Trabalho Prático

Relatório

Marcos Teixeira *Diego Mendes*

Este relatório descreve os detalhes de implementação do trabalho prático da disciplina de Computação Gráfica do ano de 2015. O trabalho prático consiste na implementação de um traçador de raios simples com superamostragem adaptativa para tratamento de aliasing, na linguagem C++. O programa foi implementado sobre o Sistema Operacional Windows, utilizando mecanismos de Programação Orientada a Objetos. Foi utilizado o reaproveitamento do código fonte disponível no ead para facilitar o desenvolvimento. Nas seções abaixo serão apresentadas as implementações efetuadas.

1 . Leitor de Cena

Para tratar o XML que descreve a cena, foram utilizadas as funções de manipulação disponibilizadas pelo parser para C++ chamado **pugixml** (http://pugixml.org). Para processar o XML foram criados os arquivos **Parser.cpp** e **Parser.h** que contém métods específicos para leitura das especificações da cena e câmera.

A classe Parser mantém dois atributos privados: **xml** (xml_document) e **root**(xml_node). No construtor temos ***Parser(const char * filename)***, onde é passado o arquivo descrevendo a cena como entrada como parâmetro, o atributo **xml** carrega os dados do XML e o atributo **root** recebe o primeiro filho da árvore XML do arquivo. No método **parseImage(int & height, int & width)** são inicializados a altura e a largura da imagem (caso não presentes no XML toma-se o valor default *1024x728).* Os métodos **parseCamera()** e **parseScene()** fazem a leitura dos atributos da câmera e da cena, respectivamente. O método **parseScene()** tem um loop no final que realiza a obtenção dos dados da cena (malhas e luz), sendo que em cada iteração é necessário lançar um tratamento no caso de ser uma malha de triângulo dada por um arquivo, uma esfera, um cone, uma caixa, um cilíndro ou uma fonte de luz. Para isso, a cada iteração do loop verifica-se a necessidade de usar algum dos seguintes métodos: *parseMesh*, *parseSphere, parseCone, parseBox, parseCylinder* e *parseLight.*

Os métodos **parseMesh(xml_node_iterator)**, **parseSphere(xml_node_iterator)**, **parseCone(xml_node_iterator)**, **parseBox(xml_node_iterator)** e **parseCylinder(xml_node_iterator)** fazem a leitura dos atributos de uma malha de triângulos

dada através de um arquivo, uma esfera, um cone, uma caixa e um cilíndro, respectivamente. Cada um deles recebe um **xml_node_iterator** que é basicamente um iterador de nós da árvore do XML, e retornam um ponteiro para um **Actor**, que é adicionado a cena a cada iteração de leitura de cada malha. O método **parseMaterial(xml_node)** faz a leitura do material da malha que está sendo definida, recebendo um **xml_node** e retornando um ponteiro para um objeto do tipo **Material,** com os dados do material da malha. Caso não exista um material definido no XML para a malha, define-se um material default.

Finalmente, o método **parseLight(xml_node_iterator)** faz a leitura das luzes da cena. O método toma um **xml_node_iterator** como entrada e gera como saída um ponteiro para um objeto do tipo **Light** com os dados de uma luz da cena. O tipo de luz **spot** não é tratado neste programa. Foram adotados no **Parser.cpp** todos os valores default para os atributos opcionais apresentados na descrição do trabalho.

2 . Traçador de Raios simples

A classe responsável pelo algoritmo de traçado de raios, tanto o simples quanto o com superamostragem, é a **RayTracer.** Na classe original foram feitas as seguintes adições

- Criação da estrutura **Coordinate** que mantém as coordenadas x e y de um ponto no VRC e também a cor desse ponto, caso já tenha sido descoberta. Esta estrutura foi criada para ajudar na fase de superamostragem adaptativa
- Criação dos métodos: **trace(const Ray&, uint, REAL)**, **shade(const Ray&, uint, REAL)**, **subDivision(int, int, REAL, int)**, **checkVisitedPoints(Color&, double, double)**, **void adaptativeScan(Image&)** e outros métodos alternativos para manipulações básicas no buffer responsável por guardar os pontos já visitados na superamostragem adaptativa.
- Setar o peso mínimo do para 0.01, o nível máximo de recursão para 6 e criar uma constante chamada **ADAPT_DISTANCE** igual a 0.06 para uso na superamostragem adaptativa como distância de adaptabilidade.

O método **trace** recebe como parâmetro o endereço de um raio (Ray), o nível de recursão (level) e o peso associado a esse raio (weight). O procedimento do método é bem simples, ele faz uma verificação se o peso mínimo é maior ou igual a weight ou se level é maior que o nível máximo de recursão pre-estabelecido. Se for o caso, retorna a do pixel como preta, caso contrário chama o método **shade** para tonalizar o pixel, passando como parâmetros para o método o raio, o nível de recursão e o peso do raio. No método **shade** verifica-se se o raio intersecta algum objeto da cena e em caso positivo traçam-se raios de luz (representado na variável **shadowR**) em direção a cada funte de luz da cena partindo do ponto

de interceção até a origem de cada fonte de luz. Se **shadowR** não intercecta nenhum outro ator em cena, então a fonte de luz ilumina diretamente o ponto e a cor no ponto de interseção é definido de acordo com o modelo local de iluminação.

Caso o material do ator no ponto de interseção for reflexivo, ou seja, seja Or o coeficiente de reflexidade do material no ponto P de interseção, que no nosso caso é a cor de reflexão especular do ator no ponto P, se Or for diferente que (0,0,0) então o material é considerado polido suficiente para gerar o raio de reflexão, Assim, utilizamos o procedimento explicado capítulo **4** seção **4.3** para definir o valor da cor no ponto P. Isso é feito chamando recursivamente o método **trace** com o incremento de mais um no nível do raio e o atualizando o peso multiplicando o valor do peso do "pai" pelo valor da maior componente de Or.

3 . Traçador de Raios com superamostragem adaptativa

Para a superamostragem adaptativa, foi criado um novo método chamado **adaptativeScan** muito parecido com o **scan** mas que ao invés de apenas disparar um raio no meio de cada pixel, nós chamamos para cada pixel (i, j) o método **subDivision(int, int, REAL, int).** Este método recebe as coordenadas (i,j) do pixel, um valor **sub** do tipo **REAL** que representa o grau de deslocamento do pixel em cada subdivisão podendo assumir os valores 1.0, 0.5 e 0.25, e ultimamente um **level** do tipo **int** que representa o nível de subdivisão do píxel. O nível máximo de subdivisão é 3, gerando no máximo 25 pontos para serem traçados.

O método **subdivision** começa verificando se o nível é menor ou igual a 3 que é o nível máximo. Em caso positivo nós calculamos os valores dos 4 pontos do pixel ou subixel, caso eles já não tenham sido calculados. Calculamos a média dos 4 pontos e verificamos se os valores não se afastam da média, usando a constante **ADAPT_DISTANCE** como threshold. Caso os valores não se afastem da média, nós retornamos a cor média calculada, caso contrário chamamos recursivamente 4 vezes o método **subDivision** para dividir os subpíxeis, recalcular os valores, verficar novamente se os valores dos pontos se afastam ou não da média e retornar como resultado a soma dos valores de cor obtido em cada subpixel da recursão dividido por 4 (média dos valores dos subpixeis).

Como já foi dito, só é calculada a cor em um determinado ponto do pixel se este já não tiver sido calculado anteriormente. Para isso, nós criamos o método **checkVisitedPoints(Color&, double, double)** que recebe o endereço de um atributo do tipo **Color** e as coordenadas (i, j) do ponto que se deseja checar se já foi visitado.O método basicamente devolve a cor já calculada se ela já estiver no atributo **visited** e caso contrário, aplica o traçado de raios para determinar a cor no ponto, adiciona a cor e as coordenadas em

visited e retorna a cor calculada. A variável **visited** é um buffer do mesmo tamanho **W** (width) já definido que armazena os pontos já visitados (coordenadas (i, j) e cor do ponto) em blocos com 25 posições que representam o total de pontos máximo que um pixel pode gerar (no caso do últmo nível de subdivisão).O que é método **checkVisitedPoints** faz então é um mapeamento das coordenadas (i, j) recebidas por parâmetro para uma posição do buffer **visited**, tendo custo linear para definir se um ponto já tem sua cor calculada ou não. O atributo **visited** é atualizado a cada linha percorrida no método **adaptativeScan**, sendo que existe um aproveitamento das cores dos pontos na fronteira inferior de cada bloco para o caso de ser necessário para a próxima linha. Este aproveitamento se dá através do atributo **border** que é um array de **Coordinate** de tamanho **W x 5** que mantém a cada iteração, após a varredura do *for* que processa a cor do pixel na posição *i*, a última linha do buffer **visited.** No início de cada iteração de linha(tirando a primeira iteração) nós inicializamos um novo buffer **visited** e então atualizamos sua primeira linha com os valores do array **border**, assim aproveitando os pontos da linha acima que já foram calculados.

4 . Precisão Numérica

Um dos problemas do Traçador de Raios é a precisão numérica, que pode ser observada como um padrão de pixeis na sombra da imagem gerada(nuvem de pontos pretos). Isso ocorre porque em decorrência do arredondamento o ponto de interceção de um raio/objeto fica para dentro do objeto, fazendo com que o próprio objeto obstrua a luz direta fazendo com que o ponto fique na sombra.

Para tratar este problema nós avançamos o ponto de interseção do raio de pixel com o ator uma distancia δ na direção da normal. O valor de δadotado foi *0.01* obtido empíricamente.

5 . Execução

Para executar o programa basta abrir o projeto chamado **rt** no Visual Studio e clicar em **F5** . Abrirá uma tela com a cena renderizada pela OpenGL e os atores com cor preta, pois ainda não foi realizado o processo de traçado de raios. As opções são:

- **‘ o ’** : Renderizar usando OpenGL
- **‘ t ’** : Executar o traçado de raios simples na cena
- **‘ i ’** : Executar o traçado de raios com superamostragem adaptativa

O arquivo XML com a descrição da cena é passado em ***Debug->rt Properties...->Configuration Properties->Debugging->Command Arguments***. Foram elaborados 5 cenas de teste que podem ser usadas e que estão na mesma pasta do código fonte.

6 . Dificuldades

A maior dificuldade foi em relação a modelagem da estratégia de superamostragem adaptativa, pois existe a necessidade de salvar os pontos visitados em alguma estrutura e ainda por cima, conseguir obter os pontos visitados de maneira rápida nesta estrutura senão comprometeria drasticamente o desempenho do programa.

Outra dificuldade foi em tratar o problema de mais de uma instância de malhas do mesmo tipo. A ideia é que se uma malha de um determinado tipo ***M*** já tivesse sido instanciada pelo parser, então se aparecesse outra malha do tipo ***M*** ela não deveria ser instanciada novamente mas , ao invés disso, criar um ponteiro para a instancia da primeira malha. O problema veio ao tentar usar a classe *ModelInstance* para criar um ponteiro para uma instância de uma malha, pois o construtor não aceitava o tipo *Primitive* como parâmetro. Tentamos fazer de outras formas e passando até mesmo outros tipos de parâmetros mas não funcionou. Acabamos por não tratar esse caso.

Uma dificuldade menos importante, mas que tomou alguns dias, foi a execução do código fonte do *ead* no Visual Studio, pois aparecia sempre um problema relacionado a pilha e parava de funcionar. Após desinstalar a máquina virtual do Windows, formatar a máquina e colocar só Windows e usando Visual Studio 2013 funcionou direito.