Uma implementação híbrida de *raytracing* em processadores gráficos programáveis ou processadores de propósito geral

Alessandro Ribeiro da Silva[1], Carlos Augusto Paiva da Silva Martins[1](Orientador)
[1]PUC-MG - Pontifícia Universidade Católica de Minas Gerais
spdoido@yahoo.com.br, capsm@pucminas.com.br

Resumo

Neste artigo são apresentados os resultados parciais da pesquisa que envolve o desenvolvimento de um raytracer que utiliza os recursos de processadores gráficos programáveis. Foram implementados dois tipos de raytracers, um que utiliza o processador gráfico programável e outro que foi desenvolvido para utilizar somente o processador de propósito geral. Ambas as implementações são bastante semelhantes entre si, pois a linguagem OpenGL Shading Language utilizada para programar os processadores gráficos programáveis é bastante semelhante a C++ que foi a linguagem de implementação do software. Para que estes raytracers pudessem ser executados na mesma aplicação, também foi desenvolvido um ambiente gráfico que contém estas implementações. Os testes feitos em dois processadores gráficos programáveis e dois processadores de propósito geral mostraram um ganho de 6 vezes mesmo comparando os processadores de melhor desempenho, onde a GeForce6600 superou a velocidade do Pentium IV 2.4Ghz.

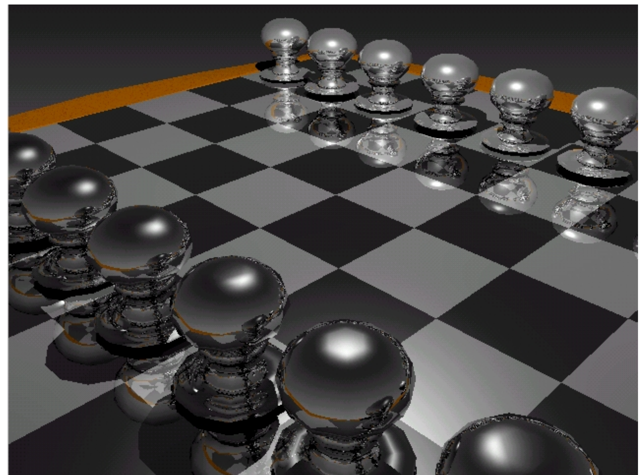

Figura 1. Renderização de uma cena realizada pelo raytracer no modelador Blender.

1 Introdução

Nesta seção são apresentados o contexto em que este trabalho se insere, o problema motivador, especificados os objetivos propostos, a meta principal e a organização das seções deste artigo.

Dentro da computação gráfica têm-se técnicas que buscam gerar imagens foto-realistas como o *raytracing*. Esta técnica é baseada na simplificação do modelo de iluminação utilizando-se do comportamento físico da luz para sua implementação através de um meio computacional [3, 5, 9]. Os resultados da implementação desta técnica são bastante interessantes no que diz respeito a qualidade da imagem gerada. Na figura 1 é possível observar a qualidade da cena gerada pelo raytracer que está implementado no modelador Blender(www.blender.org).

Umas das principais dificuldades da implementação da técnica de *raytracing* é a demanda computacional necessária que é determinada em função da resolução da imagem e/ou da complexidade da cena [3, 5].

Como tentativa de resolução deste problema da demanda computacional e do tempo necessário para se produzir uma cena existem diversas propostas que envolvem a utilização de estruturas de dados para otimização de acesso, *hardwares* especialmente desenvolvidos para realizar o *raytracing*, processamento paralelo [2] ou a partir da utilização de processadores gráficos programáveis [8, 1, 4].

O objetivo principal deste trabalho é a proposta e implementação de um *raytracer* que utilize os recursos programáveis de processadores gráficos programáveis. Entretando, quando os recursos programáveis dos processadores gráficos não estiverem disponíveis seja também possível a execução do raytracer no processador de propósito geral. Para a implementação dos módulos deste raytracer é utilizada a API (*Application Programming Interface*) OpenGL 2.0 e sua linguagem de programação de shader GLSL

(*OpenGL Shading Language*) [11, 6, 10].

Para atingir o objetivo proposto foram traçados vários objetivos secundários, que são:

- Implementar uma versão deste raytracer para que o mesmo possa ser executado em um processador de propósito geral para fins de comparação;
- Adquirir o conhecimento para programação de GPUs(*Graphics Processor Unit*) bem como utilizar a linguagem de shader;
- Definir quais características o *raytracer* deve possuir e implementar duas versões, uma que funcione no processador de propósito geral e outra que funcione na GPU programável;
- Analisar o desempenho destas implementações para verificação do ganho ou perda.

A meta principal deste trabalho é a implementação híbrida de raytracing, onde o mesmo possa ser executado em processadores gráficos programáveis e quando este recurso não for disponível também ter a possibilidade de executa-lo em processadores de propósito geral.

O processador gráfico programável é otimizado para efetuar cálculos de ponto flutuante de maneira eficiente além de ter uma capacidade de processamento considerável que se assemelha ou supera a capacidade de processamento de ponto flutuante em um processador de propósito geral.

Como o *raytracing* requer alta demanda de processamento e recursos computacionais resultando normalmente em altos tempos de resposta, se torna uma boa alternativa a utilização destes processadores gráficos programáveis para a otimização dos cálculos de todo o algoritmo. Além disto, a qualidade das imagens geradas baseadas em raytracing se assemelham a fotos reais.

Este artigo possui uma organização incremental, onde inicialmente são abordados os trabalhos relacionados seguidos de explicações sobre o funcionamento básico da técnica de raytracing, pipeline de renderização e linguagem escolhida, mostrados os resultados e apresentada a conclusão.

A seção "Trabalhos relacionados" apresenta as principais publicações que envolvem a otimização do raytracing a partir da utilização de processadores gráficos programáveis. A seção "Raytracing" apresenta uma explicação de como é a base para se implementar esta técnica. A seção "O pipeline programável e Linguagem de shader" apresenta uma visão de alto nível de como ocorre o processamento no pipeline gráfico programável e as principais linguagens de programação de processadores gráficos. A seção "Raytracer desenvolvido no processador gráfico programável e no Processador de propósito geral" apresenta os principais blocos do raytracer. A seção "Resultados" apresenta os processadores utilizados, os testes realizados e mostra uma analise sobre os tempos de resposta. A seção "Conclusão" contém um fechamento do artigo com base nos resultados apresentados na seção anterior e é feita uma listagem das continuações a serem dadas a esta pesquisa como trabalhos futuros. A seção "Referências" contém a listagem das referências utilizadas como base deste artigo.

2 Trabalhos relacionados

Existem algumas implementações de *raytracing* que se baseiam na utilização de processadores gráficos programáveis [8, 1, 4]. Uma que implementa inteiramente em *shader* [8], outra baseada em implementar em shader somente a interseção de raios com triângulos e utilizar o processador de propósito geral para gerenciar a geração de raios e demais etapas do processamento [1] e ainda outra implementação baseada na utilização da biblioteca Sh(Embedded Metaprogramming Language) definida para se utilizar recurso programável através da própria linguagem de programação c++ [4].

Na implementação de *raytracing* inteiramente no processador gráfico programável, pode se separar blocos lógicos seqüenciais, possibilitando a divisão de suas etapas de processamento em geração de raios, controle de colisões e tonalização (*shading*) [8].

A proposta de implementação parcial se baseia na utilização do processador de propósito geral como base e implementar parte do *raytracer* no processador gráfico como um co-processador aritmético (usado somente para teste de interseções). Assim se cria uma *engine* de raios que serve como ponte entre o processador gráfico programável e a aplicação, que geralmente utiliza o processador de propósito geral [1].

A biblioteca Sh foi criada para facilitar a programação de *shaders* sem que o desenvolvedor utilize uma linguagem específica de shader. O código para a construção de um *raytracer* que utiliza esta biblioteca pode ser escrito completamente em c++ [4].

3 Raytracing

Esta técnica se baseia na geração de imagens foto realistas com base no princípio de funcionamento da Câmera de Pinhole [5]. Se calcula a intensidade dos *pixels* de toda a tela de projeção de acordo com as contribuições oriundas das fontes de luzes presentes na cena [3, 5].

Estas contribuições são relativas à interação de um ponto específico do objeto com todas as luzes contidas em uma cena. Na figura 2 é mostrado um exemplo de como é o funcionamento desta técnica de raytracing. Onde se tem um olho (câmera) e a partir deste se lançam raios em toda a superfície da tela de projeção, assim podendo se verificar se este raio lançado colide ou não com algum objeto descrito

no modelo tridimensional utilizado. Se existe uma colisão então são lançados raios em direção a todas as fontes de luzes presentes com o objetivo de determinar se este é um ponto de sombra ou não [3, 9, 5].

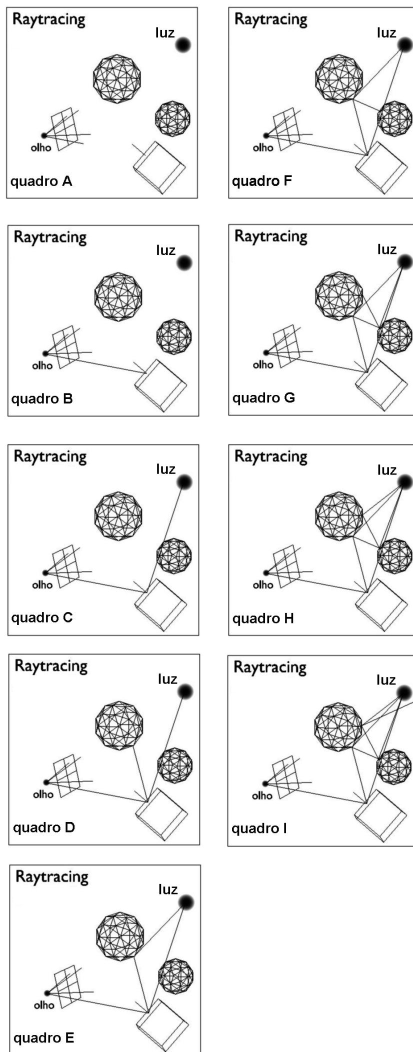

Figura 2. Seqüencia de possíveis passos da técnica de raytracing.

A figura 2 apresenta a seqüência necessária para o cálculo da intensidade de um *pixel*. No quadro A se tem uma imagem que contém duas esferas, um cubo, uma luz, um olho e na frente do olho uma tela de projeção que possui quatro *pixels*. Também são mostrados os raios que são gerados a partir do olho em direção aos *pixels* da tela de projeção, os quadros seguintes mostram a evolução do cálculo do *pixel* inferior direito da tela de projeção.

No quadro B o raio colide com o cubo, a partir deste ponto são calculadas as influências das luzes sobre este ponto ou determinado se este ponto é um ponto de sombra.

No quadro C é determinado que aquele raio de luz é obstruído por uma das esferas, sendo assim é um ponto de sombra.

No quadro D é mostrado o raio gerado a partir da reflexão do raio incidente sobre a superfície do cubo. Depois deste quadro o algoritmo é reaplicado até que no quadro I não existam mais influências para a intensidade final daquele *pixel* que está sendo calculado.

Através dos quadros da figura 2 se observa a quantidade de cálculos necessária para se obter o valor da intensidade de cada pixel na tela de projeção.

4 O pipeline programável e Linguagem de shader

Atualmente, com os avanços tecnológicos, alguns processadores gráficos vêm com um recurso interessante: a programação aberta de alguns estágios de seu *pipeline* de renderização. Como o desempenho de uma GPU (Graphics Processor Unit) é superior ao de alguns processadores de propósito geral, a utilização deste recurso se torna uma solução para problemas que envolvem uma demanda por cálculos de ponto flutuante. Estes dois estágios programáveis são a programação por vértice e programação por fragmento.

Uma possível visão lógica do *pipeline* de renderização é mostrada na figura 3 . Onde o Modelo representa os dados da cena que se deseja renderizar, como os vértices, arestas, normais, luzes, cores e etc, onde este estágio é apenas conceitual, ou seja, não faz parte da funcionalidade disponibilizada pela GPU.

Logo depois vem o estágio de processamento de vértices, que recebe todas as informações do modelo e tem o objetivo de aplicar as transformações (escala, rotação, translação) sobre os vértices do modelo. A saída deste estágio são os vértice transformados.

No estágio de rasterização das primitivas de desenho (objetos mais simples que podem ser desenhados. ex.: um triângulo), onde a partir dos polígonos visíveis na tela são gerados os fragmentos (futuros *pixels*) que preenchem estas primitivas.

A entrada do próximo estágio, o processamento de fragmentos, são os fragmentos gerados no estágio de rasterização. Aqui o objetivo é calcular a tonalidade final do fragmento a partir da informação de sua normal, posição na tela, posição no espaço, etc [11]. A saída do estágio de processamento de fragmentos são os *pixels* que são escritos no *framebuffer*(memória de quadro - região de memória que contem os *pixels* que serão mostrados na tela do dispositivo de exibição).

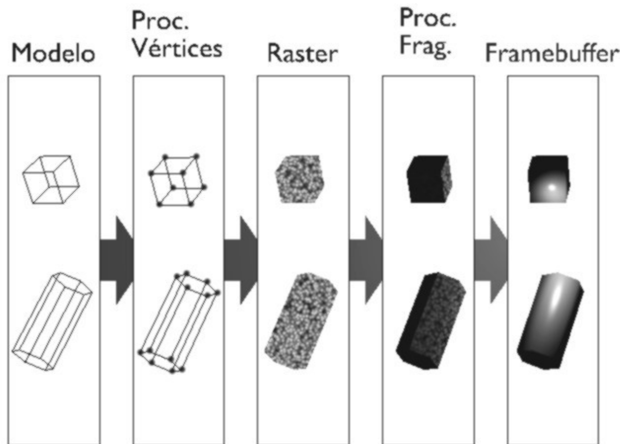

Figura 3. Possível representação lógica dos estágios de um pipeline de renderização.

Estes estágios programáveis foram projetados para permitir que o programador possa criar seus próprios efeitos, através de linguagens de programação, não se restringindo somente a utilizar recursos limitados de otimizações prontos em *hardware*, como reflexões, neblinas e etc.

4.1 Linguagem de *Shader*

Shader é o nome dado a cada instrução existente na definição de uma linguagem de programação de GPUs [10, 11].

As linguagens de alto nível mais populares para programação de processadores gráficos são HLSL (*High Level Shader Language*), GLSL (*OpenGL Shading Language*) e Cg (*Computer for Graphics*) [10, 11].

HLSL faz parte da API (*Application Programming Interface*) Direct3D do conjunto de APIs de programação multimídia DirectX da Microsoft. A não portabilidade é uma característica marcante desta linguagem. Sua estrutura baseia-se na compilação dos *shaders*, onde se gera na aplicação o código de comunicação com o *driver* de vídeo para acessar o processador gráfico. Necessita da instalação do SDK (*Software Development Kit*) do DirectX no sistema operacional Windows [10].

GLSL (*OpenGL Shading Language*) é a linguagem de shader do padrão OpenGL 2.0 (GLSL) onde seu funcionamento baseia-se na compilação do código de shader em tempo de execução através do *driver* do fabricante do processador gráfico. Necessita do *driver* de vídeo e dos arquivos de cabeçalhos da biblioteca para compilação [10, 11].

Cg (*Computer for Graphics*) é a linguagem de shader criada pela empresa Nvidia(www.nvidia.com). Tem o objetivo de ser independente de API gráfica e ser otimizada pelo *hardware*. A sintaxe é baseada nas linguagens HLSL e GLSL, onde tenta seguir o mesmo padrão da programação em código baseado em C. A desvantagem da utilização desta linguagem é a necessidade de suporte exclusivo de hardware para a utilização da mesma [7].

Como visto nesta seção, existem diversas linguagens para se trabalhar com processadores gráficos programáveis, e cada uma destas linguagens apresenta suas características próprias.

Ao se escolher uma destas linguagens deve se considerar a plataforma operacional, processadores disponíveis, portabilidade ou ainda a característica de compilação de seu código para construir uma determinada aplicação.

5 Raytracer desenvolvido no processador gráfico programável e no Processador de propósito geral

De acordo com a divisão do raytracing em etapas, como proposto por Purcell [8], foi delimitado o escopo do trabalho em três blocos lógicos (lembrando que cada um destes blocos pode ter sub-blocos internos):

- Geração de Raios: responsável pela geração de raios de luzes (ponto, direção) a partir da informação de rasterização da tela e configuração da câmera;
- Colisão: responsável por lançar o raio gerado e testar se o mesmo colide com uma lista de triângulos já armazenada;
- Tonalização: responsável por atribuir uma cor final ao *pixel* que foi rasterizado.

A estrutura proposta por Purcell [8] possui um bloco adicional denominado "estruturas de dados acelerada", que armazena uma lista de cubos com informações sobre quais triângulos estão dentro deles para efetuar os cálculos de colisões de forma eficiente. Este trabalho não implementa este tipo de estrutura, pois o objetivo do mesmo é a análise de desempenho em relação a qual processador está sendo utilizado (processador de propósito geral ou processador gráfico).

A figura 4 é uma representação da lógica básica de funcionamento do *raytracer* dentro do contexto do processador gráfico programável.

Partindo do modelo se tem uma primitiva do OpenGL chamada QUAD (um polígono que possui quatro vértices). Para se utilizar a funcionalidade de shader no OpenGL 2.0 é necessário que se criem dois programas, um que tenha as instruções do processador de vértices e outro que tem as instruções para o processador de fragmentos [11]. Como o objetivo deste trabalho é implementar o raytracer de forma

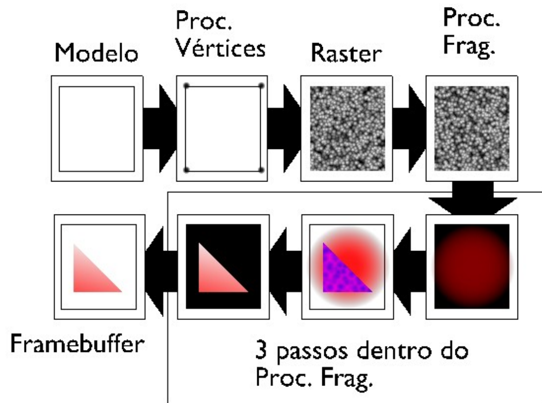

Figura 4. Diagrama do raytracer implementado dentro do pipeline de renderização. Dentro do estágio de processamento de fragmentos estão as três etapas do algoritmo de raytracing (geração de raios, colisão e tonalização).

otimizada então se utilizou principalmente o estágio de processamento de fragmentos já que este é mais eficiente que o estágio de processamento de vértices.

O programa de processamento de vértices não altera a funcionalidade de transformação, deixando os vértices de acordo com a configuração isométrica (paralela) da projeção da câmera. Veja o código escrito em GLSL para o processador de vértices a seguir:

```
void main(void){
   gl_Position = ftransform();
}
```

O estágio de rasterização recebe esta primitiva definida (o QUAD que ocupa a tela inteira) e a preenche gerando fragmentos (um para cada *pixel* final da tela).

E no estágio de processamento de fragmentos está o raytracer implementado com suas três etapas: geração de raios, colisão e tonalização. Onde ao final é escrito o resultado no *framebuffer*. Veja o código em GLSL para o processador de fragmentos a seguir:

```
void main(){
   vec3 finalColor = vec3(1.0,0.0,0.0);
   vec3 n0, n1, n2;
   property material;
   ray currentRay = getCurrentRay();
   colisionResult colision =
         getColisionResult( currentRay,
                            n0, n1, n2, material );
   if ( colision.L == INFINITE )
      discard; //background
   finalColor = shade( colision, currentRay,
                       n0, n1, n2, material );
   gl_FragColor = vec4( finalColor, 1.0 );
}
```

5.1 geração de raios

Este bloco é bem simples e é baseado na configuração de perspectiva da câmera. Onde o plano mais próximo é a tela de projeção e os raios devem possuir uma direção que aponte para esta tela. Na Figura 5 é mostrada uma possível representação da configuração da câmera, onde o vetor P0-P1 é o sentido positivo do eixo x e o vetor P1-P3 é o sentido positivo do eixo y sobre a tela de projeção.

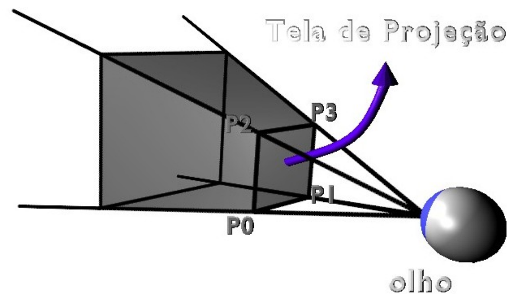

Figura 5. Representação da configuração da câmera, onde o sentido positivo do eixo x é o vetor P0-P1 e o sentido positivo do eixo y é o vetor P1-P3.

5.2 Colisão

Este foi o bloco mais complexo do raytracer implementado porque o mesmo deve definir uma estrutura de acesso a estrutura do modelo 3D utilizado e fazer os cálculos de interseções entre triângulos e raios.

Algoritmo de interseção de triângulos

O algoritmo de interseção de raios com triângulos utilizado primeiramente e apresentado por Carr [1], aborda a escolha do mesmo dentre vários voltados à implementação em processadores gráficos programáveis. Posteriormente, este mesmo algoritmo foi utilizado por Purcell [8] e Fortuny [4]. A seguir é mostrado o trecho de código de shader para efetuar o cálculo de interseção de um raio com um triângulo:

```
#define EPSILON .001
#define EPSILON2 .0001
void intersectTriangle( ray r,
                  float currentTriangleIndex,
                  vec3 v0, vec3 v1, vec3 v2,
                  inout colisionResult luvi ){
  vec3 e1=v1-v0;
  vec3 e2=v2-v0;
  vec3 pvec = cross( r.direction, e2 );
  colisionResult tuvd;
```

```
  float det = dot( e1 , pvec );
  if( abs(det) >= EPSILON ){
    det = 1.0/det;
    vec3 tvec = r.position - v0;
    tuvd.U = ( dot( tvec, pvec ) * det );
    vec3 qvec = cross( tvec, e1 );
    tuvd.V = ( dot( r.direction, qvec ) * det );
    tuvd.L =( dot( e2, qvec ) * det );
    if( tuvd.U >= -EPSILON2 &&
        tuvd.V >= -EPSILON2 &&
        (tuvd.U + tuvd.V) <= (1.0 + EPSILON2) &&
        tuvd.L <= luvi.L &&
        tuvd.L > 0.0) {
      luvi = tuvd;
      luvi.I = currentTriangleIndex;
    }
  }
}
```

5.3 Tonalização

Este bloco implementa um cálculo de iluminação simples, onde a partir da propriedade do material do objeto, da luz e do ponto de colisão se calcula a intensidade do pixel final. O código em GLSL pode ser visto a seguir:

```
vec3 lightGetIntensity( ray r, light l,
              property material, vec3 p, vec3 N ){
   vec3 ambientColor = l.emission.A * material.A;
   vec3 objToLight = normalize(  l.position - p );
   float difuseEscalar = dot( objToLight, N )*(-1.0);
   vec3 difuseColor = vec3(0.0);
   if ( difuseEscalar > 0 )
      difuseColor = l.emission.D *
                    material.D *
                    difuseEscalar;
   vec3 reflectedLightVector = -reflect(objToLight, N);
   float specularEscalar =
               dot(r.direction, reflectedLightVector);
   vec3 specularColor = vec3(0.0);
   if (specularEscalar > 0){
      if (material.shininess == 1.0)
         specularColor = l.emission.R *
                         material.R *
                         specularEscalar;
      else
         specularColor = l.emission.R *
                         material.R *
               pow(specularEscalar,material.shininess);

   }
   return ambientColor + difuseColor + specularColor;
}
```

5.4 Codificação dos raytracers

Ambos os raytracers possuem codificações semelhantes, pois a linguagem GLSL, utilizada para programar o processador gráfico programável, é baseada em C.

Veja o exemplo do código comum de geração de raios usando as linguagens GLSL e C++ a seguir:

```
ray getCurrentRay( void ){
  vec3 position = cameraP0P1 * gl_FragCoord.x +
                  cameraP1P3 * (rh - gl_FragCoord.y)+
                  cameraP0;
  vec3 direction =
               normalize(vec3(position - cameraEyePos));
  return ray (position, direction);
}
```

5.5 Ambiente Gráfico

Para que ambas as implementações de raytracers pudessem ser executadas na mesma aplicação foi necessária a criação de um ambiente. Este ambiente possui diversos blocos e um destes blocos, o Raytracer, é que define qual dos processadores será utilizado pelo raytracer: processador gráfico programável ou processador de propósito geral.

O ambiente gráfico fornece a gerência de janelas, controle de eventos de entrada, leitura e gravação de arquivos de imagens e um mecanismo de fácil configuração. Este ambiente pode ser dividido em blocos característicos: AutoStart, ConfigReader, ModelLoader, Raytracer, shared e WindowControl.

O bloco "AutoStart" é responsável pelo início do sistema, onde todos os módulos importantes são iniciados e ligados de acordo com suas necessidades.

O bloco "ConfigReader" recupera as configurações para a execução do aplicativo, como: resolução do *framebuffer*, arquivo para salvar as telas de renderizações, controle de eventos, mapeamento de teclas, etc.

O bloco "ModelLoader" carrega o modelo (descrição da cena) e alteração do mesmo.

O bloco "*Raytracer*" é responsável por gerenciar qual dos dois *raytracers* está ativo e também por fornecer uma interface da carga de modelo para cada um dos *raytracers* bem como suas respectivas chamadas em tempo de execução.

O bloco "shared" contem as definições básicas de todo o programa, possui funções de controle de tempo, definição de tipos vetoriais e matriciais e também de objetos mais complexos como câmeras, frustum (volume gerado pelas configurações da matriz de modelo e projeção da câmera, normalmente somente o que está dentro do frustum é renderizado) e planos. Além de incluir também a definição de leitura e escrita de imagens.

Finalmente, o módulo WindowControl é responsável por controlar o funcionamento da janela fazendo a gerência de eventos e atualização da imagem exibida na tela. Possui um sub-bloco importante, o "GL"(GL é o nome do sub-bloco que trabalha com as funções da API OpenGL), que é responsável pela funcionalidade genérica de manipulação de áreas de imagens na memória principal, como cópia desta para o *framebuffer* ou vice-versa.

6 Resultados

Os testes do raytracer foram realizados em dois processadores de propósito geral (Pentium III 800Mhz e Pentium IV 2.4Ghz) e em dois processadores gráficos programáveis (GeForce5200 e GeForce6600). Onde se utilizou a média aritmética de vinte medições de cada um dos testes que são explicados a seguir.

Cada etapa dos testes foi nomeada de acordo com a tecla de atalho (tecla que é associada a uma determinada função no ambiente gráfico) que o mesmo é representado no programa. Na figura 6 , podem ser vistos os seguintes testes:

- F5 - Teste de rasterização no eixo X;
- F6 - Teste de rasterização no eixo Y;
- F7 - Teste de Geração de raios;
- F8 - interseção de um triângulo;
- F9 - Cena com 22 triângulos e uma luz.

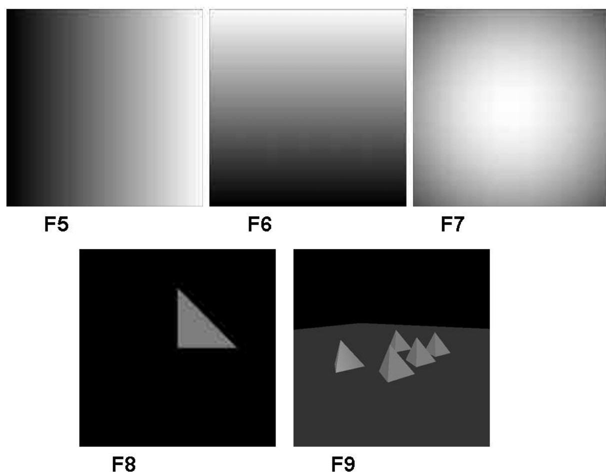

Figura 6. Testes realizados nos raytracers. (F5 e F6) Rasterização; (F7) Geração de raios; (F8) intercessão com um triângulo; (F9) renderização com iluminação

Os testes F5 e F6 são os mais simples e que ainda se assemelham mais à essência da GPU. Já os testes F7, F8 e F9 são respectivamente, de forma gradativa, níveis de implementações da técnica de *raytracing*.

De acordo com esta classificação foram obtidos os tempos de resposta mostrados na figura 7 .

Como os testes F5 e F6 são puramente de rasterização, então os mesmos equivalem à rasterizações comuns onde o processador gráfico é otimizado para este tipo de operação. Estes testes não podem representar uma base significativa para a realização do raytracer, apesar de importantes para a geração da imagem.

No teste F7, o que envolve a geração de raios, já se tem uma parte fundamental do *raytracing*. É neste estágio que a partir da configuração da câmera se geram os vetores da tela de projeção para que a mesma possa ser utilizada na geração dos raios. Este teste, sendo uma parte do raytracer,

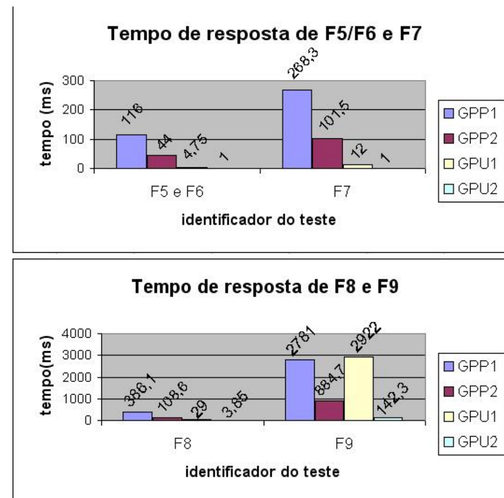

Figura 7. Gráficos dos tempos de respostas obtidos a partir dos testes (F5 até F9) sobre os processadores: (GPP1) - Pentium III 800Mhz; (GPP2) - Pentium IV 2.4 Ghz; (GPU1) - GeForce5200; (GPU2) - GeForce6600.

apresentou um *speedup* de 2.6 vezes no GPP2, de 22.3 vezes na GPU1 e de 268.3 vezes na GPU2 em relação ao GPP1.

O teste F8 é um dos mais importantes, pois o mesmo possui a geração de raios (o passo equivalente ao teste F7) e também o cálculo de interseção de raios com um triângulo. Esta interseção pode ser vista como o modo pelo qual o raytracer entende o modelo a ser renderizado. Mesmo sendo uma operação um pouco mais complexa (interseção de raio com triângulo) teve um *speedup* de 3.5 vezes no GPP2, de 13.3 vezes na GPU1 e de 100.2 vezes na GPU2 em relação ao GPP1.

O último teste, o F9, apresentou um resultado não esperado na GPU1. O speedup da GPU1 foi de 0.95 vezes em relação ao GPP1, o que quer dizer que houve uma perda de desempenho neste caso. Mas não ocorreu o mesmo com a GPU2, que apresentou um speedup de 19.5 vezes enquanto no GPP2 o speedup foi de 3.1 vezes em relação ao GPP1.

A partir destes gráficos é possível notar a diferênça do comportamento dos dois tipos de processadores gráficos, onde a GPU1 apresentou uma limitação na execução do raytracer ao se ativar a iluminação na cena.

7 Conclusão

A utilização de processadores gráficos programáveis para otimizar o processamento do raytracing é uma abordagem recente, visto que existem poucas referências internacionais e nenhuma brasileiras foi encontrada. E mesmo a

linguagem utilizada neste trabalho (GLSL) foi lançada oficialmente em agosto de 2004.

De acordo com os tempos de resposta mostrados na figura 7 podemos notar que em todos os testes com exceção do F9, as GPUs obtiveram um desempenho superior a todos os demais processadores de propósito geral envolvidos. No teste F9, a GeForce5200 pode ter apresentado um tempo de resposta muito grande em relação à GeForce6600 devido às restrições arquiteturais presentes na GeForce5200, que por exemplo não possui desvios condicionais no estágio de processamento de fragmentos.

Uma outra técnica para otimizar o raytracing, apresentada por Câmara [2], implementação baseado em máquinas multiprocessadas, obteve no máximo um speedup de 3.5 vezes, sendo que nos testes apresentados neste artigo, o speedup da GeForce6600 em relação ao PentiumIV foi de 6.2 vezes.

Este artigo apresenta de forma simplificada como funciona o processamento do *pipeline* de renderização presente na API OpenGL 2.0 bem como é o funcionamento de seus estágios programáveis. Logo após, apesenta a comparação de desempenho dos processadores gráficos programáveis GeForce5200 e Geforce6600 com os processadores de propósito geral Pentium IV 2.4Ghz e Pentium III 800Mhz.

Este trabalho foi a proposta e implementação de um raytracer que tem por objetivo utilizar mais eficientemente os recursos disponíveis em processadores gráficos programáveis atualmente bastante comuns.

Deste modo, a principal contribuição deste trabalho é a apresentação de uma implementação de dois *raytracers* onde uma utiliza os recursos dos processadores gráficos programáveis e a outra utiliza os recursos do processador de propósito geral.

Os processadores gráficos programáveis podem ser utilizados para otimizar o processamento de imagens e até mesmo outras técnicas de síntese, como no caso do Scheidegger [12] que utiliza da GPU para otimizar uma técnica de simulação de fluídos.

Entre os possíveis trabalhos futuros, podem-se destacar:

- Efetuar mais testes que envolvam a complexidade da cena renderizada, tanto em relação a resolução da imagem quanto em relação à quantidade de triângulos presentes no modelo;

- Implementar complementos ao algoritmo de raytracer de modo a permitir a renderização de reflexões, sombras e texturas;

- Utilizar diversas GPUs para implementar uma versão paralela do raytracer de modo a utilizar o paralelismo;

- Implementar alguma técnica de suavização da cena baseada em superamostragem;

- Implementar e testar este raytracer em outras GPUs do mesmo e de outros fabricantes.

Referências

[1] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[2] D. Câmara and A. L. P. Guedes. Divisão dinâmica em anel do cálculo do "ray tracing". In *X Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens*, 1997.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Pratice, 2nd ed. in C*. Addison-Wesley, 1996.

[4] G. M. Fortuny and M. McCool. Unified stream processing raytracer. In *GP2 - ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster*, 2004.

[5] A. S. Glassner. *An Introduction to Ray Tracing*. Academia Press Inc. SanDiego, 1990.

[6] J. Kessenich, D. Baldwin, and R. Rost. The opengl® shading language. 3DLabs, 2004.

[7] W. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a c-like language. pages 896–907. ACM Transactions on Graphics, 2003.

[8] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. volume 21, pages 703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[9] P. Rademacher. Ray tracing: graphics for the masses. *Crossroads*, 3(4):3–7, 1997.

[10] R. Rost. Introduction to the opengl shading language. 3Dlabs, Real Time Shading Course, 2004.

[11] R. J. Rost. *OpenGL(R) Shading Language*. Addison-Wesley Professional, 2004.

[12] C. E. Scheidegger, J. L. D. Comba, and R. D. da Cunha. Navier-stokes on programmable graphics hardware using smac. In *Computer Graphics and Image Processing, XVII Brazilian Symposium on (SIBGRAPI'04)*, pages 300–307, 2002.