3.
Otimização em Nível de Execução

A primeira etapa do processo de construção de um sistema de renderização eficiente é a arquitetura do renderizador, pois é nele que o trabalho é propriamente executado. Dessa forma, pode-se entender o renderizador com o operário de uma fábrica.

Os renderizadores focam apenas em cumprir a tarefa de renderizar um frame, sem se preocupar em descrever como o trabalho foi feito e as dificuldades encontradas. Nesse sentido, o trabalho do gerenciador fica muito limitado pela falta de informações para atuar.

3.1.1. O Processo de Renderização

A renderização de um frame consiste em diversas etapas para atingir o objetivo. Na figura 14 pode-se observar um fluxograma básico para a renderização.

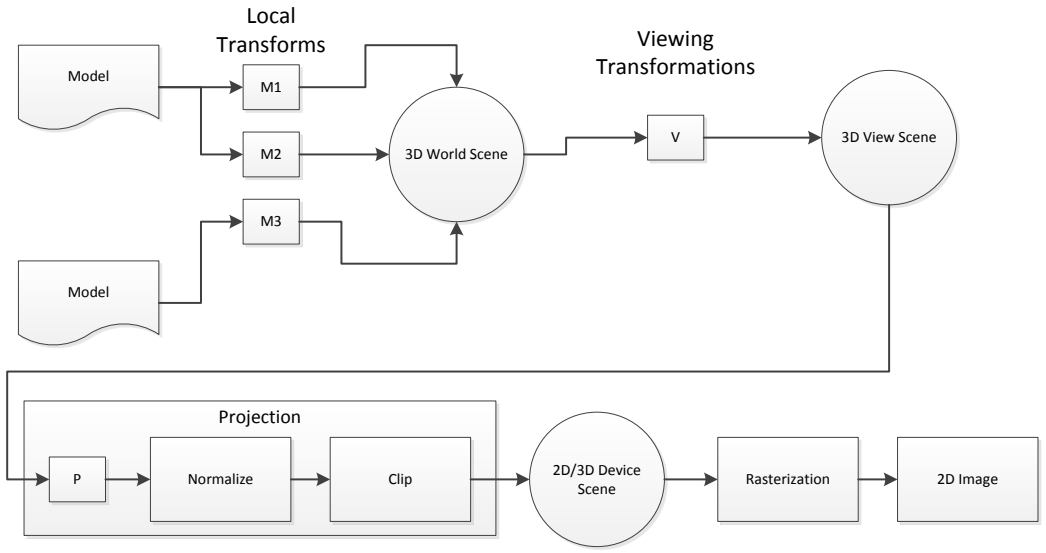


Figura 14: Workflow de Renderização (Rasterização) (imagem produzida)

Observando o fluxo de renderização é sabido que o mesmo é altamente paralelizável, sendo tal fato explorado pelas placas de vídeo e renderizadores atuais. Contudo, do ponto de vista da imagem renderizada, existem áreas que consomem muito mais recurso computacional do que outras. Na maioria dos renderizadores, o processo de subdivisão de áreas para renderização não leva em consideração nenhum métrica ou estimativa, ou seja, normalmente emprega-se a subdivisão por blocos (ou *buckets*) de tamanho fixo.

Quando se trata de métodos de iluminação global e é feito o uso de processadores de múltiplos núcleos, a má divisão limita a utilização dos núcleos, sobrecarregando um núcleo desnecessariamente. Na figura 15 o tempo necessário para renderizar a imagem foi de 50 minutos em uma máquina de 24 núcleos, sendo que 32 minutos foram gastos em apenas 1 *bucket*, ou seja, 64% do tempo foi gasto em 1 *bucket* que utilizou apenas cerca de 4% (1 núcleo) da capacidade da máquina.

Figura 15: Problemas na subdivisão da renderização. Cena do Maracanã, gerado pelo autor para a novela Avenida Brasil (TV Globo, 2012). Imagem reproduzida sob a política de “fair use”.

Esse exemplo mostra que apenas uma parte do desperdício que pode ocorrer, pois, se avaliarmos que temos uma sequência de imagens para processar, esse desperdício pode ser ainda maior.

Conforme exposto, Abraham et al (2004) percebeu esse problema na implementação de um renderizador OpenGL distribuído. Apesar de se

tratar de um rasterizador em tempo real, foi desenvolvido uma forma de se compensar os servidores que estavam com as maiores cargas de processamento. Esse processo utilizou as métricas de consumo de processamento de um frame anterior, assim os novos frames puderam se melhor divididos.

No caso de um renderizador fotoreal, existe o problema do mesmo não conhecer as métricas do frame anterior, assim, o renderizador desenvolvido foi concebido para gerar uma métrica estimada do custo de cada pixel que o mesmo irá renderizar (módulo de predição). A equação 1 mostra a fórmula utilizada para calcular o custo inicial de renderização de um pixel, nela é levado em consideração não só a complexidade geométrica da cena, mas também, a complexidade dos *shaders* e luzes presentes no frame.

$$c_0(p) = k_{geometria} \cdot complexidade(geometrica) + k_{material} \cdot complexidade(material, luzes)$$

Equação 1: Fórmula de custo inicial de um *pixel*

As variáveis iniciadas por k representam os pesos de cada elemento na renderização, esses fatores podem ser modificados dinamicamente. O cálculo desses fatores é realizado durante a carga de geometria (complexidade geométrica) e durante a geração de estrutura de aceleração (complexidade geométrica, material e luz). No fim do processo é criado uma matriz de custos, onde cada célula (i,j) corresponde ao custo do pixel (j,i).

As constantes $k_{geometria}$ e $k_{material}$ são computadas como:

$$k_{geometria} = \frac{\left(c(p) - k_{material_{old}} \cdot complexidade(material, luzes)\right)}{complexidade(geometrica)}$$

Equação 2: Fórmula de atualização de $k_{geometria}$

e

$$k_{material} = \frac{\left(c(p) - k_{geometria_{old}} \cdot complexidade(geometrica)\right)}{complexidade(material, luzes)}$$

Equação 3: Fórmula de atualização de $k_{material}$

Depois ambas são reescaladas pelo fator $1/\left(k_{geometria} + k_{material}\right)$. Nesse processo elas podem ser enviadas a um gerenciador, o qual modificá-las e atualizar o renderizador.

3.1.2. Complexidade Geométrica

Para calcular a complexidade geométrica de um pixel, optou-se por considerar três métricas. A primeira utiliza o processo de rasterização em hardware para obter o número de triângulos que projetados em um pixel (*nt*), ou seja, utilizando o *Query Object* do OpenGL (com teste de profundidade).

A segunda métrica é obtida durante a construção da estrutura de aceleração; nela projeta-se a estrutura (folha da *BVH*) na imagem e computa-se duas informações para o pixel, o número de *BVHs* projetadas (*na*) e o número de triângulos contidos pela mesma (*nat*), o qual é computado dividindo-se o número de triângulos pela área projetada (em pixels) na imagem (figura 16). Dessa maneira, cada folha da *BVH* soma ao resultado anterior de *nat*, assim, *nat* é maior que *nt*, na maioria dos cálculos, pois a folha da *BVH* é relativamente pequena quando projetada.

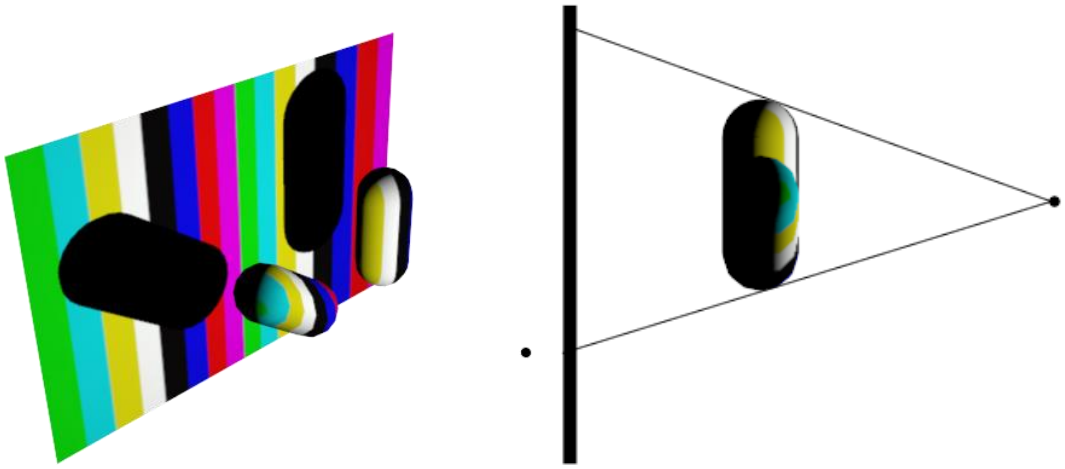


Figura 16: Projeção de nós da BVH (imagem produzida)

Dessa forma, cada pixel contém a complexidade da seguinte forma:

$$complexidade(geometrica) = t_t * (nt + nat) + t_a * na$$

Equação 4: Fórmula de custo geométrico de um *pixel*[2]

Na expressão, t_t é o tempo médio de processamento de um triangulo e t_a é o tempo médio de processamento de um nó folha da *BVH*, ambos possuem um valor inicial obtido por testes (1us e 20us, respectivamente)[3] e a cada frame renderizado são atualizados. No entanto, a atualização durante a execução é feita medindo cerca de 10% das operações envolvidas para não prejudicar o desempenho, ao contrário dos testes feitos para o valor inicial, que se calculou a média de todo o processamento envolvido em várias cenas.

3.1.3. Complexidade de Material e Luzes

Durante a geração da *BVH*, o sistema inicia o cálculo da complexidade da iluminação da cena. Essa complexidade representa o maior custo em cenas fotorealísticas, isso se dá pelo uso de estimadores (e.g. Monte Carlo), os quais aumentam muito o número de raios ou partículas necessárias para computar o pixel.

De forma muito similar ao que é realizado na computação da complexidade geométrica, as folhas da *BVH* são projetadas na imagem que será renderizada. No entanto, a computação da contribuição é dada pelo custo do material ($c_{material}$), o qual depende das opções das luzes da cena. Além disso, considera-se que o material é composto pelos componentes (reflexão, refração e difusão). Cada material (*IMaterial*) no

[2] na = 0 e nat = 0 quando o cálculo é feito para a computação em GPU.

[3] O cálculo faz a divisão por core e por GHz do processador no caso de Processadores convencionais, no caso de *GPU* o valor é tabelado por modelo.

renderizador deve retornar os parâmetros, mesmo que não os utilize. Ademais, considera-se que o material respeita a equação de *fresnel* 1D (aproximação de Schlick) (Schlick, 1994). Assim, a equação do custo do material pode ser descrita como:

$$c_{material}(material) = \sum_{l=0}^{n_{luzes}} \left(k_{fl}(l) \sum_{i=0}^{depth_{refl}} t_{refl} \cdot n_{refl}(i) + k_{fr}(l) \sum_{i=0}^{depth_{refr}} t_{refr} \cdot n_{refr}(i) + t_{dif} \right)$$

Equação 5: Fórmula de custo de material de um *pixel*

sendo:

$$k_{fl}(l) = IOR \cdot \frac{distance(l.position, center)}{scene_{diameter}}$$

$$k_{fr}(l) = (1 - IOR) \cdot \frac{distance(l.position, center)}{scene_{diameter}} \cdot SSS_{depth}$$

Equação 6: Fatores de Refração e Reflexão

Na expressão, t_{reft} é o tempo médio de processamento de um raio de reflexo, $n_{refl}(i)$ é o número médio de subraios disparados na reflexão no nível de recursão i, e t_{refr} e n_{refr} seguem o mesmo princípio. O fator t_{dif} é o tempo de processamento do componente difuso. Para a computação dos fatores k_{fl} e k_{fr}, é utilizado o índice de refração do material (IOR) e a distância da luz em processamento em relação ao centro da esfera que circunscreve a cena, em proporção ao diâmetro da daquela, assim, esse parâmetro serve como uma heurística (na qual fontes luminosas distantes do centro de ação tem menor influência nas cenas); isso é observado em muitas cenas, mas pode ser errado para outras. No caso da refração, multiplica-se o custo se o material utilizar *subsurface scattering*. Observa-se que, simplificadamente, o fator IOR atua como uma probabilidade. Além disso, o cálculo não considera a conservação de energia.

Todos os tempos médios seguem o mesmo princípio utilizado no custo geométrico, ou seja, eles possuem um valor inicial, mas são ajustados a cada frame renderizado. Esse processo é executado após a renderização, durante o salvamento dos arquivos.

Por fim, para computar o custo em um *pixel* específico, projetam-se as folhas da BVH na imagem, os pixels acertados somam o custo da *BVH* (que consiste na soma dos custos do $c_{material}$ de todos os materiais presentes na folha, sem repetir a contagem, divididos pela área ocupada pela projeção da folha. Dessa forma, o tempo de processamento estimado do frame é a soma dos custos de todos os *pixels*.

3.1.4. Subdivisão baseada em custo

Tendo o processo de estimativa da complexidade de cada *pixel* terminado, o sistema fica apito realizar a subdivisão de forma a concentrar o processamento nas áreas de maior necessidade de computação. Abraham et al (2004) utilizou a metodologia de subdivisão em blocos (BSP), conforme ilustra a imagem da figura 17:

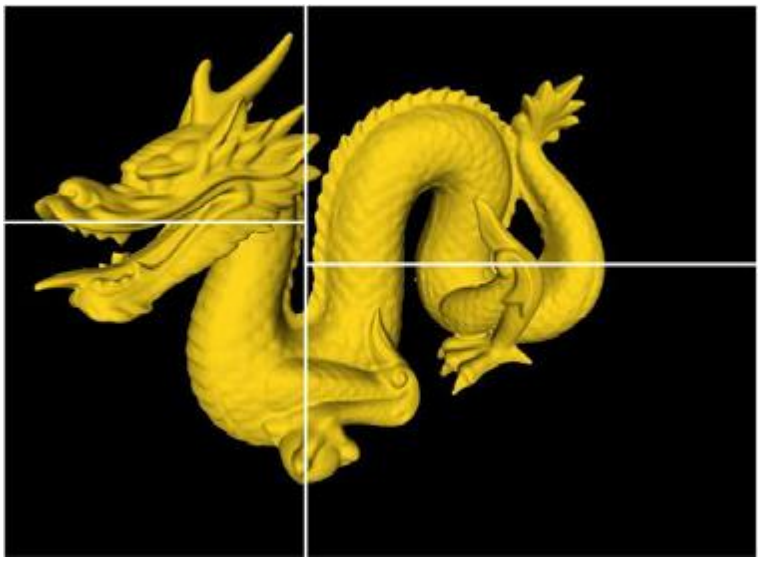


Figura 17: Subdivisão utilizada por Abraham et al (2004)

No entanto, a metodologia de Abraham et al. (op. cit.) não é adequada para um renderizador fotorealista. Desta maneira, foram analisadas as seguintes três estratégias:

- **Ordenamento de pixel**:
  Nessa estratégia, cada núcleo processa o *pixel* de maior custo disponível numa fila de *pixel* a serem renderizados. A estratégia é adequada para otimizar o uso dos núcleos, porém, ela não aproveita a coerência espacial, o que pode piorar o desempenho. Por esse motivo, não foi implementada.
  Kd-trees:
  As kd-trees são estruturas muito eficientes para realizar subdivisões, assim, o uso da mesma favorece à subdivisão eficiente e aproveita a coerência espacial. No entanto, o custo de computação é alto e as áreas de subdivisão são mais complexas.
- **Quad-trees:**
  As *Quad-trees* (Samet, 1984) são naturalmente adaptadas para imagens, sendo que sua estrutura é regular e de fácil implementação e rápida construção.

No desenvolvimento do renderizador, foram realizados os testes com *o kd-tree* e a *Quad-tree* e optou-se pelo uso da *Quad-tree* por não ter afetado significativamente o tempo de renderização.

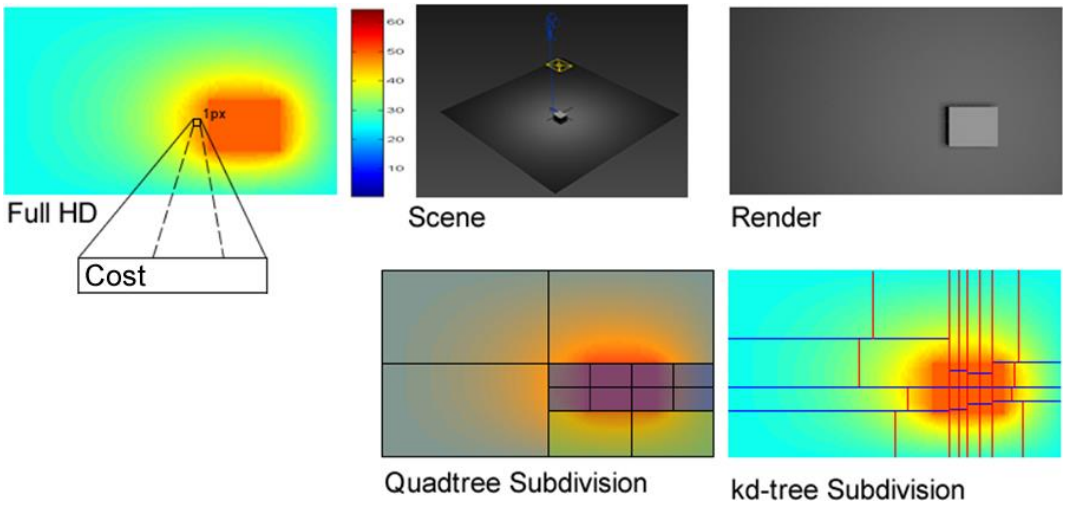


Figura 18: Subdivisão utilizada pelo sistema (imagem produzida (algoritmo))

A figura 18 ilustra o processo de subdivisão da imagem proposto no presente trabalho, levando como parâmetro o custo do *pixel*. Observa-se que a *quad-tree* simplifica visualmente como o processo ocorrerá. Outra vantagem da *quad-tree* é a possibilidade de ser usada no processo de *encoding*, pois a mesma é utilizada no algoritmo h.265 (Sullivan et al., 2012).

3.1.5. Aproveitamento de Custos

Após a computação dos custos estimados e início da renderização, a estimativa torna-se estatística, visto que é possível computar o tempo gasto em cada pixel e associá-lo. Nesse processo cria-se uma matriz de estruturas que armazenam as informações de performance do *pixel*, as quais representam o custo efetivo de cada um. Essa matriz representa a Imagem de Performance (PI, *Performance Image*).

Na PI procura-se armazenar as informações de performance e de consumo de recursos que cada pixel utilizou. Assim, além do tempo de computação, armazena-se o número de raios, memória utilizada, pilha de raios, número de acessos às texturas, números de computações de Monte Carlo e o número de raios que não colidiram. Essas informações serão úteis para os processos apresentados nos próximos capítulos. Na imagem da figura 19 é apresentado o uso da imagem de performance,

onde se observa que ela é muito similar à matriz de custos. Apesar da PI possuir diversas informações, apenas o tempo de computação é relevante para a subdivisão.

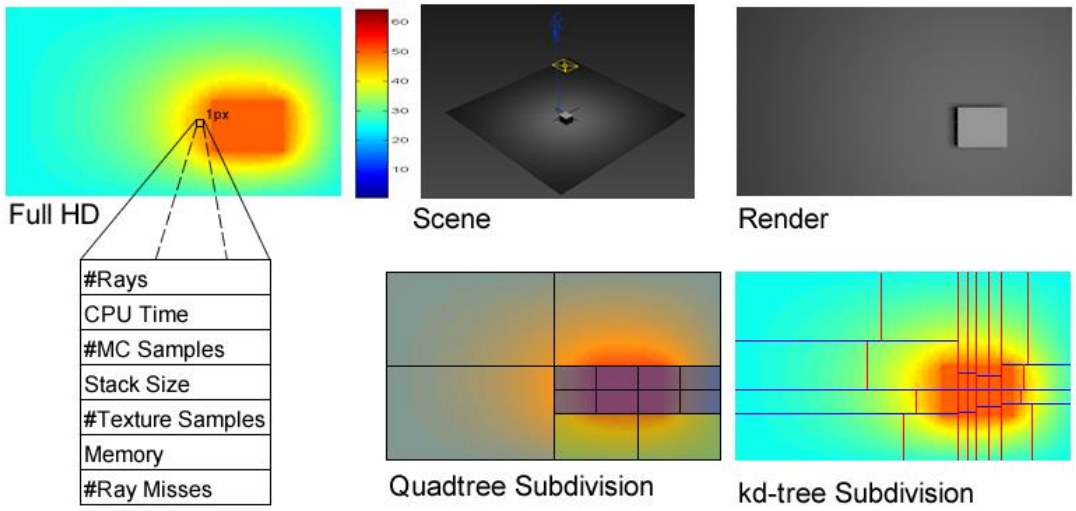


Figura 19: Imagem de Performance: cada pixel de um frame tem as informações de desempenho armazenadas (#Rays, CPU Time, #MC Samples, Stack Size, #Texture Samples, Memory, #Ray misses). Estas informações permitem a subdivisão otimizada do frame (exemplificada com as opções Quadtree e kd-tree). (imagem produzida (algoritmo))

As informações contidas na PI são úteis para o frame que já foi renderizado, ou seja, os custos computados permitem a subdivisão otimizada para o frame correspondente. Nesse contexto, a *PI* só seria útil numa nova renderização do mesmo frame. Porém, em muitas cenas virtuais, pode-se considerar que existe coerência temporal entre os *frames*, assim, o *frame* t é muito similar ao *frame* $t-1$ e $t+1$; portanto, parte das informações contidas na *PI* de $t-1$ ou $t+1$ são similares às de t.

Essa similaridade pode ser melhor ajustada utilizando algoritmos de fluxo ótico (Lucas & Kanade, 1981) (Horn et al., 1980), que tem como

base a coerência temporal e espacial, e passes de velocidade[4]. Assim, tendo dois *frames* é possível fazer a correspondência entre os *pixels* desses.

No renderizador é possível passar como parâmetro imagens de performance, juntamente com a matriz de fluxo ótico do algoritmo de Lucas & Kanade (1981), (Kondermann et al, 2011), implementado pela biblioteca OpenCV (Bradski, 2000), ou os passes de velocidade entre os mesmos (figura 20).

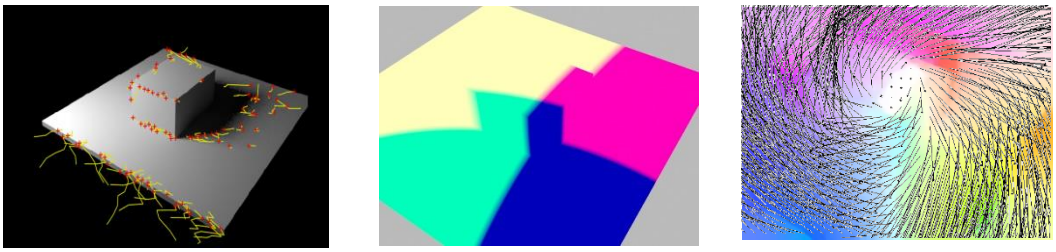


Figura 20: Compensação da Imagem de Performance (imagem produzida (algoritmo))

O renderizador admite que o intervalo entre o *frame* atual e o *frame* que possui a imagem de performance seja arbitrário, nesse caso a matriz do fluxo ótico é calculada entre os mesmos, para gerar o módulo dos vetores de forma correta. Caso seja usado passes de velocidade, o sistema faz soma vetorial do deslocamento dos passes e aplica sobre a imagem de performance final.

No caso do uso de algoritmo de fluxo ótico, utilizam-se os dados produzidos pelo OpenCV (cvGoodFeaturesToTrack e cvCalcOpticalFlowPyrLK). O algoritmo de Lucas & Kanade (1981) produz uma série de vetores de forma esparsa. Assim, para se gerar a

[4] O passe de velocidade consite em uma imagem na qual se armazena o vetor de deslocamento de pixel nos canais RGB. Isso é possível pois a cena sendo animada, pode-se calcular o próximo frame da animação.

compensação da imagem de performance aplica-se o deslocamento calculado sobre a imagem original.

Outra opção é utilizar as imagens do passe de velocidade para produzir a imagem de performance compensada. Supondo que se deseja computar o *frame t* e se possua a PI dos *frames t – 4* e *t + 2*, deverá ser disponibilizado os passes de velocidade de *t-4, t-3*, *t-2*, *t-1*,*t+1* e *t+2*, sendo que *frames* com índice maior do que *t* calculam reversamente. Assim, um *pixel* $\overrightarrow{p_t(x,y)}$ da imagem de performance em *t*, com compensação da imagem de performance *p*, terá seu valor computado através da equação 7.

$$\overrightarrow{p_t(x,y)} = \overrightarrow{p_p}\left(\overrightarrow{\left(\sum_{\iota=p}^{t^*} \frac{(t-p)}{|t-p|} \cdot \overrightarrow{v_\iota(x,y)} \cdot \Delta t\right)} + \overrightarrow{(x,y)}\right)$$

Equação 7: Fórmula da computação da imagem de performance resultante

Na equação 7, *t** é o *frame* imediatamente anterior ao *frame t*, ou seja, *t + 1* ou *t-1*. Um fator necessário para a computação do deslocamento com passes de velocidade é o intervalo de tempo Δt, o qual é computado como $\frac{1}{fps}$, ou seja, é dependente da taxa de quadros. Tal fato é natural, visto que o processo de geração do passe usa essa constante para projetar essa informação. Nesse ponto, poder-se-ia considerar a hipótese de utilizar um passe de deslocamento, no entanto, adotou-se o passe de velocidade por ser comum nos renderizadores. Observa-se ainda que um passe de velocidade apresenta o deslocamento em 3 dimensões, por isso o vetor é projetado em duas dimensões para os cálculos, visto que a imagem do passe se dá no espaço da câmera.

As imagens de performance resultantes desse processo são então utilizadas no cálculo do custo do *pixel*, alterando a equação 1 para a equação 8.

$$c(p) = \left(1 - k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} \left(e^{-k_{img}|\Delta f|}\right)}{n_{imagens}}\right) c_0(p) + k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} \cdot complexidade\left(perf_{imagem}(p)\right) \cdot \left(e^{-k_{img}|\Delta f|}\right)}{n_{imagens}}$$

Equação 8: Fórmula de custo de um pixel considerando a imagem de performance

Nessa equação, utiliza-se o custo estimado ponderado pela contribuição de cada imagem de performance computada. Essa ponderação é dada pela expressão $e^{-k_{img}|\Delta f|}$, que utiliza uma constante k_{img} multiplicada pelo módulo da diferença entre a imagem de performance atual e a utilizada como base *(*Δf*)*; assim, quanto maior for a distância entre elas, menor será a contribuição. A função exponencial foi utilizada por se avaliar que o erro associado à transformação dos *pixels* é complexo de calcular, devido a transformação bidimensional ser aplicada em um espaço tridimensional. Assim, a função exponencial tende a reduzir a influência de frames mais distantes do item em análise.

Além disso, as constantes k_{img} e $k_{perf_{imagem}}$ são recalculadas a cada renderização através da diferença entre o obtido e o estimado. Assim, temos a seguinte expressão:

$$k_{perf_{imagem_{new}}} = \frac{c(p)}{k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} \cdot complexidade\left(perf_{imagem}(p)\right) \cdot \left(e^{-k_{img_{old}}|\Delta f|}\right)}{n_{imagens}}}$$

Equação 9: Fórmula da atualização de $k_{perf_{imagem}}$

Onde, para garantir a aplicação da constante limita-se $k_{perf_{imagem}}$ da seguinte forma:

$$k_{perf_{imagem}} < \frac{1}{\frac{\sum_{p=0}^{n_{imagens}} \left(e^{-k_{img}|\Delta f|}\right)}{n_{imagens}}}$$

Equação 10: Limite da atualização de $k_{perf_{imagem}}$

A derivação de k_{img} segue:

$$c(p) = k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} \cdot complexidade\left(perf_{imagem}(p)\right) \cdot \left(e^{-k_{img}|\Delta f|}\right)}{n_{imagens}}$$

$$c(p)$$

$$= k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} \left(e^{-k_{img}|\Delta f|}\right)$$

$$= k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} \left(e^{-k_{img}}\right) \sum_{p=0}^{n_{imagens}} \left(e^{|\Delta f|}\right)$$

$$\sum_{p=0}^{n_{imagens}} \left(e^{-k_{img}}\right)$$

$$= {c(p)} \Big/ {k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} \left(e^{|\Delta f|}\right)}$$

$$e^{-k_{img} \cdot n_{imagens}}$$

$$= {c(p)} \Big/ {k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} \left(e^{|\Delta f|}\right)}$$

$$-k_{img} \cdot n_{imagens}$$

$$= \ln\left({c(p)} \Big/ {k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} \left(e^{|\Delta f|}\right)} \right)$$

$$k_{img} = -\ln \frac{\left(c(p) \Big/ k_{perf_{imagem}} \frac{\sum_{p=0}^{n_{imagens}} complexidade\left(perf_{imagem}(p)\right)}{n_{imagens}} \sum_{p=0}^{n_{imagens}} (e^{|\Delta f|}) \right)}{n_{imagens}}$$

Equação 11: Fórmula da atualização de k_{img}

Após a computação do valor, k_{img} pode ser utilizado em uma nova renderização. Esse valor pode ser enviado para um gerenciador, o qual pode modificá-lo ou sugerir outro valor para o renderizador.

Dessa forma, o cálculo do custo e tempo de renderização fica cada vez mais preciso, além de melhorar o processo de subdivisão da renderização. Essa técnica é inovadora e não é encontrada em nenhum outro sistema de renderização na literatura.

3.1.6. Otimização na Renderização

Tendo um estimava adequada para a subdivisão da renderização, inicia-se o processo de renderização em si. Para isso, a maioria dos algoritmos de renderização com iluminação global só atinge uma performance adequada quando utilizando estruturas de aceleração.

Nesse ponto, a estrutura adotada foi a *BVH* com aproveitamento temporal, mais precisamente, baseada na *SBVH* (Stich et al, 2009), *LBVH* (Lauterbach et al, 2009) e *HLBVH* (Pantaleoni et al, 2010), utilizando técnicas de rotação de nós (Kensler, 2008) e inserções (Bittner et al, 2013), tal estrutura é implementada pela Biblioteca NVidia Optix para *GPU* e Intel Embree para *CPU*. Dessa maneira, a técnica permite uma acomodação da *BVH* sem a necessidade de sua completa reconstrução, otimizando a renderização. No entanto, essa acomodação da *BVH* consegue ser eficiente por alguns frames apenas, sendo necessário

descarta-las após esse processo. Isso é exatamente o que o renderizador precisa, pois nesse trabalho a coerência temporal é primordial para o ganho de desempenho.

As bibliotecas Optix e Embree realizam dois trabalhos no renderizador: a criação e manutenção da *BVH* e o traçado de raios eficiente. A opção por essas bibliotecas se deu pela tendência acadêmica e industrial de utilizá-las para esses fins, além de permitir melhores comparações com os sistemas da indústria. Um exemplo é o Chaos Group VRay (Chaos Group, 2006), que utiliza ambas bibliotecas em sua versão mais atual. Da mesma forma que o VRay, este trabalho, inicialmente, implementou uma versão própria das mesmas. No entanto, a eficiência das implementações da Embree e Optix é difícil de se alcançar, pois ambas utilizam muitos recursos de baixo nível do hardware para ganharem desempenho.

Uma vez observado tal fato, o renderizador atua para manter a coerência da *BVH* quando acionado para a renderização de vários frames próximos, aumentando o ganho em eficiência com o uso da *PI* e com o uso da *BVH* adaptada. Essas otimizações não são utilizadas na maioria dos sistemas observados, pois esses não consideram o uso de frames anteriores no processo de renderização.

3.1.7. Otimização de Shade Tree

Além da eficiência na manipulação dos dados geométricos, o renderizador deve possuir um sistema robusto para trabalhar com os materiais da cena, isso devido ao custo associado aos materiais ser cada vez maior em relação à geometria. Dessa forma, os trabalhos de Cook (1984), McGuire et al. (2006) e Foley & Hanrahan (2011) são os instrumentos mais adequados para a aceleração da renderização de materiais.

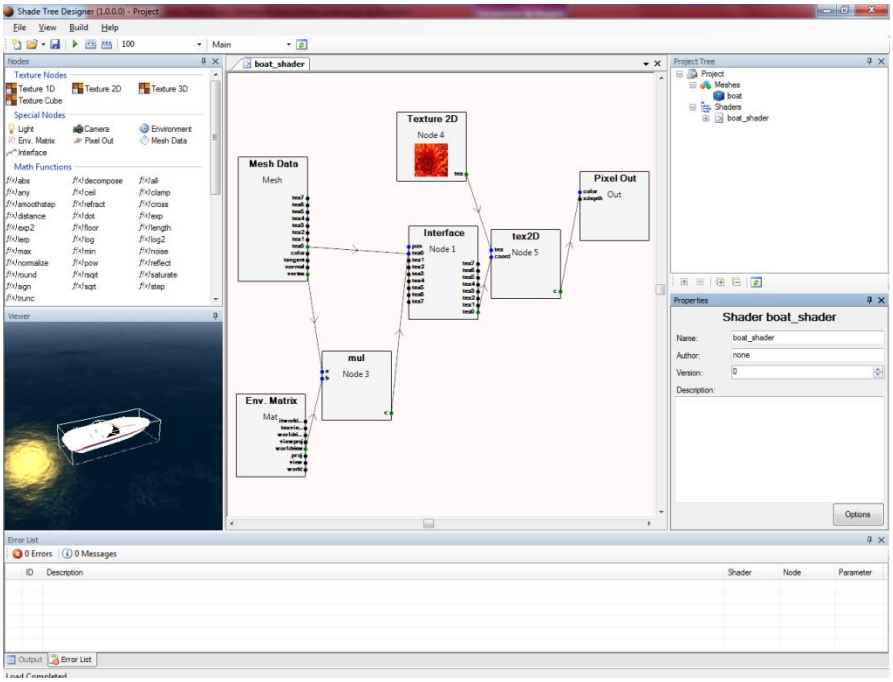


Figura 21: Sistema de Criação de Shade Trees Desenvolvido

Assim, foi desenvolvido o sistema Shade Tree Designer (STD) (figura 21) para atender as demandas de se projetar materiais, independente de linguagem ou tipo específico de hardware. O sistema desenvolvido permite a construção e a visualização imediata do shader desenvolvido, assim, o processo de teste e validação é dinâmico. No entanto, o sistema limita-se a exibir o resultado em uma cena teste com poucas luzes e modelos de pequeno porte.

O STD foi desenvolvido em 2011, num contexto em que apenas o RenderMan se preocupava com otimizações de *Shade Tree*, devido a ser baseado em rasterização, porém, não havia um sistema que fosse independente de linguagem ou hardware. Em 2013, sistemas como o Autodesk 3D Studio Max criaram estruturas para que os renderizadores possam usar *Shade Trees*, mas que ainda são dependentes de hardware e software. Porém, apenas no final de 2014, a The Foundry criou um sistema que se comprometesse em ser independente de software: o Katana (The Foundry, 2011). A necessidade disso é devida à grande demanda de renderizadores especializados, focados em pele, cabelos,

água, vegetação, nuvens e outros. Dessa forma, a descrição de material dos elementos da cena deve ser independente do *software*, caso contrário, seria necessário a criação de diversos materiais idênticos para cada tipo de renderizador.

No entanto, mesmo o Katana foca em ser um tradutor de parâmetros entre os sistemas, além de não se preocupar na eficiência do material quando convertido. Assim, o STD difere fortemente nesse quesito, pois o objetivo do mesmo é gerar o shader no formato final de uso.

Dessa maneira, uma vez descrito um material, o STD é capaz de gerar a descrição final do material, no formato escolhido. Assim, o renderizador solicita o material para o compilador de STD e o mesmo devolve o material no formato correto.

Atualmente, o STD é capaz de gerar os materiais para as linguagens GLSL, HLSL, Cg, RSL (renderman), Biblioteca Optix (CUDA) e Biblioteca Embree. Portanto, o mesmo atende estruturas de *Hardware* e *Software* distintas. O STD foi desenvolvido no início da presente pesquisa e a preocupação, na ocasião, com padrões abertos, levou o autor a optar pelo uso do engine OGRE (1999). Esta decisão de projeto seria atualmente revista, por questões de desempenho e de compatibilidade com outros módulos, aplicativos e padrões.

3.1.8. Geração de Material

O STD utiliza uma série de blocos básicos para descrever um material, como expressões matemáticas, blocos de acesso a texturas e operações de cor e vetor. Essas expressões podem ser combinadas e criar blocos maiores (figura 22) em um processo sucessivo de combinações. Assim, blocos estritamente de baixo nível de abstração se tornam extremamente intuitivos para designers e artistas de 3D.

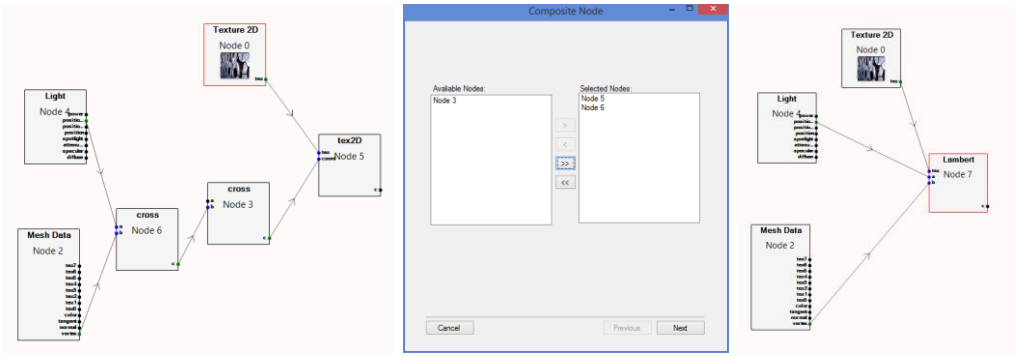


Figura 22: Criação de Bloco Composto

O procedimento de converter o material consiste em processar as expressões dispostas em um grafo orientado. Nesse momento o compilador do STD avalia qual será a saída do sistema e cria o código corresponde ao que foi solicitado. No caso de descrição por linguagem de programação como o GLSL, Cg, RSL e HLSL, o sistema cria o material em formato de código. No caso do Optix e Embree, criam-se duas saídas, a primeira sendo um código em CUDA e C++, respectivamente, os quais contém as instruções de chamadas e o descritor de classe IMaterial (Interface de material para o sistema). A outra saída é uma *DLL* que possui a chamada *getMaterial* (função obrigatória e definida como *dllexport*), a qual retorna um ponteiro para o material contido na *DLL*. Ou seja, após a geração do código, são chamados os programas de compilação final vc++ e cuda (nvcc) + Optix. As *Dlls* geradas são carregadas dinamicamente pelo renderizador.

Observa-se que cada material gera um identificador único (*GUID*), o qual serve para otimizar o gerenciador de recursos a gerar shader iguais apenas uma vez. É importante notar que no caso do Optix, a *DLL* só pode ser utilizada em arquiteturas no mínimo Kepler, por permitirem uso de *DLLs* fora do contexto de memória do software executor, além disso, nem todas as placas NVidia suportam a execução do Optix. Desse modo, esse procedimento garante que o *shader* produzido utiliza os recursos de hardware disponíveis. Porém, os testes que foram realizados utilizaram um único kernel CUDA para o processamento, sendo que o

shader utilizado foi combinado manualmente. Isso foi necessário por ser o método realizado pelo VRay e ser mais fácil de testar em arquiteturas mais antigas de placas de vídeo, que não contam com o *Dynamic Parallelism*.

O procedimento descrito acima pode ser resumido em um fluxograma, como o apresentado na figura 23.

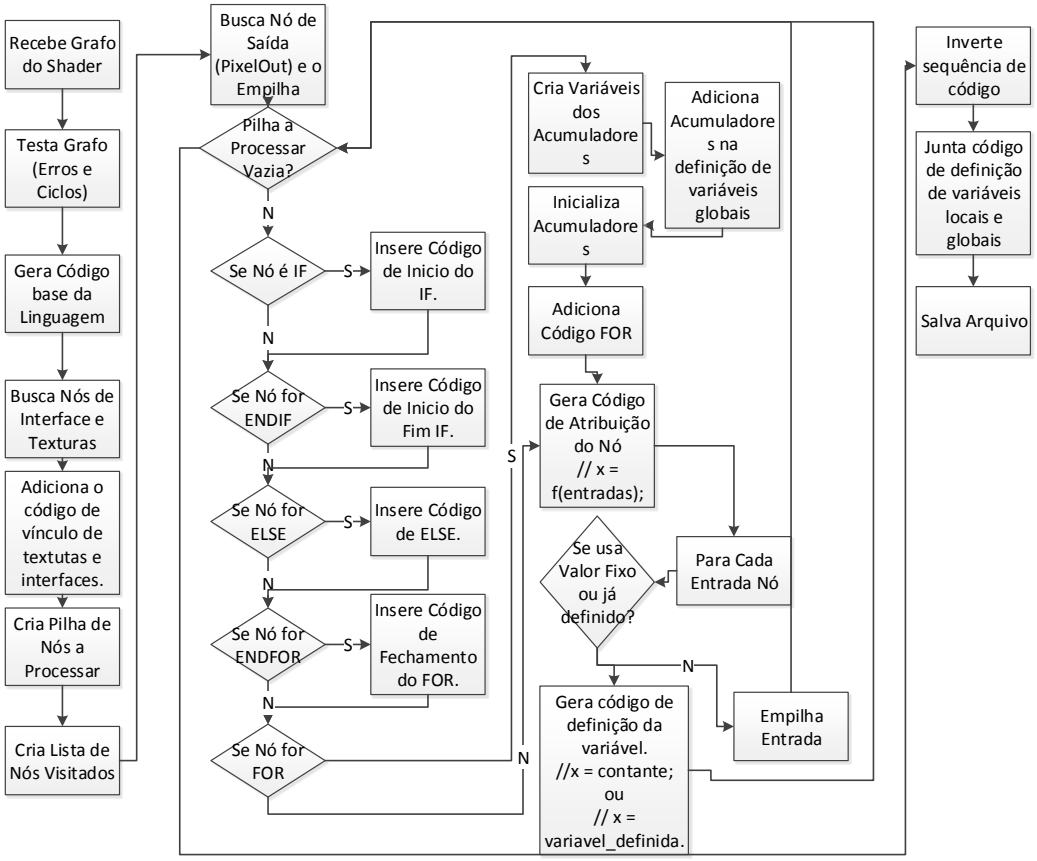


Figura 23: Algoritmo de Geração de Material (image produzida)

Observa-se que o procedimento assume que o material deve possuir no mínimo uma saída de dados, que no caso, é a cor do material. Assim, caso o material não apresente essa característica, o STD apresenta um erro ao usuário. Além disso, são verificados problemas como ciclos, incompatibilidade de dados e outros.

O algoritmo utiliza uma pilha que realiza o caminho do material de forma reversa, montando as expressões de forma única, ou seja, caso duas ou mais saídas compartilhem o mesmo cálculo com os mesmos dados de entrada, a computação será realizada apenas uma vez e armazenada de forma eficiente em registros ou memórias locais nas placas de vídeo.

Assim, num procedimento recursivo o sistema vai montando a sequência de expressões necessárias para gerar o material. No entanto, existem duas estruturas que são processadas de forma especial: condicionais (nós IF, ELSE e ENDIF) e repetidores (nós FOR e ENDFOR). Esses que podem usar nós especiais chamados acumuladores, os quais são vinculados ao um FOR e tem um valor inicial, sendo que a cada interação uma das entradas pode ser o valor do mesmo (acumulação). O motivo é a restrição do uso desses recursos em muitos *hardwares*; assim, eles possuem tratamento específico dependo do sistema e, em muitos casos, o número de repetições pode ser limitado por *hardware*.

Outro recurso não permitido é a recursão, pois gera um ciclo no grafo. Cabe lembrar que os materiais muito básicos são desenvolvidos por pessoas que conhecem bem a estrutura matemática dos modelos de iluminação. Assim, estruturas específicas, como a pilha, são implementadas dentro de nós especiais e são disponibilizadas somente em equipamentos [5] e linguagens compatíveis. Sendo realizado desse modo, os artistas 3D e designer não enxergam essas estruturas, preocupando-se apenas na combinação de estruturas muito maiores e com parâmetros muito mais próximos da realidade dos mesmos.

[5] No momento da escrita dessa tese apenas as placas da séria NVidia Tesla superiores a K20 possuiam o recurso de Dynamic Parralelism, que permite o uso de recursões na GPU. Os resultados obtidos com esse equipamento foram colocados como anexo por serem muito específicos.

Um exemplo disso são as chamadas do nó de traçado de raio e chamadas de amostragem de Monte Carlo. Esses nós são considerados nós base e não são suportados em todos os hardware e tipos de materiais (materiais que usam o pipeline de renderização similares ao OpenGL) não o suportam e seu uso é alertado no processo de geração do material.

Por fim, as estruturas fixas dos códigos, como cabeçalhos e funções de integração, bem como estruturas de dados são carregadas de arquivos com modelos feitos de forma manual. Além disso, procedimentos específicos também são escritos manualmente e colocados como templates.

3.1.9. Renderização Multitécnica

Uma vez que se possui o controle de todo o processo de geração dos *shaders*, uma possibilidade é avaliar o tipo de *hardware* e algoritmo necessário para executar o material. Além disso, pode-se construir mecanismos de se adaptar um material para ser executado em outro tipo de renderização. Assim, o sistema pode optar por escolher entre a qualidade e velocidade, sem necessitar da criação de outro material para cada tipo de sistema.

A adaptação de *hardware* é presente nos *shader* HLSL, onde é possível escolher o tipo de função a ser executada dependo do *Shader Model* do *hardware* em uso; no entanto, essa escolha é manual. SABINO et al (2012) apresenta um algoritmo que utiliza Optix e OpenGL com GLSL para a renderização de cenas, sendo que a opção por Optix é armazenada no arquivo de descrição. Assim, o sistema opera de forma a utilizar múltiplas técnicas de renderização.

Essa característica é inovadora, pois, mesmo em sistemas comerciais, como VRay e RenderMan, essa característica não é presente. No caso do RenderMan, o motor de rasterização de micropolígonos é o que o torna robusto e rápido para cenas de alta complexidade, porém, o limita na capacidade de iluminar as cenas de forma adequada, sem o uso do *Fakeosity*. Em 2002 o renderman passou a permitir o uso de *Ray tracing* em alguns materiais especiais, para simular espelhos e reflexões. Nesse caso, uma vez que a cena utilizasse um material desse tipo, a mesma passava a ser renderizada todo por *ray trace*, aumentando muito o tempo de renderização. Em 2007 o RenderMan passou a assumir funções do RSL que suportavam o *ray trace* (trace).

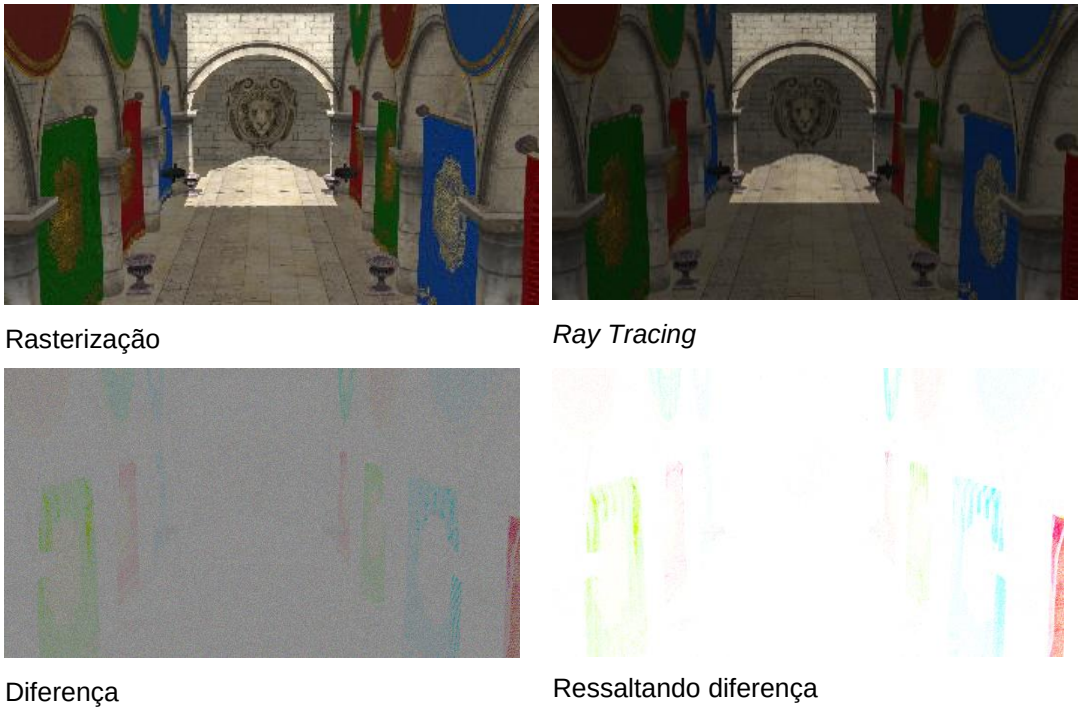
De forma similar, o VRay, que é naturalmente baseado em *Ray tracing*, suporta *photon tracing*. Porém, uma vez habilitado o uso de uma técnica no VRay, exclui-se o processamento com outra.

Essa característica cria um comportamento na produção de cenas no cinema e TV, forçando as equipes a criarem duas ou mais versões de uma cena. Assim, elementos específicos são renderizados com técnicas diferentes e depois passam por um processo de composição de imagens.

No entanto, como observado, a escolha das técnicas de renderização é diretamente ligada ao material utilizado e a disposição do objeto que o utiliza na cena. Assim, uma esfera com material difuso do tipo Blinn poderá ser rasterizada ou ser usado o *ray tracing*, dependo de quanto de luz ele recebe e a proximidade com outros elementos. Outro fator importante é a proximidade com a câmera, objetos distantes da cena podem usar técnicas mais simples para renderizar, e, mesmo quando têm influência de espalhamento de cor na cena, eles podem usar *VRLs* ao invés de *ray tracing*.

O renderizador então pode sugerir uma técnica baseada na disposição e tipo de material. Outra forma de se analisar o impacto visual de uma

técnica sobre a outra é comparar imagens produzidas por técnicas diferentes (figura 24). No entanto, o custo de se computar o resultado de múltiplas técnicas durante a execução é impeditivo, visto que consumirá recursos. Porém, observando a forma de trabalho na produção de uma cena, percebe-se que a sequência de frames é renderizada diversas vezes uma mesma cena (um artista 3D muito experiente chega a renderizar uma cena mais de 4 vezes). Além disso, dificilmente são alteradas as posições dos objetos e câmeras, sendo as alterações mais concentradas em parâmetros de material.



Rasterização *Ray Tracing*

Diferença Ressaltando diferença

Figura 24: Diferenças entre técnicas, usando a cena “San Miguel” (Barringer R. et al, 2014)

Tendo isso em vista e o fato de que as *Render Farms* têm uma ociosidade razoável, principalmente em turnos noturnos, o renderizador, quando instalado como serviço, pode iniciar a renderização dos *frames* usando outras técnicas mais simples durante esse período. Uma vez que o mesmo tenha como comparar os resultados (subtração de imagens) é possível saber quais os efeitos dessa decisão e quais os objetos menos

afetados (e observe-se que as PIs desses testes também são consideradas).

Assim, o renderizador deste trabalho pode armazenar as possíveis trocas de materiais de forma muito mais correta (quando habilitado pelo usuário). O objetivo dessa análise do renderizador é permitir que o tempo total de renderização possa ser reduzido caso haja uma demanda emergente da cena e não haja tempo hábil para a entrega com técnicas mais demandantes computacionalmente. Outra adequação abordada quando a troca do tipo de renderização é muito discrepante [erros de alta magnitude e objetos em planos médios (entre o objeto principal, primeiro plano, e o fundo) com materiais que usam muitos traçados de raios], o renderizador pode adotar a redução do número de amostras. Esse processo consiste em uma busca similar à busca binária onde, a cada tentativa, o renderizador reduz a amostragem pela metade e faz a seguinte avaliação: caso ainda esteja com pouco erro (dado definido pelo usuário), ele persiste na redução, caso contrário, ele dobra o valor da amostra em teste e continua até atingir um erro aceitável. Cabe lembrar que esses testes consomem recursos dos servidores e devem ser autorizados pelos usuários.

A esse processo chama-se de renderização multitécnica, pois o processo de geração da imagem utiliza diversas técnicas de forma conjunta para produzir a imagem final. Essa característica automatizada é outro ponto de grande inovação deste trabalho. As técnicas empregadas no sistema foram a rasterização (usando o OpenGL), o *Ray Tracing* (mais precisamente o *Path Tracing*) e as *VRLs* (ambas usando Embree e Optix). Optou-se por não adentrar nos mecanismos de renderização de cada uma delas por não haver mudanças significativas nos processos de renderização apresentados nos artigos correspondentes.

3.1.10. Combinação de Técnicas

Apesar de ser extremamente útil o uso de múltiplas técnicas para a produção da imagem final, esse processo cria alguns artefatos quando combinados de forma inadequada.

Foram avaliadas diversas alternativas para se realizar essa compensação dos resultados, no entanto, a solução mais adequada foi o uso de *Blend Zones*, ou zonas de suavização, que consistem em volumes nos quais as técnicas em troca serão usadas, criando-se, assim, uma impressão suave de transição. A figura 25 ilustra a situação.

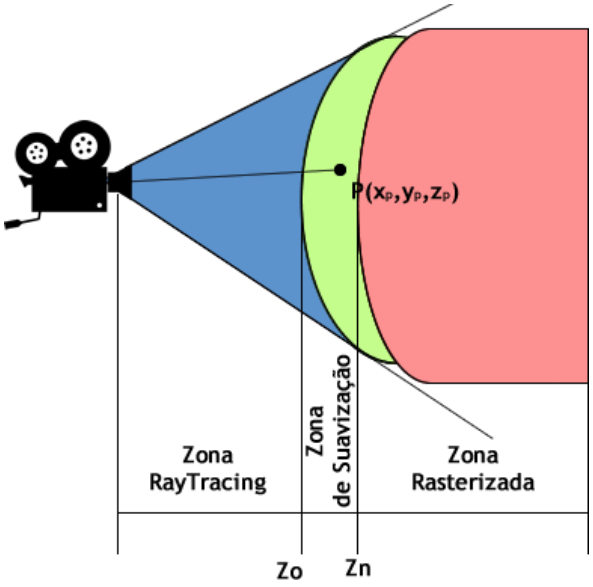


Figura 25: Blend Zones (imagem produzida)

O tamanho da zona de suavização é dependente da distância para o observador, ou seja, depende da área projetada dos elementos na imagem. A forma de como se realiza a suavização, depende das técnicas envolvidas. Dessa forma, uma transição de uma técnica qualquer para a outra envolverá o processo de renderizar todos os elementos do subvolume comum e armazenar o mapa de profundidade do mesmo, assim, quando um *pixel* é consultado, verifica-se sua profundidade e, dependendo disso, o mesmo é copiado para imagem final ou o mesmo é

computado com a técnica de maior complexidade e o resultado combinado usando a expressão $c_f = \alpha\left(1 - \frac{z_p - z_o}{z_n - z_o}\right) c_o + (1 - \alpha)\frac{z_p - z_o}{z_n - z_o}\, c_n$, onde α é um fator de controle, c_0 é a técnica de maior complexidade e c_n é a de menor complexidade.

Essa forma de compor zonas é similar ao que a indústria de cinema chama de *Deep Compositing* (Lokovic & Veach, 2000), (Heckenberg et al, 2011), que consiste em juntar elementos usando o canal de profundidade. No entanto, realizada dessa forma, a composição utiliza os dados de forma mais precisa, uma vez que toda a cena é interpretada como objetos 3D com geometria e materiais, diferente das nuvens de pontos orientadas que são propostas pelo *Deep Compositing*. A figura 26 ilustra o uso de múltipla técnicas.

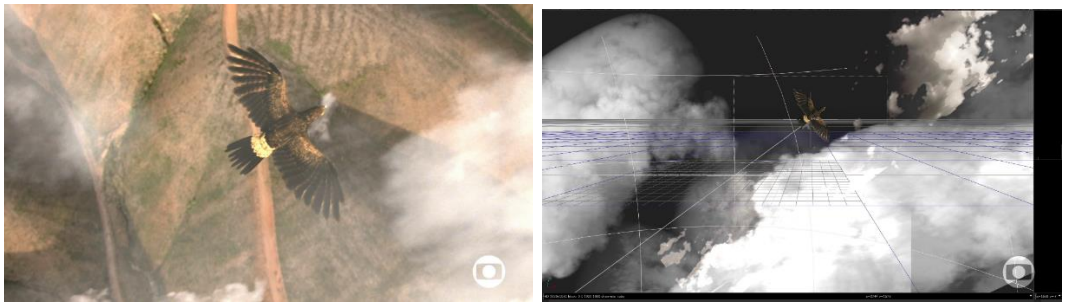


Figura 26: Múltiplas técnicas técnicas usadas em uma cena de Saramandaia (TV Globo, 2014). Imagem reproduzida sob a política de *fair use*.

3.1.11. Cache

Mesmo com diversas otimizações, a maioria dos sistemas de renderização baseados em traçado de raios utilizam o artifício do *irradiance cache* (*IRC*), que consiste em uma pré-amostragem de irradiância da cena, armazenada de forma 3D, como mostra a figura 27.

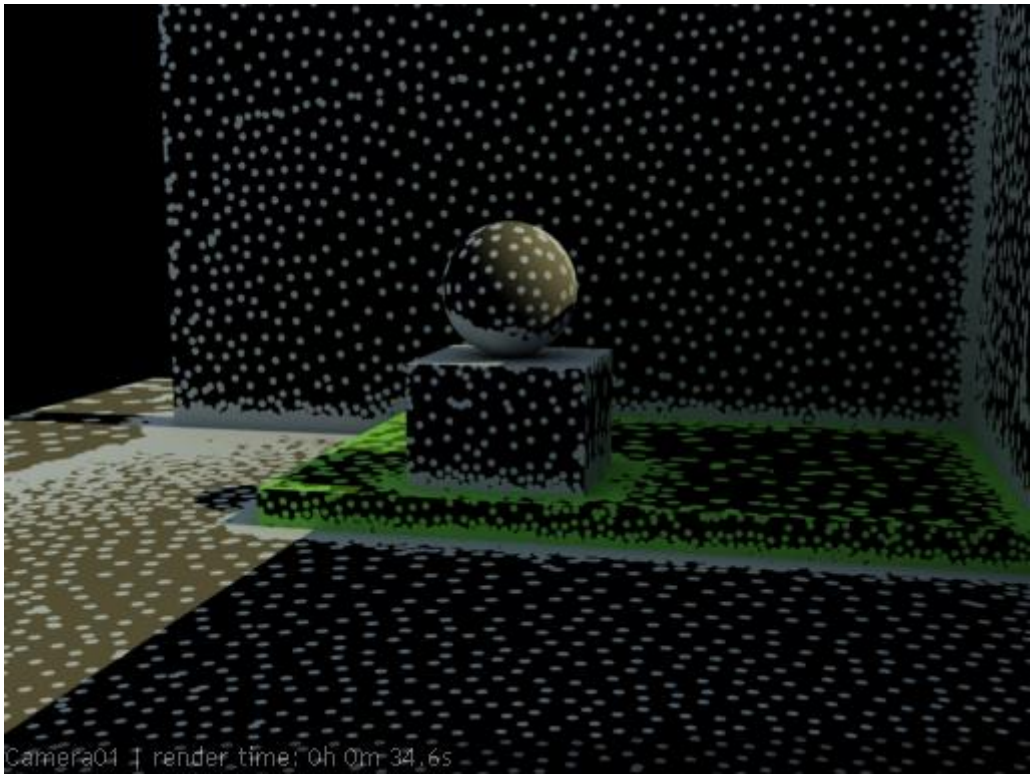


Figura 27: *Irradiance Cache (Krivanek et al., 2009)*

Dessa forma, o objetivo do *irradiance cache* é criar uma amostragem que possa ser usada por *frames* adjacentes ao *frame* no qual o mesmo foi gerado. Assim, além de otimizar o tempo de renderização, ele serve como um suavizador, reduzindo problemas de *flickering*, que são variações bruscas de luz devido à baixa amostragem.

No entanto, para o funcionamento do *IRC*, é necessário que, antes da renderização final, seja realizada uma etapa de criação do *IRC* – o que basicamente é selecionar o número de frames de distância entre os *caches* e calcular os *IRCs* desses *frames*. Esse processo é extremamente custoso, pois deve-se ter muitas amostras do espaço para conseguir um bom reaproveitamento e interpolação. Essa etapa consiste no uso de algoritmos sofisticados de amostragem como o *importance sampling* (Krivanek & Gautron, 2009) (figura 27).

Uma vez computados os caches, inicia-se o processo de renderização, que, simplificadamente, pode-se considerar que ocorre de forma similar ao algoritmo padrão. A diferença é que, ao se solicitar uma amostragem a partir de um ponto no espaço, ao invés de inicializar um motor de amostragem de Monte Carlo, consulta-se os *caches* adjacentes ao *frame* em questão, combinando-se linearmente os resultados (Walter et al, 1999) (figura 28).

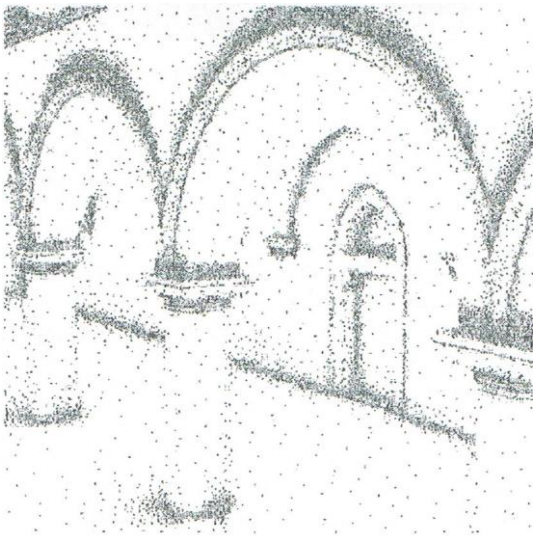


Figura 28: Amostragem dos Irradiance Caches *(Krivanek et al., 2009)*

Caso um ponto de partida não possua a irradiância pré-computada pelo *IRC*, é realizado um processo de interpolação dos pontos adjacentes, ponderados pela distância ao ponto desejado, sendo considerada a orientação da amostra pelo produto escalar da geometria do ponto considerado. Krivanek & Gautron (2009) apresentam a forma mais adequada de se implementar um sistema de *IRC*, de forma a ser utilizada por sistema multiparalelos. O código e a descrição dessas técnicas foram usados na implementação do renderizador.

Observando o funcionamento do *IRC*, o renderizador foi projetado para gerar os *irradiance caches* com a duração do número de *frames* que as *BVHs* são reconstruídas e considera-se que o processo de distribuição

de *frames* fornece *frames* próximos para um mesmo servidor, assim, quando o *frame* inicial de um servidor estiver disponível, o próximo *frame* poderá utilizar o *IRC* gerado pelo servidor e, como o intervalo entre os mesmo é pequeno, o próximo *IRC* já deve estar pronto, sendo que, caso não esteja pronto, o mesmo deve esperar a completude.

Uma tendência observada em (Sintorn et al, 2014) e (Barringer & Akenine-Möller, 2014) aponta para renderizadores que não utilizam *IRC*, pois o uso do mesmo cria o artefato de imagens borradas, ou seja, muito suavizadas. Esses trabalhos criam novas formas de se realizar amostragem de forma mais eficiente e que evitam o surgimento de zonas de *flickering*.

3.1.12. Arquitetura do Renderizador

O processo de se produzir um renderizador com as características expostas necessita de uma arquitetura modular e expansível, tendo em vista a necessidade de se utilizar *plug-ins* gerados dinamicamente e funções que necessitam de reflexão de dados. A figura 29 apresenta um diagrama esquemático em alto nível do sistema desenvolvido.

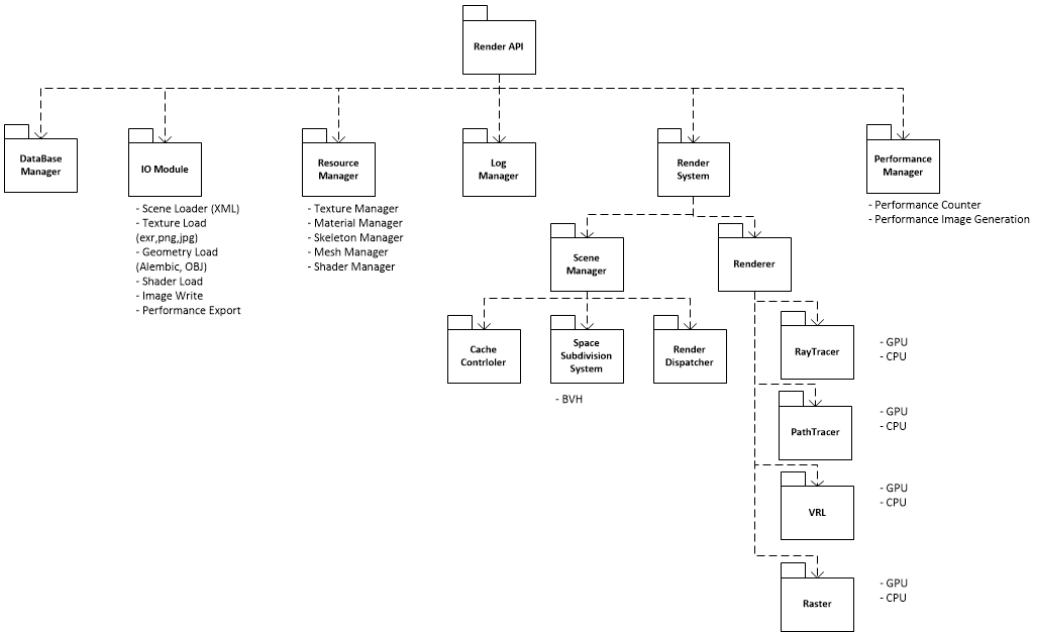


Figura 29: Arquitetura do sistema local proposto (imagem produzida)

Pode-se perceber que o sistema conta com um subsistema de carregamento e salvamento de dados e um gerenciador de recursos, visto que muitos dados são compartilhados numa cena (texturas, geometrias e materiais). Assim, todos os elementos da cena, e até mesmo o arquivo e o frame, são identificados de forma única com o uso de um identificador único (*GUID*).

Isso é necessário para o sistema economizar memória e processamento, além de ser fundamental para o sistema de análise e registro dos dados de performance. Os dados de performance são armazenados em um banco de dados, o qual é local. O sistema armazena somente os dados de referência de um frame, como os dados de complexidade, anotados a cada variação, os dados de diferença de técnicas de renderização e as decisões de trocas. A figura 30 ilustra o modelo de dados utilizado para o armazenamento local.

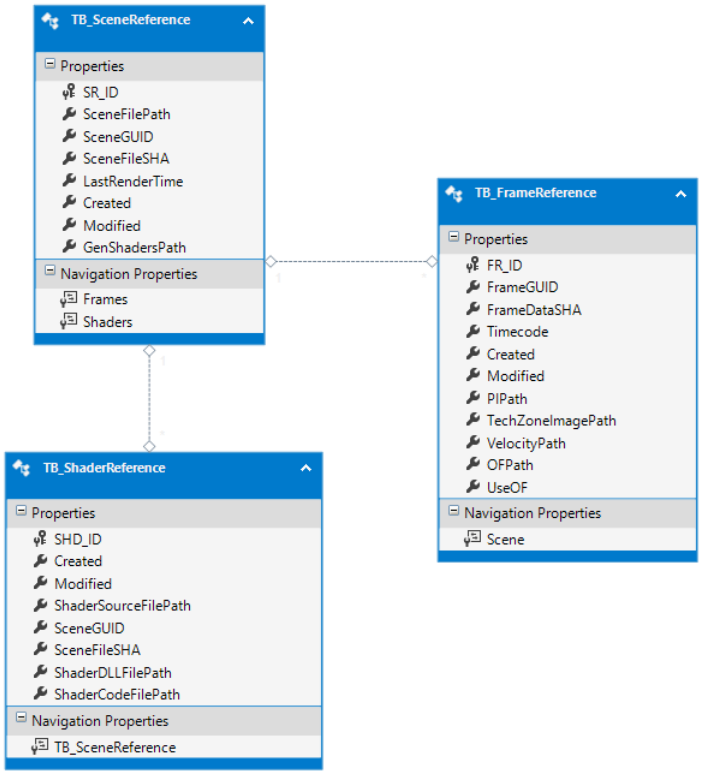


Figura 30: Modelo básico de dados do banco de dados (imagem produzida)

No protótipo desenvolvido para esta tese, o armazenamento é feito pelo SGBD Microsoft SQL Server 2012 Express e o acesso é abstraído pela *API Entity*, para simplificar as consultas ao banco através de expressões *LINQ*. O módulo de gerenciamento e comunicação com o Banco de Dado foi desenvolvido em linguagem C#.

No entanto, desenvolver um renderizador exige um esforço muito grande de pesquisa e de desenvolvimento. Dessa forma, optou-se por restringir o protótipo a algumas funcionalidades chave que sejam suficientes para validar a metodologia e os conceitos propostos. Assim, o renderizador utiliza apenas modelos triangulares e objetos paramétricos. A leitura desses arquivos se dá por meio do formato *Alembic* (Alembic, 2010) e *Wavefront OBJ*, para malhas triangulares. O *Alembic* é um formato industrial de armazenamento e compartilhamento de modelos 3D, sendo utilizado pela grande maioria dos sistemas. Da mesma forma, a leitura de texturas é feita através do formato *OpenEXR* (Lucas Digital Ltd, 2006), o qual suporta armazenamento preciso de imagens, sendo aceitos ainda PNG, TGA e JPEG. Ainda, somente a animação de câmera foi implementada.

As cenas geradas para o sistema podem ser produzidas pelo 3D Studio Max 2015 através de um *plugin* (figura 31) desenvolvido em *MaxScript*. Esse *plugin* realiza o procedimento de exportar todos os dados de geometria para o formato Alembic (por isso a necessidade do uso da versão 2015), copiar as texturas e criar um arquivo de mapeamento de materiais, onde cada material no 3D Studio Max é referenciado de forma única, da mesma forma que as geometria e texturas, utilizando *GUID*. Esse arquivo de texto é apenas uma tabela com o seguinte formato: *<nome do item dentro do 3D Studio Max>=<GUID>=<arquivo>*. No caso de materiais, *<arquivo>* é vazio.

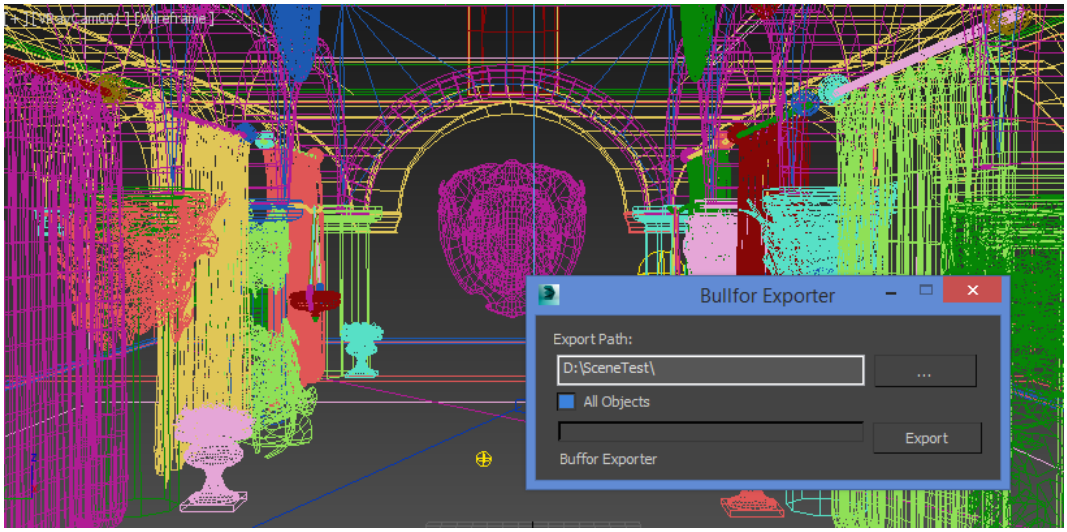


Figura 31: Interface *MaxScript*, usando a cena “San Miguel” (Barringer R. et al, 2014)

Uma vez exportado, a edição dos materiais deve ser feita no Shade Tree Designer, importando o arquivo texto. Por fim, O STD gera o arquivo XML final da cena e o arquivo comprimido ZIP com todos os elementos necessário para a renderização. Dessa forma, uma vez recebido o arquivo de renderização final, o renderizador pode renderizar qualquer frame da cena. O STD não consegue visualizar o modelo exportado em Alembic, visto que o mesmo utiliza a engine Ogre. Apenas se houver a conversão para o formato próprio do Ogre é possível visualizar os shaders na malha final (quando utilizado HLSL e GLSL).

O sistema suporta luzes pontuais e área, usando como emissores espectros conhecidos (D65), espectros do corpo negro e arquivos CSV. Dentre as funcionalidades, apenas *fallout* é implementado.

Ainda, os modelos de câmeras disponíveis são apenas Pin-Hole e um modelo físico (Kolb et al, 1995) que considera as informações de obturador, exposição, sensor, lente e sensibilidade.

Conforme já informado, o sistema de criação e manutenção das estruturas de aceleração, traçado de raios e *streaming* de dados são executados pelas bibliotecas Optix e Embree. Sendo assim, toda a parte

de renderização que utiliza traçado de raios foi implementada em C++ CLR (com Optix utilizando CUDA através do compilador nvcc); assim, os módulos desenvolvidos em C# podem ser utilizados para executar procedimentos. A parte de Rasterização utiliza a API OpenGL, com a linguagem de Shader GLSL e foram implementadas em C#. O Shade Tree Designer foi totalmente desenvolvido em C#. Muitas estruturas foram simplificadas por conta da enorme quantidade de código necessária para funcionar todo o sistema.

O sistema de renderização funciona como um serviço do sistema operacional, pois necessita permanecer computando testes para o sistema multitécnica. Dessa forma, o renderizador é uma biblioteca que é chamada quando a renderização é necessária. Essas chamadas são solicitadas pelo gerenciador de renderização (capítulo 4).

3.1.13. Rotina Principal do Renderizador

Esquematicamente, a rotina apresentada pelo fluxograma da figura 32 é a forma de operação do renderizador.

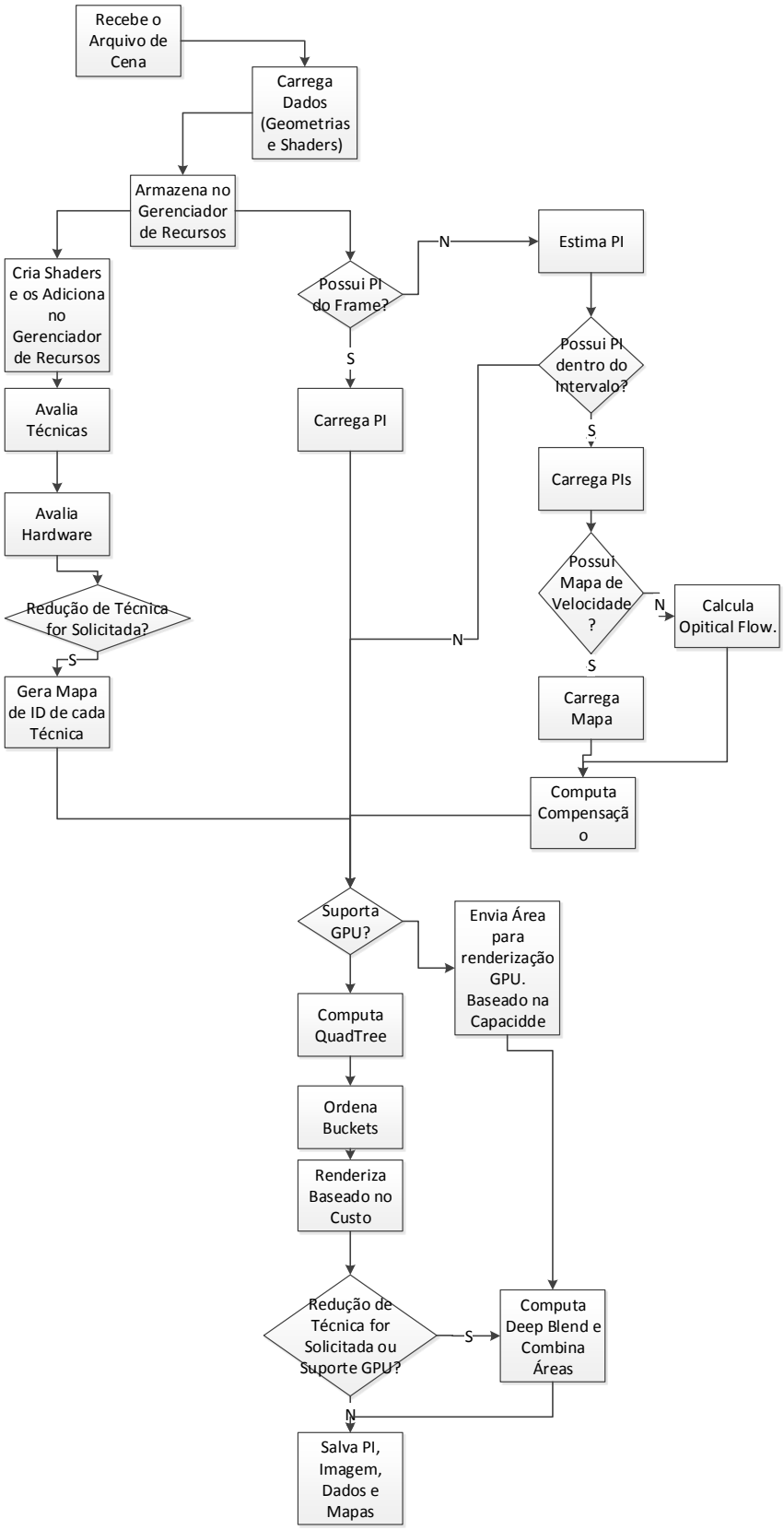


Figura 32: Fluxograma de operação do sistema proposto

Assim, as otimizações da renderização local são apresentadas. No capítulo 4 inicia-se a análise e desenvolvimento das metodologias para a renderização global de uma cena.