Implementação em OpenCL™ e Análise Comparativa de Três Algoritmos de Interseção Raio-Triângulo em FPGA

Jorgeluis Andrade Guerra

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2020

Jorgeluis Andrade Guerra

Implementação em OpenCL™ e Análise Comparativa de Três Algoritmos de Interseção Raio-Triângulo em FPGA

Monografia apresentada ao curso Engenharia de Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em
Engenharia de Computação

Orientador: Christian Azambuja Pagot
Coorientador: Alisson Vasconcelos de Brito

Agosto de 2020

Catalogação na publicação
Seção de Catalogação e Classificação

G934i Guerra, Jorgeluis Andrade.
Implementação em openCL e análise comparativa de três algoritmos de interseção raio-triângulo em FPGA / Jorgeluis Andrade Guerra. - João Pessoa, 2020.
69 f. : il.

Orientação: Christian Azambuja Pagot.
Coorientação: Alisson Vasconcelos de Brito.
TCC (Graduação) - UFPB/CI.

1. Algotitmo. 2. OpenCL. 3. Interseção raio-triângulo. 4. Circuito integrado. I. Pagot, Christian Azambuja. II. Brito, Alisson Vasconcelos de. III. Título.

UFPB/CI CDU 004.021

Elaborado por ROGERIO FERREIRA MARQUES - CRB-15/690

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

ATA DE DEFESA PÚBLICA DO TRABALHO DE CONCLUSÃO DE CURSO

Aos **13** dias do mês de **Agosto** de **2020**, às **10 horas** e **00 minutos**, em sessão pública *online* por videoconferência, na presença da banca examinadora presidida pelo professor orientador **Dr. Christian Azambuja Pagot** e pelos professores **Dr. Elmar Uwe Kurt Melcher**, **Dr. Gustavo Charles Peixoto de Oliveira** e **Dr. Andrei de Araújo Formiga**, o aluno **Jorgeluis Andrade Guerra** apresentou o trabalho de conclusão de curso intitulado **Implementação em OpenCL e Análise Comparativa de Três Algoritmos de Interseção Raio-Triângulo em FPGA** como requisito curricular indispensável para a integralização do **Curso de Bacharelado em Engenharia de Computação**.

Após a exposição oral, o candidato foi arguido pelos componentes da banca que reuniram-se reservadamente e decidiram **APROVAR** a monografia, com nota **10.0**. Divulgando o resultado formalmente ao aluno, e demais presentes, eu, na condição de Presidente da Banca, lavrei a presente ata que será assinada por mim, pelos demais examinadores e pelo aluno.

Prof. Dr. Christian Azambuja Pagot

Prof. Dr. Elmar Uwe Kurt Melcher

Prof. Dr. Gustavo Charles Peixoto de Oliveira

Prof. Dr. Andrei de Araújo Formiga

Jorgeluis Andrade Guerra

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

"Compiling for FPGA.
This process may take a long time, please be patient."
AOC message

RESUMO

Em técnicas de renderização baseada no traçado de raios, cerca de 85% do custo computacional se concentra no cálculo de interseções raio-triângulo. Este trabalho consiste na implementação em FPGA e na análise comparativa de três algoritmos que realizam esse cálculo: Möller-Trumbore, Ingo Wald e Hanika. Para simplificar os circuitos obtidos, foram propostas implementações em ponto fixo dos algoritmos de Möller-Trumbore e Ingo Wald, originalmente propostos em ponto flutuante, realizando a comparação destes com o algoritmo de Hanika, cujo cálculo já é realizado em ponto fixo. O *framework* OpenCL™ foi utilizado para possibilitar e acelerar o desenvolvimento da implementação e síntese de alto nível dos algoritmos em *hardware*. Os resultados apresentados incluem: tempos de execução em FPGA, CPU e GPU; comparação entre imagens renderizadas em FPGA e as estatísticas da síntese do circuito.

Palavras-chave: OpenCL™ , FPGA, interseção raio-triângulo, ray tracing.

ABSTRACT

The calculation of ray-triangle intersections represents nearly 85% of the computational cost of rendering methods based in ray tracing. This monograph introduces a hardware implementation and a comparative study of three algorithms to determine ray-triangle intersection for FPGA: Möller-Trumbore, Ingo Wald and Hanika algorithms. To reduce the size of the synthesized circuits, we proposed fixed point implementations of Möller-Trumbore and Ingo Wald algorithms, originally developed under floating point representation. We compare these implementations against Hanika's algorithm, which uses the fixed point representation by default. To accelerate the design of the digital hardware and synthesize the kernel code in high-level, we resorted to the OpenCL™ framework. The results reported comprise: execution times on FPGA, CPU and GPU; comparison between rendered images and the statistics of synthesized designs.

Key-words: OpenCL™ , FPGA, ray-triangle intersection, ray tracing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TERMOS E ABREVIATURAS

OpenCL™ – *Open Computing Language.*
SDK – *Software Development Kit.*
API – *Application Programming Interface.*
BSP – *Board Support Package.*
FPGA – *Field-Programmable Gate Array.*
ALM – *Adaptive Logic Module.*
ALUT – *Adaptive Look-Up Table.*
RAM – *Random-Access Memory.*
SDRAM – *Synchronous Dynamic Random-Access Memory.*
DSP – *Digital Signal Processor.*
M10K – Unidade de bloco de memória configurável de 10 kb presente em FPGA da Intel® FPGA.
HDL – *Hardware Description Language.*
RTL – *Register-Transfer Level*
E/S – Entrada e Saída.
I/O – *Input and Output.*
RGB – *Red Green Blue.*
PPM – *Portable Pixel Map* - formato de imagem do Netpbm.
Netpbm – Um *toolkit* para manipulação de imagens gráficas.
KD-Tree – *k-dimensional tree.*
BVH – *Bounding volume hierarchy.*
BIH – *Bounding interval hierarchy.*

Sumário

1 INTRODUÇÃO

Técnicas baseadas em *ray tracing*, com aplicações que vão desde a geração de imagens (PHARR; JAKOB; HUMPHREYS, 2017) a simulações de propagação do som (KROKSTAD; STROM; SøRSDAL, 1968) (TANG; MANOCHA, 2020), por exemplo, precisam verificar a interseção entre raios e triângulos. O número de testes de interseção que precisam ser avaliados é em geral muito alto, dominando os tempos de computação. Whitted (1980, p. 349) estimou que os testes de interseção raio-primitiva consomem de 75% a 95% do custo computacional do *ray tracing*.

Dado este fato, existem diversas técnicas que podem ser utilizadas para reduzir o tempo de renderização. Entre elas podemos citar:

- Redução do número de testes a serem feitos por meio do uso de estruturas de aceleração, como KD-Trees e BVH, por exemplo;
- Redução do custo de cada teste de interseção por meio de rotinas de teste eficientes;
- Avaliação dos testes em paralelo, visto que são independentes entre si.

Muitos *hardwares* paralelos têm sido utilizados na avaliação dos testes de interseção, tais como CPUs com múltiplos *cores*, GPUs e FPGAs. A vantagem das FPGAs em relação às outras alternativas reside no fato de serem dedicadas e de permitirem a implementação de projetos voltados para a diminuição do consumo de energia, características desejáveis, por exemplo, na construção de *rendering farms*. Por outro lado, FPGAs mais acessíveis apresentam em geral um número bastante limitado de elementos lógicos, o que pode se tornar um problema, em especial para a implementação de rotinas que demandam computações complexas e em ponto flutuante, por exemplo.

Este trabalho compara o algoritmo de Hanika (2007), baseado em ponto fixo, com a conversão para ponto fixo de outros dois algoritmos bastante conhecidos, idealizados originalmente para operarem em ponto flutuante, que são os algoritmos propostos por Möller e Trumbore (1997) e por Wald (2004). Estes dois algoritmos são bastante utilizados em *renderers* populares, como o Mitsuba (JAKOB, 2010), por exemplo.

FPGAs contêm conjuntos de funções lógicas básicas implementadas em *hardware* e são postas à disposição do projetista para combiná-las e usá-las com amplo grau de liberdade. É ainda possível agrupar diversas FPGAs em uma espécie de fazenda (ou *farm*, em inglês) para trabalharem em conjunto, cada uma processando um subconjunto do cálculo paralelizável a ser realizado, o que aumenta o apelo das FPGAs na construção de *rendering farms*.

Neste trabalho utilizamos um compilador OpenCL™ , conhecido como OpenCL™ C, capaz de sintetizar a partir de um código em alto nível, o circuito lógico a ser mapeado e

programado na FPGA.

Conforme Singh (2011), a síntese de alto nível aliada a um sistema heterogêneo, composto de CPU e FPGA, usando o padrão OpenCL™ pode nos dar uma vantagem significativa de *time-to-market* em comparação ao fluxo de desenvolvimento tradicional de *hardware* em FPGA, usando uma descrição de *hardware* em nível RTL.

Nas tradicionais metodologias de projeto para FPGA, muito esforço é feito para descrever o funcionamento do *hardware* a cada ciclo de *clock* de forma que seja implementado o algoritmo desejado, o que involve a criação de *datapaths*, máquinas de estado para controlar os *datapaths*, a conexão dos *IP cores* de baixo nível usando ferramentas a nível de sistema e a busca em alcançar os requisitos de *timing*, visto que as interfaces externas impõem restrições de *timing* que devem ser alcançadas.

Assim, o objetivo de um compilador OpenCL™ para FPGA que permita síntese de alto nível é realizar todas essas etapas, inerentes a um projeto de baixo nível, de forma automática e transparente para o projetista, permitindo que o esforço seja direcionado na definição dos algoritmos em vez de ser direcionado para esses detalhes e complicações do projeto de *hardware*. Em se tornando viável, a síntese de alto nível possibilitará que o fluxo de desenvolvimento de sistemas em *hardware* seja aproximado do fluxo de desenvolvimento de sistemas em *software*.

1.1 Objetivos

O objetivo geral deste trabalho é:

- Analisar a implementação de três métodos distintos de teste de interseção raio-triângulo baseados em ponto fixo usando FPGA.

Os objetivos específicos são:

- Experimentar a síntese de alto nível possibilitada pelo compilador OpenCL™ utilizado e verificar, dentro dos casos de teste realizados, quão eficaz tal síntese pode ser;
- Comparar imagens renderizadas usando cálculos em ponto flutuante e em ponto fixo;
- Analisar e comparar as estatísticas de área, velocidade e consumo de energia reportadas pela síntese.

2 FUNDAMENTAÇÃO TEÓRICA

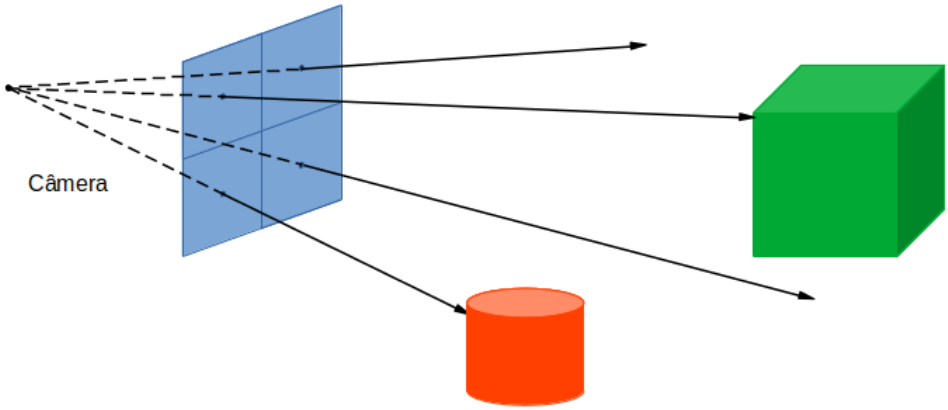
Para facilitar o entendimento dos capítulos posteriores, neste capítulo introduzimos uma ideia geral de como funciona o *ray casting*, tratamos de alguns conceitos de forma prática da aritmética de ponto fixo, descrevemos também as FPGAs e discutimos aspectos do padrão OpenCL™ relevantes para este projeto.

2.1 Visão Geral sobre o Funcionamento do Ray Casting

A técnica de *ray casting* gera renderizações bem simples, porém é a base para técnicas mais realistas como o *ray tracing* e o *path tracing*, por exemplo. Basicamente o que fazemos no *ray casting* é modelar matematicamente uma entidade que representa uma câmera com projeção perspectiva e da qual os raios serão traçados em direção à cena que contém os objetos a serem renderizados. E são calculados tantos raios quantos forem os *pixels* da imagem que se quiser obter.

Na figura a seguir, por exemplo, a imagem resultante teria 4 *pixels* (com resolução 2 x 2) e, portanto, seriam traçados 4 raios contra a cena representada. Cada raio é, então, testado para determinar se este possui interseção ou não com algum objeto da cena.

Figura 1 – Exemplo de uma cena com dois objetos e uma entidade câmera de onde os raios são disparados contra a cena.



2.2 Aritmética de Ponto Fixo

Operações matemáticas em ponto fixo diferem de simples operações com números inteiros pelo fato de que para manter a representação coerente é preciso realizar correções, nos operandos ou no resultado, dependendo da operação matemática realizada.

Nas operações de soma e subtração não há necessidade alguma de ajustes nos resultados. As operações de multiplicação e divisão, no entanto, sempre necessitam de

ajuste, como mostraremos a seguir.

Considere um número qualquer x_f representado em ponto flutuante, podemos representá-lo ainda em ponto flutuante usando o seguinte artifício matemático:

$$x_f = (x_f \cdot 2^m) \cdot 2^{-m} \tag{1}$$

Se retirarmos da equação o termo 2^{-m} e se realizarmos o arredondamento do termo restante $x_f \cdot 2^m$, o que obteremos será um número inteiro x_i que podemos chamar de representação em ponto fixo de x_f. Supondo uma função *int()* que retorna a parte inteira de um número flutuante, o raciocínio anterior equivale a dizer que:

$$x_i = int(x_f \cdot 2^m) \tag{2}$$

A variável m representa a largura de bits utilizada para a representação da parte fracionária nos números representados em ponto fixo.

Vejamos um exemplo com números em representação decimal:

$$\begin{aligned} x_f &= 2,543 \\ m &= 2 \\ x_i &= int(2,543 \cdot 10^2) = int(254,3) = 254 \end{aligned}$$

Façamos agora um outro exemplo: utilizando a representação em inteiros vamos tentar realizar uma multiplicação de dois números fracionários. Considere novamente dois números, $a_f = 2,543$ e $b_f = 1,727$, $m = 2$ e suponha uma função *float()* que retorna a representação em ponto flutuante de um número inteiro:

$$\begin{aligned} a_i &= int(2,543 \cdot 10^2) = int(254,3) = 254 \\ b_i &= int(1,727 \cdot 10^2) = int(172,7) = 172 \\ c_i &= a_i \cdot b_i = 254 \cdot 172 = 43688 \\ c_f &= float(43688) \cdot 10^{-2} = 436,88 \end{aligned}$$

Percebe-se então que houve um problema na multiplicação. Não obtivemos o valor aproximado de c_f que era esperado $a_f \cdot b_f = 2,543 \cdot 1,727 \approx 4,39$. Comparando os resultados percebemos que fazendo um ajuste no resultado $c_i = 43688$ obtido na multiplicação em ponto fixo, após a conversão para ponto flutuante, obteremos, então, o valor aproximado que era esperado:

$$\begin{aligned} c_i &= a_i \cdot b_i \cdot 10^{-2} = 254 \cdot 172 \cdot 10^{-2} = 43688 \cdot 10^{-2} = 436 \\ c_f &= float(436) \cdot 10^{-2} = 4,36 \end{aligned}$$

Com o ajuste feito obtivemos, então, o resultado coerente. Para a divisão, ajuste equivalente tem que ser realizado, para manter coerente a representação em ponto fixo.

O comportamento da representação em ponto fixo é equivalente na representação binária. E vale reforçar que, como também observado no resultado do exemplo anterior ($4,36 \approx 4,39$) esse processo de conversão acrescenta um erro de quantização ao valor convertido.

Neste tópico decidimos apresentar a representação em ponto fixo seguindo uma abordagem mais pragmática e simplificada. No entanto, existem na literatura referências que apresentam esta teoria com maior rigor formal, que corroboram os artifícios matemáticos aqui aplicados e que, se necessário, podem ser consultadas pelo leitor (LINZ; WANG, 2003; YATES, 2010).

Outra consideração importante a fazer é que há também problemas númericos em métodos de renderização baseada em traçados de raios quando realizada usando números em ponto flutuante: são os casos da quantização de números reais que se encontram distantes da origem e o conhecido problema de auto-interseção (HANIKA; KELLER, 2007, p. 3). Quando se usa a aritmética de ponto flutuante, artefatos de quantização surgem pelo fato de que os números representados não são equidistantes entre si, o que não acontece em ponto fixo, onde temos equidistância entre os números. Porém, em ponto fixo, pela falta de um expoente na representação (como temos em ponto flutuante) o intervalo representável de números fica reduzido. Assim, artefatos podem aparecer na renderização, decorrentes da limitação do intervalo de representação.

É, ainda, uma boa prática normalizar os números em ponto flutuante dentro do intervalo $[-1, 1]$ antes de converter para a representação em ponto fixo, sempre que possível. E para representarmos números binários fracionários em ponto fixo, multiplicamos os valores originais por 2^m, as computações são feitas usando aritmética inteira e, por fim, o resultado é convertido de volta para o intervalo anterior multiplicando-o por 2^{-m} (HANIKA; KELLER, 2007, p. 9).

2.3 Introdução às FPGAs

FPGAs são *chips* que podem conter diversas unidades básicas lógicas, de memória, de processamento de sinais e outras funções básicas úteis no projeto de circuitos digitais complexos (SINGH, 2011). A flexibilidade das FPGAs vem justamente do fato destas serem *hardwares* reprogramáveis: o circuito montado dentro do *chip* pode ser atualizado, bastando apenas um processo de reprogramação a partir de novos arquivos de programação gerados atráves da compilação e síntese da descrição de *hardware* implementada.

FPGAs podem ser usadas para acelerar processamentos pesados e reduzir gargalos de processos que podem ser paralelizados, principalmente aqueles que realizam cálculos iguais sobre um grande conjunto de dados.

A FPGA utilizada neste trabalho dispõe dos seguintes elementos de lógica programáveis:

- ALM: são as unidades básicas de lógicas que implementam funções simples, podem ser agrupadas para realizar funções mais complexas e dispõem de registradores locais;
- RAM *block*: são pequenas unidades de memória integradas na FPGA, configuráveis e que podem ser agrupadas para implementar memórias maiores;
- DSP *block*: são blocos de lógica básica, semelhantes aos ALM, porém específicos e otimizados para funções bastante comuns em processamento digital de sinais.

2.4 O Padrão OpenCL™

OpenCL™ é um padrão inicialmente proposto pela Apple em 2008 ao grupo Khronos, que possibilita a implementação de algoritmos paralelos que podem ser portados de uma plataforma para outra com pouca ou nenhuma necessidade de recodificação. A linguagem é baseada na linguagem de programação C e possui extensões que possibilitam diferentes tipos de especificação de paralelismo (SCARPINO, 2011, p. 4).

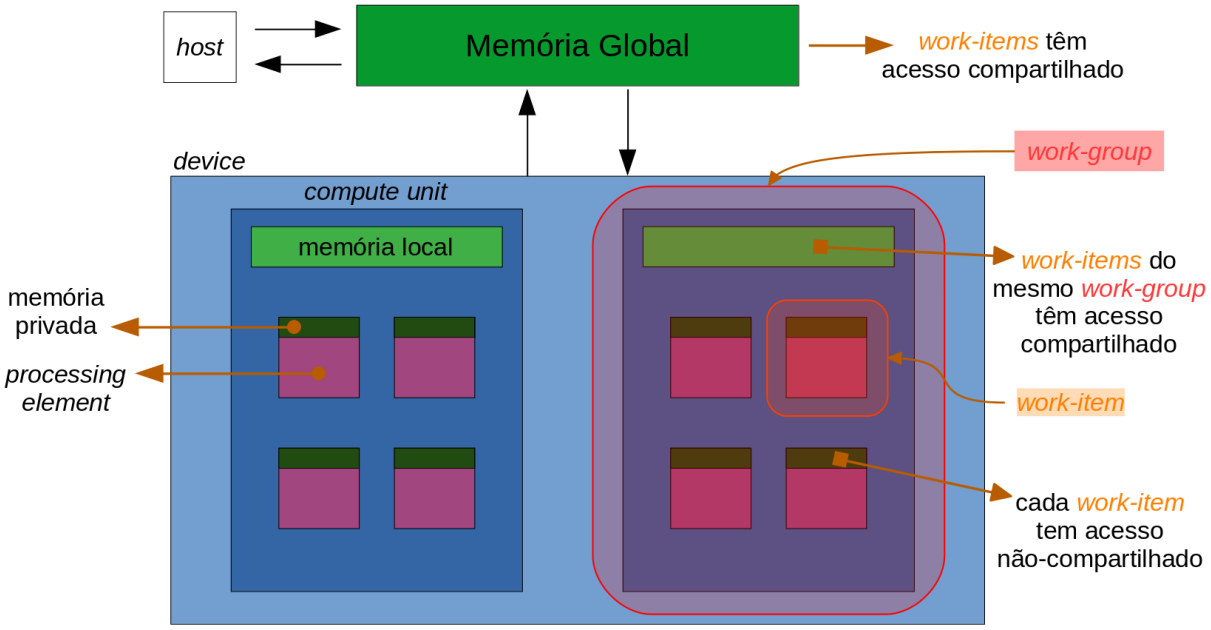
O padrão OpenCL™ possibilita que o programador explicitamente especifique e controle o paralelismo codificando em uma linguagem de alto nível. Espera-se, portanto, que códigos escritos seguindo o padrão OpenCL™ se aproximem da característica altamente paralela das FPGAs mais naturalmente do que os programas sequenciais escritos puramente em C.

Uma aplicação OpenCL™ é sempre composta por duas partes: o *host* e o *kernel*. O programa *host* é um *software* escrito em C/C++ padrão executado em CPU. Utilizando as funções da API OpenCL™ , o *host* pode selecionar um *device* para que este execute uma tarefa chamada *kernel*. O *device* pode ser escolhido entre diversos hardwares disponíveis e compatíveis com o padrão OpenCL™ , como CPU, GPU ou FPGA, por exemplo. O *kernel* é uma função escrita em OpenCL™ C, cujo propósito principal em geral é explorar o paralelismo do *device*. Geralmente o código do *kernel* contém parte da aplicação que é mais computacionalmente custosa e que pode ser acelerada através de execuções em paralelo. E a linguagem OpenCL™ C basicamente é o C padrão acrescentado de sintaxes que permitem especificação de paralelismo e hierarquia de memória.

Ao ser instanciado em uma unidade de processamento, que o padrão OpenCL™ designa como *processing element*, o *kernel* passa a ser chamado de *work-item* e é identificado através de um *global ID*. Os *hardwares* de aceleração normalmente contêm diversos *processing elements* agrupados em um ou mais *compute units*. Os *work-items*, por sua vez, são agrupados em um ou mais *work-groups*. Um *work-group* é sempre executado em um *compute unit*.

O padrão OpenCL™ (MUNSHI, 2009) ainda define três hierarquias de memória: global, local e privada. Todos os *work-items* têm acesso compartilhado à memória global, mas apenas os *work-items* de um mesmo *work-group* têm acesso compartilhado à memória local do *compute unit* no qual estão sendo executados. E, por fim, a memória privada é a memória de acesso não-compartilhado, exclusiva de um *work-item* e pertencente ao *processing element*.

Figura 2 – Arquitetura OpenCL



Na FPGA, em geral a memória global é mapeada em uma memória externa, como uma SDRAM, por exemplo; a memória local é mapeada em blocos de RAM; e a memória privada pode ser mapeada em registradores dos ALM da FPGA ou em blocos de RAM, mas preferencialmente em registradores. Mais detalhes acerca da implementação física da hierarquia de memórias podem ser consultados nos manuais de programação do SDK OpenCL™ (INTEL. . . , 2018a, 2018b).

Diferente de CPUs e GPUs, onde *work-items* podem ser executadas em diferentes núcleos, nas FPGA o *kernel* é sintetizado em um circuito dedicado equivalente a um *compute unit*. O *compute unit* pode ainda ser replicado várias vezes (até o limite da quantidade de lógica disponível na FPGA) para prover ainda mais paralelismo. Segundo Singh (2011) o processo de síntese é iniciado pela tradução do código OpenCL™ C do *kernel* em *hardware* criando circuitos que implementam cada operação presente. Em seguida, os circuitos são roteados de acordo com o fluxo de dados do *kernel*, gerando ao final um *hardware* paralelo com a mesma funcionalidade do código OpenCL™ C.

Além do *hardware* do *kernel*, o compilador cria *hardware* para interface com memórias internas e externas. As unidades de carregamento e armazenamento de cada *compute*

unit são conectadas à memória externa através de uma estrutura global de interconexão. Da mesma forma, acessos a memórias locais são conectados através de uma estrutura especializada de interconexão aos elementos de memória RAM presentes na FPGA.

Com isso, encerramos o capítulo de fundamentação teórica. Acreditamos que o que foi sucintamente mostrado aqui seja suficiente para o entendimento dos capítulos posteriores. Todo caso, lembramos sempre que informações mais detalhadas sobre cada assunto tratado nesse capítulo podem ser encontradas nas referências desta monografia.

3 ESTADO DA ARTE

Neste capítulo iniciamos com a descrição do algoritmo de Badouel, que pode ser considerado como um algoritmo geral para a determinação da interseção raio-triângulo. Em seguida apresentamos as particularidades dos algoritmos de Möller-Trumbore, Ingo Wald e Hanika em relação ao algoritmo de Badouel.

Para facilitar o entendimento do leitor, procuramos estabelecer e usar uma notação matemática unificada em todas as descrições dos algoritmos, em que letras minúsculas (u e v, por exemplo) representam valores escalares e letras maiúsculas e em negrito representam pontos ou vetores no espaço ($\mathbf{P}$ e $\mathbf{N}$, por exemplo).

Vale ainda ressaltar que os trabalhos de Möller-Trumbore e Ingo Wald foram originalmente desenvolvidos para serem executados em CPU, sendo apenas o trabalho de Hanika concebido originalmente para síntese em *hardware*.

3.1 O Algoritmo de Badouel

Em renderizações baseadas no traçado de raios, como o *ray casting*, o *ray tracing* e o *path tracing*, as imagens são produzidas a partir de modelos que são constituídos por polígonos básicos, chamados de primitivas. Neste trabalho escolhemos os triângulos como primitiva, por questão de simplicidade e visto que triângulos são formas bastante convenientes e ainda bastante utilizados para a renderização de superfícies 3D (MARTENS, 2011).

O algoritmo de Badouel (1995) é tido como o algoritmo geral para o cálculo de interseção raio-triângulo e pode ser dividido em duas etapas básicas: o cálculo da interseção com o plano que contém o triângulo, seguido pela determinação da interseção com o próprio triângulo, conforme mostramos a seguir.

3.1.1 Etapa 1: Interseção com o Plano

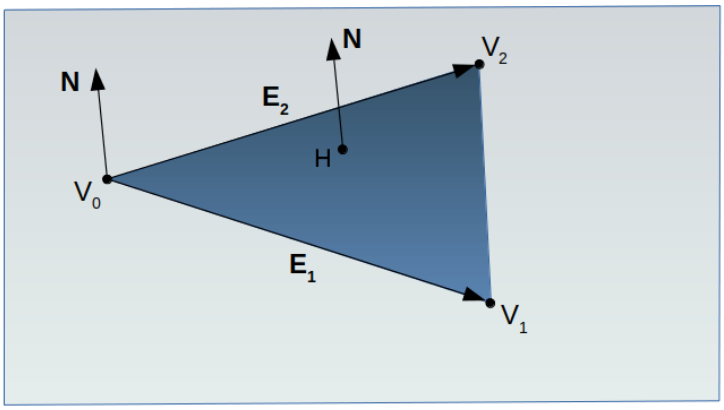
Seja um triângulo qualquer definido pelos seus três vértices $\mathbf{V_0}$, $\mathbf{V_1}$ e $\mathbf{V_2}$, podemos calcular os vetores de duas arestas do triângulo, ambas com origem em $\mathbf{V_0}$, a partir da equação a seguir:

$$\mathbf{E_j} = \mathbf{V_j} - \mathbf{V_0}, \quad j \in \{1, 2\} \tag{3}$$

Depois de calculados os vetores das arestas $\mathbf{E_1}$ e $\mathbf{E_2}$, a partir deles podemos calcular o vetor normal ao plano que contém o triângulo. O vetor normal $\mathbf{N}$ é dado, então, por:

$$\mathbf{N} = \mathbf{E_1} \times \mathbf{E_2} \tag{4}$$

Figura 3 – Representação gráfica de um triângulo e seus vértices, o plano que o contém, os vetores das arestas e o vetor normal.



Tendo calculado o vetor normal ao plano, podemos fazer uso de uma propriedade que afirma que o produto interno de qualquer ponto $\mathbf{P}$ pertencente ao plano pelo seu vetor normal $\mathbf{N}$ é igual a uma constante d (BADOUEL, 1995, p. 390). Calculamos, assim, a constate d do plano em questão utilizando o vértice $\mathbf{V_0}$:

$$d = \mathbf{V_0} \cdot \mathbf{N} \tag{5}$$

Considerando agora o raio a ser traçado como sendo definido por um ponto de origem $\mathbf{O}$ e um vetor de direção de comprimento unitário $\mathbf{D}$, esse raio define uma reta no espaço definida pela seguinte equação paramétrica:

$$\mathbf{R}(t) = \mathbf{O} + t\mathbf{D} \tag{6}$$

A reta definida pelo raio irá intersecionar o plano em algum ponto, aqui denotado por $\mathbf{H}$ e o valor da distância até essa interseção é o valor do parâmetro t. Como o valor d é constante para qualquer ponto pertencente ao plano, temos que:

$$d = \mathbf{H} \cdot \mathbf{N} \tag{7}$$

Fazendo o produto interno de todos os termos da equação (6) pelo vetor $\mathbf{N}$, temos:

$$\mathbf{R}(t) \cdot \mathbf{N} = \mathbf{O} \cdot \mathbf{N} + t \cdot \mathbf{D} \cdot \mathbf{N} \tag{8}$$

Assim, podemos agora encontrar a interseção da reta definida pelo raio e o plano que contém o triângulo, igualando os termos das equações (7) e (8):

$$\mathbf{H} \cdot \mathbf{N} = \mathbf{R}(t) \cdot \mathbf{N} \tag{9}$$

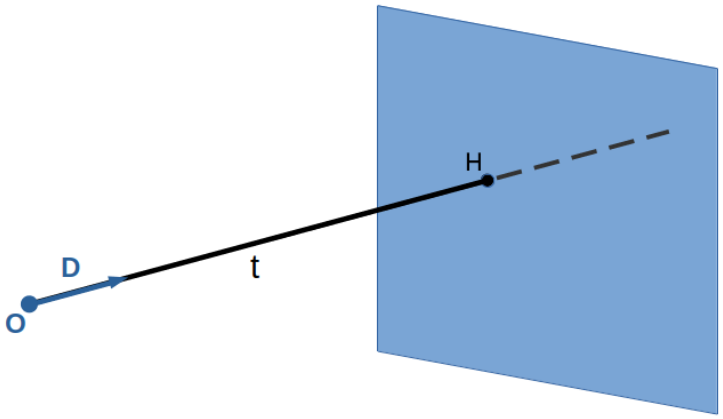
Temos, então, que:

$$d = \mathbf{O} \cdot \mathbf{N} + t \cdot \mathbf{D} \cdot \mathbf{N} \tag{10}$$

E, assim, obtemos a equação que nos fornece o valor da distância t:

$$t = \frac{d - \mathbf{O} \cdot \mathbf{N}}{\mathbf{D} \cdot \mathbf{N}} \tag{11}$$

Figura 4 – Representação gráfica de um raio, sua origem e vetor de direção e a distância t até o ponto de interseção **H** *com o plano.*



O cálculo de t, porém, requer ainda a realização de três testes para verificar se há interseção e se esta é válida (BADOUEL, 1995, p. 391):

- Se o plano que contém o triângulo e a reta definida pelo raio são paralelos (se $\mathbf{N} \cdot \mathbf{D} = \mathbf{0}$), ou seja, não há interseção;
- Se a interseção acontece atrás da origem do raio (se $t < 0$);
- E se uma interseção válida mais próxima já foi encontrada para o raio em questão (se $t > t_{ray}$).

3.1.2 Etapa 2: Interseção com o Triângulo

Tendo calculado a interseção do plano com a reta definida pelo raio, agora nos resta saber se o ponto **H** encontrado pertence ao triângulo. Para este fim partiremos da seguinte equação paramétrica (BADOUEL, 1995, p. 391):

$$\mathbf{E_0} = u\mathbf{E_1} + v\mathbf{E_2} \tag{12}$$

Onde $\mathbf{E_0}$ é o vetor com origem em $\mathbf{V_0}$ e extremidade em **H** (portanto, $\mathbf{E_0} = \mathbf{H} - \mathbf{V_0}$), $\mathbf{E_1}$ e $\mathbf{E_2}$ já são conhecidos (da seção 3.1.1) e u e v são coordenadas baricêntricas do ponto

H.

O ponto **H** pertencerá ao triângulo se e somente se:

$$u \geq 0, \quad v \geq 0 \quad \text{e} \quad u + v \leq 1 \tag{13}$$

As condições acima são derivadas das propriedades das coordenadas baricêntricas. E vale ressaltar que para triângulos representados em um espaço tridimensional, faltaria então a terceira coordenada baricêntrica w. Porém, essa terceira coordenada, se necessário, pode ser obtida a partir das duas outras. A determinação das coordenadas u e v é suficiente para o problema do cálculo de interseção raio-triângulo. Mais informações sobre o assunto podem ser encontradas nas referências (COXETER, 1989, p. 216).

Precisamos, então, determinar os valores de u e v. Para isso, reescrevemos a equação 12 em termos de suas componentes cartesianas:

$$\begin{cases} e_{0x} = u \cdot e_{1x} + v \cdot e_{2x} \\ e_{0y} = u \cdot e_{1y} + v \cdot e_{2y} \\ e_{0z} = u \cdot e_{1z} + v \cdot e_{2z} \end{cases} \tag{14}$$

O sistema anterior possui solução e esta é única. Porém, como temos duas incógnitas e três equações, podemos eliminar uma das equações simplificando o sistema e os cálculos. Isso é feito a partir da projeção do triângulo em um dos planos cartesianos: xy, yz ou zx.

Para escolher o plano em que é feita a projeção do triângulo, devemos escolher o plano cartesiano no qual a área do triângulo projetado é a maior, garantindo, assim, a preservação das características do triângulo. Se escolhêssemos ao acaso um plano e o triângulo fosse perpendicular a esse plano, por exemplo, a projeção seria apenas uma reta, resultando em um triângulo degenerado. Portanto, precisamos escolher a dimensão de projeção como sendo aquela em que a componente do vetor normal **N** possui maior valor absoluto e o plano de projeção será aquele perpendicular a essa dimensão.

Sendo assim, a dimensão de projeção r é definida como sendo:

$$\begin{cases} 0 \text{ se } ||\mathbf{N}||_\infty = n_x \\ 1 \text{ se } ||\mathbf{N}||_\infty = n_y \\ 2 \text{ se } ||\mathbf{N}||_\infty = n_z \end{cases} \tag{15}$$

Os índices de números 0, 1 e 2 correspondem às dimensões x, y e z, respectivamente. Considerando s e q (s e $q \in \{0, 1, 2\}$) como sendo os índices diferentes de r, temos que eles representam o plano cartesiano sq no qual o triângulo deverá ser projetado ($sq \in \{xy, yz, zx\}$). Assim, r, s e q são representações genéricas dos eixos x, y e z. Sejam, então, (e_{0s}, e_{0q}), (e_{1s}, e_{1q}) e (e_{2s}, e_{2q}) as coordenadas bidimensionais dos vetores $\mathbf{E_0}$, $\mathbf{E_1}$ e

$\mathbf{E_2}$ nesse plano, respectivamente, estas serão dadas por:

$$\begin{aligned} e_{0s} &= h_s - v_{0s} & e_{1s} &= v_{1s} - v_{0s} & e_{2s} &= v_{2s} - v_{0s} \\ e_{0q} &= h_q - v_{0q} & e_{1q} &= v_{1q} - v_{0q} & e_{2q} &= v_{2q} - v_{0q} \end{aligned} \tag{16}$$

Como exemplo, caso a dimensão de projeção escolhida seja $r = 1$ (ou seja, $r = y$), o plano projetado sq seria o plano zx e as equações anteriores ficariam assim reescritas:

$$\begin{aligned} e_{0s} &= h_z - v_{0z} & e_{1s} &= v_{1z} - v_{0z} & e_{2s} &= v_{2z} - v_{0z} \\ e_{0q} &= h_x - v_{0x} & e_{1q} &= v_{1x} - v_{0x} & e_{2q} &= v_{2x} - v_{0x} \end{aligned} \tag{17}$$

Voltando ao caso geral, podemos agora simplificar o sistema de equações (14) para duas equações:

$$\begin{cases} e_{0s} = u \cdot e_{1s} + v \cdot e_{2s} \\ e_{0q} = u \cdot e_{1q} + v \cdot e_{2q} \end{cases} \tag{18}$$

Em seguida, para encontrar a solução deste último sistema, podemos fazer uso da regra de Cramer, o que nos dá como resultado:

$$u = \frac{\begin{vmatrix} e_{0s} & e_{2s} \\ e_{0q} & e_{2q} \end{vmatrix}}{\begin{vmatrix} e_{1s} & e_{2s} \\ e_{1q} & e_{2q} \end{vmatrix}} \qquad v = \frac{\begin{vmatrix} e_{1s} & e_{0s} \\ e_{1q} & e_{0q} \end{vmatrix}}{\begin{vmatrix} e_{1s} & e_{2s} \\ e_{1q} & e_{2q} \end{vmatrix}} \tag{19}$$

E, por fim, resolvendo os determinantes anteriores, obtemos as equações para os valores de u e v:

$$u = \frac{e_{0s} \cdot e_{2q} - e_{0q} \cdot e_{2s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} \qquad v = \frac{e_{1s} \cdot e_{0q} - e_{1q} \cdot e_{0s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} \tag{20}$$

Com os valores de u e v determinados, basta testarmos as condições expressas pelas inequações (13), para descobrirmos se o ponto de interseção $\mathbf{H}$ pertence ou não ao triângulo.

Essa última etapa encerra o algoritmo geral para o cálculo da interseção raio-triângulo e, caso uma interseção válida seja encontrada, temos os valores de t, u e v determinados. As coordenadas (u, v) do ponto $\mathbf{H}$ também podem ser usadas, nas aplicações de renderização, em cálculos posteriores como os de mapeamento de textura, interpolação da normal e interpolação de cor, por exemplo (MöLLER; TRUMBORE, 1997), o que justifica a preferência dada a algoritmos que realizam o cálculo dessas coordenadas.

A seguir, iniciaremos com a descrição das particularidades dos algoritmos estudados neste trabalho, iniciando pelo algoritmo de Möller-Trumbore, passando, em seguida, pelo algoritmo de Ingo Wald e, por fim, descrevendo o algoritmo de Hanika.

3.2 O Algoritmo de Möller e Trumbore

O trabalho de Möller e Trumbore (1997) apresenta um algoritmo para o cálculo de interseção raio-triângulo otimizado para execução em CPU. Neste trabalho utilizamos a versão *non-culling* do algoritmo (não definimos nem diferenciamos se a face do triângulo atingida é a face anterior ou posterior).

Para o carregamento do conjunto de triângulos que formam o modelo a ser renderizado, o algoritmo utiliza uma estrutura de dados que armazenam as coordenadas dos vértices dos triângulos.

O algoritmo de Möller-Trumbore é tido como o algoritmo mais rápido dentre os que requerem menos memória. Isso se dá pelo fato de que com o algoritmo original não é necessário realizar nenhum pré-processamento na malha de triângulos, não gerando nenhuma informação extra que necessite ser armazenada junto com a informação dos vértices, o que pode representar uma considerável economia de memória para malhas com um grande número de triângulos.

Em geral, como vimos na seção anterior, resolve-se o cálculo da interseção raio-triângulo primeiramente calculando a interseção entre o raio e o plano que contém o triângulo e, em seguida, testando se o ponto encontrado pertence ao triângulo. O algoritmo de Möller-Trumbore, no entanto, faz uma série de transformações nas equações de tal forma que a primeira etapa, a que determina a interseção do raio com o plano, não é necessária. Também é desnecessário realizar a projeção do triângulo em um dos planos cartesianos. Tal fundamentação do algoritmo de Möller-Trumbore é dada a seguir.

Conforme descrito no artigo de Möller e Trumbore (1997, p. 2), considerando o ponto $\mathbf{H}$, com coordenadas cartesianas e pertencente a um triângulo, expresso em termos de suas coordenadas baricêntricas (u, v), pela seguinte equação:

$$\mathbf{H}(u, v) = (1 - u - v)\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2} \tag{21}$$

Assim, para computar a interseção entre o raio $\mathbf{R}(t)$, dado pela equação (6), e o triângulo em $\mathbf{H}(u, v)$, conforme a equação (21), basta fazermos $\mathbf{R}(t) = \mathbf{H}(u, v)$, o que resulta em:

$$\mathbf{O} + t\mathbf{D} = (1 - u - v)\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2} \tag{22}$$

Fazendo os seguintes desenvolvimentos na equação anterior:

$$\mathbf{O} = \mathbf{V_0} - u\mathbf{V_0} - v\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2} - t\mathbf{D}$$

$$\mathbf{O} - \mathbf{V_0} = u(\mathbf{V_1} - \mathbf{V_0}) + v(\mathbf{V_2} - \mathbf{V_0}) + t(-\mathbf{D})$$

Obtemos o seguinte sistema de equações lineares:

$$\begin{bmatrix} -\mathbf{D} & \mathbf{V_1} - \mathbf{V_0} & \mathbf{V_2} - \mathbf{V_0} \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{O} - \mathbf{V_0} \tag{23}$$

Neste sistema, temos o vetor de incógnitas como sendo o vetor $(t \;\; u \;\; v)^T$. Lembrando que t é a distância da origem do raio até a interseção, basta resolvermos o sistema que obteremos os valores de t, u e v. Assim, o cálculo da interseção raio-plano é evitado.

Definindo $\mathbf{T} = \mathbf{O} - \mathbf{V_0}$, substituindo $\mathbf{V_1} - \mathbf{V_0} = \mathbf{E_1}$ e $\mathbf{V_2} - \mathbf{V_0} = \mathbf{E_2}$, a solução do sistema definido pela equação (23) é dada a seguir (utilizando a regra de Cramer):

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\mathbf{D},\ \mathbf{E_1},\ \mathbf{E_2}|} \begin{bmatrix} |\mathbf{T},\ \mathbf{E_1},\ \mathbf{E_2}| \\ |-\mathbf{D},\ \mathbf{T},\ \mathbf{E_2}| \\ |-\mathbf{D},\ \mathbf{E_1},\ \mathbf{T}| \end{bmatrix} \tag{24}$$

Das propriedades do produto triplo (ou misto) de vetores, temos que $|\mathbf{A},\ \mathbf{B},\ \mathbf{C}| = -(\mathbf{A} \times \mathbf{C}) \cdot \mathbf{B} = -(\mathbf{C} \times \mathbf{B}) \cdot \mathbf{A} = -(\mathbf{B} \times \mathbf{A}) \cdot \mathbf{C}$. Assim, podemos reescrever a equação (24) da seguinte forma:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\mathbf{D} \times \mathbf{E_2}) \cdot \mathbf{E_1}} \begin{bmatrix} (\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{E_2} \\ (\mathbf{D} \times \mathbf{E_2}) \cdot \mathbf{T} \\ (\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{D} \end{bmatrix} \tag{25}$$

Em que $(\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{E_2}$ é obtido a partir da igualdade $\mathbf{A} \times \mathbf{B} = -(\mathbf{B} \times \mathbf{A})$ (o produto vetorial é anticomutativo). Portanto:

$$-(\mathbf{E_1} \times \mathbf{T}) \cdot \mathbf{E_2} = (\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{E_2} \tag{26}$$

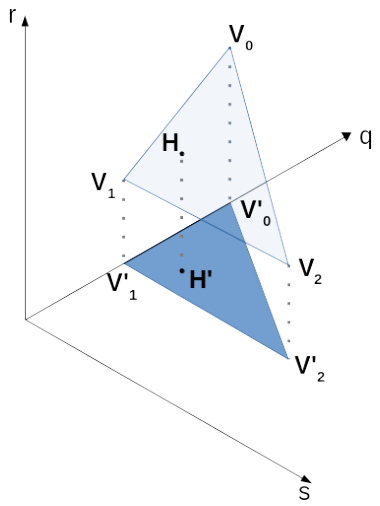
Por fim, fazendo $\mathbf{S} = (\mathbf{D} \times \mathbf{E_2})$ e $\mathbf{Q} = (\mathbf{T} \times \mathbf{E_1})$, obtemos a equação principal do algoritmo de Möller-Trumbore:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\mathbf{S} \cdot \mathbf{E_1}} \begin{bmatrix} \mathbf{Q} \cdot \mathbf{E_2} \\ \mathbf{S} \cdot \mathbf{T} \\ \mathbf{Q} \cdot \mathbf{D} \end{bmatrix} \tag{27}$$

3.3 O Algortimo de Ingo Wald

O algoritmo proposto por Wald (2004) também é otimizado para execução em CPU e usa a ideia de projetar o triângulo em um dos planos cartesianos não ortogonal ao plano do triângulo, visto que a projeção diminui a quantidade de coordenadas (de três para duas) e não altera as coordenadas baricêntricas do ponto de interseção (WALD, 2004, p. 91).

Figura 5 – Exemplo de projeção de triângulo em um dos planos cartesianos.



O ponto de interseção, aqui denotado por $\mathbf{H}$, é, então, dado por:

$$\mathbf{H'} = (1 - u - v)\mathbf{V'_0} + u\mathbf{V'_1} + v\mathbf{V'_2} \tag{28}$$

Onde $\mathbf{V'_0}$, $\mathbf{V'_1}$, $\mathbf{V'_2}$ e $\mathbf{H'}$ são as projeções de $\mathbf{V_0}$, $\mathbf{V_1}$, $\mathbf{V_2}$ e $\mathbf{H}$, respectivamente. O plano escolhido para a projeção é sempre o que melhor preserva a área do triângulo. Rearranjando os termos da equação anterior, obtemos:

$$u(\mathbf{V'_1} - \mathbf{V'_0}) + v(\mathbf{V'_2} - \mathbf{V'_0}) = \mathbf{H'} - \mathbf{V'_0} \tag{29}$$

Em seguida, usando o esquema de Horner, obtemos:

$$u = \frac{|\mathbf{E'_0},\ \mathbf{E'_2}|}{|\mathbf{E'_1},\ \mathbf{E'_2}|}, \quad v = \frac{|\mathbf{E'_1},\ \mathbf{E'_0}|}{|\mathbf{E'_1},\ \mathbf{E'_2}|} \tag{30}$$

Onde $\mathbf{E'_2} = \mathbf{V'_2} - \mathbf{V'_0}$, $\mathbf{E'_1} = \mathbf{V'_1} - \mathbf{V'_0}$ e $\mathbf{E'_0} = \mathbf{H'} - \mathbf{V'_0}$. Chegamos, portanto, aos mesmos resultados obtidos nas equações (20), que nos dão o cálculo das coordenadas baricêntricas utilizando apenas duas dimensões:

$$u = \frac{e_{0s} \cdot e_{2q} - e_{0q} \cdot e_{2s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}, \quad v = \frac{e_{1s} \cdot e_{0q} - e_{1q} \cdot e_{0s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}$$

A distância t é calculada através da equação (11), fazendo apenas algumas modificações demonstradas a seguir. Substituindo o valor de d dado pela equação (5) na equação (11), obtemos:

$$t = \frac{\mathbf{V_0} \cdot \mathbf{N} - \mathbf{O} \cdot \mathbf{N}}{\mathbf{D} \cdot \mathbf{N}} \tag{31}$$

O algoritmo de Ingo Wald, no entanto, usa valores constantes em relação aos triângulos, que foram antecipadamente calculados e armazenados, através de um pré-

processamento da malha de triângulos.

Primeiro, portanto, foi identificado que não é necessário armazenar o vetor normal completo: se r é a dimensão de projeção, logo a componente n_r do vetor normal nunca será zero e, assim, podemos dividir $\mathbf{N}$ por n_r obtendo:

$$\mathbf{N}' = \frac{\mathbf{N}}{n_r} \tag{32}$$

Podemos ainda substituir $\mathbf{N}$ por $\mathbf{N}'$ na equação (31), sem alterar-lhe o resultado:

$$t = \frac{\mathbf{V_0} \cdot \mathbf{N}' - \mathbf{O} \cdot \mathbf{N}'}{\mathbf{D} \cdot \mathbf{N}'} \tag{33}$$

Resolvendo alguns dos produtos internos da equação (33), obtemos ainda outra equação alternativa para o cálculo de t:

$$t = \frac{\mathbf{V_0} \cdot \mathbf{N}' - o_s \cdot n'_s - o_q \cdot n'_q - o_r \cdot n'_r}{d_s \cdot n'_s + d_q \cdot n'_q + d_r \cdot n'_r} \tag{34}$$

Da equação anterior podemos destacar os valores constantes em relação a cada triângulo: $d' = \mathbf{V_0} \cdot \mathbf{N}'$, $n'_s = \frac{n_s}{n_r}$ e $n'_q = \frac{n_q}{n_r}$. Essas constantes podem, portanto, ser pré-computadas e armazenadas. Como $n'_r = \frac{n_r}{n_r} = 1$, não precisa ser calculado nem armazenado. Com isso, obtemos a equação final otimizada do algoritmo de Ingo Wald para o cálculo do valor de t, destacando os valores constantes que podem ser pré-computados:

$$t = \frac{d' - o_r - o_s \cdot n'_s - o_q \cdot n'_q}{d_r + d_s \cdot n'_s + d_q \cdot n'_q} \tag{35}$$

Podemos agora, usando o mesmo raciocínio anterior, rearranjar as equações (20) de u e v para destacar os termos constantes em relação a cada triângulo e que podem ser pré-calculados. Iniciando pela equação de u e nela substituindo $e_{0s} = h_s - v_{0s}$ e $e_{0q} = h_q - v_{0q}$, obtemos:

$$u = \frac{(h_s - v_{0s}) \cdot e_{2q} - (h_q - v_{0q}) \cdot e_{2s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}$$

$$u = \frac{1}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} (e_{2q} \cdot h_s - e_{2q} \cdot v_{0s} - e_{2s} \cdot h_q + e_{2s} \cdot v_{0q})$$

$$u = \frac{e_{2q}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} h_s + \frac{-e_{2s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} h_q + \frac{e_{2s} \cdot v_{0q} - e_{2q} \cdot v_{0s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}$$

$$u = k_{us} \cdot h_s + k_{uq} \cdot h_q + k_{ud} \tag{36}$$

Os termos pré-calculáveis da equação de u, são, portanto:

$$k_{us} = \frac{e_{2q}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}, \quad k_{uq} = \frac{-e_{2s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}, \quad k_{ud} = \frac{e_{2s} \cdot v_{0q} - e_{2q} \cdot v_{0s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} \tag{37}$$

De maneira análoga, podemos rearranjar a equação de v, obtendo:

$$v = k_{vs} \cdot h_s + k_{vq} \cdot h_q + k_{vd} \tag{38}$$

Em que os termos constantes são dados por:

$$k_{vs} = \frac{-e_{1q}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}, \quad k_{vq} = \frac{e_{1s}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}}, \quad k_{vd} = \frac{e_{1q} \cdot v_{0s} - e_{1s} \cdot v_{0q}}{e_{1s} \cdot e_{2q} - e_{1q} \cdot e_{2s}} \tag{39}$$

Logo, a estrutura de dados resultante do pré-processamento da malha de triângulos, para o algoritmo de Ingo Wald, é a seguinte:

```
struct {
    r,
    d',
    n_s', n_q',
    k_us, k_uq,
    k_ud,
    k_vs, k_vq,
    k_vd
}
```

3.4 O Algoritmo de Hanika

No seu trabalho, Hanika e Keller (2007) escolheram o algoritmo de Badouel como base, acrescentaram pré-computações para facilitar os cálculos em ponto fixo e fizeram alguns ajustes e análises para garantir estabilidade numérica: foram investigados os limites númericos e os efeitos das quantizações em ponto fixo (HANIKA; KELLER, 2007, p. 1). O autor também faz uso do artifício de projetar os triângulos em um plano e, assim, reduzir as três dimensões em apenas duas.

Sendo assim, o algoritmo de Hanika pré-calcula inicialmente os valores constantes em relação a um triângulo, alguns idênticos aos calculados por Ingo Wald e listados a seguir (lembrando que r é a dimensão de projeção, s e q são as dimensões do plano projetado):

$$n'_r = \frac{n_r}{n_r} = 1, \quad n'_s = \frac{n_s}{n_r}, \quad n'_q = \frac{n_q}{n_r} \tag{40}$$

$$d' = v_{0r} + v_{0s} \cdot n'_s + v_{0q} \cdot n'_q \tag{41}$$

A equação da distância t é, então, dada pela mesma equação (35) do algoritmo de Ingo Wald, com as mesmas constantes destacadas:

$$t = \frac{d' - o_r - o_s \cdot n'_s - o_q \cdot n'_q}{d_r + d_s \cdot n'_s + d_q \cdot n'_q}$$

Lembrando que d é a constante do plano que contém o triângulo e d_r, d_s e d_q são as componentes do vetor $\mathbf{D}$ de direção do raio.

No cálculo das componentes bidimensionais dos vetores das arestas $\mathbf{E'_1}$ e $\mathbf{E'_2}$, no entanto, o algoritmo de Hanika as normaliza também por n_r, visto que tal normalização simplifica os produtos internos (HANIKA; KELLER, 2007, p. 4), resultando em:

$$e'_{ik} = \frac{v_{ik} - v_{0k}}{n_r}, \quad i \in \{1,\ 2\},\ k \in \{s,\ q\} \tag{42}$$

E por fim, o algoritmo calcula as coordenadas s e q do ponto de interseção $\mathbf{H}'$, bem como as coordenadas baricêntricas u e v. Destacando os termos constantes em relação a um triângulo, temos:

$$\begin{aligned} h_s &= o_s + t \cdot d_s - v_{0s} \\ h_q &= o_q + t \cdot d_q - v_{0q} \end{aligned} \tag{43}$$

$$\begin{aligned} u &= e'_{1s} \cdot h_q - e'_{1q} \cdot h_s \\ v &= e'_{2q} \cdot h_s - e'_{2s} \cdot h_q \end{aligned} \tag{44}$$

Assim, a estrutura de dados resultante do pré-processamento da malha de triângulos para o algoritmo de Hanika é a seguinte:

```
struct {
   r,
   d',
   v_0s, v_0q,
   n_s', n_q',
   e_1s', e_2s',
   e_1q', e_2q'
}
```

Vale ressaltar que os intervalos destes valores pré-computados também foram analisados no trabalho original, para que nenhuma informação importante se perca, principalmente nesses valores que são os valores iniciais do cálculo de interseção.

O autor comenta e mostra alguns resultados no artigo que um *hardware* foi projetado com o seu algoritmo de cálculo de interseção raio-triângulo, implementado em um *ray tracing* e usando aritmética de ponto fixo conseguiu resultados comparáveis aos de um *software* de *ray tracing* utilizando aritmética de ponto flutuante. O *hardware* foi descrito

em VHDL, incluindo recursos de pipelining no teste da interseção com o triângulo e um estrutura de aceleração BIH para o percorrimento da lista de triângulo. O *hardware* e particularidades de sua arquitetura é descrito com mais detalhes em sua dissertação (HANIKA, 2007).

Aqui termina a descrição dos trabalhos originais dos três algoritmos estudados e experimentados por esse trabalho, além do algoritmo de Badouel, que pode ser visto como um algoritmo geral e básico para o cálculo de intersecção raio-triângulo. No próximo capítulo abordaremos as principais particularidades da implementação desses algoritmos nesse trabalho, como as modificações, ajustes e simplicações que foram realizadas.

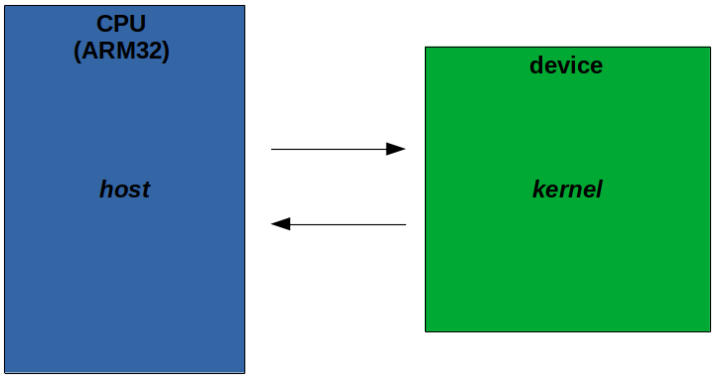
4 DESENVOLVIMENTO

Neste capítulo faremos uma descrição geral da implementação dos algoritmos baseados nos trabalhos de Möller-Trumbore, Ingo Wald e Hanika. Inicialmente mostramos uma visão geral de como foi organizada a aplicação.

4.1 Visão Geral da Aplicação

A estrutura da nossa aplicação OpenCL™ é a mesma descrita na seção 2.4. Em particular, o *host* define o objeto da câmera e copia os parâmetros desta na memória global do *device* para que o *kernel* tenha acesso a eles. Como o *ray casting* dispara um raio por *pixel*, cada *work-item* fica responsável por um *pixel* e constrói o seu raio (vetores de origem e direção) a partir dos parâmetros da câmera e de seu *global ID*. O *work-item*, então, testa a interseção do raio contra cada triângulo presente na malha que foi carregada e, em seguida, havendo interseção, define a cor e a escreve no *buffer* também presente na memória global, destinado a conter a imagem renderizada. Ao final da execução de todos os *work-items*, o *host* copia o *buffer* de imagem da memória global do device de volta para a memória da CPU (processador ARM integrado junto com a FPGA), monta a imagem e escreve o arquivo no disco em formato PPM.

Figura 6 – Visão geral de uma aplicação OpenCL™ .



O formato PPM salva os valores RGB de cada *pixel* exatamente como foram gerados pela renderização, garantindo que esses valores não sofram nenhum ajuste posterior, correção ou compressão, o que precisamos garantir para que houvesse uma comparação coerente entre as imagens que seriam geradas.

Os parâmetros usados na função que coloca o *kernel* na fila de comandos, foram convenientemente definidos para que o particionamento dos dados a serem processados fosse automático, levando em consideração a quantidade de *compute units* disponíveis no

device alvo. Isso foi assim implementado visto que o *kernel* foi executado em três diferentes *devices*: uma CPU, uma GPU e a FPGA.

A seguir, apresentamos um pseudocódigo da estrutura geral do *kernel*:

```
1  kernel rendering_kernel (camera, triangleAccelList)
2     Ray ray;
3     ray.origin = camera.position;
4
5     pixel_position = camera.top_left_pixel_center
6                      + camera.pixel_width * global_id_x
7                      - camera.pixel_height * global_id_y;
8
9     ray.direction = pixel_position - camera.position;
10
11    for i in triangleAccelList:
12       intersectTriangle (triangleAccelList[i], ray);
13
14    if there is a intersection:
15       return pixel_color;
16    otherwise:
17       return black;
18 endkernel
```

De acordo com a listagem anterior, dentro do laço no *kernel*, a função *intersectTriangle* (linha 12), que é chamada para realizar o teste do raio com o triângulo, é justamente uma das três implementações dos algoritmos selecionados para esse estudo. Assim, para alterar o algoritmo a ser usado pela aplicação, basta trocar esta função pela implementada para o algoritmo desejado. E são as diferentes implementações da função que faz apenas o teste de interseção de um dado raio com um dado triângulo que serão discutidas nas seções seguintes deste capítulo, ressaltando as particularidades e dificuldades de cada uma das três variações.

Dado que *hardwares* que realizam cálculos usando números em ponto flutuante são mais complexos do que *hardwares* que realizam cálculos usando números em ponto fixo (números inteiros), como foi evidenciado em nossos testes e será apresentado com detalhes na seção 5.5, foi preciso ainda realizar a modificação dos algoritmos de Möller-Trumbore e Ingo Wald para que estes também executassem seus cálculos em ponto fixo, na expectativa de que os circuitos sintetizados fossem mais simples e, portanto, viáveis de serem executados na DE10-Nano.

Para mais detalhes, os códigos da implementações que foram realizadas dos algoritmos estudados neste trabalho, estão disponibilizados na seção A dos anexos deste texto.

4.2 Implementação do Algoritmo de Möller-Trumbore

O algoritmo de Möller-Trumbore original não possui nenhum pré-processamento a ser realizado, como descrevemos na seção 3.2 do capítulo anterior. Porém, já no trabalho de Ingo Wald, as comparações são feitas com uma versão otimizada do algoritmo de Möller-Trumbore, que inclui o pré-processamento da malha de triângulos. O que fizemos então foi implementar também uma versão otimizada do algoritmo de Möller-Trumbore, incluindo o pré-processamento da malha de triângulos e adiantando alguns dos cálculos do algoritmo no *host*, visto que tais cálculos só precisavam ser executados uma única vez para uma mesma malha de triângulos e, em seguida, repassados para *kernel*. Não eram variáveis que dependiam das propriedades de cada raio.

Descreveremos agora as modificações realizadas no algoritmo de Möller-Trumbore para incluir o pré-processamento da malha de triângulos. Utilizando novamente a propriedade do produto misto $(\mathbf{A} \times \mathbf{C}) \cdot \mathbf{B} = (\mathbf{C} \times \mathbf{B}) \cdot \mathbf{A}$, a propriedade do produto vetorial $\mathbf{A} \times \mathbf{B} = -(\mathbf{B} \times \mathbf{A})$ e partindo da equação (25), podemos fazer a seguinte mudança nos seus termos:

$$(\mathbf{D} \times \mathbf{E_2}) \cdot \mathbf{E_1} = (\mathbf{E_2} \times \mathbf{E_1}) \cdot \mathbf{D} = -(\mathbf{E_1} \times \mathbf{E_2}) \cdot \mathbf{D}$$

$$(\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{E_2} = (\mathbf{E_1} \times \mathbf{E_2}) \cdot \mathbf{T}$$

$$(\mathbf{D} \times \mathbf{E_2}) \cdot \mathbf{T} = (\mathbf{T} \times \mathbf{D}) \cdot \mathbf{E_2}$$

$$(\mathbf{T} \times \mathbf{E_1}) \cdot \mathbf{D} = (\mathbf{D} \times \mathbf{T}) \cdot \mathbf{E_1}$$

Usando esses resultados, lembrando que $\mathbf{N} = \mathbf{E_1} \times \mathbf{E_2}$ e definindo $\mathbf{TD} = \mathbf{T} \times \mathbf{D}$, a equação (25) pode ser reescrita como:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{-\mathbf{N} \cdot \mathbf{D}} \begin{bmatrix} \mathbf{N} \cdot \mathbf{T} \\ \mathbf{TD} \cdot \mathbf{E_2} \\ -\mathbf{TD} \cdot \mathbf{E_1} \end{bmatrix} \tag{45}$$

Com a equação (27) já poderíamos substituir a estrutura de dados que continha um triângulo por uma estrutura (que chamamos de estrutura de aceleração) que ao invés de conter os vértices do triângulo, conteria os valores de $\mathbf{E_1}$, $\mathbf{E_2}$ e $\mathbf{T}$ adiantando o cálculo desses valores em um pré-processamento da malha de triângulos realizado no *host*. Com a equação (45), podemos remover o cálculo de $\mathbf{N}$ (um produto vetorial) do *kernel*, utilizando como estrutura de aceleração uma estrutura de dados que contenha os valores de $\mathbf{E_1}$, $\mathbf{E_2}$, $\mathbf{T}$ e também $\mathbf{N}$ e acrescentando o cálculo da normal $\mathbf{N}$ ao pré-processamento realizado no *host*. Sendo assim, um dos argumentos que o *kernel* recebe, deixa de ser a malha dos triângulos e passa a ser a seguinte estrutura de aceleração:

```
typedef struct {
  int3 edge1, edge2;
  int3 normal;
  int3 tvec;
} TriAccel;
```

Para a correta representação em ponto fixo, os membros da estrutura são assim definidos:

```
triaccel.edge1 = (int3) edge1 * 2^m
triaccel.edge2 = (int3) edge2 * 2^m
triaccel.normal = (int3) normal * 2^m
triaccel.tvec = (int3) tvec * 2^m
```

Nos pseudocódigos anteriores a sintaxe *int3* indica um vetor com três componentes (x, y e z) e faz referência ao tipo de variável similar disponível em OpenCL™ .

Antes da estrutura de aceleração ser disponibilizada para o *kernel* precisamos converter seus valores para a representação em ponto fixo. Como demonstrado anteriormente, a conversão é feita pela multitiplicação da variável em ponto flutuante pelo fator 2^m, que, em seguida, é truncada e atribuída a uma variável inteira. Lembramos que m é o valor da largura de *bits* da parte fracionária que será representada em ponto fixo.

Apesar do algoritmo ser otimizado para a execução em CPU, não foram feitas alterações nesse trabalho para otimizá-lo para execução em *hardware*, mas especificamente a FPGA, tampouco foi feita alguma otimização para execução em GPU.

A seguir, destacamos as modificações e ajustes necessários feitos na implementação para a mudança dos cálculos em ponto flutuante para ponto fixo.
Em ponto flutuante:

$$\det = -\mathbf{N} \cdot \mathbf{D}$$

$$\det = -n_x \cdot d_x + n_y \cdot d_y + n_z \cdot d_z$$

Em ponto fixo:

$$\det = -[(n_x \cdot d_x) >> m + (n_y \cdot d_y) >> m + (n_z \cdot d_z) >> m]$$

O símbolo $>>$ denota aqui a operação de deslocamento de m *bits* para a direita. Como cada multiplicação em ponto fixo adiciona um termo 2^m ao resultado, é necessário compensar com a multiplicação do resultado por um termo 2^{-m}. Essa compensação é mais facilmente obtida com o deslocamento à direita de m *bits*. Fizemos ainda o deslocamento logo após a multiplicação de cada componente e antes da soma das multiplicações do produto interno, para simplificar a síntese das operações de soma.

Um dos pontos mais problemáticos na conversão para ponto fixo é o cálculo do

inverso do determinante, que em ponto flutuante é facilmente obtido pelo cálculo:

$$inv\_det = \frac{1}{\det}$$

O cálculo do inverso do determinante se mostrou um problema com relação ao uso dos recursos da FPGA. Foram feitos testes com algumas diferentes formas de converter esse trecho do cálculo, como por exemplo, trocar a multiplicação pelo inverso do determinante, nas três equações em que ele aparece (mostrados em detalhe a seguir), pela divisão pelo valor do determinante, excluindo assim o cálculo do inverso. Porém, os resultados melhores, em termos de uso de recursos, foram obtidos com a manutenção do cálculo do inverso do determinante em ponto flutuante, retornando ao cálculo de ponto fixo logo após a multiplicação pelo inverso e, em seguida, fazendo os ajustes necessários. A implementação final é, portanto, mista: tem uma pequena parte do cálculo que foi feita em ponto flutuante: uma divisão e três multiplicações em ponto flutuante de 32 *bits* requereram menos recursos da FPGA do que 3 divisões em ponto fixo de 64 *bits*.
Em ponto flutuante:

$$\mathbf{TD} = \mathbf{T} \times \mathbf{D}$$

$$\mathbf{TD} = \begin{bmatrix} t_y \cdot d_z - t_z \cdot d_y \\ t_z \cdot d_x - t_x \cdot d_z \\ t_x \cdot d_y - t_y \cdot d_x \end{bmatrix}$$

Em ponto fixo:

$$\mathbf{TD} = \begin{bmatrix} (t_y \cdot d_z - t_z \cdot d_y) >> m \\ (t_z \cdot d_x - t_x \cdot d_z) >> m \\ (t_x \cdot d_y - t_y \cdot d_x) >> m \end{bmatrix}$$

A correção do resultado da multiplicação é novamente feita aqui através do deslocamento para a direita de m *bits*, como demonstrado no resultado anterior. No entanto, para esse cálculo em particular, foi possível fazer a correção apenas após a subtração advinda do produto vetorial.
Cálculos de u, v, t:

$$u = (\mathbf{TD} \cdot \mathbf{E_2}) \cdot inv\_det$$

$$v = -(\mathbf{TD} \cdot \mathbf{E_1}) \cdot inv\_det$$

$$t = (\mathbf{N} \cdot \mathbf{T}) \cdot inv\_det$$

A solução encontrada para os cálculos anteriores foi: os produtos internos $\mathbf{TD} \cdot \mathbf{E_n}$ e $\mathbf{N} \cdot \mathbf{T}$ são resolvidos em ponto fixo, em seguida é realizado o cálculo em ponto flutuante da multiplicação pelo inverso do determinante e o resultado final é atribuído truncado e novamente representado em ponto fixo. Estes cálculos não demandaram nenhuma compensação por deslocamentos, visto que as multiplicações são compensadas pela divisão

implícita realizada ao multiplicarmos o resultado do produto interno pelo inverso do determinante.

Estes foram os ajustes necessários para a implementação em ponto flutuante do algoritmo de Möller-Trumbore para que pudesse ser sintetizado e executado na FPGA, em sua maior parte, realizando apenas cálculos em ponto fixo.

4.3 Implementação do Algoritmo de Ingo Wald

Diferentemente do algoritmo de Möller-Trumbore, o trabalho de Ingo Wald não disponibiliza o código de seu algoritmo, apenas alguns trechos e a descrição do restante. Seguindo as instruções e fazendo uso dos trechos disponibilizados, foi possível implementar e verificar o algoritmo de Ingo Wald com sucesso.

Na modificação para ponto fixo, conseguimos uma conversão mais fácil e praticamente literal do algoritmo de Ingo Wald, o que nos permitiu esperar uma boa taxa de redução na utilização da área da FPGA, que foi confirmada e está apresentada na seção 5.5.

A seguir, apresentamos as modificações necessárias feitas na implementação do algoritmo para a mudança dos cálculos em ponto flutuante para ponto fixo.
Em ponto flutuante:

$$den = d_r + n_x \cdot d_p + n_y \cdot d_q$$

Em ponto fixo:

$$den = d_r + ((n_x \cdot d_p + n_y \cdot d_q) >> m)$$

Seguindo o mesmo raciocício utilizado na modificação da implementação do algoritmo de Möller-Trumbore, no cálculo de denominador da equação que calcula a distância t, fazemos a correção das multiplicações com o deslocamento para a direita de m *bits*.
Em ponto flutuante:

$$t = \frac{n_d - o_r - n_x \cdot o_p - n_y \cdot o_q}{den}$$

Em ponto fixo:

$$t = \frac{((n_d - o_r) << m) - n_x \cdot o_p - n_y \cdot o_q}{den}$$

No cálculo da distância t temos que os termos $n_x \cdot o_p$ e $n_y \cdot o_q$ do numerador, terão suas multiplicações automaticamente compensadas pela posterior divisão pelo denominador. O termo $n_d - o_r$, porém, não envolve uma multiplicação e necessita de correção após a divisão pelo denominador. No entanto, escolhemos fazer esse ajuste antes da divisão, para evitar perdas de precisão.
Em ponto flutuante:

$$H = \begin{bmatrix} o_s \\ o_q \end{bmatrix} + t \cdot \begin{bmatrix} d_s \\ d_q \end{bmatrix}$$

Em ponto fixo:

$$H = \begin{bmatrix} o_s \\ o_q \end{bmatrix} + \begin{bmatrix} (t \cdot d_s) >> m \\ (t \cdot d_q) >> m \end{bmatrix}$$

O cálculo apresentado encontra nas coordenadas s e q do ponto de interseção (lembrando que s e q são as coordenadas após a projeção do triângulo). Em ponto fixo a multiplicação pelo escalar t requer correção, podemos observar que foi realizada uma pequena modificação na equação e o deslocamento para a direita de m *bits* após a multiplicação de cada componente do vetor direção **D** pelo escalar t.
Em ponto flutuante:

$$u = h_s \cdot b_{ns} + h_q \cdot b_{nq} + b_d$$
$$v = h_s \cdot c_{ns} + h_q \cdot c_{nq} + c_d$$

Em ponto fixo:

$$u = ((h_s \cdot b_{ns} + h_q \cdot b_{nq}) >> m) + b_d$$

$$v = ((h_s \cdot c_{ns} + h_q \cdot c_{nq}) >> m) + c_d$$

Nos cálculos das coordenadas baricêntricas u e v, as duas multiplicações são corrigidas após a soma de seus resultados, que é deslocada em m para a direita. Após essas modificações na implementação do algoritmo de Ingo Wald, este pode ser sintetizado e executado na FPGA, realizando todos os seus cálculos em ponto fixo.

4.4 Implementação do Algoritmo de Hanika

Dos algoritmos selecionados, é o único que já é proposto usando apenas operações em ponto fixo. E aqui precisamos apenas discutir uma questão referente a estrutura de aceleração utilizada por ele, transcrita a seguir usando a nossa notação unificada:

```
typedef struct {
   int d;
   unsigned r;
   unsigned ps;
   unsigned int pq;
   int ns, nq;
   int e1s, e1q;
   int e2s, e2q;
} TriAccel;
```

Todos os valores da estrutura de dados anterior foram pré-calculados no *host* em ponto flutuante e convertidos para ponto fixo antes de serem escritos na estrutura, conforme mostramos no seguinte pseudocódigo:

```
triaccel.d = (int) d * 2^m
triaccel.ps = (unsigned) ps * 2^m
triaccel.pq = (unsigned) pq * 2^m
triaccel.ns = (int) ns * 2^m
triaccel.nq = (int) nq * 2^m
triaccel.e1s = (int) e1s * 2^m * 2^e
triaccel.e1q = (int) e1q * 2^m * 2^e
triaccel.e2s = (int) e2s * 2^m * 2^e
triaccel.e2q = (int) e2q * 2^m * 2^e
```

Podemos observar que para os cinco primeiros membros fizemos a multiplicação já usual pelo fator 2^m, porém, para os quatro últimos membros ($e1s$, $e1q$, $e2s$ e $e2q$) há um fator extra sendo multiplicado: 2^e. A explicação para a necessidade desse fator extra, segundo Hanika e Keller (2007, p. 7), se deve ao fato de que os valores dessas componentes das arestas se tornam muito pequenos, menores que 0,001 e a maior parte das vezes menores até que 0,0001. Assim, mapeando esses valores para o intervalo usado na aritmética de ponto fixo, resultaria em consideráveis erros de quantização. Os autores sugerem, então, que antes de mapear esses valores das arestas para o intervalo de ponto fixo, seja feito um deslocamento extra que recebe a denominação de deslocamento de aresta (*edge shift*). Realizamos esse ajuste proposto, portanto, acrescentando a multiplicação pelo fator 2^e, como mostrado no pseudocódigo anterior.

Hanika utiliza ainda a estrutura de aceleração BIH na implementação do seu algoritmo. Porém, por simplificação e para manter uma maior equivalência com os outros algoritmos que foram estudados não utilizados nenhuma estrutura de aceleração do tipo da BIH neste trabalho.

5 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

Neste capítulo as imagens renderizadas com os algoritmos são apresentadas e comparadas. São apresentados também parâmetros de velocidade (performance), área de circuito e estimativa de consumo de energia. Para a extração das métricas de comparação e análise foi renderizada uma imagem de referência utilizando um *ray casting* em ponto flutuante executado em CPU. A imagem tomada como referência foi então comparada com as imagens geradas nas implementações em ponto fixo executadas na DE10-Nano. Os demais parâmetros são retirados dos relatórios da síntese das implementações em FPGA.

A FPGA utilizada para a implementação e testes foi uma FPGA da família Cyclone® V SE, modelo 5CSEBA6U23I7NDK, presente no kit DE10-Nano produzido pela Terasic. Para a compilação e síntese dos algoritmos em hardware foi utilizado o SDK OpenCL™ da Intel® FPGA instalado em um computador com as seguintes especificações:

- Intel® FPGA SDK for OpenCL™ 18.1.1 Build 646 Standard Edition
- Processador Intel® Core™ i5-3570K
- GPU GeForce GT 1030 GDDR5

Este mesmo computador foi também utilizado para a realização dos testes e extração das métricas de comparação.

5.1 A Cena e Alguns Parâmetros Utilizados

Buscando um melhor resultado na conversão para ponto fixo, e seguindo a recomendação descrita e explicada no trabalho de Hanika e Keller (2007, p. 6), aproximamos os limites da cena aos limites da *bounding box* que continha o modelo a ser redenrizado. Outro ajuste proposto também para facilitar a renderização em ponto fixo foi escalar e mover a cena para o primeiro octante (positivo) do espaço tridimensional, assim apenas valores positivos são válidos para as coordenadas dos vértices dos triângulos.

Outra precaução foi armazenar os valores intermediários do cálculo (computações internas do *kernel*), por garantia, em variáveis de 64 bits. O valor da largura de *bits* m utilizado foram sempre valores próximos aos que foram apresentados no artigo de Hanika e obtiveram os melhores resultados, não coincidentemente foram valores próximos de $m = 23$, que é a quantidade de bits comumente utilizada para armazenar o valor da mantissa na representação em ponto flutuante de 32 bits (IEEE..., 2019).

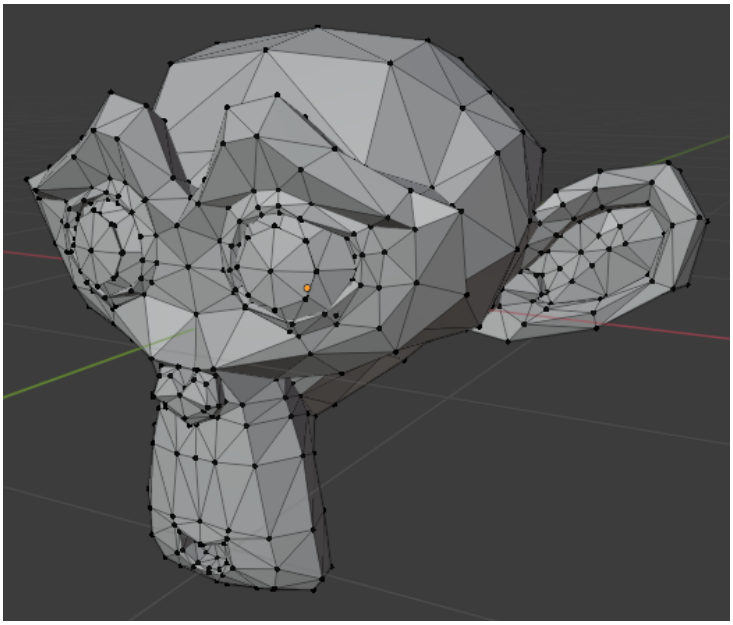
Por fim, a seguir apresentamos os valores dos parâmetros da câmera emulada de projeção perspectiva (escolhemos a convenção do sistema de mão direita) e as coordenadas limitantes da *bounding box* do modelo e, portanto, da cena utilizada:

```
Setup da camera perspectiva
   position = (2, 2, 0)
   look at =  (2, 2, 1)
   up =       (0, 1, 0)
   fov =      55
Bounding box da cena
   xmin = 0,720249; xmax = 3,185574
   ymin = 1,022053; ymax = 2,990803
   zmin = 2,282166; zmax = 4,421446
```

5.2 Imagens Renderizadas

O modelo utilizado nos testes e na extração das métricas de comparação foi a malha *Suzanne* do *software* Blender (COMMUNITY, 2018), composta de um total de 967 triângulos que, em geral, são de boa qualidade, segundo os parâmetros normalmente analisados para medir a qualidade de malhas de triângulos, como, por exemplo, a presença de triângulos degenerados, de ângulos extremos e a diferença de tamanho entre as arestas (PéBAY; BAKER, 2003).

Figura 7 – Malha de triângulos (Suzanne) usada como modelo principal para testes dos algoritmos e geração das imagens.

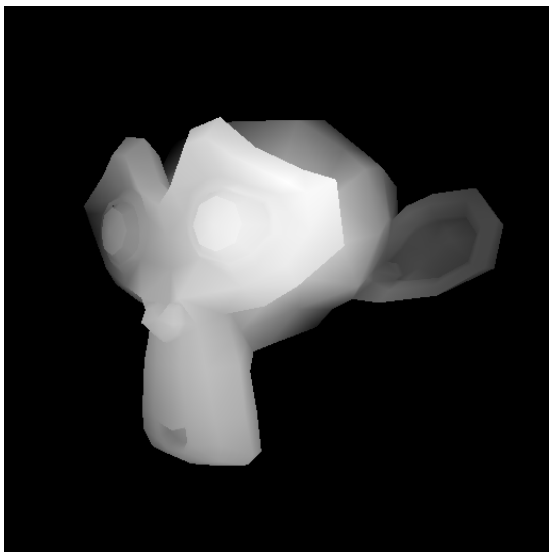


Além da análise visual, extraímos também métricas númericas das imagens renderizadas, fazendo uso, por simplicidade, de duas normas que são comumente utilizadas em processamento de imagens e visão computacional, a norma Zero, também chamada de norma L0 (GARCIA, 2018) e a norma ou distância Manhattan, também conhecida como norma L1 (KRAUSE, 1987). A norma L0 corresponde ao total de elementos em um vetor cujos valores são diferentes de zero. Já a norma L1 é a soma total das magnitudes dos elementos de um vetor.

Cada imagem renderizada possui 512 x 512 *pixels* e cada *pixel* pode assumir valores inteiros que variam no intervalo de $[0, 255]$ (para imagens em escala de cinza). Para análise e comparação das imagens, então, subtraímos da imagem de referência a imagem a ser comparada, *pixel* a *pixel*. E sobre o vetor resultante dessa diferença, aplicamos as normas L0 e L1. A norma L0, portanto, nos informa quantos *pixels* são diferentes entre as imagens e a norma L1 nos informa o valor acumulado das diferenças entre os *pixels*.

Para gerar a imagem de referência, o modelo *Suzanne* foi renderizado utilizando implementações próprias em *software* de um *ray casting* com os algoritmos originais de Möller-Trumbore e Ingo Wald. As execuções de ambos os algoritmos resultaram exatamente na mesma imagem: tanto a norma L0 quanto a norma L1 do vetor de diferença das imagens são iguais a zero (as imagens que serão apresentadas podem ser melhor observadas em tamanhos mais próximos do original na seção B dos Anexos):

Figura 8 – Imagem de referência renderizada por meio de ray casting em CPU utilizando uma implementação em ponto flutuante do algoritmo de Möller-Trumbore



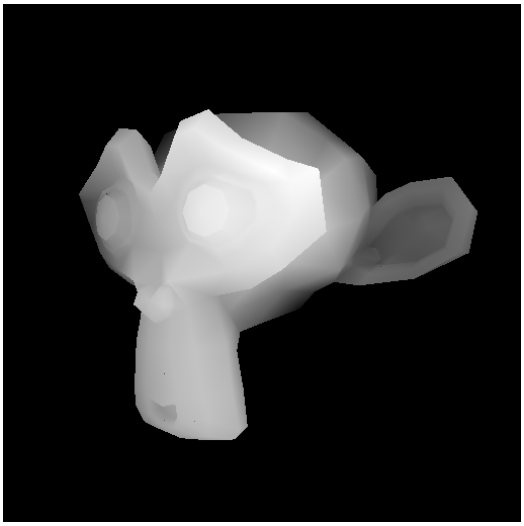
Por simplicidade e para permitir a visualização das caracteristicas de profundidade da imagem, colorimos a imagem usando tons de cinza de acordo com a distância da interseção do raio com o triângulo e a câmera. Utilizamos valores de profundidade normalizados no intervalo $[0, 1]$, onde 1 (branco) representa a menor distância em relação à câmera e 0 (preto) representa a maior distância. Em ponto fixo o intervalo normalizado corresponde a $[0, 255]$.

Para uma comparação justa, apenas a área correspondente à área de *pixels* onde obtivemos interseção foi utilizada na análise, o que corresponde a 68.932 *pixels* do total de 512x512, representados em branco na figura a seguir:

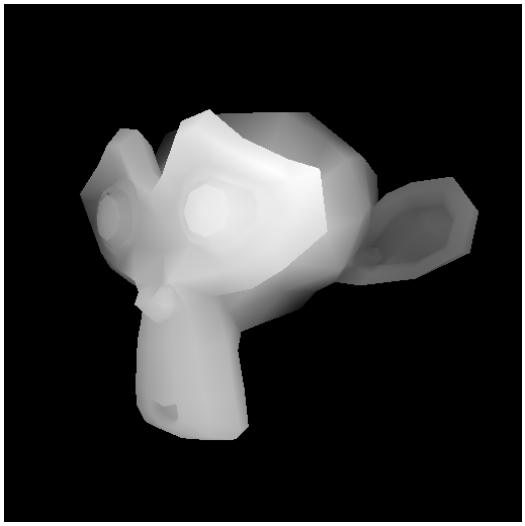
Figura 9 – Imagem destacando em branco os pixels onde houve interseção de raios com triângulos da malha.

Executando as implementações em FPGA obtivemos os seguintes resultados:

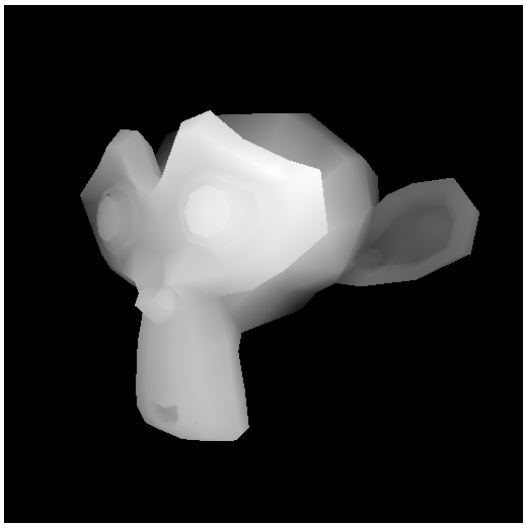
Figura 10 – Modelo Suzanne renderizado utilizando implementações OpenCL e a placa DE10-Nano.



(a) Möller-Trumbore



(b) Ingo Wald



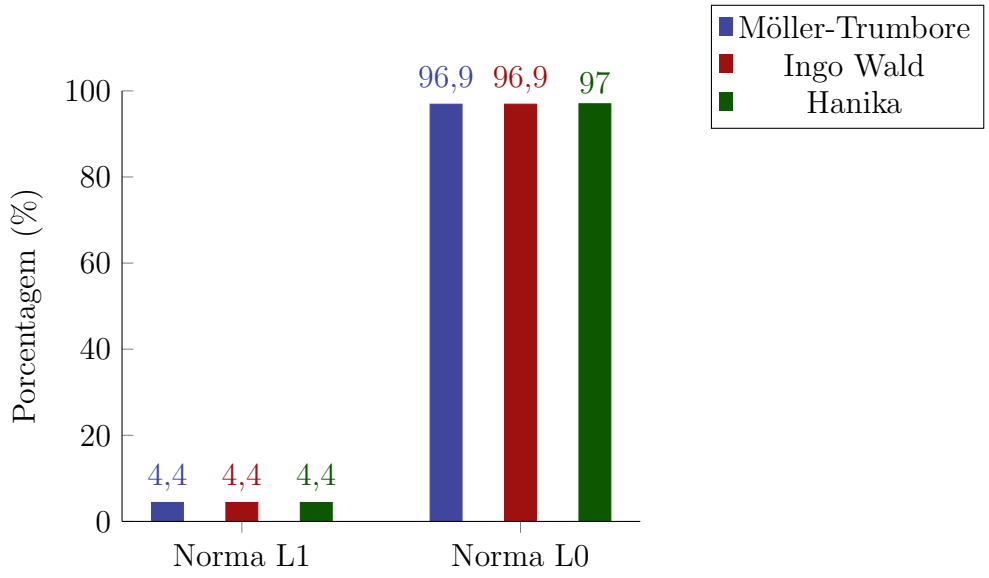
(c) Hanika

Apresentamos na tabela a seguir os resultados das normas L0 e L1, extraídas da diferença entre cada imagem renderizada na FPGA e a imagem de referência renderizada em CPU.

Tabela 1 – Comparações quantitativas entre as imagens geradas na FPGA e a imagem de referência.

Algoritmo	Norma L1	Média (L1)	Norma L0
Möller-Trumbore	778431	11,3	66817
Ingo Wald	776695	11,3	66813
Hanika	775386	11,2	66833

Figura 11 – Gráfico percentual da tabela 1



Como a tabela e gráfico anteriores nos evidencia, as três imagens renderizadas possuem resultados praticamente iguais em termos de qualidade visual: a diferença média entre os valores de nível de cinza de cada imagem e a imagem de referência foi de cerca de 11 níveis, o que corresponde a 4,4% de variação total, que é uma diferença pequena, observada tanto visualmente como em números, apesar de 97% dos *pixels* apresentarem alguma variação.

A diferença apresentada em números na tabela 1, pode ser observada também nas imagens a seguir, que são as imagens obtidas pela subtração das imagens renderizadas pelos algoritmos na FPGA e a imagem de referência:

Figura 12 – Imagens geradas pela subtração da imagem de referência e das imagens geradas em FPGA pelos três algoritmos estudados.

(a) Möller-Trumbore

(b) Ingo Wald

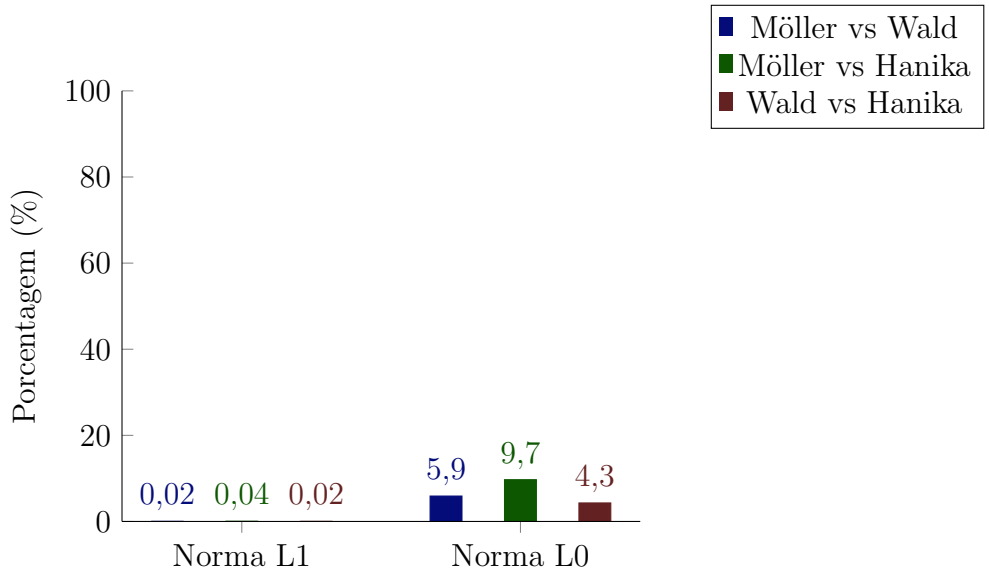
(c) Hanika

Apresentamos ainda na tabela a seguir a comparação feita com as imagens geradas em FPGA entre si. Pode-se observar que a diferença visual e numérica entre as imagens geradas utilizando cálculos em ponto fixo é ainda menor do que quando comparadas com a imagem de referência.

Tabela 2 – Comparações entre as imagens geradas pelos algoritmos em ponto fixo de Möller-Trumbore, de Ingo Wald e de Hanika.

Comparação	Norma L1	Média	Norma L0
Möller / Ingo Wald	4.350	0,06	4.044
Möller / Hanika	7.233	0,10	6.715
Ingo Wald / Hanika	3.413	0,05	2.931

Figura 13 – Gráfico percentual da tabela 2



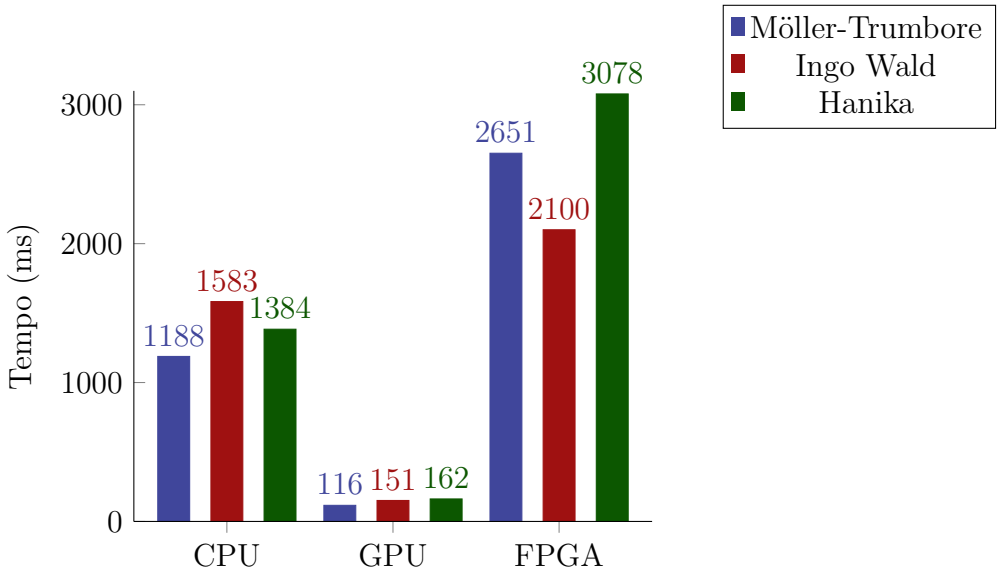
5.3 Performance

Para a extração das estatísticas de tempo foram realizadas 30 execuções seguidas do rendering do modelo Suzanne e o valor de tempo apresentado é a média destas 30 execuções. Para estes testes foram utilizados recursos da OpenCL™ API que fornecem os tempos de execução apenas do *kernel*. Foram comparadas as execuções dos três algoritmos implementados utilizando cálculos em ponto fixo em três OpenCL™ *devices*: uma CPU, uma GPU e a FPGA da DE10-Nano. As especificações relevantes na comparação dos *devices* e os resultados são apresentados nas tabelas a seguir:

Tabela 3 – Tempos de espera na fila (OpenCL™ command queue) e de execução dos três diferentes kernels nos três devices utilizados para os testes.

Device	Algoritmo	Tempo na fila	Tempo de execução
CPU	Möller-Trumbore	0,911 μs	1,188 ms
CPU	Ingo Wald	1,23 μs	1,583 ms
CPU	Hanika	0,845 μs	1,384 ms
GPU	Möller-Trumbore	1,76 μs	116 ms
GPU	Ingo Wald	1,824 μs	151 ms
GPU	Hanika	1,824 μs	162 ms
FPGA	Möller-Trumbore	44,13 μs	2,651 ms
FPGA	Ingo Wald	44,71 μs	2,100 ms
FPGA	Hanika	45,92 μs	3,078 ms

Figura 14 – Gráfico percentual da tabela 1



Os tempos de espera na OpenCL™ *command queue* são referentes ao tempo de carregamento dos argumentos do *kernel* na memória global do *device*.

Tabela 4 – Devices utilizados para a comparação de tempo entre as execuções dos três algoritmos estudados

Device	Frequência base (MHz)	*Compute units*
CPU	3800	4
GPU	1468	3
FPGA	50	1

Como mostrado na tabela 3, o *device* que obteve os melhores resultados de tempo foi a GPU, seguida pela CPU e por último pela FPGA. O *kernel* com o algoritmo de Möller-Trumbore obteve melhores tempos de execução tanto na CPU quanto na GPU, porém, na FPGA foi o *kernel* com o algoritmo de Ingo Wald que obteve o melhor tempo de execução.

Os resultados de velocidade das implementações em FPGA até aqui parecem ser negativos, no entanto, como veremos na próxima seção, a seção 5.4, o uso da FPGA pode ser justificado se normalizarmos a comparação de velocidade pela potência consumida.

5.4 Consumo de energia

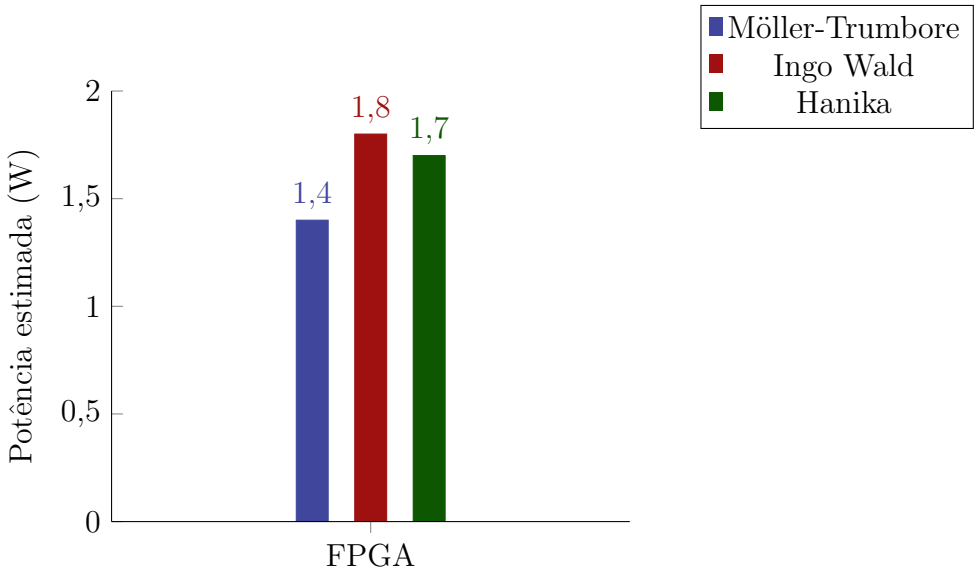
O compilador do SDK OpenCL™ da Intel® FPGA não possui nenhum recurso para análise e estimativa de potência, tampouco a placa DE10-Nano possui sensores para facilitar a para medição de consumo de energia da FPGA. Assim, por simplificação,

as estatísticas de potência de consumo que serão apresentadas são estimativas e foram obtidas, após a síntese dos circuitos, através da ferramenta *Power Analyzer* da suíte Quartus™ Prime que acompanha o SDK OpenCL™ da Intel® FPGA. As estimativas de potência apresentadas possuem uma margem de variação possível de 20 a 30%, segundo documentação própria da ferramenta (POWERPLAY..., 2017, p. 4).

Tabela 5 – Estimativas aproximadas de potência de cada uma das implementações em FPGA.

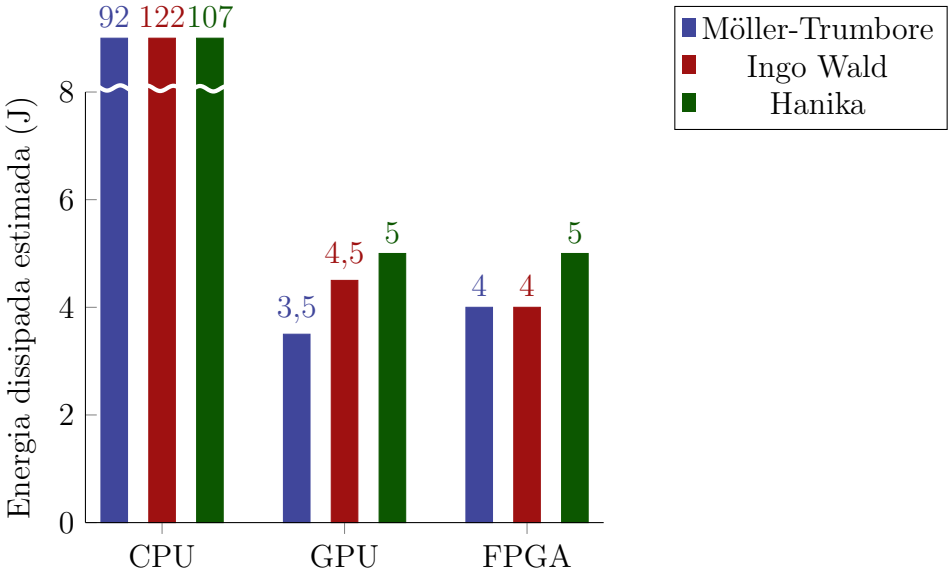
Algoritmo	Potência dinâmica	Potência estática	Potência de E/S	Potência total
Möller-Trumbore	840 mW	500 mW	46 mW	1,4 W
Ingo Wald	1240 mW	530 mW	47 mW	1,8 W
Hanika	1110 mW	520 mW	47 mW	1,7 W

Figura 15 – Gráfico da tabela 5



O algoritmo de Möller-Trumbore, portanto, é o que apresenta melhor estimativa de potência de consumo. E levando em consideração as especificações de potência dos outros dois *devices* utilizados nos testes, 77 W para a CPU (INTEL, 2012) e 30 W para a GPU (NVIDIA, 2018), podemos estimar o gasto de energia em cada execução de cada kernel nos três *devices*. Os resultados destas estimativas são apresentados a seguir:

Figura 16 – Gráfico das estimativas aproximadas de gasto de energia na execução de cada um dos três algoritmos nos três hardwares utilizados.



Observando os resultados do gráfico anterior, vemos que em termos de performance por energia consumida, os resultados obtidos em FPGA são comparáveis aos obtidos em GPU. E a implementação do algoritmo de Möller-Trumbore se destaca como a mais eficiente em todos os três *devices*.

No seu estudo de caso, Singh (2011) obteve resultados mais satisfatórios em termos de consumo de energia e afirma que soluções em FPGA consomem cerca de um quinto (20%) da energia quando comparadas a GPUs similares e executando o mesmo código.

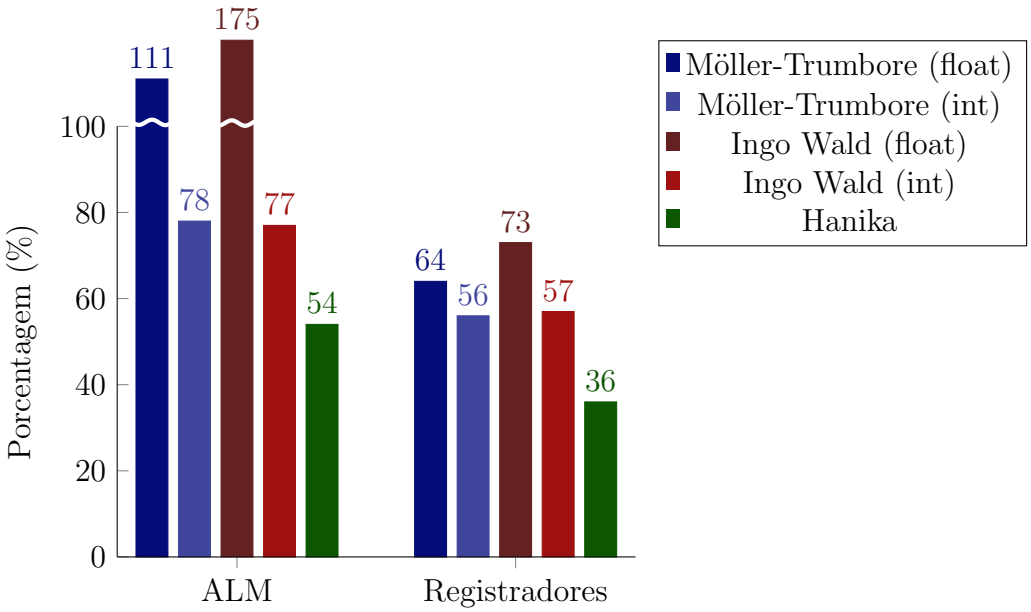
5.5 Resultados de Uso dos Recursos da FPGA: Área do Circuito

Ao final do processo de compilação e síntese, as informações dos recursos utilizados da FPGA podem ser obtidas dos arquivos de relatório gerados pela suíte Quartus® , que é a ferramenta utilizada pelo compilador para a síntese do circuito a ser programado na FPGA. As estatísticas dos três algoritmos selecionados são apresentadas e também para fins de comparação, as estatísticas da síntese dos algoritmos de Möller-Trumbore e Ingo Wald originalmente implementadas usando ponto flutuante. Nos gráficos, as taxas indicam a porcentagem que a quantidade de recursos ali exibida representa do total disponível na FPGA. E ao final de cada tabela os recursos totais da FPGA Cyclone® V são apresentados.

Tabela 6 – Recursos de ALM e registradores requeridos por cada um dos 5 casos de experimentação.

Algoritmo	ALM	Registradores
Möller-Trumbore (float)	46400	107714
Möller-Trumbore (int)	32626	94671
Ingo Wald (float)	73333	122168
Ingo Wald (int)	32435	95118
Hanika	22469	61171
Cyclone® V SE	41910	167640

Figura 17 – Gráfico percentual da tabela 6



Compo podemos observar, o algoritmo de Hanika foi o que obteve menor utilização de ALM da FPGA. Dos algoritmos modificados para trabalhar com ponto fixo, o de Ingo Wald foi o que obteve maior eficiência na redução de área. A menor eficiência da conversão do algoritmo de Möller-Trumbore para ponto fixo já era prevista no desenvolvimento e se explica pelos problemas encontrados nos cálculos que envolvem o inverso do determinante, como foi explicado no capítulo anterior.

O algoritmo de Möller-Trumbore, portanto, demonstrou ser um algoritmo de moderada dificuldade na adaptação para processamento usando apenas ponto fixo, chegando até a demonstrar aumento na utilização de recursos em alguns testes, ao invés da diminuição esperada. Por esse motivo, a implementação final do algoritmo de Möller-Trumbore é mista, utiliza na maior parte dos cálculos a representação em ponto fixo, mas no cálculos que envolvem o inverso do determinante é usada a representação em ponto flutuante. Tal

implementação mista do algoritmo de Möller-Trumbore, foi facilmente obtida graças à síntese de alto nível proporcionada pela utilização da linguagem OpenCL™ . Se tivéssemos utilizado uma HDL, essa mistura de cálculos ora em ponto fixo ora em ponto flutuante seria bem mais difícil de ser realizada.

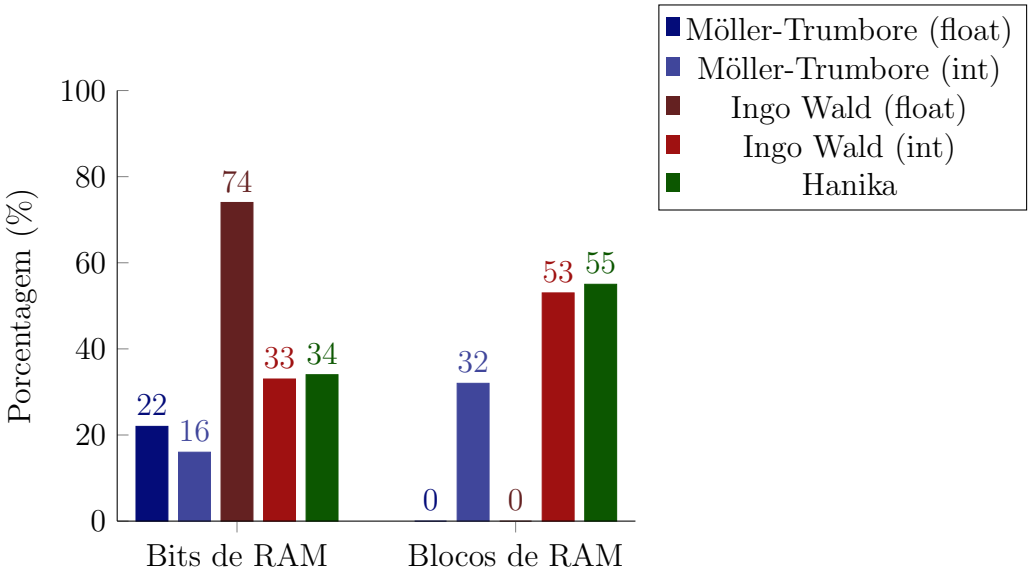
Algo interessante de se observar, é que sabendo que uma ALM possui 4 registradores, talvez esperássemos que o número de registradores alocados fosse exatamente igual ao quádrupulo do número de ALM requeridas. Porém, como pode-se ver na tabela anterior, não há essa equivalência em nenhum dos 5 casos experimentados. O que nos faz perceber na prática que apesar de a ALM ser o elemento lógico "fundamental", ele ao ser alocado não necessariamente terá todos os seus recursos internos aproveitados, dependendo da aplicação.

A tabela e o gráfico a seguir mostram a quantidade de bits de memória requeridos pela síntese e a quantidade de blocos de M10K da FPGA de fato alocados:

Tabela 7 – Recursos de bits de RAM requeridos e a quantidade real de blocos de RAM alocados.

Algoritmo	*Bits* de RAM	Blocos de RAM
Möller-Trumbore (float)	1233800	–
Möller-Trumbore (int)	915744	179
Ingo Wald (float)	4178240	–
Ingo Wald (int)	1842464	292
Hanika	1938224	302
Cyclone® V SE	5662720	553

Figura 18 – Gráfico percentual da tabela 7



Essas estatísticas mostram a eficiência na alocação dos recursos dedicados de memória e o quanto deles é "desperdiçado", dado que os blocos M10K são as menores (e únicas) unidades de memória alocáveis na FPGA utilizada nesse trabalho. Em um projeto de *hardware* digital para ser programado em uma FPGA, é interessante buscar a alocação mais eficiente o possível dos elementos de memória disponíveis. As implementações em ponto flutuante não apresentam a quantidade de blocos de RAM alocados porque como os circuitos requerem mais ALM do que a FPGA tem de disponível, o processo de síntese é automaticamente interrompido antes da alocação dos blocos de RAM.

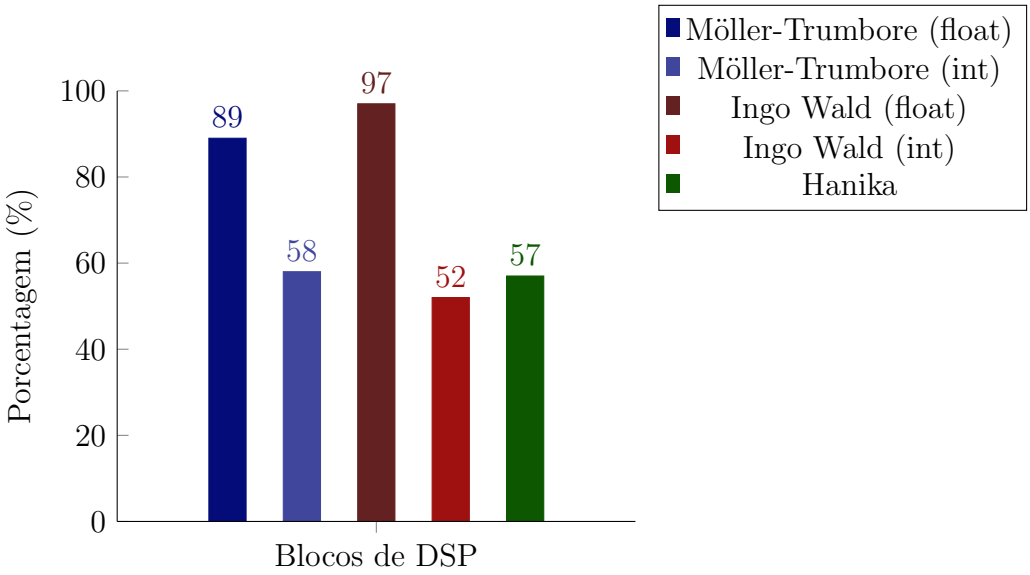
É interessante observar ainda que o algoritmo de Möller-Trumbore, apesar das otimizações de pré-processamento acrescentadas, manteve sua proposição original de ser o algoritmo existente que menos requer memória RAM.

Modernamente, as FPGAs da Intel® FPGA também contém blocos extras de lógica especializados e otimizados para a realização de funções comuns do processamento digital de sinais, assim permitindo que a alocação de recursos seja mais eficiente do que se blocos ALM fossem alocados para tais funções. Apresentamos as estatísticas de utilização desses recursos especializados a seguir:

Tabela 8 – Quantidade requerida por cada algoritmo dos blocos especializados de DSP.

Algoritmo	Blocos de DSP
Möller-Trumbore (float)	100
Möller-Trumbore (int)	65
Ingo Wald (float)	109
Ingo Wald (int)	58
Hanika	64
Cyclone® V SE	112

Figura 19 – Gráfico percentual da tabela 8



Os três projetos que foram programados na FPGA e testados, executam o seu *kernel* a uma frequência de *clock* de 50 MHz. A tabela a seguir mostra ainda o valor máximo estimado para o qual essa frequência poderia ser aumentada sem causar problemas de *timing* e, assim, perda de funcionalidade:

Tabela 9 – Valores de operação e estimativa da frequência base máxima para o hardware na FPGA

Algoritmo	Clock FPGA	Fmax
Möller-Trumbore	50,0 MHz	94,86 MHz
Ingo Wald	50,0 MHz	94,09 MHz
Hanika	50,0 MHz	98,36 MHz

A frequência máxima apresentada nestes resultados é a estimativa de menor valor entre as frequências máximas, obtidas através da análise de *timing* usando dois modelos, o *Slow 1,1V 85C* e o *Slow 1,1V 0C*. O modelo *slow* é o que considera os valores mais lentos estimados da operação dos transistores. A temperatura também influencia a velocidade de transição, pois afeta as características do silício, corrente de fuga e a mobilidade dos elétrons, por exemplo (MAC; WYSOCKI, 2009).

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Fazendo a ressalva de que foram realizados relativamente poucos testes com outras malhas de triângulos, os resultados não apontam para um único algoritmo melhor em todos os aspectos investigados nas implementações para a FPGA. Temos o algoritmo de Hanika como o que requereu a menor quantidade total de recursos da FPGA, porém apresentou a maior estimativa de consumo no teste realizado e o foi também o mais lento na renderização da imagem. O algoritmo de Ingo Wald foi o mais rápido e o algoritmo de Möller-Trumbore foi o que obteve a menor estimativa de consumo de energia.

Uma outra consideração que pode ser feita a partir dos resultados obtidos, acreditamos ser o relativamente baixo ganho em potência de consumo que se esperava. Talvez isso revele o custo da síntese de alto nível em contrapartida ao considerável ganho de tempo que nos possibilitou no desenvolvimento. É possível que a síntese de alto nível se consolide como uma alternativa rápida de prototipação em FPGA ou como solução para aplicações sem muitas restrições de performance ou de consumo de energia, mas provalmente não superará os resultados obtidos por uma implementação em baixo nível utilizando alguma HDL e o fluxo convencional de projeto de *hardware* digital.

Uma das principais vantagens observadas no SDK OpenCL™ para FPGA foi, de fato, permitir a criação de circuitos digitais a partir de uma linguagem de alto nível, o OpenCL™ C utilizado nos *kernels*, sem haver necessidade de se preocupar com nenhum aspecto do baixo nível dos circuitos. Uma outra vantagem, que se mostrou na implementação em ponto fixo do algoritmo de Möller-Trumbore, foi a facilidade em utilizar no mesmo código representações numéricas ora em ponto flutuante ora em ponto fixo. Tal feito em um projeto de baixo nível certamente demandaria uma quantidade considerável de esforço.

Uma possível desvantagem que pode ser apontada no uso de OpenCL™ em FPGA é que o projeto fica limitado ao framework OpenCL™ e como este é organizado, necessitando sempre que uma parte da funcionalidade, mesmo que pequena, seja executada em CPU através do *host*.

Por fim, como possibilidades de trabalhos futuros, elencamos as seguintes:

- Implementar os algoritmos de Möller-Trumbore, Ingo Wald e Hanika em SystemVerilog RTL e verificar os ganhos obtidos, em termos de área, velocidade e potência de consumo, em relação às implementações de alto nível em OpenCL™ C;
- Investigar a possibilidade de instanciar módulos RTL dentro dos *kernels* OpenCL™ e como essa possibilidade poderia ajudar a aumentar a eficiência e eficácia das implementações que foram feitas neste trabalho;
- Otimizar e tentar obter a quantidade mínima de *bits* necessários para cada variável dos *kernels*, verificar o ganho obtido e se não afetaria a qualidade dos cálculos, visto que a maioria das variáveis internas foram declaradas com 64 *bits*;

- Explorar e modificar os arquivos contidos no BSP disponibilizado pela Terasic para a DE10-Nano na tentativa de melhorar a síntese levando em consideração as especificidades das nossas implementações, além de poder testar o direcionamento do esforço da síntese para parâmetros específicos, como área e velocidade;
- Sintetizar um maior número de compute units, verificar os ganhos e como escalam os requisitos de área, e a potência de consumo;

REFERÊNCIAS

BADOUEL, D. An efficient ray-polygon intersection. *Graphics Gems*, 1995.

COMMUNITY, B. O. *Blender - a 3D Modelling and Rendering Package*. Stichting Blender Foundation, Amsterdam, 2018. Disponível em: ⟨http://www.blender.org⟩.

COXETER, H. S. M. *Introduction to Geometry*. 2nd. ed. [S.l.]: Wiley, 1989. (Wiley Classics Library).

GARCIA, S. I. *L0 Norm, L1 Norm, L2 Norm and L-Infinity Norm*. Medium, 2018. Disponível em: ⟨https://medium.com/@montjoile/l0-norm-l1-norm-l2-norm-l-infinity-norm-7a7d18a4f40c⟩.

HANIKA, J. *Fixed Point Hardware Ray Tracing*. Dissertação (Mestrado) — Ulm University, 2007.

HANIKA, J.; KELLER, A. Towards hardware ray tracing using fixed point arithmetic. In: . [S.l.: s.n.], 2007. p. 119 – 128.

IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, p. 1–84, 2019.

INTEL. *Intel Core i5-3570K Processor Product Specifications*. 2012. Disponível em: ⟨https://ark.intel.com/content/www/us/en/ark/products/65520/intel-core-i5-3570k-processor-6m-cache-up-to-3-80-ghz.html⟩. Acesso em: 17 jun. 2020.

INTEL FPGA SDK for OpenCL Standard Edition: Programming guide. [S.l.], 2018.

INTEL FPGA SDK for OpenCL Standard Edition: Best practices guide. [S.l.], 2018.

JAKOB, W. *Mitsuba Renderer*. 2010. Https://www.mitsuba-renderer.org.

KRAUSE, E. F. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. [S.l.]: Dover Publications, 1987.

KROKSTAD, A.; STROM, S.; SøRSDAL, S. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound and Vibration*, v. 8, n. 1, p. 118–125, 1968.

LINZ, P.; WANG, R. *Exploring Numerical Methods: An Introduction to Scientific Computing Using MATLAB*. [S.l.]: Jones and Bartlett Publishers, 2003.

MAC, M.; WYSOCKI, C. Guaranteeing silicon performance with fpga timing models. *Intel FPGA whitepaper*, 2009.

MARTENS, S. *Why do 3D engines primarily use triangles to draw surfaces?* Stack Exchange Inc., 2011. Disponível em: ⟨https://stackoverflow.com/questions/6100528/why-do-3d-engines-primarily-use-triangles-to-draw-surfaces⟩.

MUNSHI, A. (Ed.). *The OpenCL Specification*. 1.0. ed. [S.l.], 2009.

MöLLER, T.; TRUMBORE, B. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, Taylor & Francis, v. 2, n. 1, p. 21–28, 1997.

NVIDIA. *GeForce GT 1030 Specifications*. 2018. Disponível em: ⟨https://www.nvidia.com/en-us/geforce/graphics-cards/gt-1030/specifications⟩. Acesso em: 17 jun. 2020.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically Based Rendering: from Theory to Implementation*. 3. ed. [S.l.]: Elsevier, 2017.

POWERPLAY Early Power Estimator User Guide. [S.l.], 2017.

PéBAY, P.; BAKER, T. Analysis of triangle quality measures. *Math. Comput.*, v. 72, p. 1817–1839, 10 2003.

SCARPINO, M. *OpenCL in Action*: How to accelerate graphics and computation. [S.l.]: Manning, 2011.

SINGH, D. Implementing fpga design with the opencl standard. *Intel FPGA whitepaper*, 2011.

TANG, Z.; MANOCHA, D. Scene-aware sound rendering in virtual and real worlds. In: *2020 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. [S.l.: s.n.], 2020. p. 535–536.

WALD, I. Fast triangle intersection in rtrt. In: _____. *Realtime Ray Tracing and Interactive Global Illumination*. Saarbrucken, Germany: [s.n.], 2004. cap. 7, p. 89–100.

WHITTED, T. An improved illumination model for shaded display. *Communications of the ACM*, v. 23, n. 6, p. 343–349, 1980.

YATES, R. Practical considerations in fixed-point arithmetic: Fir filter implementations. Digital Signal Labs, 2010.

APÊNDICES

A Códigos das Implementações dos Algoritmos

A.1 Implementação do algoritmo de Möller-Trumbore em OpenCL™ C

```
// Non-culling optimized function
bool intersectTriangle (
   global TriAccel* triaccel,          // Pointer to triangle
   private const long3* ray_direction, // Pointer to ray_direction
   private IntersectionRecord* iRec) { // Pointer to intersection record

   // if determinant is near zero, ray lies in plane of triangle
   const long det =
                  - dot (int3tolong3(triaccel->normal), *ray_direction);

   if (det == 0)
      return false;  // No intersection

   const float inv_det = 1.0f / ((float) det);
   const long3 tdvec = cross (triaccel->tvec, *ray_direction);

   // calculate U parameter and test bounds
   const int u = dotM (tdvec, int3tolong3(triaccel->edge2)) * inv_det;
   const int v = - dotM (tdvec, int3tolong3(triaccel->edge1)) * inv_det;

   if ((u < 0) || (v < 0) || ((u + v) > (1U << M)))
      return false;  // No intersection

   // calculate t, ray intersects triangle
   const int t =
         dotM (int3tolong3(triaccel->normal), triaccel->tvec) * inv_det;

   if (t > 0 && t < iRec->t) {
      iRec->t = t;
      iRec->u = u;
      iRec->v = v;
      return true;
   }
   return false;
}
```

A.2 Implementação do algoritmo de Ingo Wald em OpenCL™ C

```
bool intersectTriangle (
   global TriAccel* triaccel, // Pointer to triangle acceleration struct
   private const Ray* ray,            // Pointer to ray
   private IntersectionRecord* iRec,  // Pointer to intersection record
   private uchar p, private uchar q) {

   // Denominator of the equation for the calculus of t
   const long den = ray->direction[triaccel->r]
                    + ((triaccel->n_.x * ray->direction[p]
                       + triaccel->n_.y * ray->direction[q]) >> M);

   // Test to avoid division by zero
   if (den == 0)
      return false;

   // Calculus of t, the distance to intersection
   const long t = (((triaccel->n_d - ray->origin[triaccel->r]) << M)
                  - triaccel->n_.x * ray->origin[p]
                  - triaccel->n_.y * ray->origin[q]) / den;

   // If the new t calculated is not valid or greater than the previous
   // t stored
   if ((t < 0) || (t > iRec->t))
      return false;  // No intersection

   // Compute hitpoint positions on the projected plane
   const long2 h = (long2) (ray->origin[p], ray->origin[q])
                   + (long2) ((t * ray->direction[p]) >> M,
                              (t * ray->direction[q]) >> M);

   // Compute the first barycentric coordinate u (originally beta in the
   // paper)
   const long u = ((h.x * triaccel->b_n.x + h.y * triaccel->b_n.y) >> M)
                  + triaccel->b_d;
   if ((u < 0) || (u > (1LU << M)))
      return false;  // No intersection

   // Compute the second barycentric coordinate v (originally lambda in
   // the paper)
   const long v = ((h.x * triaccel->c_n.x + h.y * triaccel->c_n.y) >> M)
                  + triaccel->c_d;
   if ((v < 0) || ((u + v) > (1LU << M)))
      return false;  // No intersection


```

```
46      // Have a valid hitpoint here, so store it and return success
47      iRec->t = t;
48      iRec->u = u;
49      iRec->v = v;
50      return true;
51  }
```

A.3 Implementação do algoritmo de Hanika em OpenCL™ C

```
bool intersectTriangle (
   global TriAccel* triaccel, // Pointer to triangle acceleration struct
   private const Ray* ray,            // Pointer to ray
   private IntersectionRecord* iRec,  // Pointer to intersection record
   private uchar p, private uchar q) {

   long den = ((ray->direction[triaccel->r]
              + (ray->direction[p] * triaccel->np >> (M - 1))
              + (ray->direction[q] * triaccel->nq >> (M - 1)))) >> HT;
   // Mask to assure precision correctness in hardware implementations
   long mask = 0xFFFFFFFFE0000000L;
   if(den == 0)
      return false;
   // Calculus of t, the distance to intersection
   int t =
         (((((triaccel->d - ray->origin[triaccel->r]) * (1L << (M - 1)))
            - ray->origin[p] * triaccel->np
            - ray->origin[q] * triaccel->nq) >> HT) & mask) / den;

   // If the new t calculated is valid and shorter than the previous t
   // stored
   if((t <= iRec->t) && (t > 0)) {
      int kp = ray->origin[p] + ((t * ray->direction[p]) >> (M - 1))
               - triaccel->pp;
      int kq = ray->origin[q] + ((t * ray->direction[q]) >> (M - 1))
               - triaccel->pq;

      long u = ((long) triaccel->e1p * kq
                - (long) triaccel->e1q * kp)  >> (M - 1);
      long v = ((long) triaccel->e2q * kp
                - (long) triaccel->e2p * kq)  >> (M - 1);

      if(u < 0 || v < 0 || ((u + v) >> E) > (1UL << (M - 1)))
         return false;
      else {
         // Have a valid hitpoint, so store it and return success
         iRec->t = t;
         iRec->u = u;
         iRec->v = v;
         return true;
      }
   }  // if((t <= hit->t) && (t > 0))
   else
      return false;
}  // intersectTriangle
```

B Imagens Renderizadas em Tamanho Próximo do Original

Figura 20 – Modelo Suzanne renderizado usando o software de referência.

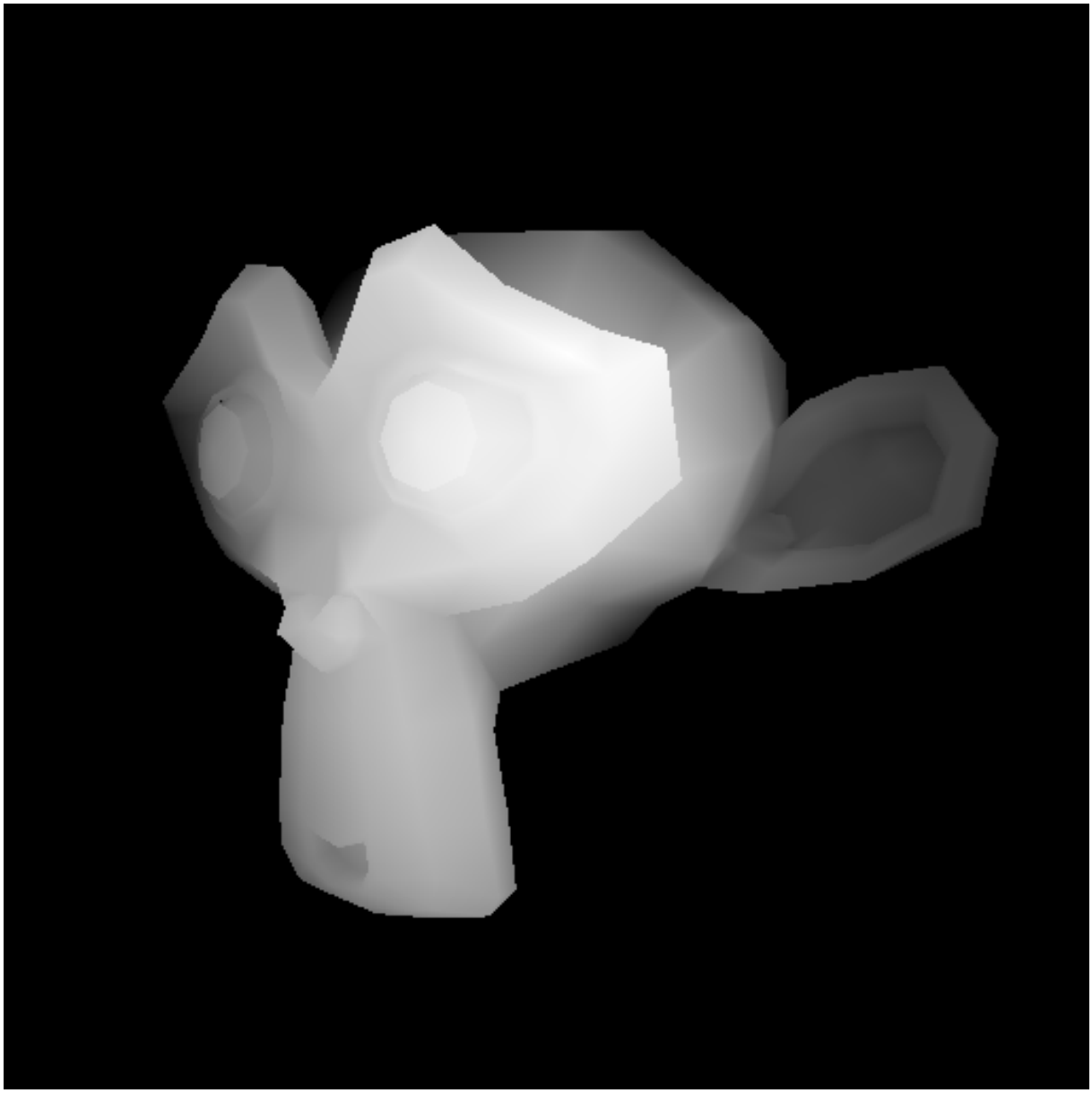


Figura 21 – Modelo Suzanne renderizado usando o algoritmo de Möller-Trumbore em FPGA e em ponto fixo.

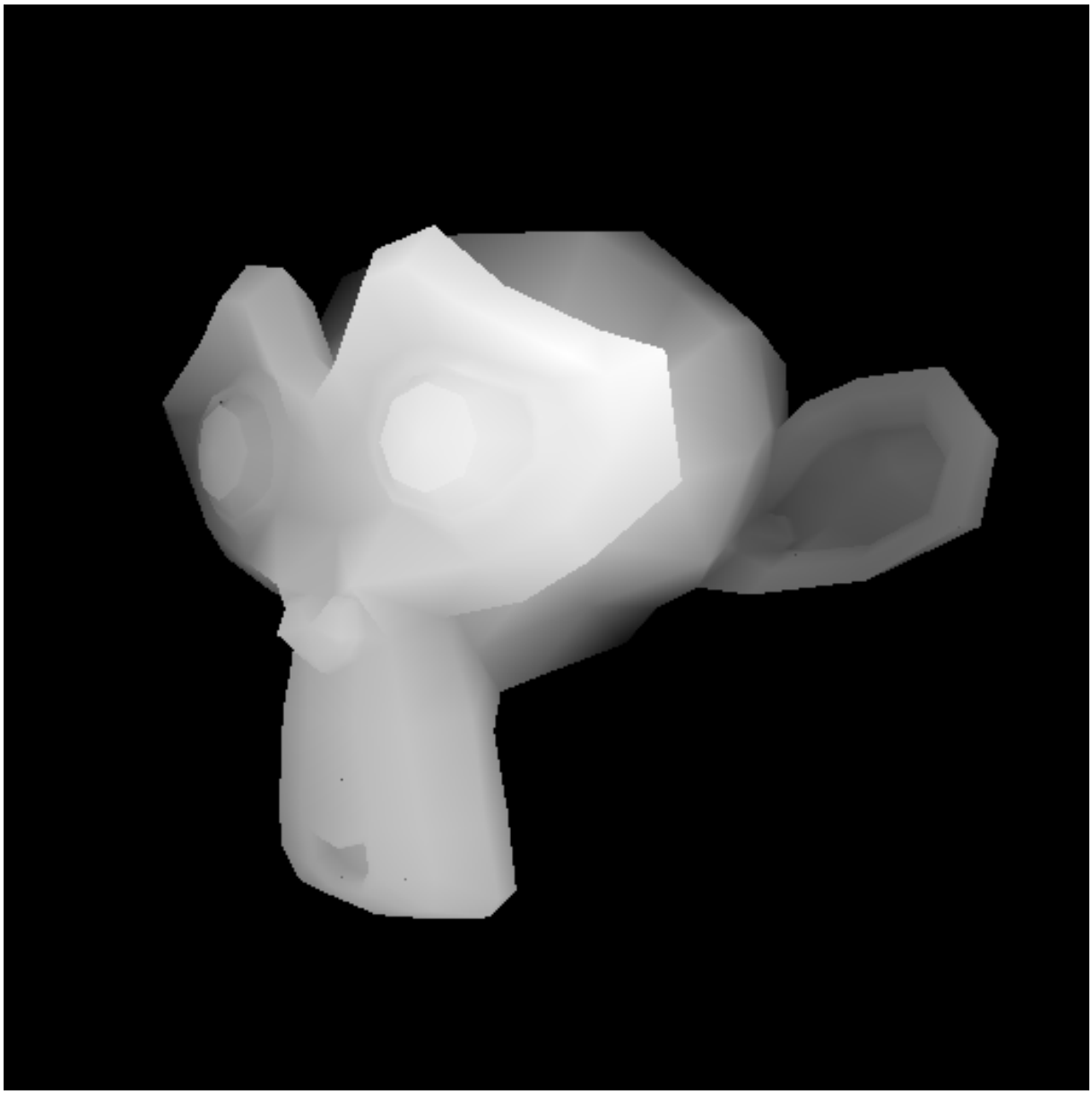


Figura 22 – Modelo Suzanne renderizado usando o algoritmo de Ingo Wald em FPGA e em ponto fixo.

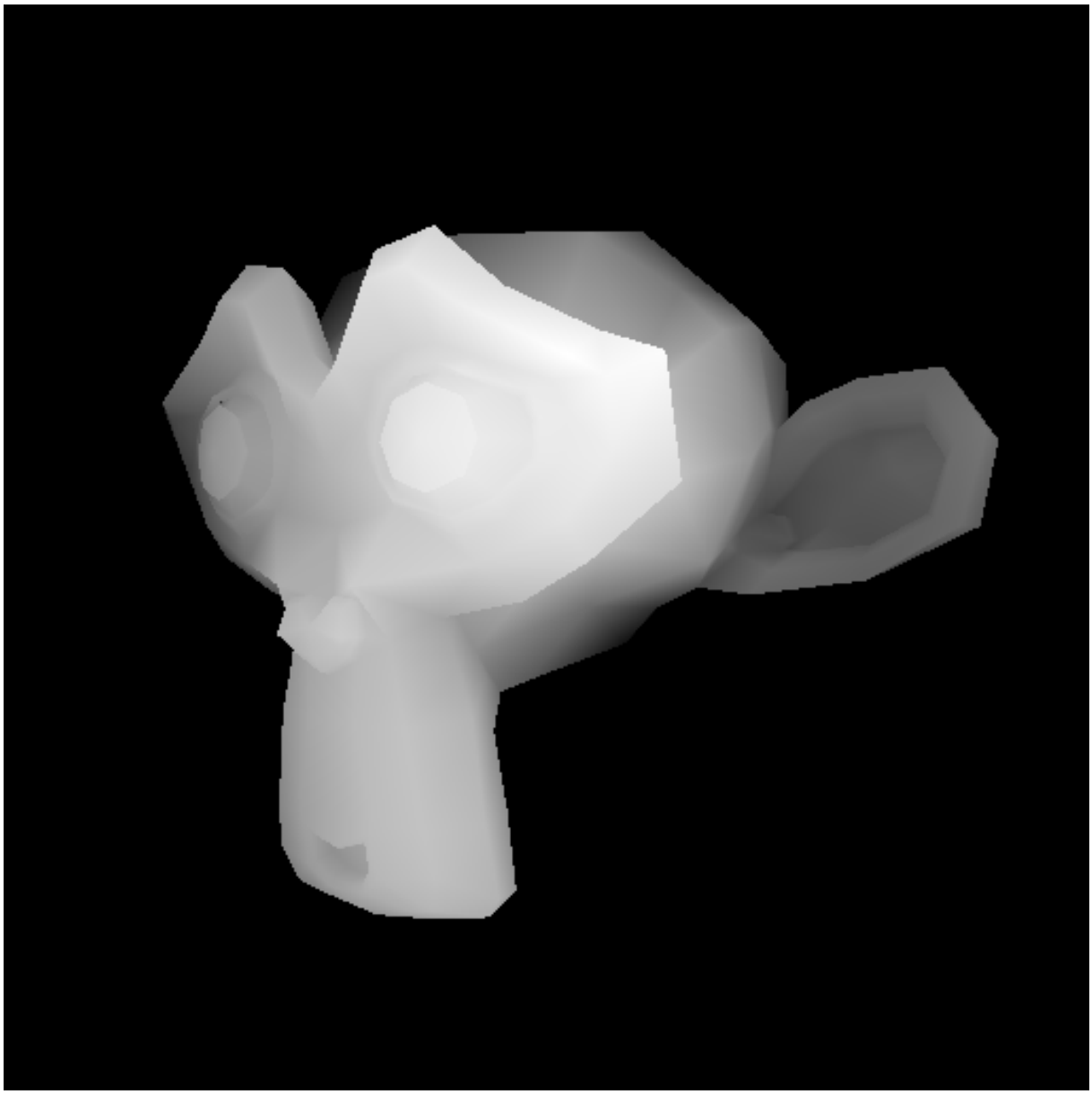
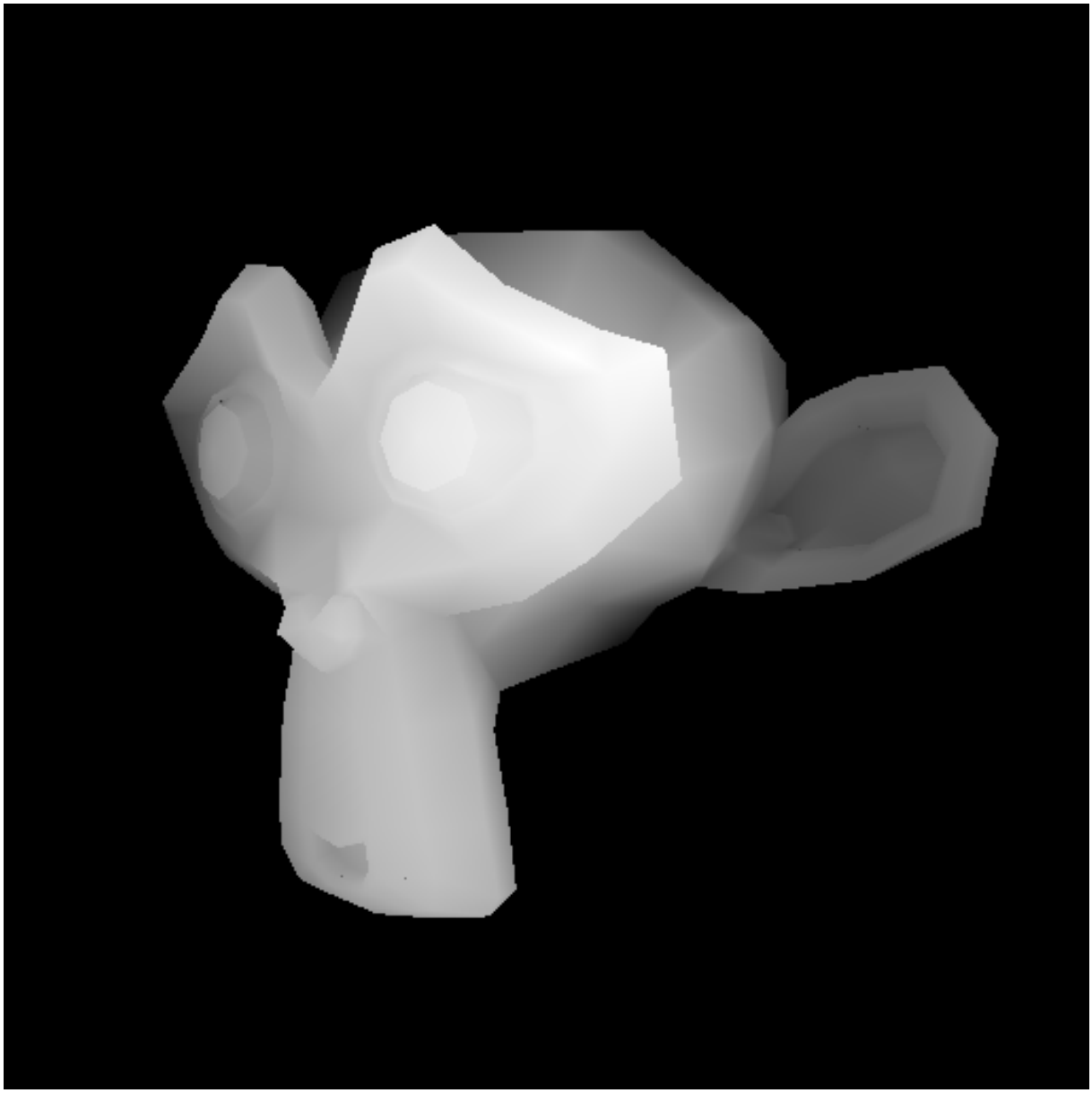


Figura 23 – Modelo Suzanne renderizado usando o algoritmo de Hanika em FPGA.



C Pequeno guia de instalação do OpenCL™ SDK da Intel® FPGA no Linux (Ubuntu 18.04 LTS)

Versão do Ubuntu: 18.04.4 LTS
Kernel atual: 5.3.0-61-generic
FPGA: DE10-Nano revision C

- Da Intel® FPGA baixar:
– **AOCL**
* Intel® FPGA SDK for OpenCL™ Standard 18.1
(Tamanho: 18,9GB MD5: AED22A6F659ACBC78D5DA50EA7F07582)
** O arquivo aqui baixado é o AOCL-18.1.0.625-linux.tar que pode ser descompactado e contém os seguintes arquivos:
1. Quartus Prime Standard Edition
2. Intel® FPGA SDK for OpenCL™
3. Arria 10 Part 1
4. Arria 10 Part 2
5. Arria 10 Part 3
6. Arria V
7. Cyclone® V
8. Stratix V
** Destes, neste trabalho, precisamos apenas dos arquivos:
1. Quartus Prime Standard Edition
(Tamanho: 2,8GB MD5: 7D26DB3BB0ED8EAB62D30FDA4EE316B1)
2. Intel® FPGA SDK for OpenCL™
(Tamanho: 897MB MD5: 4AD6209A9EC0397341169EBBEF028CCC)
7. Cyclone® V
(Tamanho: 1,2GB MD5: 75F5029A9058F64F969496B016EE19D4)
** Os demais podem ser excluídos antes de executar o *script setup.sh* para instalação. Assim o tamanho total instalado diminui consideravelmente, ao se retirar da instalação os modelos de FPGA que não serão utilizados.

– **EDS**
* Intel® SoC FPGA Embedded Development Suite Standard Edition
(Tamanho: 2.6GB MD5: CAD961FE44FCB94639070A076B7ED937)
- Baixar também os updates:
– **AOCL**
* Quartus Prime Software v18.1 Update 1
(Tamanho: 5,4 GB MD5: C81D42FAB13BD0408354644D1942D889)

* Intel® FPGA SDK for OpenCL™ v18.1 Update 1
(Tamanho: 226,9MB MD5: 24B3086701A083BD809B0D638B8E41C2)
– **BSP**
* Intel® FPGA OpenCL™ BSP for Cyclone® V SoC Linux x86-64 TGZ
(Tamanho: 8,8 MB MD5: 5E2546687DA716F286A182C91013BCCB)

- Da Terasic baixar o BSP:
– **DE10-Nano**
* BSP for Intel® FPGA SDK OpenCL™ 18.1 (.tar.gz) 1.0 2019-01-02
(Tamanho: 93MB MD5: 28304ABD0905451C59A28018973D2DA9)