4
Implementação

Neste capítulo serão explicados detalhes da implementação do pré-processador e do visualizador.

Por estarmos lidando com modelos compostos por vários milhões de triângulos, mostrou-se importante que o algoritmo de geração de voxels fosse desenvolvido de forma eficiente, apesar dessa ser uma etapa de pré-processamento. Uma implementação descuidada poderia elevar o tempo de pré-processamento de algumas horas para alguns dias. A operação mais freqüente realizada durante essa etapa é a amostragem de pontos usando o algoritmo de traçado de raios, o que criou a necessidade de uma estrutura espacial que permitisse que o tempo gasto na realização dos testes de interseção dos raios traçados com a geometria não fosse diretamente proporcional à complexidade da cena.

Foram implementadas duas versões dessa estrutura. Na primeira, usou-se uma *octree* semelhante à usada na geração da hierarquia, mas com mais níveis, de forma que as folhas da hierarquia fossem formadas por uma quantidade pequena de triângulos. Porém, essa implementação simples não apresentou desempenho satisfatório.

Na segunda implementação foi usado o motor de simulação física PhysX (2005), que tem recebido bastante atenção dos desenvolvedores de jogos e aplicações interativas por conseguir acelerar a simulação física usando hardware dedicado. Essa biblioteca, como qualquer motor físico, exige que seja criada uma estrutura de indexação para organizar os objetos da cena e permitir que testes de colisão entre objetos sejam realizados de forma eficiente. O algoritmo de pré-processamento implementado nesta dissertação tira vantagem dessa estrutura usando o suporte a traçado de raios oferecido pela biblioteca, que é mais eficiente que a primeira estrutura simples implementada.

Um problema em se usar o PhysX, uma biblioteca voltada para o mundo dos jogos, para trabalhar com modelos massivos é que ela não é planejada para lidar nem com tantos objetos nem com objetos tão complexos compostos por

muitos triângulos. Nenhuma dessas limitações está descrita no manual da biblioteca, e foram descobertas durante o desenvolvimento do pré-processador. Essa característica teve que ser contornada através da criação de um novo grafo de cena antes dos dados serem fornecidos ao motor do PhysX.

A biblioteca impõe que os objetos usados com ela tenham no máximo 60.000 triângulos. O tratamento dado aos objetos muito tesselados é simples, bastando quebrá-los em 2 ou mais objetos menores de forma que cada parte tenha uma quantidade aceitável de faces.

Outra limitação é de que cada "cena" do PhysX contenha no máximo 32.000 objetos. Uma solução simples consistiu em quebrar o modelo todo em diversas cenas com a quantidade de objetos suportados. Para evitar que o tempo gasto no teste de interseção dos raios comece a aumentar de forma proporcional ao número de cenas e, consequentemente, ao tamanho do modelo, podemos particionar o modelo em pequenos pedaços de forma que cada cena do PhysX seja criada apenas com objetos de uma região limitada do modelo. Assim, podemos fazer um teste de interseção do raio com a caixa envolvente de cada uma dessas regiões, e pedir que o motor físico trabalhe apenas com as regiões em que houve interseção com a sua caixa envolvente.

As operações usadas com mais freqüência no pré-processador foram desenvolvidas de forma a usar múltiplos processadores caso esses estejam disponíveis. Esse suporte foi dado usando a API OpenMP 2.0 criada por Dagum & Menon (1998), que hoje é suportada em diversos compiladores e linguagens. Essa API permite executar em paralelo códigos que possam ser estruturados em laços (*loops*). Seu uso é simples e utiliza diretivas *#pragma* como *#pragma omp parallel for* para realizar a execução de laços em paralelo e *#pragma omp critical* para obrigar que trechos do laço sejam executados em apenas uma *thread* por vez. Por utilizar diretivas *#pragma*, o mesmo código paralelizado ainda funciona, sem precisar de alterações, em compiladores mais antigos. As diretivas também podem ser facilmente desabilitadas para que o código funcione em apenas um processador. Um código típico usando OpenMP é escrito da seguinte forma:

```
função gerarVoxelsDeUmNó( Nó nó )
   // Esse for será executado em paralelo, com uma thread
   // para cada processador disponível na máquina
   #pragma omp parallel for
   para i = 1 até totalDeRaios faça
      raio = calcularRaioAleatório();

      // Calcular a interseção do raio com a cena
      // é a operação mais cara do pré-processamento
      // e pode ser executada concorrentemente
      // em várias threads[1].
      inter = calcularInterseção( nó, raio )

      // Essa é uma região crítica, que altera dados.
      // e consequentemente tem que ser executada em
      // apenas um processador por vez
      #pragma omp critical
         salvarResultado( voxels, raio, inter )
```

O visualizador foi implementado usando a biblioteca de grafos de cena *OpenSceneGraph* (Burns & Osfield., 2004). Essa biblioteca nos forneceu as classes necessárias para criarmos a hierarquia da cena e percorrê-la para gerar a renderização com descarte dos objetos fora da pirâmide de visão. O comportamento desejado do visualizador foi obtido através da extensão das classes existentes no grafo de cena, sendo que duas delas merecem destaque.

A primeira classe estendida tem a função de renderizar os nós de voxels, sendo responsável por carregar o *shader* apropriado para cada nó e enviar para a placa os vetores de parâmetros que contêm os dados que são acessados no *shader* para renderizá-lo. Essa classe também é responsável por verificar a

[1] Na realidade, por uma limitação da versão atual do PhysX este passo não pode ser paralelizado. O paralelismo foi, entretanto, utilizado em diversos outros pontos do programa.

disponibilidade de memória em vídeo para a criação de um *VertexBufferObject*, que permite a reutilização dos dados enviados à placa nos quadros seguintes.

A segunda classe reimplementada é responsável por redirecionar o percurso da hierarquia de cena de forma a escolher os níveis de detalhe apropriados em cada momento da visualização. Uma instância dessa classe é criada e inserida no grafo em cada posição em que for necessário decidir sobre usar a representação simplificada de voxels existente naquele nível ou continuar percorrendo os nós inferiores da hierarquia para usar representações de maior resolução. Essa decisão é baseada em parâmetros como a distância da câmera àquele nó, a resolução da superfície de visualização, a abertura da câmera e dados como tamanho e posição da representação de voxels disponível. Essa classe também é responsável por enviar os pedidos de carregamento de dados ao carregador que está rodando em segundo plano.