontos e depois mapear a imagem resultante na malha final é a aparição de pequenos defeitos no objeto devido ao *unwrap* da malha. Isso ocorre porque regiões conectadas no espaço tridimensional podem estar separadas no espaço 2D. Assim, as convoluções do mapa de irradiância podem fazer com que espaços do mapa que não pertencem ao objeto (um objeto 3D levado para o espaço 2D não necessariamente ocupa a imagem toda) "invadam" uma região válida, fazendo a cor de *background* se misturar com as cores do objeto, criando artefatos no resultado final não condizentes com a realidade. As Figuras 30 e 31 ilustram esse problema.

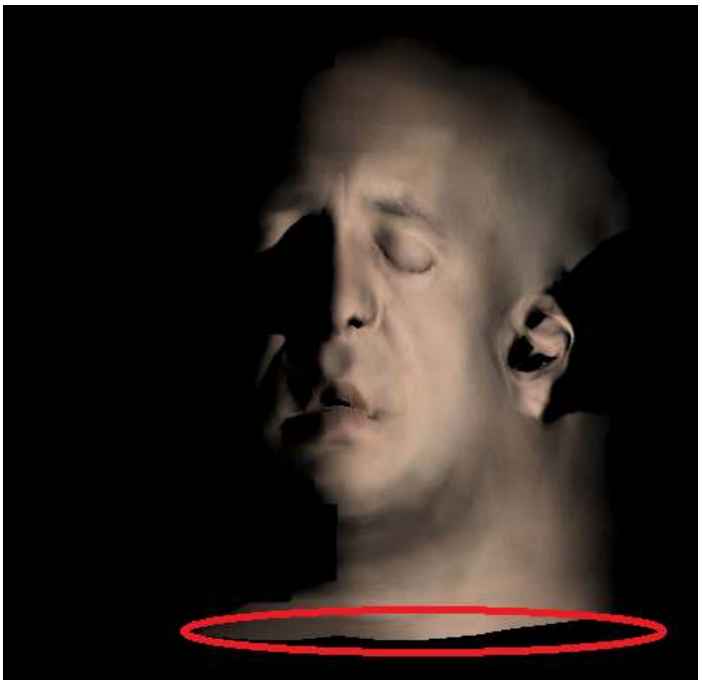


Figura 30 – Convoluções nas regiões de borda, como, por exemplo, a região selecionada na imagem, podem fazer com que as cores de *background* se misturem com as cores do objeto.

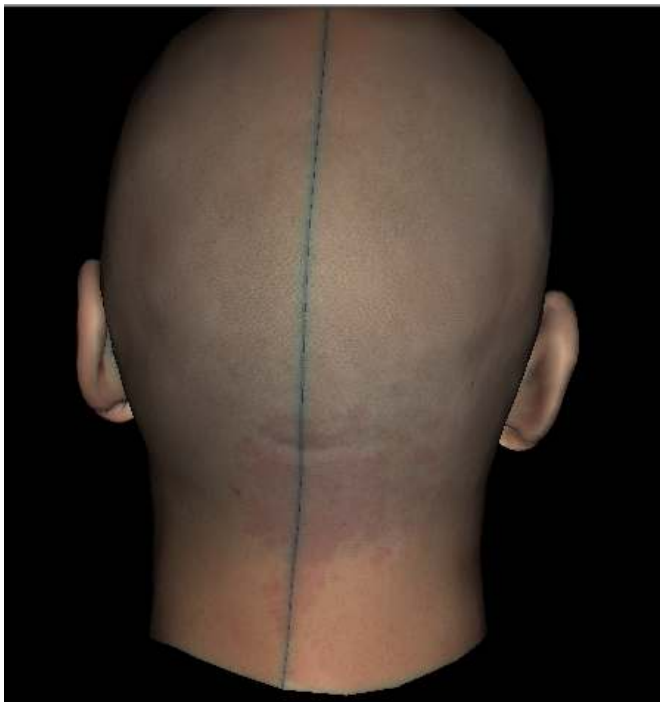


Figura 31 – Aparição de artefatos indesejados devido ao *unwrap* da malha e as convoluções da textura de irradiância.

d'Eon e Luebke (2007) apenas mencionam em seu trabalho que esse problema pode ocorrer e citam possíveis soluções, porém não fornecem muitos

detalhes sobre essas correções. Como ambas as técnicas estudadas apresentam esse defeito, é utilizada a solução de Gosselin et al. (2004a) para remover os artefatos indesejados, com algumas pequenas adaptações. A idéia deles baseia-se basicamente em usar o canal alfa (A) na convolução para verificar se um pixel processado está fora da região do objeto e com isso substituí-lo por pixels vizinhos para preencher os espaços vazios.

Para colocar em prática a idéia de Gosselin et al. (2004a), antes do início do passo 4 das duas técnicas estudadas (Tabelas 3 e 4), o *buffer* deve ser limpo com a cor (0,0,0,1). Ao criar o mapa de irradiância, os pixels que fazem parte do objeto ficam com seu canal alfa diferente de *1.0* (para a técnica de Gosselin et al. (2004a) os valores de A vêm do *Kernel Size Map* (explicado mais adiante nessa seção) e nunca devem ser *1.0* e para a técnica de d'Eon e Luebke (2007) ao escrever a distância corrigida no canal é feita uma validação para que esse valor só tenha valor máximo de *0.99*) enquanto que os pixels inalterados continuam com o canal alfa atribuído na limpeza. Durante o *blur,* se for verificado que o pixel a ser processado tem valor *1.0* no canal alfa, é utilizado um pixel vizinho que tenha A diferente de *1.0* para gerar o resultado da convolução.

Para que essa solução atenda ao problema na técnica de d'Eon e Luebke (2007) a mesma idéia de limpar e escrever valores diferentes do padrão existente no *buffer* é utilizada na geração do *Stretch Map*. Assim, quando o canal A do pixel do *Stretch Map* equivalente ao pixel a ser processado pelo *blur* for *1.0*, o valor de distorção é ignorado no momento de gerar o raio da convolução, aumentando assim a probabilidade de encontrar um pixel válido para preencher o espaço (o termo de distorção diminui o raio de atuação da gaussiana, por esse motivo é ignorado).

Por fim, outra alteração feita para corrigir os artefatos indesejados é a utilização de apenas vizinhos válidos no momento do cálculo das convoluções nas equações (13) e (26). Assim, pixels vizinhos com canal alfa igual a *1.0* são ignorados nos respectivos *blurs*. A Tabela 7 resume as condições para a correção dos artefatos indesejados.

Tabela 7 – Resumo geral das condições para corrigir artefatos gerados nas bordas dos objetos devido ao *unwrap* da malha.

Condição	Técnica de Gosselin et al. (2004a)	Técnica de d'Eon e Luebke (2007)
Pixel processado possui canal alfa igual a *1.0*.	O cálculo de convolução é resolvido utilizando o primeiro pixel vizinho "válido" (*A<1.0*).	O cálculo de convolução é resolvido utilizando o primeiro pixel vizinho "válido".
Pixel processado possui canal alfa igual a *1.0* no pixel correspondente do *Stretch Map*.	Não se aplica.	O valor de correção de distorção na equação (25) é ignorado e o primeiro vizinho "válido" é utilizado.
Pixels vizinhos ao pixel processado não são "válidos".	Cálculo do *blur* é realizado de acordo com a equação (36).	Cálculo do *blur* é realizado de acordo com a equação (37).
Pixel inválido com vizinhos inválidos.	Utiliza-se o cálculo normal de difusão *(N.L)* para preencher o espaço.	Utiliza-se o cálculo normal de difusão *(N.L)* para preencher o espaço.

$$pixel = \frac{pixel\_color\ (coord)}{13} + \frac{\sum_{i=0}^{11} validate\_pixel\ (coord + poisson\ (i) * radius)}{13} \quad (36)$$

onde *validate_pixel (x)* retorna a cor do pixel na coordenada *x* se *A < 1*.0. Caso contrário, retorna a cor do pixel na coordenada *coord* (replica o ponto processado).

$$pixel = \sum_{i=-3}^{3} validate\_pixel\ (coord.x + i * radius) * gaussian\ (i) \quad (37)$$

Na seção 4.2.6 é mencionado que o cálculo de conservação de energia pode ser realizado em coordenadas esféricas. d'Eon e Luebke (2007) utilizam esse

sistema de coordenadas para resolver a equação (31) através de algoritmos de cálculo numérico para integrais e o resultado desse processamento é armazenado em uma imagem pré-computada chamada de textura de atenuação de energia. A coordenada *Y* da textura representa o valor *m* de rugosidade do material e a coordenada *X* o cosseno do ângulo formado pelos vetores *N* e *L*. Assim as equações (32) e (34) podem ser escritas de forma mais simples através das equações (38) e (39) respectivamente. O uso dessa textura pré-computada é importante, pois não é necessária a resolução da integral da equação (31) a cada *frame* computado. A textura de atenuação de energia pode ser vista na Figura 29.

$$final\_difuse = (1 - pixel\_atten((N.L), m)) * fac\_shadow * difuse * C^{mix} \tag{38}$$

onde *pixel_atten((N.L),m)* é o resultado da equação (31) para a rugosidade *m* e cosseno entre a normal *N* e o vetor de luz *L*.

$$final\_difuse\_color = C^{(1-mix)}.(1 - pixel\_atten((N.V), m)).\left(\sum_{i=1}^{6} w_i.final\_difuse\_blurred(i) + \sum_{i=2}^{6} w_i.e^{\frac{-final\_difuse\_blurred(i-1).a^2}{v_i}}.final\_difuse\_blurred(tsm\_coord, i) \right) \tag{39}$$

Gosselin et al. (2004a) mencionam que vários *blurs* devem ser feitos para simular o efeito de suavização na pele causado pelo *subsurface scattering* (seção 4.1.3). Porém, não é explicitada por eles a quantidade de convoluções que devem ser realizadas para reproduzir esse efeito. Para o desenvolvimento realizado nessa dissertação, são considerados três *blurs* para obter o resultado final (passos 5 à 7 nas Tabelas 3 e 5).

Também na seção 4.1.3 é dito que um valor *kernel_size* é usado para determinar o raio de atuação do filtro de Poisson durante o processo de *blur*. Esse valor vem de uma textura (*Kernel Size Map*) na qual para cada pixel é armazenado um valor. Gosselin et al. (2004a) sugerem que essa textura seja feita por um artista, permitindo ao mesmo informar ao sistema o quanto aumentar ou diminuir

o tamanho das gaussianas para cada região da pele. O valor do *kernel size* é armazenado no canal alfa da textura de irradiância (passo 4 da Tabela 3), para que possa ser utilizado posteriormente nos passos onde são aplicados os filtros de Poisson (passos 5 à 7 da Tabela 3). Um exemplo de *Kernel Size Map* é demonstrado na Figura 28.

De forma a permitir que o artista consiga inibir as convoluções em certas regiões como as sobrancelhas através do *Kernel Size Map*, a equação (12) é alterada para a equação (40). Assim se um pixel do *kernel size* contiver o valor *0.0*, o raio de convolução é zerado e o filtro não é aplicado na região desejada.

$$radius = size * \text{ker}nel\_size * (scalex + scaley) \tag{40}$$

5.3.1. Otimizações

Um ponto da implementação que merece mais destaque que os demais é a velocidade na qual as duas técnicas são executadas. Devido a necessidade do processamento ser realizado em tempo real (de forma a ser incorporado à um motor de jogos), nessa seção são apresentadas algumas formas de otimizar os algoritmos estudados no capítulo 4.

Assim como existe uma textura pré-computada com os resultados da equação (31) para determinar o termo de conservação de energia, d'Eon e Luebke (2007) propõem que o *Stretch Map* também seja pré-computado. Essa pequena otimização evita o processamento a cada *frame* dessa textura, vide que a mesma é sempre igual para o mesmo objeto, pois não depende de dados variantes como o ângulo de visão ou do ponto de luz para ser criada. As Tabelas 4 e 6 já consideram que o *Stretch Map* é pré-computado para realização da técnica de renderização da pele humana.

No algoritmo desenvolvido, é considerada apenas uma fonte de luz (fonte de luz principal) para a renderização de iluminação da pele. O mapa de sombra translúcido modificado possui um canal que armazena a distância dos pontos iluminados à luz. Essa informação é suficiente para criar as sombras de uma cena (conforme visto na seção 4.1.1), considerando um único foco de luz. Dessa forma, para a técnica de d'Eon e Luebke (2007) é feita uma pequena mudança no

algoritmo de forma que as sombras sejam criadas com o *TSM* modificado ao invés do mapa de sombra, evitando um passo a mais de processamento de dados. Assim como a otimização do *Stretch Map* pré-computado, as Tabelas 4 e 6 já ignoram o *Shadow Map*, e utilizam o *TSM* modificado para o cálculo das sombras.

As otimizações apresentadas até o momento apenas visam à melhoria de desempenho da técnica de d'Eon e Luebke (2007). Uma adaptação feita no código que melhora significativamente a *performance* de ambas as técnicas é a utilização do *frustum culling* para os vértices que sofrem *unwrap* para o espaço da textura. O *frustum culling* desabilita o processamento no *fragment shader* de pixels provenientes de vértices que estão fora da pirâmide de visão da câmera. Pelo *pipeline* gráfico padrão, essa otimização é obtida de "graça" pelo hardware da placa gráfica. Porém, ao fazer o *unwrap* da malha de forma que ela ocupe toda a região de visão, o hardware não faz o corte dos vértices, pois os mesmos ficam sempre dentro do *frustum* de visão. Dessa forma, pontos que na renderização final da malha não são vistos têm seus dados processados desnecessariamente quando levados para o espaço da textura.

De forma a evitar esse processamento desnecessário e obter o corte por hardware dos pixels das texturas que não são usados na imagem final, o seguinte procedimento deve ser realizado:

1. Cria-se um passo a mais para cada uma das técnicas (passo 3 das Tabelas 3 a 6). Esse passo deve estar no início com o *z-buffer* limpo com valor *0.0* e serve para definir se um pixel deve ser processado em um passo seguinte ou não.
2. No *vertex shader* desse passo, faz-se o *unwrap* da malha de forma a levar o objeto para o espaço da textura.
3. Ainda no *vertex shader*, calcula-se se o vértice está dentro do *frustum* de visão de acordo com as equações (41) e (42).
4. Caso o vértice não esteja no *frustum* de visão, escreve-se *1.0* no *z-buffer*.
5. Para todos os passos seguintes em que se deve trabalhar no espaço da textura (passos 4 à 7 das Tabelas 3 e 5 e passos 4 à 14 das Tabelas 4 e 6) a comparação em hardware do *z-buffer* deve estar habilitada para permitir a escrita nos pixels onde o *z* do processamento for igual ao valor armazenado no z-*buffer*.

Com base no procedimento acima, o hardware corta os pixels do espaço da textura que não estejam no *frustum* de visão (considerando que os passos que fazem a transformação da malha do espaço 3D para o espaço 2D sempre escrevem *0.0* na coordenada *z* do ponto). Assim, se no passo 3 das Tabelas 3 a 6 um ponto estiver dentro do *frustum* de visão, o *z-buffer* não é alterado e quando o mapa de irradiância for gerado (passo 4 das Tabelas 3 a 6), *0.0* é comparado com o valor *0.0* anterior, permitindo o processamento no *pixel shader*. Caso o ponto esteja fora do *frustum*, *1.0* é escrito no *z-buffer*, o qual é comparado com *0.0* na criação do mapa de irradiância e, como ambos são diferentes, o hardware desabilita o processamento no *fragment shader*. As Figuras 32 e 33 mostram um exemplo de região cortada do processamento por não ser necessária na imagem final.

(a) (b)

Figura 32 – Todo objeto se encontra dentro do *frustum* de visão da câmera.

(a) (b)

Figura 33 – O objeto não se encontra totalmente na região de visão da câmera, logo não deve ser realizado todo o processamento no espaço da textura. As partes em azul à direita (b) são cortadas pelo hardware, pois seu processamento é desnecessário. Com isso o algoritmo fica mais rápido.

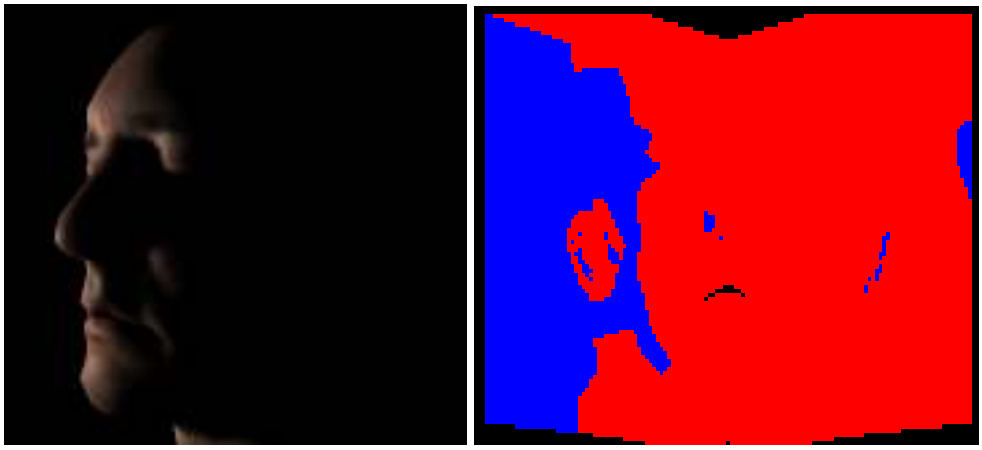
$$\begin{cases} se\ tg\left(\frac{fov}{2}\right) < \frac{p.x}{p.z}, ponto\ \ fora\ do\ frustum \\ \quad caso\ contrário, está\ dentro \end{cases} \tag{41}$$

$$\begin{cases} se\ \frac{height^{*}\, tg\left(\frac{fov}{2}\right)}{width} < \frac{p.y}{p.z}, ponto\ \ fora\ do\ frustum \\ \quad caso\ contrário, está\ dentro \end{cases} \tag{42}$$

Gosselin et al. (2004a) e Jimenez e Gutierrez (2008) apresentam técnicas de *frustum culling* para acelerar seus algoritmos de renderização da pele humana, porém baseiam-se em conceitos um pouco diferentes para delimitar quando desconsiderar o processamento de um pixel.

Gosselin et al. (2004a) e Hable et al. (2009) apresentam outra técnica eficiente para determinar se um pixel deve ser processado ou não. A idéia baseia-se no fato de que pontos não observados não precisam ser processados (*backface culling*). De forma matemática, isso se resume a observar o ângulo entre a normal N do ponto e a direção V do observador ao ponto. Caso o cosseno desse ângulo seja negativo, não é necessário o processamento do vértice no *pixel shader*.

Essa idéia pode ser facilmente incorporada ao passo 3 (Tabelas 3 a 6) de ambas as técnicas junto com o algoritmo de *frustum culling*. Caso *(N.V)<0, 1.0* deve ser escrito no *z-buffer*, seguindo assim a mesma idéia de corte pelo hardware apresentada anteriormente. A Figura 34 mostra a quantidade de processamento desnecessário evitado.



(a) (b)

Figura 34 – Em azul na figura à direita (b) está o corte realizado pelo hardware dos pixels desnecessários. Nota-se que a parte ignorada é grande, o que aumenta o desempenho do algoritmo.

Por fim, um último ponto de otimização utilizado nessa dissertação é o corte de pixels desnecessários na imagem original. O mapa de cor pode ter regiões que não contém informação alguma sobre a pele do objeto (parte escura na Figura 35 (a)). Processar esses pixels apenas onera o algoritmo. Sendo assim, o algoritmo utilizado nessa dissertação apresenta um passo a mais (passo 2 das Tabelas 3 a 6) que tem como objetivo "cortar" o processamento desses pixels desnecessários nos passos envolvendo o espaço da textura (passos 4 a 7 das Tabelas 3 e 5 e passos 4 a 14 das Tabelas 4 e 6). A idéia é análoga a apresentada anteriormente para representar o passo 3 (Tabelas 3 a 6), tendo como principal diferença o fato do processamento ser feito no *fragment shader* e não no *vertex shader* (nessa etapa é feito um processamento de imagem e não uma análise geométrica). Abaixo segue o procedimento para realizar essa otimização:

1. No *vertex shader* desse passo, apenas projeta-se para o *pixel shader* a malha retangular na qual a textura é enquadrada para processamento.

2. No *fragment shader*, verifica-se se o pixel do mapa de cor é preto (0,0,0). Caso seja, escreve-se no *z-buffer* um valor qualquer diferente de *0.0*.
3. Para todos os passos seguintes em que se deve trabalhar no espaço da textura (passos 4 a 7 das Tabelas 3 e 5 e passos 4 a 14 das Tabelas 4 e 6) a comparação em hardware do *z-buffer* deve estar habilitada para permitir a escrita nos pixels onde o *z* do processamento for igual ao valor armazenado no z-*buffer*.

A Figura 35 ilustra o quanto de processamento desnecessário é evitado ao utilizar esse passo inicial.

(a) (b)

Figura 35 – Na imagem à esquerda (a) está o mapa de cor original. Pode-se perceber que as regiões em preto não fazem parte do objeto e podem ser ignoradas no processamento. Na imagem à direita aparece em azul toda a região na qual o processamento dos pixels foi desconsiderado.