UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Eduardo Capellari Culau

ACELERADOR GRÁFICO PARA *RAY TRACING*

Santa Maria, RS
2023

Eduardo Capellari Culau

ACELERADOR GRÁFICO PARA *RAY TRACING*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia de Computação**.

ORIENTADOR: Prof. Leonardo Londero de Oliveira

Santa Maria, RS
2023

Eduardo Capellari Culau

ACELERADOR GRÁFICO PARA *RAY TRACING*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia de Computação**.

Aprovado em 31 de janeiro de 2023:

Leonardo Londero de Oliveira, Dr. (UFSM)
(Presidente/Orientador)

Everton Alceu Carara, Dr. (UFSM)

Marcelo Serrano Zanetti, Dr. (UFSM)

Santa Maria, RS
2023

RESUMO

ACELERADOR GRÁFICO PARA *RAY TRACING*

AUTOR: Eduardo Capellari Culau
ORIENTADOR: Leonardo Londero de Oliveira

O *ray tracing* é uma técnica de renderização de imagens tridimensionais (3D) em uma exibição bidimensional (2D) traçando um caminho de luz através de pixels em um plano de imagem, a tela. A técnica é capaz de produzir um grau muito alto de realismo visual, mais alto do que o dos métodos de renderização típicos, mas a um custo computacional maior. Um modo de acelerar esse processo é usando placas gráficas dedicadas como as GPUs (*Graphical Processing Units*), co-processadores ou mesmo FPGAs (*Field Programmable Gate Array*) para isso. Essa técnica está cada vez mais presente no nosso dia a dia e tende, num futuro próximo, estar presente em sistemas embarcados, que por padrão não possuem um alto poder computacional. Para que seja possível aplicar a técnica do *ray tracing* nesses dispositivos é necessário dispor de componentes de hardware específicos, focados na aceleração de algumas partes do método, assim tornando viável o uso do mesmo em tempo real. No presente trabalho, foi implementado o método simplificado/parcial do *ray tracing* em linguagem C, de forma a exemplificar o uso do método em um sistema embarcado. Foram analisados os pontos críticos, identificando otimizações em potencial. Foram estabelecidas diferentes configurações do processador para se analisar o desempenho e determinar qual a versão mais otimizada. Ao final, foram comparadas a versão original, a versão otimizada e as implementações presentes na literatura, utilizando como figura de mérito o poder de computação, medido em raios por segundo, além da área e potência de cada implementação. A melhor implementação teve como resultado médio de 108 KiloRays/sec, com um pico de 239 KiloRays/sec.

Palavras-chave: *Ray Tracing*. Processador Configurável. Otimização de Hardware. ASIC. VHDL

ABSTRACT

GRAPHICS ACCELERATOR FOR RAY TRACING

AUTHOR: Eduardo Capellari Culau
ADVISOR: Leonardo Londero de Oliveira

Ray tracing is a technique for rendering three-dimensional (3D) images into a two-dimensional (2D) display by tracing a path of light through pixels on an image plane, the screen. The technique is capable of producing a very high degree of visual realism, higher than typical rendering methods, but at a higher computational cost. One way to speed up this process is to use dedicated graphics cards such as GPUs (Graphical Processing Units), co-processors or even FPGAs (Field Programmable Gate Array) for this purpose. This technique is increasingly present in our daily lives and it tends, in the near future, to be present in embedded systems, which by default do not have high computational power. In order to apply the ray tracing technique to these devices, it is necessary to have specific hardware components, focused on accelerating some parts of the method, thus making it viable to use it in real time. In the present work, the simplified/partial method of ray tracing was implemented in C language, in order to exemplify the use of the method in an embedded system. Critical points were analyzed, identifying potential optimizations. It was performed the definition of some different processor configurations to analyze the performance with the presence of some specific component and determine which version is the most optimized. In the end, the original version, the optimized version and implementations present in the literature were compared, using as a figure of merit the computing power, measured in rays per second, in addition to the area and power of each implementation. The best implementation had an average result of 108 KiloRays/sec, with a peak of 239 KiloRays/sec.

Keywords: Ray Tracing. Configurable processors. Hardware Optimization. ASIC. VHDL

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TIE	*Tensilica® Instruction Extension*
IEEE	Instituto dos Engenheiros Eletricistas e Eletrônicos
RISC	*Reduced Instruction Set Computer*
DSP	*Digital Signal Processing*
ISA	*Instruction Set Architecture*
RTL	*Register Transfer Level*
FLIX	*Flexible Length Instruction Extension*
AR	*Address-Registers*
FR	*Floating-Registers*
ASIC	*Application Specific Integrated Circuit*
SOC	*System-on-chip*
GPU	*Graphical Processing Unit*
FPGA	*Field Programmable Gate Array*
VFX	*Visual Effects*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo faz uma breve contextualização acerca do desafio de projeto proposto neste trabalho ao abordar a importância da criação e modelamento de ambientes gráficos em várias áreas de interesse. Além disso, também são elencados os objetivos e apresentada a organização do texto do trabalho.

1.1 CONTEXTUALIZAÇÃO

A indústria da animação possui uma receita na casa de centenas de bilhões de dólares. Em 2019 este valor foi de US$ 264 bilhões. Já na indústria de jogos eletrônicos o valor foi de US$ 100 bilhões em 2019 (Research and Markets, 2020). Além desses dois exemplos, existem diversos outros setores que usam efeitos visuais, chamados de *visual effects* (VFX), como o setor de efeitos especiais, que aumenta o montante monetário relacionado a toda essa indústria. Diversas técnicas de renderização são usadas pela indústria para criar ambientes, objetos e personagens virtuais em 3D, que se parecem cada vez mais reais. Exemplos dessas técnicas são: *shading*, *tessellation*, *spatial anti-aliasing*, *texture-mapping*, *motion blur*, entre outros. Praticamente a qualquer imagem gerada por computador é aplicada alguma dessas técnicas para tornar a imagem mais realista aos nossos olhos.

Alguns dos efeitos que tornam a imagem mais realista são os efeitos luminosos, como por exemplo, sombras e reflexos. O modo como a luz viaja do seu ponto de origem, reflete e refrata nos objetos até chegar aos nossos olhos é algo único. Esta alta complexidade é o que torna tão desafiadora a reprodução eficiente deste efeito no mundo digital. Para isso existe a técnica conhecida como *ray tracing*, que realiza justamente isso.

A técnica do *ray tracing* é utilizada pela indústria da animação há anos, apresentando uma ampla gama de filmes animados que já empregaram essa técnica. Alguns exemplos deles são os filmes da Pixar Animation Studios como: Os Incríveis 2 (COLEMAN et al., 2018), Carros (CHRISTENSEN et al., 2006) e Procurando Dory (HERY; VILLEMIN; HECHT, 2016). Além disso, o *ray tracing* também é usado para modelagem de fenômenos reais. Uma vez que a luz possui um comportamento análogo a uma onda eletromagnética e o *ray tracing* apresenta uma alta acurácia na representação de seus fenômenos, também é possível utilizar essa técnica para ondas usadas em comunicação *wireless*. Como exemplos disso existem os dois seguintes trabalhos: o primeiro utiliza o *ray tracing* para determinar a distribuição de um sinal *wireless* e

assim encontrar a quantidade e a posição ideal dos transmissores (YUN; LIM; ISKANDER, 2008); já o segundo apresenta um novo algoritmo para que a computação seja feita em um tempo menor, visando modelagem de canais de comunicação (WANG et al., 2016).

1.2 OBJETIVOS GERAIS E ESPECÍFICOS

1.2.1 Objetivo geral

O presente trabalho tem como objetivo geral desenvolver e acelerar um algoritmo simplificado do *ray tracing*, utilizando duas abordagens distintas, ambas com foco no emprego desta técnica em sistemas embarcados. A primeira abordagem trata do projeto de um coprocessador para a realização de uma parte crítica do algoritmo. Já a segunda abordagem envolve a utilização de um processador de propósito geral configurável.

1.2.2 Objetivos específicos

A partir do objetivo geral supramencionado, é possível elencar os seguintes objetivos específicos:

(a) Desenvolvimento do algoritmo de *ray tracing* em uma linguagem de programação de forma que seja possível gerar/renderizar uma imagem;

(b) Projeto de um Circuito Integrado de Propósito Específico[1] (ASIC) que acelere uma parte crítica do algoritmo;

(c) Utilização da arquitetura de um processador de propósito geral configurável para executar a aplicação do *ray tracing*, levando em conta características como área, potência e velocidade;

(d) Análise do desempenho de renderização de uma cena simples para ambas as abordagens e, de 4 cenas complexas no processador de propósito geral;

[1] Circuito Integrado de Propósito Específico, do inglês, *Application-Specific Integrated Circuit* (ASIC) é um circuito integrado construído com o intuito de um uso em particular, diferentemente de circuitos de propósito geral, como processadores de computadores.

(e) Realizar a comparação entre o trabalho desenvolvido pelo autor e o estado da arte da indústria e da academia.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho é dividido em 6 capítulos (incluindo esta introdução). O Capítulo 2 denominado Fundamentação Teórica, detalha tópicos essenciais para o correto entendimento desse trabalho bem como fornece um panorama geral dos trabalhos sendo desenvolvidos nesse meio. O Capítulo 3 detalha a metodologia aplicada ao trabalho, apresentando as ferramentas empregadas, o fluxo de projeto e de análise. O Capítulo 4 detalha as principais etapas do trabalho, tais como implementações e ponderações acerca das escolhas de projeto. O Capítulo 5 mostra os resultados obtidos, bem como realiza uma comparação de desempenho entre as implementações em ASIC e processador de propósito geral, além de por em perspectiva estes dados a outros trabalhos da literatura. Por fim, o Capítulo 6 traz as conclusões e possíveis direcionamentos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este Capítulo irá introduzir a técnica de *ray tracing*. Serão abordados aspectos teóricos relacionados com a interseção de objetos gráficos em duas e três dimensões, além de apresentar a questão matemática envolvida neste processo. Parte do desenvolvimento deste trabalho está apoiado em uma tecnologia capaz de permitir o projeto rápido do conjunto de instruções de um processador. Os principais pontos que demonstram a versatilidade do ambiente de projeto deste processador também serão mostrados.

2.1 *RAY TRACING*

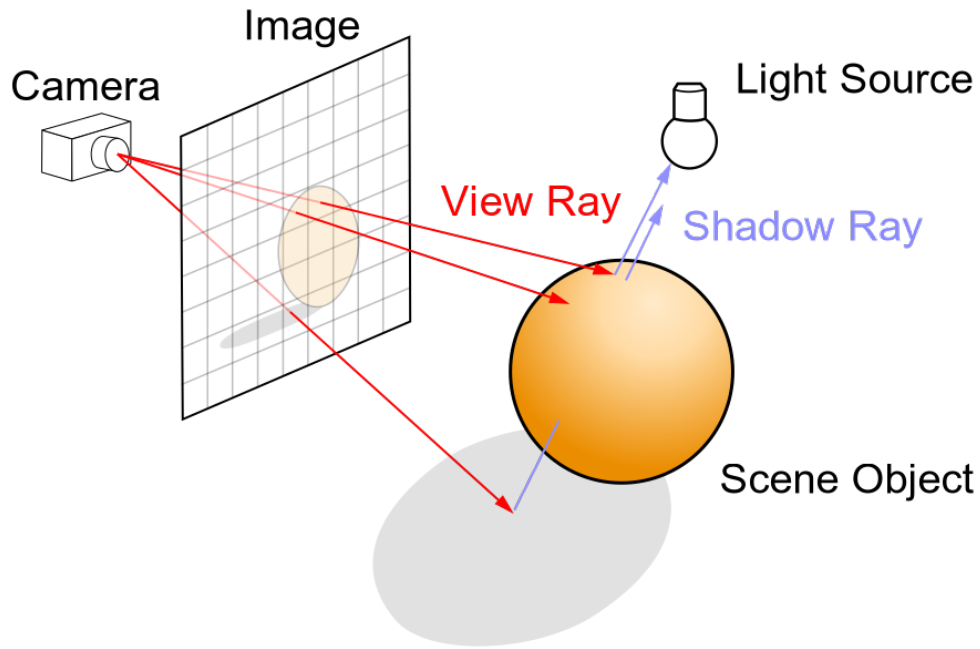
Primeiramente, deve ficar claro para o leitor que o *ray tracing* não é a única técnica de renderização existente. Apesar de hoje em dia estar em alta, ela não é a técnica mais amplamente utilizada, principalmente por demandar um alto poder de processamento. Como não se tinha esse poder, até os últimos anos, a técnica de renderização mais comumente aplicada era a rasterização, ou *rasterization* em inglês. Com os avanços do poder computacional dos circuitos eletrônicos, hoje é possível aplicar o *ray tracing* de maneira minimamente satisfatória para o público em geral.

O *ray tracing* nada mais é que uma técnica para síntese de imagens, ou seja, criar uma imagem de duas dimensões (2D) a partir de um ambiente de três dimensões (3D). Como em diversas outras técnicas e conceitos usados na ciência e engenharia, o *ray tracing* foi traduzido do mundo real, ou da natureza, para a computação. De forma breve, para cada pixel da tela é gerado um raio de luz que sai do olho do observador, entra no ponto central do pixel (podendo ter algum ângulo em relação à malha de pixels) e, é rastreado pelo ambiente 3D. Dessa forma, pode-se calcular os objetos que colidem com esse raio de luz, onde ele é refletido, refratado e outros fenômenos físicos que ocorrem com a luz. Tudo isso é baseado nas características físicas do material no qual o objeto 3D é feito, além de sua forma física.

Na Figura 1 temos uma imagem da aplicação dessa técnica. Pode-se observar que o raio de luz viaja da origem até colidir, ou não, com os objetos, sendo refletido em direção à fonte luminosa. Assim, o pixel ficará com uma intensidade luminosa maior se existir um caminho entre o pixel e a fonte de luz (reflexo da fonte de luz), ou ficará mais escuro se não existir um caminho direto, logo tendo sombra (levando em conta que o objeto é opaco e não existe refração da luz). Quando um raio de luz reflete em direção à fonte luminosa, é preciso levar em conta o ângulo para que se tenha

uma ideia de qual a intensidade da reflexão e, como essa cor afetará a cor final do pixel. Para isso é necessário fazer o produto escalar (*dot*) entre a normal do ponto de colisão no objeto e a direção desse ponto até a fonte luminosa. Caso o ângulo seja zero, temos uma reflexão perfeita, logo uma alta intensidade da reflexão. Já se o ângulo for alto (tendendo a 90 graus), temos uma reflexão inexistente e a intensidade tenderá à zero.

Figura 1 – Exemplificação do funcionamento da técnica do *ray tracing*



Fonte: (WIKIPEDIA, Ray Tracing 2020).

2.1.1 Um Pouco de História

A primeira vez que se pensou e que se tem registro do funcionamento/ideia análogo a técnica do *ray tracing* remonta há séculos. Esta ideia foi, nada mais nada menos, idealizada por Platão na Grécia Antiga, mais de 3 séculos antes de Cristo. Em seu texto/diálogo *Timaeus of Locri* escrito em 360 a.C, ele apresenta a especulação sobre a natureza do mundo físico e os seres humanos. Ao filosofar e descrever como a visão do ser humano deveria funcionar, ele descreve que dentro do nosso olho devemos ter uma chama que faz com que raios saiam dos nossos olhos em direção ao ambiente e assim, capturem a cor do ambiente para conseguirmos enxergar. Durante a noite essa chama se apaga e não conseguimos mais capturar as cores do ambiente e, por isso não vemos nada. (PLATO, 360 a.C)

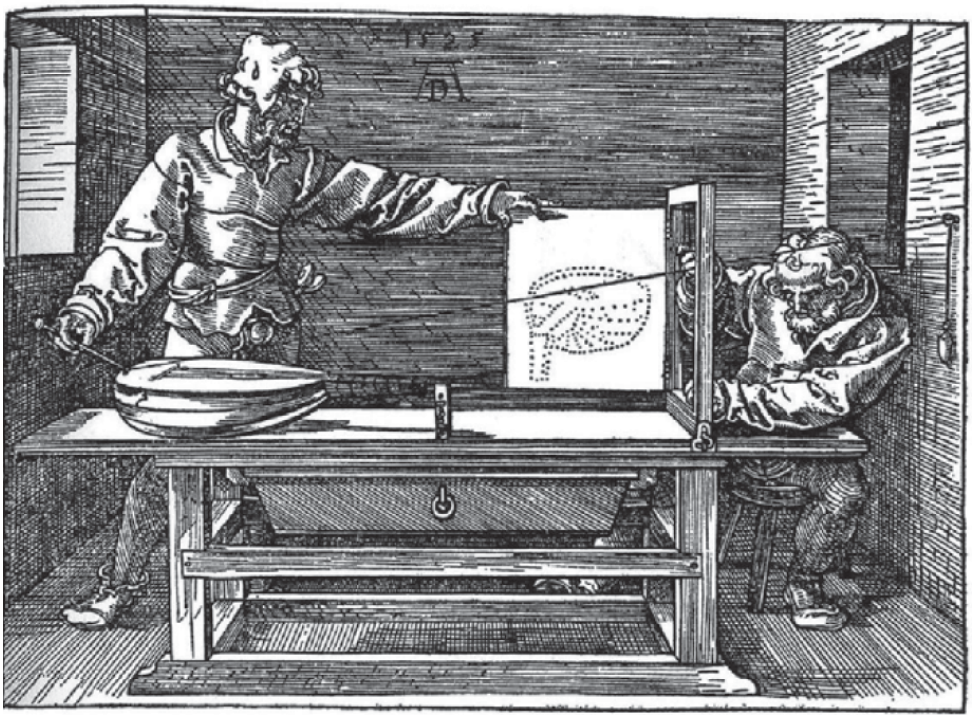
Hoje, essa ideia é totalmente irreal e sabemos que não é assim que a visão

funciona e, que na verdade os raios são fótons. Os fótons não saem dos nossos olhos, mas sim de uma fonte luminosa, por exemplo, o Sol. Os raios refletem nos objetos do ambiente e acabam entrando no nosso olho. Porém, a descrição de Platão é uma forma bem simplificada e análoga a ideia do *ray tracing*.

Já a segunda vez que temos registro do uso da técnica foi em 1525, por Albrecht Dürer. Dürer desenvolveu um aparato que demonstrava um método no qual qualquer pessoa poderia criar um desenho em perspectiva de praticamente qualquer forma. Na Figura 2 temos um exemplo de duas pessoas utilizando o mecanismo desenvolvido por Dürer para desenhar um alaúde.

O método nada mais era que a analogia do *ray tracing*, no qual uma corda era tensionada de um ponto focal (obturador) até um ponteiro que era encostado em algum lugar do objeto que estava sendo desenhado. Nos pontos em que a ponta da corda encostava no objeto, uma segunda pessoa mantinha um lápis no ponto exato em que a corda atravessava uma "janela"(que representava a visão da cena). A corda era colocada de lado e a "janela"era fechada, fazendo com que o lápis fizesse um único ponto no papel. Ao repetir o processo inúmeras vezes, com centenas de pontos, um desenho com grande fidelidade e perspectiva era formado. Assim surgia o primeiro "motor gráfico" da história.

Figura 2 – Duas pessoas utilizam o mecanismo de Albrecht Dürer para desenhar um alaúde



Fonte: (HUGHES et al., 2014).

Mas foi somente em 1968 que Arthur Appel propôs a primeira aplicação computacional para o *ray tracing*, a qual ficou conhecida e chamada de *ray castings*, que nada mais é do que um forma primitiva, mais simplificada e não recursiva, do *ray tracing* (APPEL, 1968).

Já em 1979, John Turner Whitted introduziu o *ray tracing* da forma como ele é conhecido hoje. Turner Whitted recebeu seu PHD da *North Carolina State University* em 1978 e logo depois ingressou na Bells Labs. O seu artigo publicado é considerado como um ponto de inflexão para a história da computação e, por esse motivo podemos considerar Turner Whitted como o pai do *ray tracing* como o conhecemos hoje (WHITTED, 1979).

2.1.2 Técnica do *Ray Tracing*

O conceito do *ray tracing*, como apresentando de forma breve na introdução, é realmente bem simples. Como desmastrado em pseudocódigo, no Algoritmo 1, o algoritmo nada mais consiste em duas iterações, onde para cada pixel é "disparado" um raio de luz na direção que a câmera está apontando, em seguida testando, para cada objeto, se esse raio colide e se existe um caminho direto até à fonte luminosa.

Algoritmo 1: Pseudocódigo do algoritmo do *ray tracing*.

```
1     for each pixel do
2         generate ray from viewpoint
3         for each object in the scene do
4             find closest intersection to the ray origin
5             if there is intersection then
6                 compute intersection point P
7                 shade(ray,P) //Aplicar cores e sombreamento;
8             end if
9         end for
10    end for
```

Como já comentado, a técnica mais utilizada atualmente é a da rasterização e a diferença entre ela e o *ray tracing* é muito sutil. Simplesmente invertendo as iterações temos a técnica da rasterização. Dessa forma, varremos o ambiente 3D por objeto e, para cada objeto, calculamos o seu impacto na cor final de cada pixel. Obviamente esta ideia apresentada para o algoritmo é bem simplificada e não está se levando em conta diversas técnicas para acelerar o mesmo, assim removendo a quantidade de interações. Também não está considerando técnicas que simulam o ambiente 3D

de forma mais precisa com o mundo real, e que acabam gerando uma imagem de qualidade muito superior e mais acurada da realidade. Todas essas otimizações são essenciais para se extrair o melhor do *ray tracing* e torná-lo minimamente utilizável. (HUGHES et al., 2014) (GLASSNER, 1989) Entre essas técnicas podemos citar:

- Técnicas para aumentar o desempenho:
    1. *Bounding-Boxes*;
    2. Hierarquia dos objetos;
    3. *Ray Coherence*;
    4. Otimização estatística;
    5. Paralelização e Vetorização.
- Técnicas para melhorar a qualidade da imagem gerada:
    1. *Shading*;
    2. *Tessellation*;
    3. *Spatial Anti-Aliasing*;
    4. *Texture-Mapping*;
    5. *Motion Blur*;
    6. Diversos efeitos Difusão Luminosa.
- Efeitos físicos que devem ser levados em conta para melhorar a simulação do ambiente:
    1. Efeitos da luz: Difusão, Refração, Irradiação, Efeito de Fresnel, etc.;
    2. Conceitos atrelados ao tipo do objeto: Rugosidade, Refletividade, etc.;
    3. Variação na percepção de cores dos nossos olhos.

2.1.3 Matematicamente Falando

Uma das principais e mais básicas dificuldades da computação gráfica é como definir o ambiente 3D, uma cena e um objeto. Para tanto e também para podermos processar tudo isso em computadores, utiliza-se massivamente da linguagem e estudos prévios da matemática. Isso implica que devemos descrever todo o ambiente de forma matemática, mais precisamente, em formato de equações matemáticas.

Para isso, partimos de objetos simples nos quais a matemática consegue descrever com perfeição. Caso o leitor queira explorar de forma mais aprofundada as definições e equações matemáticas, indicam-se os seguintes dois livros-texto: *Computer Graphics: Principles and Practice - Thrid edition* (HUGHES et al., 2014) e *An Introduction to RAY TRACING* (GLASSNER, 1989). E, para uma melhor compreensão do equacionamento adotado, definiremos previamente o padrão utilizado nas operações matemáticas como segue: letras maiúsculas para identificar vetores com 3 coordenadas cartesianas (x, y, z) e letras minúsculas para variáveis escalares.

2.1.3.1 Definindo o Ambiente

2.1.3.1.1 Raio de Luz

O raio de luz percorrendo um meio físico é o cerne de toda a técnica do *ray tracing*. Para a sua definição, levamos em conta o fato de um raio de luz sempre seguir uma trajetória em linha reta[1], logo definiremos um raio de luz com a equação da reta, representada pela Equação 2.1:

$$R(t) = O + D\mathbf{t} \tag{2.1}$$

onde O e D são vetores 3D de origem e direção, respectivamente, do raio de luz.

A equação determina a posição do raio de luz no ambiente 3D com coordenadas cartesianas (x, y, z), em função do tempo, após o seu disparo. t é o tempo, ou seja, em $t = 0$ o raio está na sua origem (ponte de criação) e ele se move na direção D com o avanço do tempo. Adotaremos o "$\cdot$" como o simbolo para produto escalar, "$\times$" para produto vetorial e "$*$" para multiplicação simples entre escalares. Para multiplicação entre vetor e escalar, optou-se por não colocar nenhum símbolo.

2.1.3.1.2 Plano

Agora que temos um raio definido, podemos partir para o ambiente em si, começando pelo plano. Ele é usado como base para os outros objetos, ou seja, o chão.

[1]Um raio de luz segue sempre uma linha reta no espaço-tempo. Desta forma, caso o espaço-tempo tenha sido deformado por um objeto massivo, como uma estrela ou buraco-negro, a linha reta será deformada e, para um observador num ponto distante verá a luz se curvando ao retor do objeto (seguindo um caminho curvilíneo), mas na realidade ela continua seguindo uma linha reta no espaço-tempo (que se encontra deformado perto do objeto).

A Equação 2.2 mostra a equação do plano mais conhecida. Já a Equação 2.3 apresenta uma forma mais elegante da equação, definida por um ponto Q e uma normal N. Um ponto qualquer de um plano X é definido e está contido no plano, quando ele satisfaz a Equação 2.3.

$$a * x + b * y + c * z + d = 0 \tag{2.2}$$

$$(X - Q) \cdot N = 0 \tag{2.3}$$

2.1.3.1.3 Triângulo

Um triângulo nada mais é do que um plano limitado por 3 bordas, ou seja, ele é muito similar à equação do plano, porém com algumas restrições. Essas restrições são os vértices do triângulo (A, B, C) e, por causa deles a equação fica um pouco mais complexa, visto que seus pontos devem estar contidos no plano formado por esses 3 pontos.

Para facilitar, podemos primeiramente nos concentrar a somente uma borda do triângulo, logo teremos que um ponto qualquer $Q = (1-t)A+tB$, onde $0 \leq t \leq 1$, está sobre o segmento de reta formado pelos pontos A e B. Já um ponto $R = (1-s)Q+sC$, onde $0 \leq s \leq 1$ está sobre o segmento reta formado pelos pontos Q e C.

Ao expandir, substituindo Q pela sua fórmula, temos a Equação 2.4, que define qualquer ponto dentro do triângulo.

$$R = (1 - s)(1 - t)A + (1 - s)tB + sC \tag{2.4}$$

Ao analisar os coeficientes de A, B e C, e seus limites, podemos chegar à conclusão que a soma de todos eles é igual a 1. Dessa forma, conseguimos definir um triângulo pela Equação 2.5.

$$P = \alpha A + \beta B + \gamma C \tag{2.5}$$

Um ponto qualquer P deve satisfazer a Equação 2.5 além de, $\alpha + \beta + \gamma = 1$ com $\alpha, \beta, \gamma \geq 0$. Sendo esses coeficientes chamados de coordenadas baricêntricas do ponto P em relação ao triângulo ABC.

2.1.3.1.4 Esfera

A Equação 2.6 define uma esfera, sendo a mesma determinada por um ponto central Q e o seu raio r. Um ponto qualquer X está na superfície da esfera quando satisfaz a equação.

$$(X - Q) \cdot (X - Q) = r^2 \tag{2.6}$$

O interessante desse objeto é que, diferentemente dos outros, ele é um objeto tridimensional. A equação da esfera cria um objeto fechado, ou seja ela possui volume. Dessa forma começamos a ter propriedades interessantes, como o interior do objeto, além de sua superfície.

2.1.3.2 Interseção do Raio

Como já apresentado anteriormente para determinar a cor do pixel, temos de saber se o raio, que saiu do olho do observador, colide ou reflete em algum objeto de forma a estar em uma linha reta com a fonte luminosa. Para que isso seja possível devemos determinar SE e ONDE o raio intersecta os objetos. Portanto, mais importante do que sabermos as equações das formas, é determinar as equações que nos indicam se houve ou não colisão entre o raio e o triângulo. Para isso contamos novamente com a álgebra linear, utilizando as equações de interseção raio-triângulo, raio-plano, raio-esfera, etc.

2.1.3.2.1 Interseção Raio-Plano

Como vimos na Equação 2.3, basta satisfazer a igualdade para que um ponto esteja no plano. Substituindo X pelo raio de luz (Equação 2.1) temos a Equação 2.7.

$$((O + D\mathbf{t}) - Q) \cdot N = 0 \tag{2.7}$$

Ao simplificar e resolver para **t**, teremos o “tempo” no qual o raio intersectou o plano, isso se ele realmente o intersectar. A Equação 2.8 apresenta isto de forma simplificada.

$$\mathbf{t} = \frac{(Q - O) \cdot N}{D \cdot N} \tag{2.8}$$

2.1.3.2.2 Interseção Raio-Esfera

A esfera, definida pela Equação 2.6, é um objeto fechado e convexo, logo existe um interior. Antes mesmo de entrar nas equações, podemos imaginar que teremos uma situação diferente em relação aos outros casos, visto que quando um raio colidir com a esfera ele terá 3 possibilidades:

1. Não colidir com a esfera;
2. Intersectar de forma tangencial, ou seja, passando de raspão e encostando em um único ponto da esfera;
3. Atravessar a esfera, assim tendo 2 pontos de interseção com o raio: um de entrada e um de saída.

Essa característica de resultado é típica de uma equação de segundo grau. Substituindo a equação do raio dentro da equação da esfera, e simplificando, temos a Equação 2.9.

$$(v \cdot v - r^2) + \mathbf{t}(2D \cdot v) + \mathbf{t}^2(D \cdot v) = 0 \quad \text{Para: } v = O - Q \tag{2.9}$$

Felizmente a fórmula quadrática nos indica facilmente as raízes para uma equação de segundo grau do estilo $c + b\mathbf{t} + a\mathbf{t}^2$. A fórmula quadrática nos diz que:

$$\mathbf{t} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.10}$$

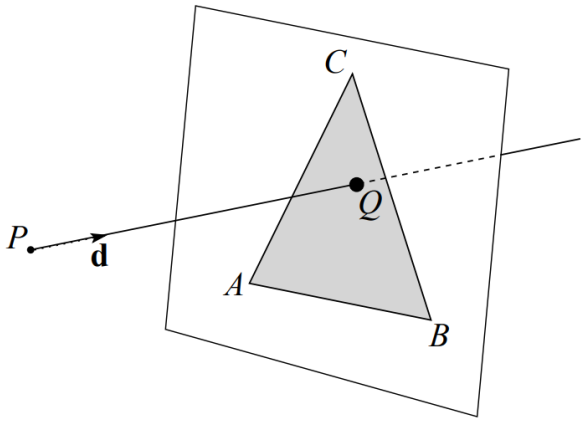
Pensando na aplicação do *ray tracing*, quando tivermos 2 raízes, teremos os tempos nos quais o raio entrou e saiu da esfera. Como neste trabalho estamos assumindo diversas simplificações, sendo que uma delas é que os objetos são sólidos e não translúcidos, não temos de nos preocupar com o ponto de saída do raio. Para interseções raio-esfera, que possuem 2 raízes, iremos sempre assumir a menor delas, considerando o primeiro ponto como o de colisão e de entrada do raio.

2.1.3.2.3 Interseção Raio-Triângulo

Como o triângulo é muito importante para a computação gráfica, serão apresentadas duas formas de computar a interseção raio-triângulo. Sendo a primeira mais intuitiva porém mais lenta que a segunda, que por ser mais otimizada foi utilizada nesse trabalho. A Figura 3 apresenta a situação na qual a reta é formada pelo raio

com seu ponto de partida P e direção **d**. O ponto Q é onde o raio intersecta o plano formado pelo triângulo. Será testado se Q está dentro do triângulo ABC.

Figura 3 – Esquematização de um raio intersectando um triângulo, o qual está posicionado sobre o seu plano implícito



Fonte: (CURLESS, 2006).

A forma mais intuitiva de computar a interseção raio-triângulo é realizada em dois passos:

1. Determinar se o raio intersecta o plano no qual o triângulo delimita e qual é esse ponto Q;

2. Testar se o ponto Q está dentro do triângulo ABC.

Neste primeiro passo é verificado se houve a interseção com o plano. Logo, temos de definir o plano no qual o triângulo está e, para isso basta determinar a normal do plano em um dos ponto dele. Como os três vértices do triângulo estão sobre o plano, conseguimos utilizar esses três pontos para determinar a sua normal. Sendo $\mathbf{n} = (B - A) \times (C - B)$.

Ao aplicarmos a Equação 2.7, determinamos o tempo **t** no qual o raio intersecta o plano. Nessa equação temos o ponto do plano Q, que pode ser substituído por A, B ou C do triângulo. Resolvendo a equação teremos o ponto Q de interseção do plano e teremos de testar se o mesmo está dentro do plano através da Equação 2.12:

$$\begin{aligned} [(B - A) \times (Q - A)] \cdot \mathbf{n} \geq 0 \\ [(C - B) \times (Q - B)] \cdot \mathbf{n} \geq 0 \\ [(A - C) \times (Q - C)] \cdot \mathbf{n} \geq 0 \end{aligned} \tag{2.11}$$

Como é possível perceber, o conjunto de equações acima calcula as coordenadas baricêntricas e realiza o teste para ver se $\alpha, \beta, \gamma \geq 0$.

Já a segunda forma de computar a interseção raio-triângulo foi retirada do artigo "*Fast, Minimum Storage Ray-Triangle Intersection*" (MöLLER; TRUMBORE, 1997) que, apesar de antigo, continua sendo considerado umas das formas mais eficientes de calcular até os dias de hoje. Por não ser muito intuitiva, não haverá detalhamento matemático igual ao que foi feito no primeiro caso, limitando-se a uma explicação breve dos conceitos utilizados.

De forma resumida, ao igualar a equação baricêntrica (Equação 2.5) com a do raio (Equação 2.1), simplificar e aplicar a regra de Cramer, ficamos com a Equação 2.12.

$$\begin{bmatrix} \mathbf{t} \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{2.12}$$

onde $E_1 = B - A$; $E_2 = C - A$; $T = O - A$; $P = D \times E_2$; $Q = T \times E_1$.

Da mesma forma das outras equações, ao determinar o **t**, temos o tempo em que o raio colidiu ou não com o triângulo. Utilizando esse método, e como já diz no nome do artigo (MöLLER; TRUMBORE, 1997), temos uma redução na quantidade de dados armazenados, visto que não há a necessidade de pre-computar e armazenar nada, além de ser relativamente mais rápido do que o outro método.

2.1.3.2.4 Dentro, Fora ou Sobre?

Destaca-se que, em todos os casos, resolvemos as equações para **t**, ou seja, em quanto tempo após ser disparado, o raio vai colidir com o objeto. Dessa forma podemos obter 2 cenários distintos. Pode ocorrer uma interseção e assim teremos um valor para o tempo. Caso o valor do tempo seja infinito, visto que nas equações temos denominadores e eles podem ser 0, teremos um tempo também infinito. Esse tempo infinito significa que o raio não colide no objeto, ou seja, ele viaja infinitamente sem colidir com o objeto, não havendo consequentemente um tempo exato para sua colisão. Existem os casos em que o raio está perpendicular com as formas. Para esses casos alguns testes/condicionais simples já identificam e podemos forçar um retorno do **t** como sendo infinito.

2.1.4 Objetos complexos

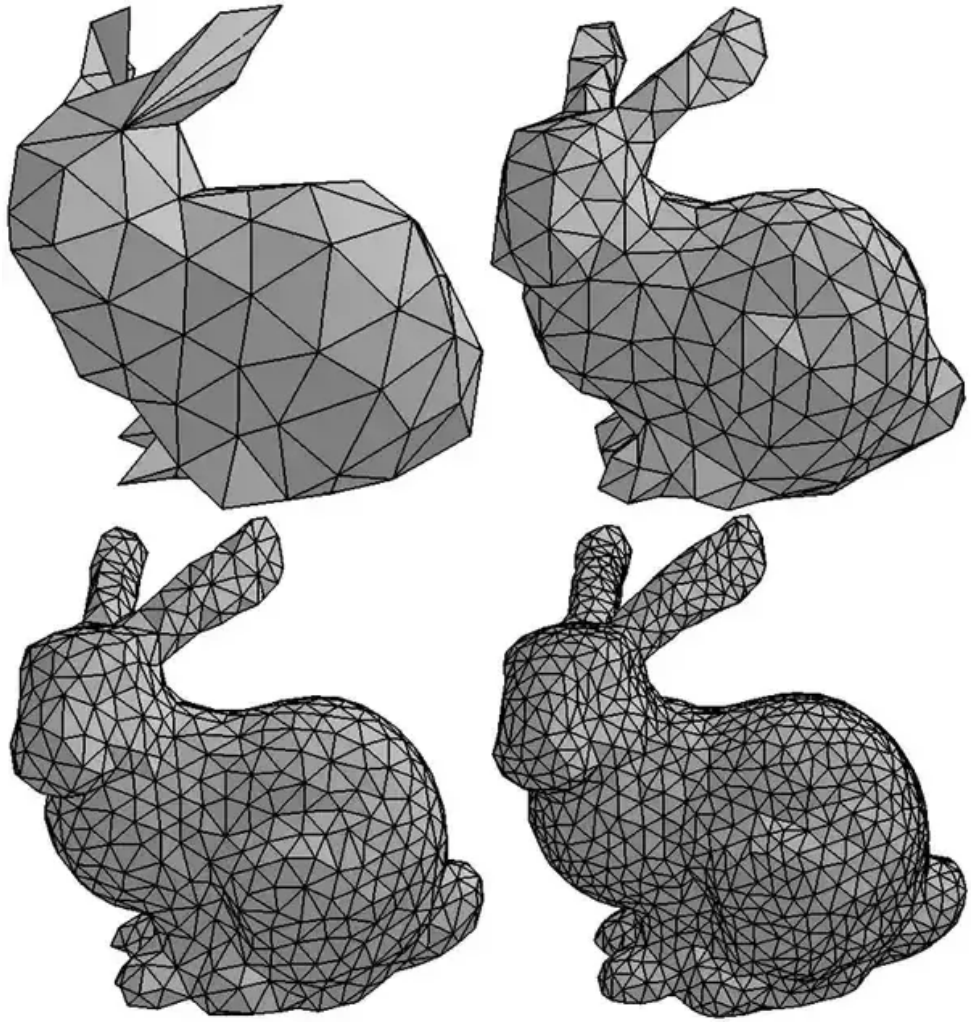
Como pode ser notado, o *ray tracing* e a computação gráfica como um todo se utilizam fortemente da álgebra linear para definir os cenários, mas existe uma limitação

quanto a esse uso. É desejável gerar imagens de cenas/objetos complexos, como os que são utilizados no dia a dia. E a primeira pergunta que se deve fazer é como podemos definir em equações o mundo ao nosso redor? Qual a equação matemática que descreve uma mesa, um carro, um coelho, um navio ou o corpo humano?

Certamente existe uma equação matemática única que descreve todos esses objetos no espaço 3D, só que é extremamente difícil de se chegar a essa equação. E para cada novo objeto demoraríamos muito tempo para o descrever. O que podemos fazer é descrevê-los como combinações de outros objetos mais simples e que sabemos trabalhar facilmente. A partir dos objetos simples apresentados anteriormente, principalmente o mais simples deles, podemos descrever qualquer objeto. Por esse motivo, o padrão adotado pela indústria é de descrever os objetos mais complexos principalmente por quadriláteros e triângulos. De forma mais precisa, conseguimos descrever todo e qualquer objeto complexo através de uma malha de polígonos, podendo existir milhares, milhões e até mesmo bilhões de polígonos por cena. (Bart Veldhuizen, 2012)

Na Figura 4 podemos ter uma clara visão da modelagem de um coelho através de uma malha de polígonos. Percebe-se que quanto mais polígonos são utilizados, maior a precisão e conseguimos descrever com mais detalhes o objeto.

Figura 4 – Modelagem de um coelho através de malhas com diferentes quantidades de polígonos



Fonte: (Harold Serrano, 2015).

2.2 ARMAZENAMENTO DE UM OBJETO

Como vimos, um objeto será definido por essa malha de polígonos, além das definições de formas mais simples. Como para qualquer outra área da computação, existe uma diversa gama de formatos voltados a armazenar informações, cada qual com a sua peculiaridade. Os formatos podem ser proprietários ou não, com ou sem compactação, otimizados para uma ferramenta/linguagem, com textura/cor anexa, com os movimentos do objetos e etc. Alguns exemplos são:

- STL (*Standard Tessellation Language*);
- OBJ (*Object*), formato aberto definido pela empresa *Wavefront Technologies*;
- FBX (Filmbox), formato proprietário desenvolvido pela empresa Kaydara, subsidiária da AutoDesk;
- COLLADA (COLLAborative Design Activity), originalmente criada pela empresa *Sony Computer Entertainment*;
- 3DS , formato usado pelo *software Autodesk 3ds Max*;
- IGES (Initial Graphics Exchange Specification), definida pelo governo dos EUA (*U.S. National Bureau of Standards*);
- STEP (Standard for the Exchange of Product Data, ISO 10303)

Vale um destaque maior para os dois primeiros formatos. O STL (STereoLithography), mais comumente referenciado como *"Standard Tessellation Language"*, trata de um formato muito utilizado por softwares de CAD (*Computer-Aided Design*), impressão 3D. Ele é um dos mais antigos formatos, criado em 1987 por Chuck Hull, atualmente CTO (*Chief Technology Officer*) da 3D Systems (Dibya Chakravorty, 2019).

Mas o maior destaque fica para o formato OBJ, desenvolvido pela Wavefront Technologies por volta de 1990, a qual disponibilizou as especificações do formato para difundir seu uso e prover maior interoperabilidade entre os softwares. O formato é reconhecido como aberto, ou seja, não proprietário desde a sua criação. Hoje em dia ele continua sendo amplamente utilizado, principalmente para impressão 3D de objetos, mas não suporta nenhum tipo de animação, sendo utilizado somente para objetos inanimados (Library of Congress, 2020).

O formato é baseado no formato ASCII (*American Standard Code for Information Interchange*, ou em português, Código Padrão Americano para o Intercâmbio de Informação) e não é compactado. Por não ser compactado, ou seja, ficar como um arquivo texto bruto o formato é extremamente pesado, principalmente para objetos

complexos que possuem muitos triângulos em sua composição, mapeamento de texturas e outras características. Existe uma versão mais compacta, sendo codificada binariamente, porém essa versão é proprietária (Dibya Chakravorty, 2019). De forma sucinta, o formato OBJ contém uma lista enumerada de vértices do objeto (pontos no espaço 3D) os quais são utilizados, de forma indexada, para construir as superfícies, podendo ser triângulos, quadrados, esferas e outras formas simples. Mas o formato não se limita a isso, contendo diversos outros pontos que não serão abordados. Caso haja interesse, recomenda-se a referência a seguir onde é possível encontrar um detalhamento com toda a especificação (Wavefront, 1992).

2.3 PROCESSADORES CADENCE® TENSILICA®

Os processadores Tensilica® da Cadence® são processadores de dimensões reduzidas, que permitem um alto grau de customização e são adequados para sistemas embarcados dos mais diversos tipos e aplicações. Existem arquiteturas exclusivas para processamento digital de sinais, processamento de áudio, vídeo, visão computacional entre outros (Cadence Inc., 2020). A Cadence® comercializa mais de 2 bilhões de unidades desses processadores por ano (Cadence, 2016), sendo utilizados por diversas gigantes da tecnologia em inúmeros projetos. Um exemplo é o projeto do *Hololens* da *Microsoft* (Chris Williams, 2016).

2.3.1 Processador Cadence® Tensilica® Xtensa® LX

O processador Cadence® Xtensa® LX é um processador RISC (*Reduced Instruction Set Computer*, ou conjunto de instruções reduzido) que possui como base cerca de 85 instruções, arquitetura de 32 bits, 32 ou 64 registradores, 5 ou 7 estágios de *pipeline* e uma ampla possibilidade de integrar co-processadores como, por exemplo, unidades de ponto flutuante de precisão simples ou dupla, DSPs, co-processadores dedicados para áudio e vídeo, entre outros.

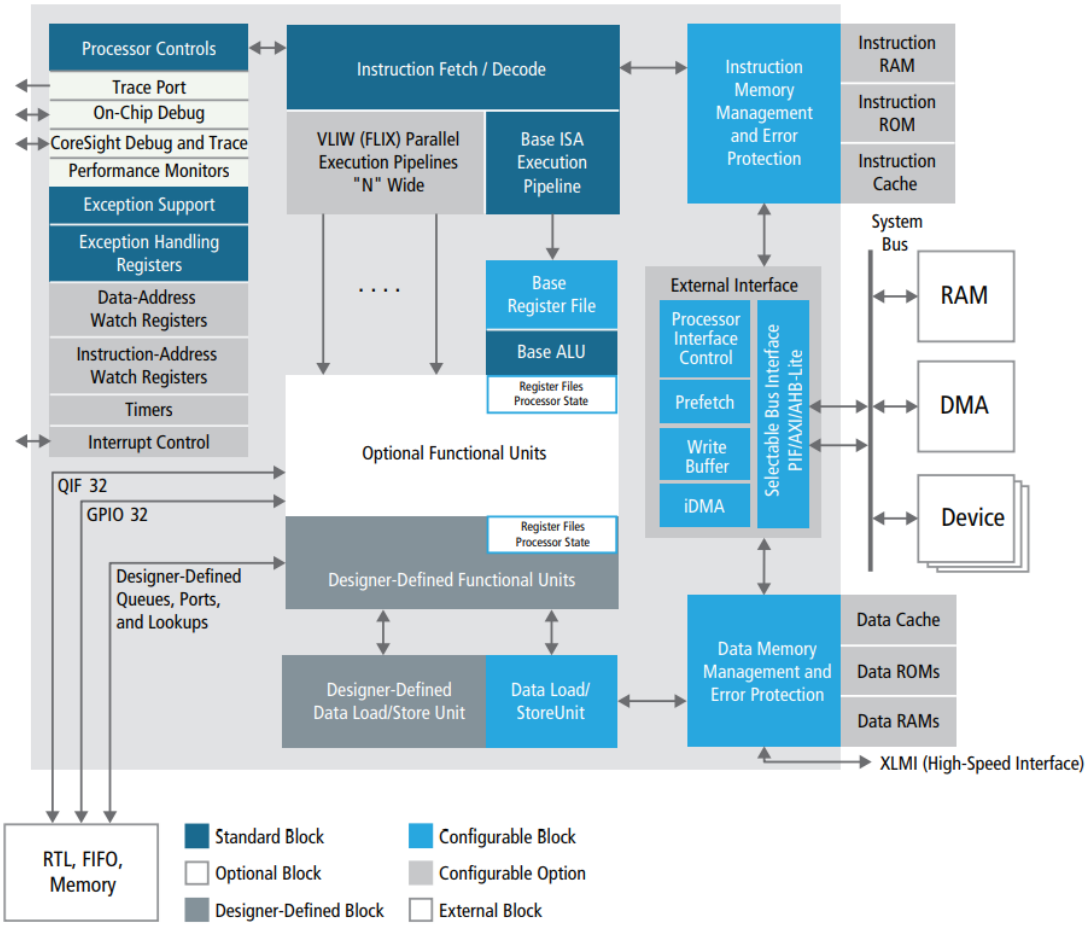
Contudo, a sua maior vantagem é a customização, sendo possível adicionar um hardware dedicado para determinada tarefa, desenvolvido pelo próprio engenheiro do projeto. Essa linha de processadores vem com compiladores em C e C++ e, usando a linguagem TIE[2] (*Tensilica Instruction Extension*), é possível adicionar hardware ao processador de modo simples e automático. Quando uma chamada operação em hardware é criada, a plataforma da Cadence® automaticamente cria instruções *as-*

[2]Esta linguagem será melhor detalhada na sub-seção 2.3.2.

sembly para ativar esse hardware e protótipos em C. Assim, é possível utilizar o que foi implementado em hardware diretamente no código de alto nível, sem ter de se preocupar com toda a integração e validação do mesmo dentro do *pipeline* e controle do processador.

A Figura 5 apresenta em alto nível o diagrama de blocos desse processador, além de proporcionar uma percepção sobre o quão versátil o mesmo pode ser e, os seus graus de customização. Nesse sentido, com a ampla gama de customizações diferentes para esse processador é possível criar diferentes versões do mesmo, as chamadas **configurações**. Uma configuração é uma organização específica do processador, na qual o número de registradores, tamanho do *pipeline*, frequência de operação, etc, podem ser escolhidos de acordo com a necessidade da aplicação.

Figura 5 – Arquitetura em diagrama de blocos do Xtensa® LX7



Fonte: (Cadence Inc., 2018).

2.3.2 TIE - Tensilica® Instruction Extension

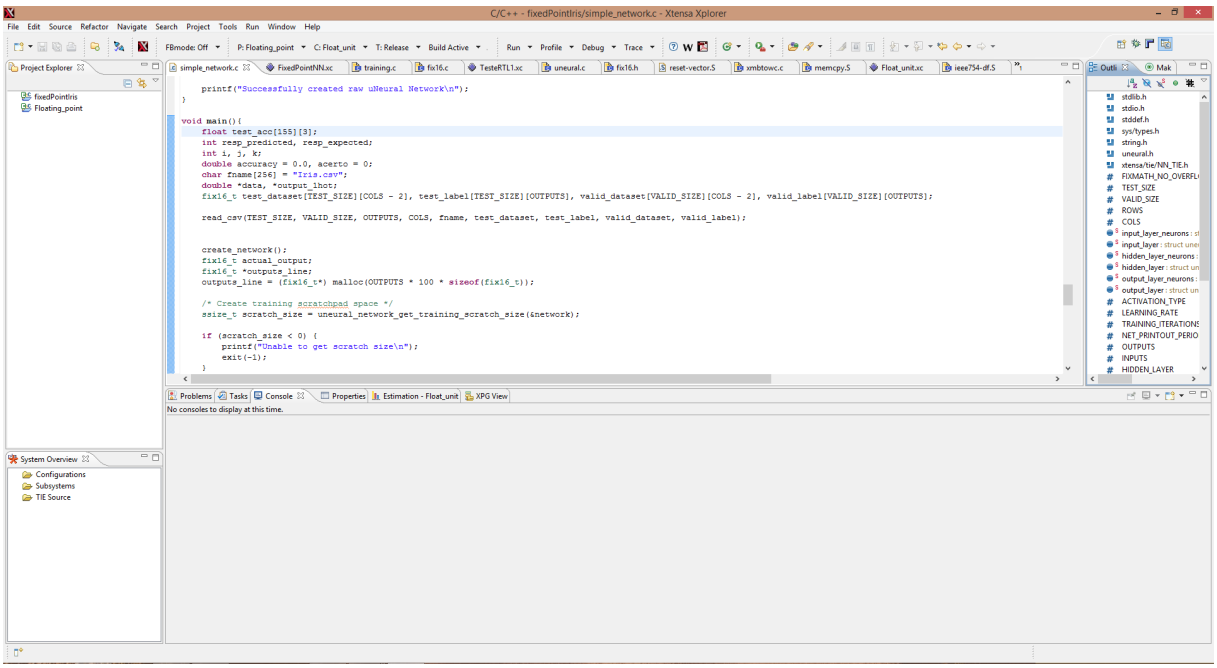
A *Tensilica® Instruction Extension* é uma linguagem proprietária da empresa Cadence® Inc., sendo empregada para descrever novas operações e o hardware que funcionará no processador Cadence® Xtensa® LX. Desse modo, quando um módulo é descrito em TIE, ele é automaticamente integrado ao ISA (*Instruction Set Architecture*) do processador. A linguagem é baseada em Verilog, sendo portanto muito parecida com esta. Com a TIE, é possível criar bancos de registradores com tamanho e número maiores do que a base do processador, criar multiplicadores, somadores e subtratores com uma largura maior ou menor de bits. Desse modo, é possível transformar operações de 32 bits em operações de 64 bits, por exemplo, permitindo expressivos ganhos em desempenho (CULAU et al., 2018).

2.3.3 Cadence® Xtensa® Xplorer

O *Xplorer* é um software proprietário da empresa Cadence® Inc. que possui uma interface gráfica baseada em Eclipse e um conjunto de ferramentas para análise de programas (*profile*), configuração de processadores, compilação de programas em C, C++ ou *assembly*, compilação do arquivo TIE, simulação e ferramentas de *debug*. A Figura 6 ilustra a interface básica do programa. Ele trabalha com diferentes visões, com objetivos específicos como *profile*, *debug*, execução e compartilhamento entre equipe, por exemplo. Assim, cada uma das visões oferece todo o conjunto de ferramentas para análise, correção e verificação do código.

Para que se realize a análise (*profile*) do algoritmo, o Cadence® Xtensa® Xplorer simula o *pipeline* do processador, permitindo ver quantos ciclos leva para a execução de uma aplicação, o número de ciclos de cada função, exceções que ocorreram (para chamadas de funções, por exemplo) entre outras informações. Essa ferramenta permite a simulação de 2 modos principais, com ou sem modelagem do atraso ao acesso à memória. O primeiro modo é o chamado *Fast Functional* e permite analisar o correto funcionamento do programa executado, não capturando exatamente a quantidade de ciclos utilizada. Esse modo, é extremamente mais rápido. O segundo modo é o *Cycle Accurate* e faz a modelagem e avaliação do *pipeline*. Ele é focado em descobrir a quantidade de ciclos gasta pelo programa de forma acurada, ou seja contando uma a uma. Por levar em conta todos os ciclos, esse modo é mais lento do que o anterior.

Figura 6 – Interface básica do Xtensa® Xplorer



Fonte: Cadence® Inc.

2.4 ESTADO DA ARTE

Como mencionado na Seção 2.1.1, a técnica do *ray tracing* é explorada há séculos. Por esse e outros motivos, existem trabalhos acadêmicos utilizando o mesmo na computação gráfica há décadas. Apesar de sempre ter sido utilizado, o tema ficou um pouco dormente para o público em geral, principalmente por não ser possível aplicar a técnica por uma limitação de desempenho dos computadores comuns. Porém, na indústria ele sempre se manteve ativo, sendo utilizado vastamente pelo setor de entretenimento, principalmente pelo cinema, mais focado no segmento de animação.

Alguns exemplos de animações são os famosos filmes da Pixar Animation Studios: Os Incríveis 2 (COLEMAN et al., 2018), Carros (CHRISTENSEN et al., 2006) e Procurando Dory (HERY; VILLEMIN; HECHT, 2016). Além da disso, o *ray tracing* também é usado para modelagem de fenômenos reais. Uma vez que a luz é uma onda e o *ray tracing* apresenta uma alta acurácia na representação de seus fenômenos, também é possível utilizar essa técnica para ondas usadas em comunicação *wireless*. Como exemplos disso existem os dois seguintes trabalhos: o primeiro utiliza o *ray tracing* para determinar a distribuição de um sinal *wireless* e assim encontrar a quantidade e a posição ideal dos transmissores (YUN; LIM; ISKANDER, 2008); já o segundo apresenta um novo algoritmo para que a computação seja feita em um tempo menor, visando modelagem de canais de comunicação (WANG et al., 2016).

Para o público em geral, inclusive para o autor deste trabalho, o tema ressurgiu

em 2018, quando a NVIDIA lançou suas novas placas gráficas (NVIDIA, 2018c). As novas *GPUs* possibilitaram a aplicação da técnica em tempo real em computadores comuns e, para o público em geral, o tema voltou à tona. Algo bem similar ao que aconteceu com o tema de inteligência artificial, redes neurais e afins. Por esse motivo vale um destaque especial para o trabalho realizado pela NVIDIA, visto que o estado da arte em termos de um hardware compacto e focado no *ray tracing* é da empresa[3].

Os hardwares da NVIDIA, lançados em 2018, formam as placas gráficas da família RTX, sendo a mais potente a *Quadro RTX 8000*. Ela é considerada a primeira GPU com suporte/aceleração de hardware para *ray tracing* em tempo real, atingindo até 10 GigaRays/sec[4] (NVIDIA, 2018c), sendo voltada para o mercado consumidor que produz animações, edição de vídeo e renderização em geral. Além dessa placa, também foram anunciadas GPUs para o consumidor final, assim podendo usufruir da tecnologia em jogos eletrônicos (NVIDIA, 2018b).

Utilizando essas GPUs é possível realizar os cálculos de *ray tracing* em tempo real, desse modo é possível renderizar cenas complexas, com várias fontes de luz em tempo real, mesclando *ray tracing* com *rasterization* e formando cenas com detalhes incríveis, como é possível ver nos vídeos: *Project Sol: A Real-Time Cinematic Scene Powered by NVIDIA RTX* (NVIDIA, 2018d) e *NVIDIA RTX and GameWorks Ray Tracing Technology Demonstration* (NVIDIA, 2018a). Já em 2020, a família foi renovada e uma nova série foi adicionada, com ainda mais poder de processamento, sendo esta a *Quadro RTX A6000* (NVIDIA, 2020), tendo um poder de processamento que chega na casa de até 20 GigaRays/sec. Contudo, pelo fato deste ser um valor máximo, é provável que o desempenho médio se encontre na casa dos 15 GigaRays/sec.

Já mais na área acadêmica, temos alguns trabalhos voltados para a otimização do *ray tracing* em hardware e em melhorias de implementação (software). Como exemplo, em *A Hardware Acceleration Engine for Ray Tracing* (GAO et al., 2014), é apresentado um hardware para realizar as operações necessárias para gerar toda uma imagem a partir de uma cena, ou seja um motor gráfico completo. Esse trabalho merece destaque pois apresenta um circuito completo, abordando vários aspectos para a aceleração do *ray tracing*. O sistema do trabalho citado alcança um desempenho máximo na casa dos 43,5 MegaRays/sec, isso com um hardware extremamente compacto de 16 mm^2 (no processo de 130 nm) e uma potência de 768 mW, operando em 500 MHz.

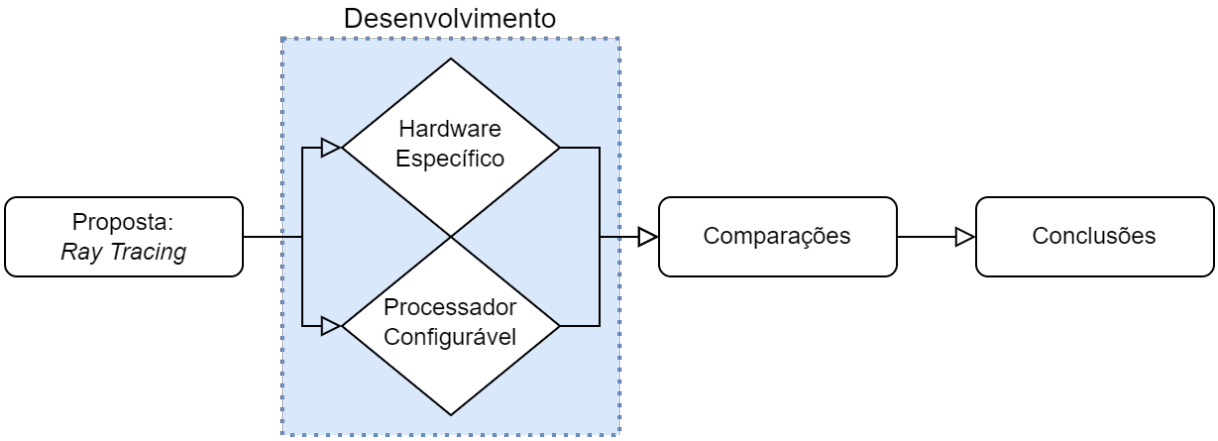
[3]Considerando um hardware único e específico para o *ray tracing*. A aplicação de supercomputadores, com petaFLOPS de desempenho, mas que ocupam salas inteiras, foi descartado. *FLoating-point Operations Per Second* é a métrica para medir a quantidade de operações em ponto-flutuante por segundo que um computador consegue processar.

[4]Métrica utilizada para medir quantos raios são processados por segundo por um determinado hardware.

3 METODOLOGIA

O desenvolvimento deste trabalho se divide em duas partes distintas. A primeira delas teve como foco a implementação de um circuito específico para melhoria do *ray tracing*, método que determina a intersecção raio-triângulo. Já a segunda parte trata da utilização do processador configurável Cadence® Tensilica® Xtensa® LX7 para executar o algoritmo, assim determinando os pontos críticos[1] (*hotspots*), cujas operações poderiam ser otimizadas através de módulos específicos do processador configurável. Desse modo, é possível realizar o ajuste fino[2] (*fine-tuning*) do processador e deixar preparado, para que no futuro, aplique-se otimizações mais específicas com a linguagem TIE. Este processo resulta em um fluxo em formato de diamante, onde ambas abordagens possuem uma origem em comum, distanciam-se durante o desenvolvimento e, encontram-se ao final, tendo as mesmas métricas para as comparações, conforme apresentado na Figura 7.

Figura 7 – Visão de topo para a divisão do fluxo adotada no desenvolvimento deste trabalho



Fonte: Autor.

Inicialmente foi utilizada uma implementação básica do algoritmo do *ray tracing* de modo a melhorar o entendimento do mesmo, mas sem entrar nos muitos detalhes da vasta área da computação gráfica. Para isso foi feita uma pesquisa em repositórios abertos na internet e em diferentes linguagens, sendo avaliados cada um deles. Ao final da avaliação, foi decidido usar um código em linguagem *Python* (ROSSANT,

[1] Pontos que demoram mais para executar e consomem uma grande quantidade de ciclos em relação a outras partes do circuito.

[2] Alteração de parâmetros da configuração do processador a fim de encontrar a melhor combinação que resulte num desempenho satisfatório, sem que o processador tenha nem área e nem consumo muito elevados.

2017). O código estava todo contido em único arquivo, de fácil manipulação e possuía quase todas as funcionalidades necessárias. Seu único ponto negativo era a falta da funcionalidade para triângulos. O código só trabalhava com o conceito de esferas e só possuía o método para intersecção raio-esfera, não tendo o conceito de triângulo e nem o método de interseção raio-triângulo.

O conceito de triângulo é o padrão da indústria, como foi apresentado na fundamentação teórica, no Capítulo 2. Por esse motivo ele é a base para produtos comerciais e para as pesquisas acadêmicas, sendo indispensável a sua implementação para esse trabalho. Desta forma, foi adicionado o conceito de triângulo no código, o método de interseção raio-triângulo, seguindo a Equação 2.12, e as funções necessárias para o correto funcionamento do método, como o produto escalar e produto vetorial. Desta forma, com o software em *Python* implementado, foi gerada uma primeira imagem utilizando a técnica do *ray tracing*. Através do devido posicionamento dos vértices de um triângulo, foi constituída uma cena onde se tem um plano base (chão) e o triângulo inclinado de maneira a ter sombra e reflexão no chão.

A partir desse ponto, temos a separação e distanciamento entre as abordagens de projeto, conforme mostrado na Figura 7. Para melhor entendimento do leitor, apresenta-se de forma cronológica o que foi feito e, sempre se referindo à primeira parte como **Implementação VHDL** e à segunda parte como **Implementação Xtensa**. Portanto cada parte seguirá independente até o final, onde os resultados serão comparados.

3.1 IMPLEMENTAÇÃO VHDL

Como o algoritmo realizava os cálculos com números em ponto flutuante, uma unidade de ponto flutuante[3] (FPU) era necessária para a realização das operações. Tendo em mente que a ideia do trabalho não era a de desenvolver uma FPU, e sim o algoritmo do *ray tracing*, uma busca foi realizada para encontrar uma FPU que suprisse todos os requerimentos:

- A FPU deveria ser capaz de realizar as operações de soma, subtração, multiplicação, divisão e comparação. Todas essas operações deveriam ser feitas entre dois números de ponto flutuante de precisão simples;
- A FPU deveria ser sintetizável, algo preferencialmente já confirmado pelo projetista original;

[3] A unidade de ponto flutuante, do inglês *Floating-Point Unit* (FPU), é um hardware específico para a realização/execução de operações matemáticas, como soma, subtração, multiplicação de dados representados em ponto flutuante.

- Caso houvesse empate, com todas as candidatas cumprindo os requerimentos anteriores, a FPU mais rápida (em termos de ciclos gastos para cada operação) seria escolhida.

Infelizmente não foi encontrada uma FPU que cumprisse todos os requisitos, mas foi observado que ao unir duas candidatas teríamos tudo que foi requerido. A primeira é a (Jidan Al-Eryani, 2017) feita em VHDL. Ela é uma FPU modular, que contém todas as operações aritméticas, menos o comparador. A outra encontrada foi a (Rudolf Usselmann, 2014) projetada em Verilog[4], que continha um comparador a parte, com precisão simples, mas todas as demais operações aritméticas em precisão dupla. A FPU final foi a união da FPU em VHDL com o comparador da FPU em Verilog.

3.2 IMPLEMENTAÇÃO XTENSA

Já com o algoritmo devidamente testado em *Python*, teve-se de converter/implementar o mesmo em linguagem C, visto que o Xplorer/Xtensa só compila/trabalha com linguagens C e C++, como apresentado na Seção 2.3. Desta forma, foram aplicados os conhecimentos sobre a linguagem para a implementação, sendo a maior dificuldade a implementação de métodos que em Python eram prontos e/ou adicionados por pacotes. Para tanto, utilizou-se de técnicas básicas como o uso de ponteiros, estruturas e criação de vetores dinâmicos para o armazenamento dos dados. Após essa conversão, o algoritmo foi validado, inicialmente no próprio computador, utilizando o compilador GCC[5] e confrontando os resultados com o Python. Além disso foram geradas algumas imagens teste e, validados os resultados finais da imagem (BitMap)[6].

Com a etapa do algoritmo concluída, partiu-se para a definição das configurações dos processadores, sendo feito o ajuste fino para se chegar a 4 configurações distintas (*Controle*, *Controle + FPU*, *Equilibrado*, *Max-Desempenho*). O objetivo é avaliar a execução do *ray tracing* em diferentes versões do processador, cada uma possuindo hardwares diferenciados com o intuito de estabelecer uma configuração de processador mais apropriada para a aplicação em foco.

[4]Verilog é uma linguagem de descrição de hardware (HDL) usada para modelar e descrever circuitos eletrônicos.

[5]GNU Compiler Collection (GCC) é um compilador produzido sob guarda do projeto GNU. Atualmente suporta diversas linguagens, mas foi inicialmente desenvolvido para compilar códigos em C, sendo sua primeira versão de 1987.

[6]BitMap é mapeamento de dados como inteiros para bits, geralmente em vetores ou matrizes. No contexto de uma imagem temos um *pixel-mapping* que nada mais é do que mapear cada pixel da imagem 2D (x,y) para a sua cor. Essa cor é descrita como a combinação de 3 cores distintas que conseguem formar todo o espectro visível, sendo elas Vermelho, Verde e Azul, em inglês *Red, Green and Blue* (RGB).

3.3 AVALIAÇÕES DE DESEMPENHO

A partir desta etapa, as duas abordagens de projeto se unem novamente a fim de realizar as avaliações de desempenho, renderizações de cenas e geração de métricas. A análise final foi feita utilizando 4 cenas distintas (Cena Padrão, Tea Pot, Bunny, Cube), que serão apresentadas com mais detalhes no decorrer do trabalho. Dessa forma foi possível comparar os dados de simulação utilizando uma gama maior de cenários, visto que o desempenho do *ray tracing* depende da quantidade de polígonos na cena.

Para comparação de desempenho foram utilizadas duas métricas:

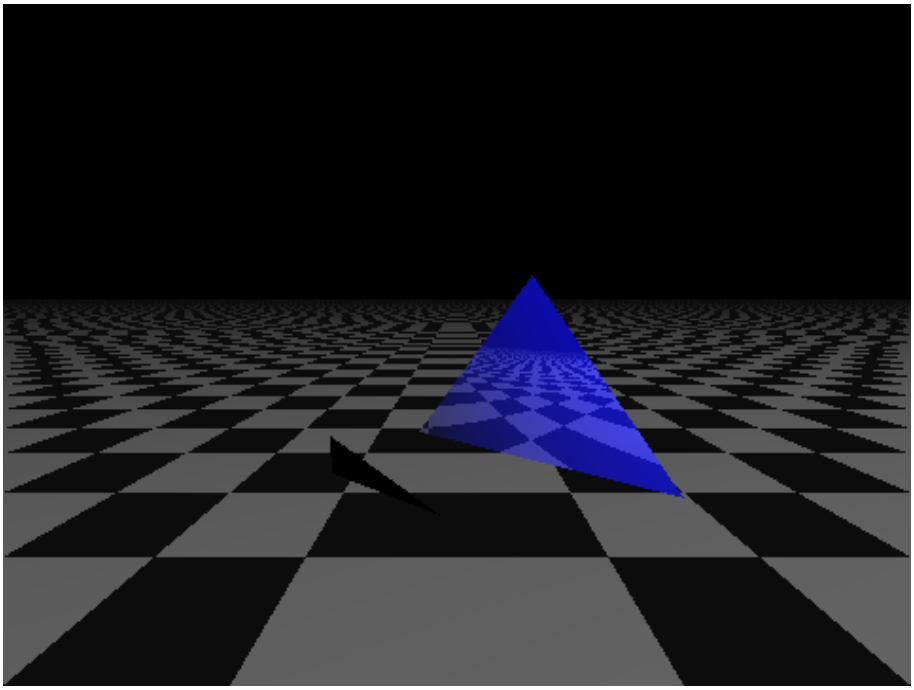
1. FoM (*Figure-of-Merit*): Relação entre área, tempo e energia dos circuitos;
2. Rays/sec: Relação de vazão do circuitos, ou seja, quantos cálculos de raios (na média) o circuito consegue realizar num período de 1 segundo.

4 DESENVOLVIMENTO DO TRABALHO

Com a metodologia estruturada, o próximo passo é o desenvolvimento do trabalho em si. É importante ressaltar, novamente, que o trabalho foi dividido em duas partes e, por isso o desenvolvimento naturalmente também terá uma separação adiante. O primeiro passo foi alterar o código Python para suportar a renderização de um triângulo. Após as alterações necessárias e a implementação dos métodos para o cálculo da interseção raio-triângulo, foi definida a cena simples. A Figura 8 apresenta a imagem gerada pelo software com a cena descrita anteriormente.

- Plano: Chão, definido por:
  - Ponto: (0.0, -0.5, 0.0)
  - Normal: (0.0, 1.0, 0.0)
- Triângulo: Único triângulo da cena, de cor azul, definido por:
  - Vértice V0: (-0.1, 0.0, 0.2)
  - Vértice V1: (0.4, 0.0, -0.2)
  - Vértice V2: (0.15, 0.4, -0.1)

Figura 8 – Imagem gerada utilizando somente o software



Fonte: Autor.

4.1 IMPLEMENTAÇÃO VHDL

O foco da Implementação VHDL foi restrito a uma única função/método, visto que pela demanda de tempo de projeto não seria possível implementar todo o algoritmo. Limitou-se à implementação do circuito para detectar a interseção raio-triângulo, utilizando a equação de (MöLLER; TRUMBORE, 1997), vista no Capítulo 2. A Tabela 1 apresenta a funcionalidade de topo do circuito ASIC: recepção dos dados digitais, processamento dos mesmos e retorno do resultado.

Tabela 1 – Etapas do funcionamento do circuito, com os dados que devem ser inseridos e os dados de saída do circuito

Função	***Direção***
1	Recebe sequencialmente cada uma das dimensões dos vetores 3D: **O, D, V0, V1, V2** OBS: Cada dado deve ser acompanhado pelo sinal *data_av* indicando sua disponibilidade.
2	Habilita a máquina de estados transmitindo mais uma vez o sinal *data_av*.
3	Computa a operação de interseção do raio com o triângulo. OBS: Retorna o instante no qual ocorre a interseção ou +INF caso não ocorra.
4	Retorna um sinal de DONE indicando o fim da operação.

Fonte: Autor.

4.1.1 Diagrama de topo VHDL

A Tabela 2 apresenta os pinos de entrada e saída do circuito. Cada valor escalar utiliza números em ponto flutuante de 32 bits (precisão simples), logo cada dimensão tem essa quantidade de bits. Para que não se tenha uma quantidade enorme de pinos de entrada, foi determinado um padrão para a entrada de valores. Cada vetor entra de forma sequencial, sendo enviada dimensão por dimensão para o circuito. A ordem dos vetores foi detalhada na Tabela 1, sendo **O**, **D**, **V0**, **V1**, **V2**. Para cada vetor, foram enviadas as dimensões seguindo a ordem: X, Y, Z.

Com esse padrão de envio, temos somente uma entrada de 32 bits, reduzindo a quantidade de pinos de entrada. Como a saída também é um *float* e, o *design kit* da IBM CMOS 7RF (CMRF7SF) (IBM, 2010) não suporta *tri-states*[1], ou seja, não suportando uma porta bi-direcional, dessa forma foi criada uma saída de dados de 32 bits. O circuito terá um total de 69 pinos, sendo 64 para entrada e saída de dados, 3 para controle, 1 para o *reset* e 1 para o *clock*. A Figura 9 apresenta o diagrama de topo do circuito com as entradas e saídas descritas anteriormente.

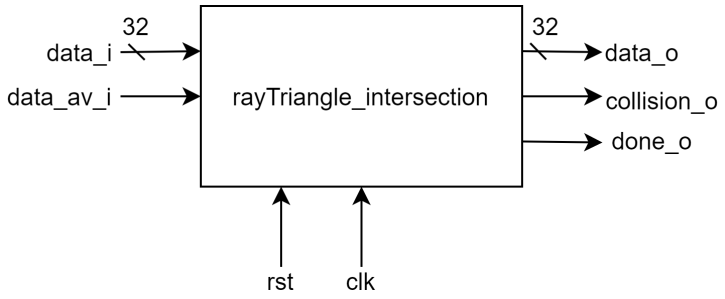
[1]Circuito digital que permite 3 estados na sua saída. Além dos binários 0 e 1 lógicos, também permite alta impedância (Z). Esse circuito é ideal para desacoplar o que vem antes dele do segue o mesmo, sendo muito utilizado quando múltiplos circuitos precisam usar uma mesma linha de sinal.

Tabela 2 – Especificação das portas do circuito topo em VHDL

Nome	**Direção**	**Tamanho (bits)**	**Descrição**
clk	in	1	*clock* global
rst	in	1	*reset* global
data_av_i	in	1	indica que o dado na entrada é válido
data_i	in	32	entrada das dimensões X, Y e Z dos vetores
done	out	1	indica que o cálculo terminou
collision_o	out	1	indica se houve ou não uma colisão
data_o	out	32	saída do resultado (instante de tempo)

Fonte: Autor.

Figura 9 – Diagrama de topo do circuito *rayTriangle_intersection*, apresentando suas entradas e saídas

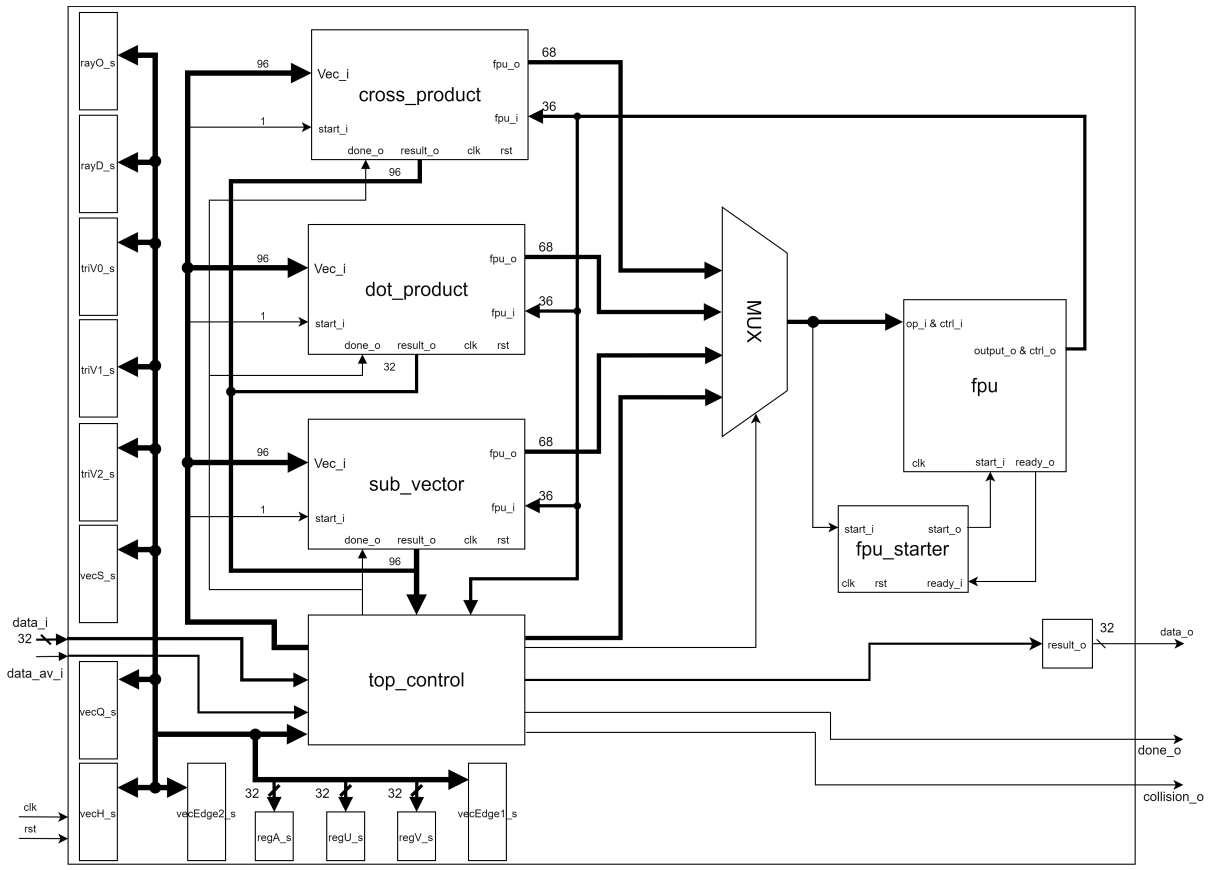


Fonte: Autor.

4.1.2 Diagrama de blocos e Interconexões

O circuito *rayTriangle_intersection* (topo) instancia vários sub-circuitos, os quais auxiliam e simplificam o controle das operações. Dentre esses sub-circuitos há a FPU, a qual foi selecionada e descrita anteriormente no Capítulo 3, além de circuitos que implementam um sub-controle para agrupar algumas funções matemáticas aplicadas sobre os vetores de entrada. O circuito de topo implementa um controle de forma comportamental e, por isso foi abstraído em um bloco chamado *top_control*. Deve ficar claro para o leitor que esse bloco não existe na realidade e não é instanciado no circuito de topo, sendo somente usado no diagrama de blocos para representar o controle comportamental. O mesmo vale para o barramento bi-direcional que liga o *top_control* aos registradores, esse sendo uma representação que existem troca de dados entre os registrados e o controle. A Figura 10 apresenta o diagrama de blocos do circuito.

Figura 10 – Diagrama de blocos do circuito *rayTriangle_intersection*, contendo todos os blocos instanciados e o bloco *top_control*, o qual é uma abstração do controle comportamental do topo, não existindo na realidade.



Fonte: Autor.

É possível identificar na Figura 10 a existência de subcircuitos que realizam as operações básicas de manipulação de vetores, visto que tais operações são necessárias para os cálculos da intersecção, como foi apresentado no Capítulo 2:

1. *sub_vector*: subtração de dois vetores;
2. *dot_product*: produto escalar entre dois vetores;
3. *cross_product*: produto vetorial entre dois vetores;
4. *fpu_starter*: circuito simples necessário para transformar o comando de *start* em um pulso para a FPU;
5. FPU: responsável por realizar as operações básicas de adição, subtração, multiplicação, divisão e comparação entre dois valores em ponto flutuante de 32 bits.

Como o circuito apresenta somente uma FPU, visando à redução de área, os subcircuitos compartilham a mesma FPU através de um multiplexador. Assim, cada

um assume o controle da mesma quando a está usando, determinando qual operação ela deve executar. Cada subcircuito possui sua máquina de estados, bem como o circuito de topo e a FPU. Os subcircuitos assumem que estão em comunicação direta com a FPU ao serem ativos, por isso enviam dados e esperam retornar o resultado. O controle do multiplexador que gerencia essa comunicação direta é feito pelo circuito de topo.

4.1.3 Máquina de estados

Como já comentado, o circuito projetado em HDL implementa o fluxo, tal qual o apresentado pelo algoritmo presente na Figura 11. O mesmo calcula a interseção do raio-triângulo com auxílio dos subcircuitos e da FPU. É realizada a carga inicial dos vetores de forma sequencial, controlando o multiplexador e a escrita de todos os registradores apresentados na Figura 10, comandando os subcircuitos e, realizando algumas operações específicas na FPU. Por realizar todas as funções do algoritmo apresentado na Figura 11 de forma comportamental, a sua máquina de estados ficou extensa. Desta forma, será apresentada de forma dividida, em blocos de funcionalidade. No Anexo A segue a imagem completa de toda a máquina de estados com os seus 36 estados, sendo quase metade dedicada à carga dos registradores.

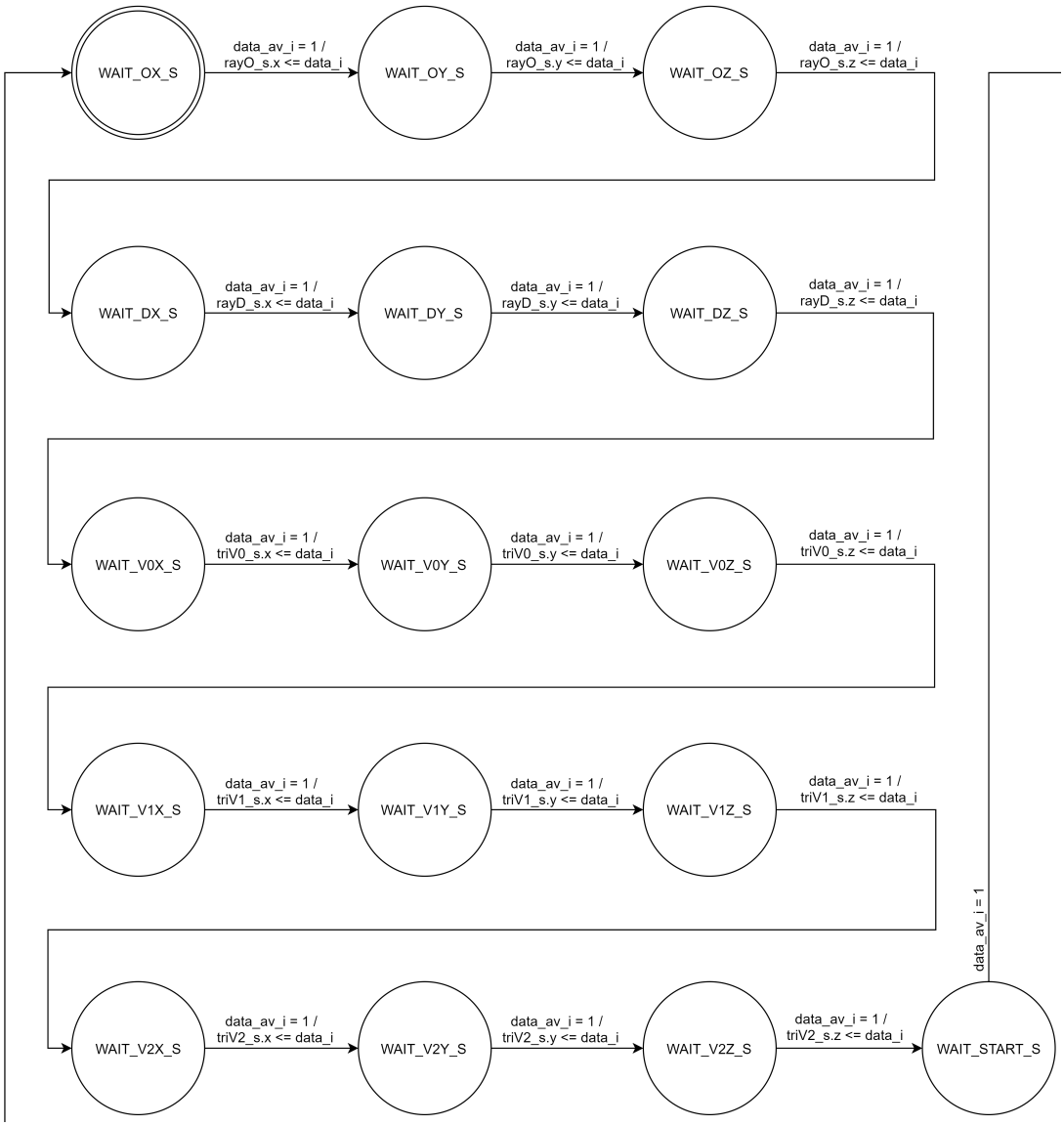
Figura 11 – Software implementado em *Python*, servindo como modelo para o projeto do hardware

```
#Recebe um triangulo e um raio.
#Resutla no tempo em que ocorre a collision ou +INF se não ocorrer.
def intersect_triangle(O, D, V0, V1, V2):
    EPSILON = 0.0000001

    # Determinar o vetor nas dias bordas que compartinlam o V0.
    edge1 = V1 - V0
    edge2 = V2 - V0

    # Calculo do determinante.
    h = cross_product(D, edge2)
    a = dot_product(edge1, h)
    # Se o determiannte for proximo de 0, o raio não colide com o plano.
    if (a > -EPSILON and a < EPSILON):
        return np.inf
    #Resolver equação
    f = 1.0/a
    s = O - V0
    u_dot = dot_product(s, h)
    u = f * u_dot
    if (u < 0.0 or u > 1.0):
        return np.inf
    q = cross_product (s, edge1)
    v_dot = dot_product(D, q)
    v = f * v_dot
    if (v < 0.0 or (u+v) > 1.0):
        return np.inf
    #Computar o t para verificar se resultou em collision.
    t_dot = dot_product(edge2, q)
    t = f * t_dot
    #Verificar se acolisao ocorre a intersection ou se so exite uma linha.
    if (t > EPSILON):
        return t
    else:
        return np.inf
```

Fonte: Autor.

Na Figura 12 têm-se os primeiros 16 estados, sendo os 15 primeiros para carga dos registradores, recebendo os vetores de forma sequencial. O 16º estado espera uma liberação para iniciar o cálculo.

Figura 12 – Máquina de estados com os 16 estados do circuito *rayTriangle_intersection*, onde há o carregamento de todos os dados de entrada, ou seja, todas as três dimensões dos cinco vetores

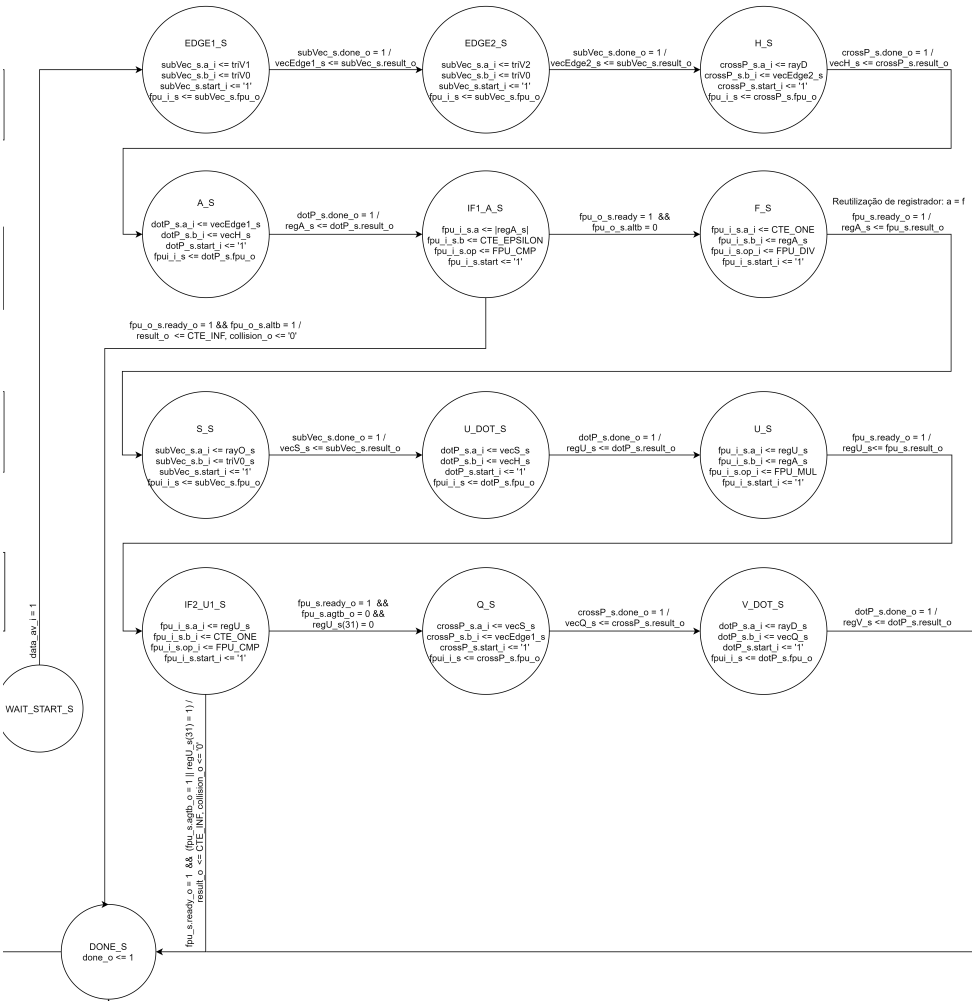


Fonte: Autor.

Na Figura 13 há os estados do 17º até o 28º e o último estado (36º, DONE_S). Esses estados são a implementação do algoritmo, sendo o estado 17º (EDGE1_S) a segunda operação do algoritmo da Figura 11 (a primeira sendo a atribuição). Nesse estado o sub-circuito *sub_vector* é ativo para que seja feita a subtração de vetores que resulta em: $edge1 = V1 - V0$. Nesse bloco já está sendo realizado o algoritmo, sendo que o nome do estado indica a operação equivalente do algoritmo. Os estados que têm ramificações representam condições (IFs) do algoritmo.

Outro exemplo é o 19º (H_S), onde a quarta operação do algoritmo é realizada. O subcircuito *cross_product* é ativado para que seja feito o produto vetorial que resulta em: $h = cross\_product(D, edge2)$. Os estados que contêm duas ramificações, como o 21º estado (IF_A_S), implementam comparações, sendo os IFs do algoritmo. As ramificações que vão para o estado DONE_S indicam que o IF era verdadeiro e o raio não colidiu com triângulo, logo o registrador de resultado recebe infinito (+INF).

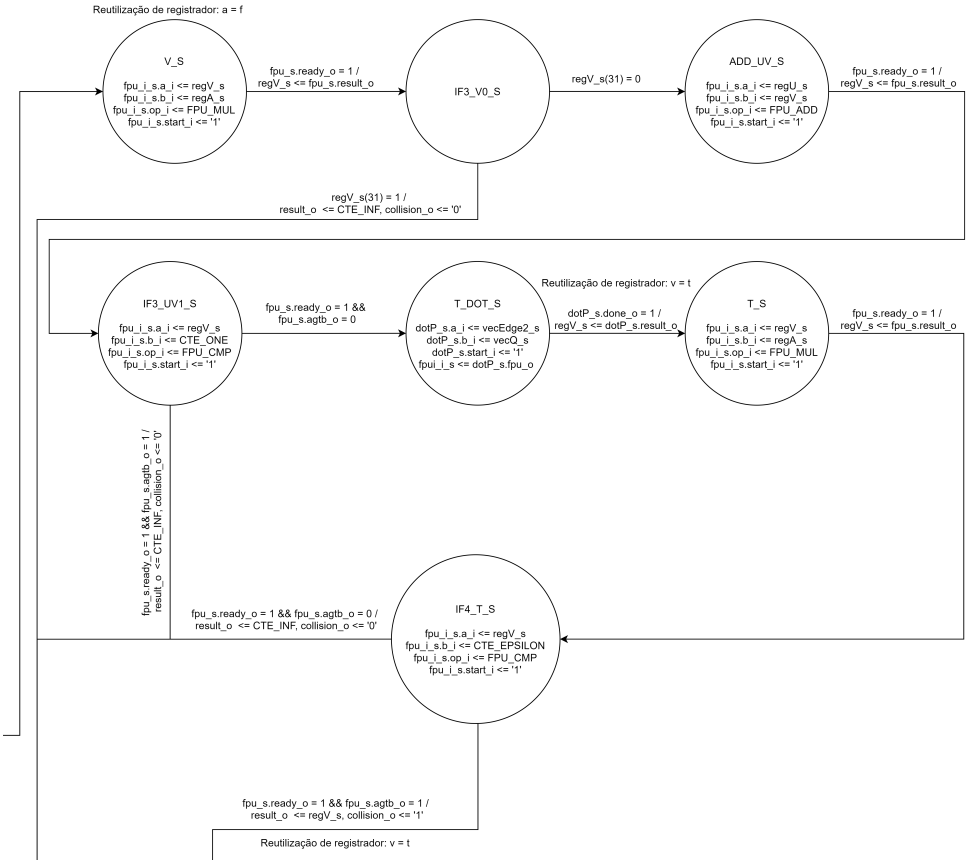
Figura 13 – Máquina de estados com os estados do 17º até o 28º e o último estado DONE_S do circuito *rayTriangle_intersection*



Fonte: Autor.

Na Figura 14 temos os estados do 29º até o 35º. Esses estados implementam a parte final do algoritmo da Figura 11, contendo os últimos testes para verificar se houve colisão do raio com o triângulo. O estado mais importante desse bloco é o 35º estado (IF4_T_S), o qual verifica se o raio colide e, diferentemente dos outros IFs, faz o registrador de resultado receber o instante da colisão e não mais infinito. Independentemente do resultado todos eles seguem para o último estado (DONE_S).

Figura 14 – Máquina de estados com os estados do 29º até o 35º do circuito *rayTriangle_intersection*



Fonte: Autor.

4.2 IMPLEMENTAÇÃO XTENSA

Diferente da seção anterior, onde apenas uma parte de todo o algoritmo foi projetada em hardware, a utilização de um processador de propósito geral permitiu desenvolver todo o algoritmo em *Python*. Porém, como já comentado, o Cadence® Xtensa® e o Cadence® Xtensa® Xplorer apenas suportam a compilação em código C. Logo, a primeira tarefa foi converter o código para essa linguagem.

4.2.1 Implementação do algoritmo em linguagem em C

A conversão demandou um certo esforço visto que a linguagem C possui um nível de abstração menor em relação ao *Python*. Muitas das funções utilizadas no código original não eram implementadas pelas bibliotecas padrões do C. Uma nova estrutura de vetores foi definida, além de funções que foram feitas para manipulá-los. Diferentemente do *Python*, a implementação em C ficou muito mais estruturada, visando a um melhor entendimento do código. Para isso foram desenvolvidas estruturas para os objetos, como segue na Figura 15. Isso acabou facilitando em diversos momentos para diminuir a complexidade das funções e a quantidade de parâmetros, sendo uma boa prática de desenvolvimento, além de buscar uma melhor alocação dos dados na memória (o que ocorreu de forma contínua).

Uma das principais partes do desenvolvimento do algoritmo em C foram os testes. Foram necessários aplicar testes unitários para cada método, desde o mais básico, até o método final do *rayTracer*. Dessa forma foi garantido que o código em C apresentava os mesmos resultados do *Pyhton*. Como a implementação em C, é muito mais rápida do que a em *Python*, foi acrescido um novo método para a leitura de um arquivo objeto, como visto no Capítulo 2. Para isso, utilizou-se do padrão "Wavefront.obj" (Library of Congress, 2020). A implementação foi feita de maneira muito simples, lendo o arquivo linha a linha, convertendo os valores de caracteres em números *float* e armazenando cada conjunto de valores (vértices) em um *array* de um tipo vértice que contém as três posições.

Figura 15 – Exemplo das estruturas criadas na implementação em *C* do Software

```
//Define a Triangle (vertice0, vertice1, vertice2, color)
typedef struct
{
    vectorT v0;
    vectorT v1;
    vectorT v2;
} triangleT;

//Define a Sphere (position, radius, color)
typedef struct
{
    positionT position;
    valueT radius;
} sphereT;


//Define a Plane (position, normal)
typedef struct
{
    positionT position;
    vectorT normal;
} planeT;

//Define a Ray (Origin, Direction)
typedef struct
{
    positionT origin;
    vectorT direction;
} rayT;

//Define an Object (type and data)
typedef struct
{
    objectType type;
    void *p;
    colorT color;
    valueT reflection;
    valueT diffuseC;
    valueT specularC;
} objectT;
```

Fonte: Autor.

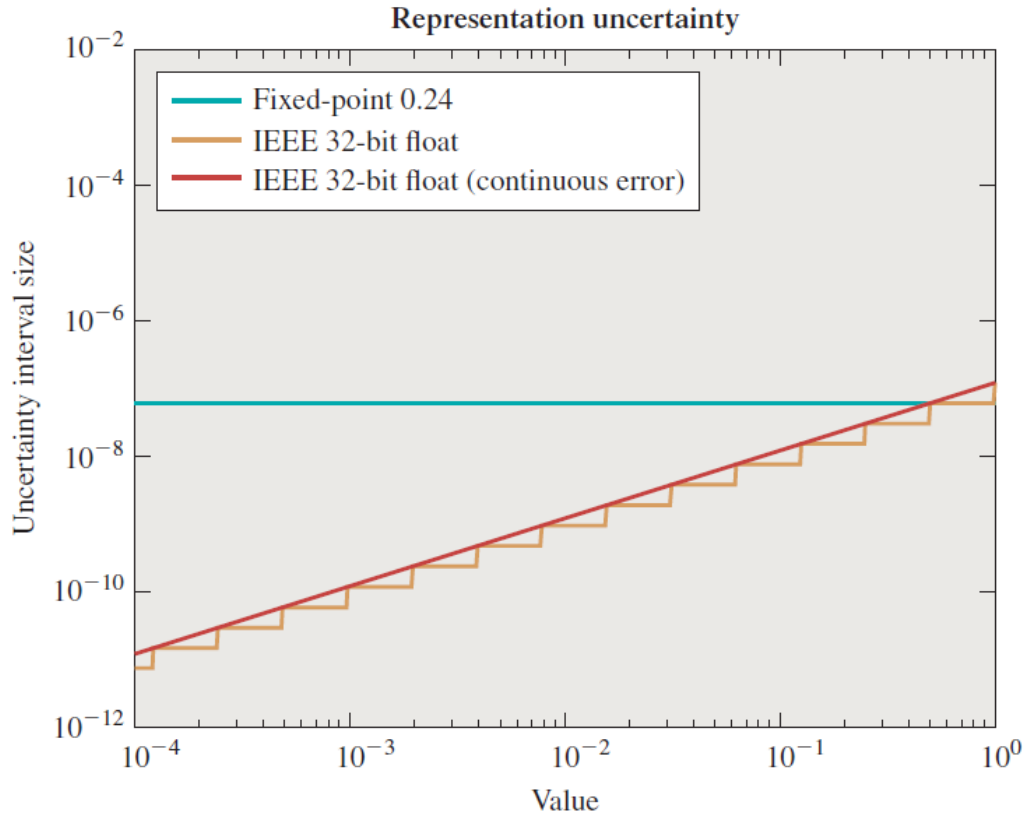
4.2.2 Ponto Fixo vs Ponto Flutuante

Uma dúvida que pode surgir para o leitor é o porquê do uso do ponto flutuante e não do ponto fixo. Para isso será dada uma breve explicação sobre os motivos da escolha. Existem muitos trabalhos que apresentam a utilização do ponto fixo para no uso do *ray tracing* e da computação gráfica em geral. Alguns exemplos são: "*Towards Hardware Ray Tracing using Fixed Point Arithmetic*" (Hanika; Keller, 2007); "*A fixed-point 3D graphics library with energy-efficient cache architecture for mobile multimedia systems*" (Min-wuk Lee et al., 2005); "*Comparison metrics for the output of a mixed fixed point and floating point graphics pipeline*" (Sicoe; Popa, 2017).

Existe, contudo, uma grande desvantagem do ponto fixo para a utilização nessa aplicação. Como muitas das cenas possuem objetos pequenos e próximos ao centro, os valores dos vetores e vértices ficam em um intervalo entre 0 e 1. E, considerando que cada unidade é um metro, muitas vezes as dimensões ficam na casa de milímetros (10^{-3}) ou em casos de objetos que se movimentam e colidem, a distância entre

os objetos tende a zero, portanto, magnitudes de micro-metros (10^{-6}), começam a ser relevantes. Para esses cenários o ponto fixo começa a apresentar um erro, ou incerteza, na ordem de magnitude do próprio valor. Enquanto que o ponto flutuante vai diminuindo o seu erro, sempre ficando algumas ordens de magnitude menor. Na Figura 16 pode-se ver, claramente, essa relação entre o erro e o valor para o ponto flutuante e o ponto fixo com 8 bits para a parte inteira e, 24 bits para representar a parte fracionária. Com esses 24 bits, temos um erro fixo na casa dos 2^{-24}, sendo na base 10, 10^{-7}.

Figura 16 – Gráfico relacionando o valor nominal e a incerteza/erro para o ponto fixo (8,24) e o ponto flutuante de 32 bits



Fonte: (HUGHES et al., 2014).

Este é um dos motivos para a maior adoção do ponto flutuante pela comunidade. Consequentemente, a indústria de hardware focou muito em otimizar o hardware para o ponto flutuante, criando uma retroalimentação em termos de esforço de desenvolvimento, que torna o uso do ponto flutuante muito mais vantajoso hoje em dia, principalmente para o uso em GPUs que possuem FPUs muito otimizadas. Já para o *mobile* ou computação de borda, até mesmo pelo objetivo da plataforma, o ponto fixo é claramente uma opção melhor ou, no mínimo, merecedora de avaliações de desempenho mais aprofundadas.

4.3 OTIMIZAÇÃO POR MEIO DO HARDWARE

4.3.1 Escolha da configuração ideal para o processador

Uma das vantagens da plataforma de projeto do Cadence® Xtensa® é a customização do ISA do processador, além da definição de características físicas como: o número de registradores; tamanho da interface de memória, estágios de *pipeline*, paralelismo, etc. Cada alteração gera implicações futuras na manufatura dos chips e em limitações no seu uso, sendo 3 das mais relevantes: a frequência de operação, a potência máxima e a área do mesmo (geralmente referido em número de *gates*).

Além da configuração de *Controle* utilizada até o momento para avaliar melhorias, criou-se 3 novas configurações onde os parâmetros básicos de potência, área e desempenho foram avaliados seguindo o critério *Figure-of-Merit* (FoM) (BALASUBRAMANIAN; NARAYANA; CHINNADURAI, 2005). Na prática, o critério FoM é o inverso da multiplicação entre área, energia e tempo de execução, permitindo uma análise balanceada e a escolha da melhor opção, a qual maximiza o FoM. Ainda, o FoM é descrito pela Equação 4.1.

$$FoM = \frac{1}{Area \cdot Energia \cdot Tempo} \tag{4.1}$$

A descrição sucinta de cada uma das novas configurações do processador está explicada abaixo, enquanto os dados podem ser analisados comparativamente pela Tabela 3. É importante ressaltar que os dados de potência são oriundos de estimativas de alto nível da ferramenta e não consideram a execução das aplicações.

1. **Configuração Controle:** Uma configuração balanceada, simples e sem o emprego de componentes de hardware especializados, resultando em uma área reduzida. No Cadence® Xtensa® Xplorer é referenciada como *sample_config* e é uma excelente configuração para obter resultados de controle para a aplicação. Todos os testes iniciais foram feitos nessa configuração para se obter uma comparação mais justa no final. Vale ressaltar que essa configuração não possui unidade de ponto flutuante, logo todas as operações são feitas em nível de software;

   Frequência de operação: 875 MHz

   Área: 132.793 (*gates*)

   Potência: 41,00 mW

2. **Configuração Controle_FPU:** É a mesma configuração anterior, só que com a adição de uma FPU (Unidade de Ponto-Flutuante). Com uma área significativa-

mente maior, mas com uma execução das operações *float* muito mais rápida, por possuir um hardware específico para isso;

Frequência de operação: 874 MHz

Área: 173.502 (*gates*)

Potência: 41,58 mW

3. **Configuração Equilibrado:** Essa configuração apresenta maiores modificações que as anteriores, sendo as duas principais: a interface de memória que foi para 64 bits, ou seja, permitindo *loads/stores* com o dobro de dados das primeiras configurações e 64 registradores AR (*Address-Registers*) para evitar o uso da pilha de memória;

Frequência de operação: 1.091 MHz

Área: 179.943 (gates)

Potência: 57,5 mW

4. **Configuração MaxDesempenho:** Essa configuração foi construída utilizando todas as possíveis *features* para se obter um maior desempenho. Além das duas alterações da configuração anterior, também temos 7 estágios de *pipeline* para uma maior frequência.

Frequência de operação: 1165 MHz

Área: 195.080 (*gates*)

Potência: 74,25 mW

Com essas 4 configurações, será avaliado o desempenho na execução do *ray tracing* em diferentes cenas. As configurações serão ranqueadas utilizando o critério da FoM, levando em conta os parâmetros de energia, área e desempenho.

Tabela 3 – Tabela comparativa entre as configurações do processador

Dado	Controle	Controle_FPU	Equilibrado	MaxDesempenho
Frequência (MHz)	875	871	1.091	1.165
Estágios de *pipeline*	5	5	5	7
Processo de fabricação	28 nm HPM	28 nm HPM	28 nm HPM	28 nm HPM
Interface de memória (bits)	32	32	64	64
Número de unidades de Load/Store	1	1	1	1
Número de registradores AR	32	32	64	64
FLIX[a]	Não	Não	Sim	Sim
Unidade de *pre-Fetch*	Não	Não	Sim	Sim
FPU	Não	Sim	Sim	Sim
Área (*gates*)	132.793	173.502	179.943	195.080
Área (mm^2)	0.131	0.176	0.183	0.2
Potência (mW)	41,00	41,58	57,50	74,25

[a] *Flexible Length Instruction Extension*

Fonte: Xtensa® Xplorer.

4.4 COLETA DE RESULTADOS

Uma vez tendo as distintas versões do Xtensa®, escolheu-se outras 4 cenas para avaliar o desempenho atingido. As cenas utilizadas foram a *Cena Padrão*, *Tea Pot*, *Bunny*, *Cube*. Abaixo, temos um breve comentário das 4 cenas e a Tabela 4 apresenta um resumo da quantidade de vértices e triângulos.

1. *Cena Padrão* (Autor, 2018): Cena construída pelo próprio autor. A imagem é bem simples, contendo somente um triângulo e o chão. Na Figura 17 podemos ver a mesma renderizada.

2. *Utah Tea Pot* (Martin Newell, 1975): Famosa cena de um bule de chá, baseada em um bule Mellita®. A cena foi feita por Martin Newell em 1975 para sua dissertação de PHD. Ficou extremamente famosa e é uma das cenas mais utilizadas

na computação gráfica. Na Figura 18 podemos ver a mesma renderizada.

3. *Stanford Bunny* (Stanford University, 1996): Cena de um coelho escaneada a partir de um figurino real em cerâmica por um sistema de lasers. Pertence ao conjunto de cenas/modelos da Stanford University, por isso o nome de *Stanford Bunny*. Na Figura 19 podemos ver a mesma renderizada.

4. *Cube* (Morgan McGuire, 2005): Cena de um cubo, onde cada lado do mesmo é formada pela união de 2 triângulos com as suas hipotenusas encostadas. A cena possui 12 triângulos. Na Figura 20 podemos ver a mesma renderizada.

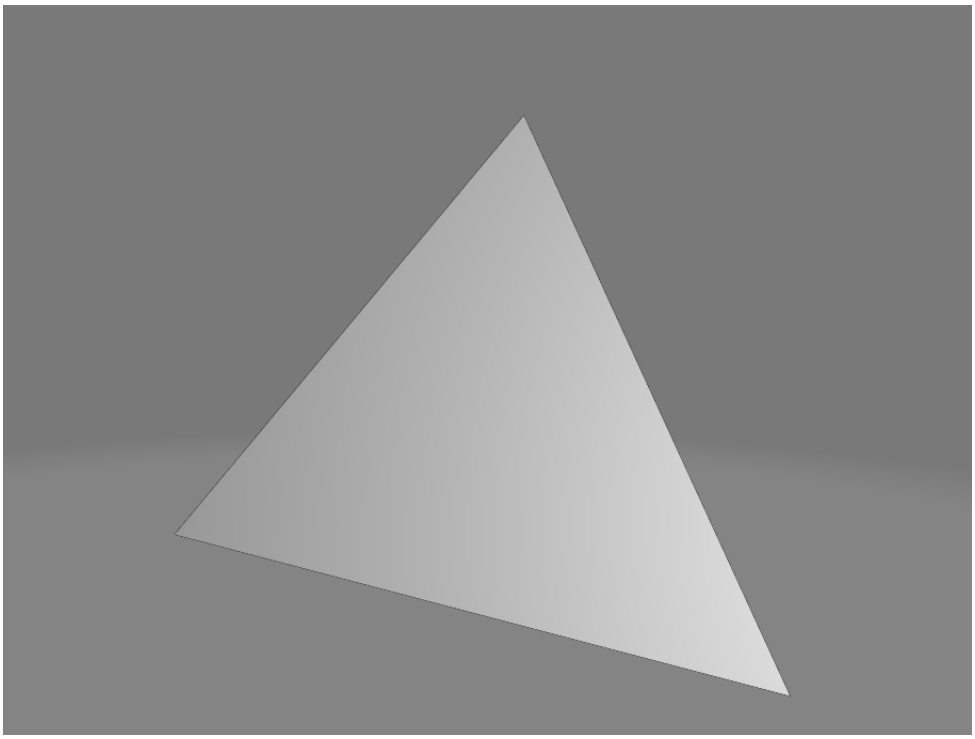
Para cada cena, foi realizado o a execução no Cadence® Xtensa® Xplorer, usando as 4 configurações e coletando dados de desempenho. Além disso, essas mesmas cenas tiveram seu tempo médio medido quando executadas no processador Intel (R) Core (TM) i7-9750H @2,60 GHz com 16GB de memória RAM (INTEL), permitindo aquisição de mais dados para posterior análise.

Tabela 4 – Cenas utilizados para avaliar o desempenho do processador

Cena	**Quantidade de Triângulos**	**Quantidade de Vértices**
CenaPadrão	1	3
Utah TeaPot	15704	8478
Stanford Bunny	144046	72378
Cube	12	24

Fonte: Autor, (Martin Newell, 1975; Stanford University, 1996; Morgan McGuire, 2005).

Figura 17 – Cena construída pelo Autor: um único triângulo flutuante

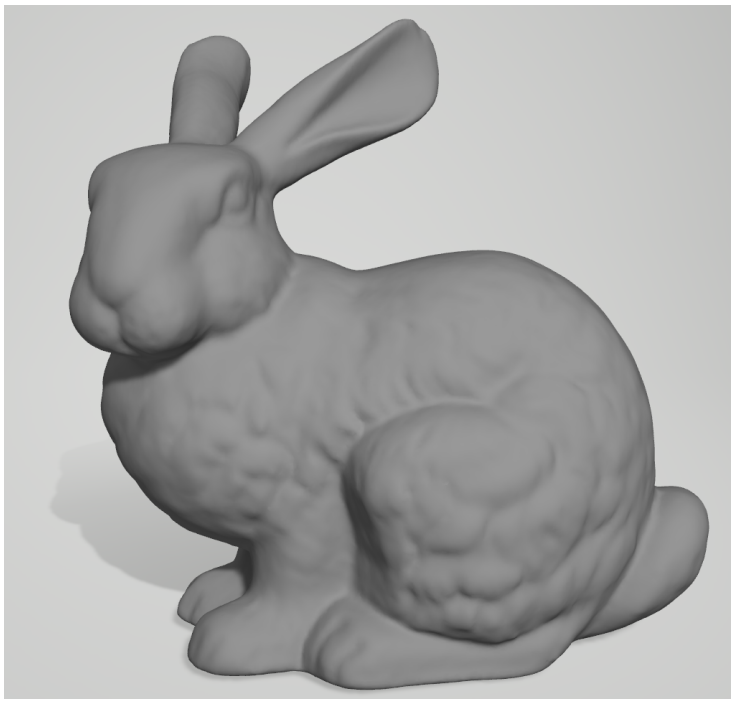


Fonte: Autor.

Figura 18 – Cena de um bule de chá, uma das cenas mais famosas da computação gráfica

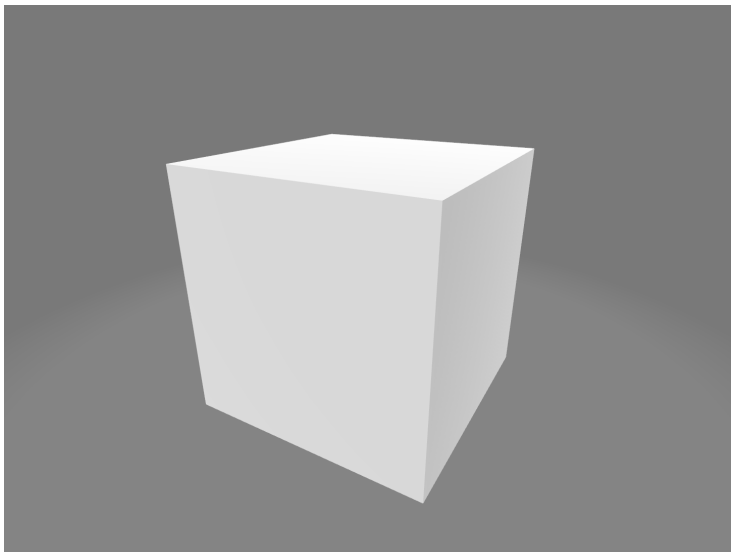
Fonte: (Martin Newell, 1975).

Figura 19 – Cena de um coelho de cerâmica digitalizado através de um sistema de lasers



Fonte: (Stanford University, 1996).

Figura 20 – Cena de um simples cubo, formado por 12 triângulos, 2 para cada lado do cubo



Fonte: (Morgan McGuire, 2005).

5 ANÁLISE DOS RESULTADOS

Todos os resultados obtidos para o Cadence® Xtensa® foram obtidos através do simulador Cadence® Xtensa® Xplorer com a funcionalidade de execução do código. Como existem diversas possíveis configurações para se realizar essa tarefa, é essencial descrever quais foram as opções definidas para uma melhor reprodução do resultado futuramente:

- Building Target: ***Release***, que provê otimizações no código compilado visando um maior desempenho.
- Simulator (ISS Simulation Mode): ***Fast-Functional***, também conhecido como *TurboXim*, que foca em uma execução mais rápida da simulação, podendo ser de 40-80 vezes mais rápido do que o modo que simula com todos os detalhes a microarquitetura.
- Simulator (Memory Modeling): ***Never***, ou seja, sem simulação dos tempos de memória.

Já para resultados obtidos no INTEL, foi realizado o *compile* com o argumento *-O*, ou seja, sem nenhuma otimização do compilador GCC. Além disso, os tempos foram obtidos utilizando o método *clock()* da biblioteca *time.h*, realizando a execução do executável gerado após compilação.

5.1 ANÁLISE DOS RESULTADOS OBTIDOS

A primeira análise feita foi para saber o desempenho das diferentes configurações de cada processador em cada cena. Para tanto, as cenas foram executadas em cada processador e aferido o tempo de execução e número de ciclos de processamento. Na Tabela 5 temos o desempenho medido pelo tempo de execução, já na Tabela 6 temos a mesma análise, mas com a quantidade de ciclos de execução.

Como mencionado no início do trabalho, a métrica mais usada para comparação das diversas implementações do *ray tracing* é o raios por segundo (*ray/sec*). Porém existe uma outra forma que também é muito usada, o *frames per second* (FPS). Este é mais utilizado para demonstrar a quantidade de imagens que são processadas por segundo, visto que o *ray tracing* é muito usado em animações e jogos que necessitam de pelo menos 24-30 FPS para apresentarem movimentos fluidos dos objetos nas cenas.

Nas nossas análises e comparações foi utilizado *ray/sec* como métrica principal, mostrando qual o *throughput* médio da implementação. Além disso, para cenas estáticas com objetos parados, como comumente usadas nas comparações dos trabalhos na literatura, não faz muito sentido apresentar o FPS, visto que se está renderizando a mesma imagem de forma repetida. Nesse tipo de cena podemos facilmente converter entre *ray/sec* e FPS, simplesmente multiplicando a quantidade de raios necessários para renderizar a imagem pela quantidade de FPS. Dessa forma sabemos que uma cena precisa de R raios para ser renderizada e que o FPS nos indica que a implementação consegue renderizar F vezes a cena por segundo. Temos que $R \times F = N\ rays/sec$. Na Tabela 8 são apresentados os dados de desempenho em *ray/sec* e FPS de cada processador em cada cena.

Tabela 5 – Dados do tempo de execução, utilizados na análise de desempenho dos processadores

Processador	**Desempenho (ms)**			
	Cena Padrão	Tea Pot	Bunny	Cube
Controle	1282	341054	3185672	264
Controle_FPU	340	127440	1824644	88
Equilibrado	217	71319	1015735	55
MaxDesempenho	204	66835	951216	52
Intel Core i7-9750H	47	12781	232844	16

Fonte: Autor.

Tabela 6 – Dados de número de ciclos, utilizados na análise de desempenho dos processadores

Processador	**Ciclos (#)**			
	Cena Padrão	Tea Pot	Bunny	Cube
Controle	1117M	298B	2787B	230M
Controle_FPU	296M	111B	1589B	76M
Equilibrado	237M	78B	1108B	60M
MaxDesempenho	238M	78B	1108B	60M
Intel Core i7-9750H	46875	13M	232M	15625

Fonte: Autor.

Ao analisar os resultados da Tabela 7, é notório o ganho de desempenho dos outros processadores em comparação ao Controle, o que é muito explicado pela adição da FPU. A configuração Controle_FPU é a mesma que a Controle, com adição da FPU, e possui ganhos de mais de 2x em todos os testes, sendo que o resultado de alguns testes chegou próximo das 4 vezes. Já entre as configurações Equilibrado e MaxDesempenho vemos que a diferença é marginal, explicado puramente pela diferença na frequência do processador. Já o processador Intel Core i7-9750H mostra

resultados superiores em todas as cenas, porém mesmo assim apresenta um desempenho pífio de 0,0043 FPS na cena Bunny, apresentado na Tabela 8, que significa um tempo de 232 segundos para computar um único *frame*. Esse desempenho baixo na cena Bunny se explica pela alta complexidade da mesma, sendo a que apresenta a maior quantidade de triângulos, como apresentado na Tabela 4.

Tabela 7 – Métrica de rays/sec médio para cada cena

Processador	**Desempenho Médio em rays/s**			
	Cena Padrão	Tea Pot	Bunny	Cube
Controle	38K	30	3	37K
Controle_FPU	143K	79	6	114K
Equilibrado	224K	141	10	181K
MaxDesempenho	239K	151	11	193K
Intel Core i7-9750H	1M	788	43	640K

Fonte: Autor.

Tabela 8 – Desempenho médio, em rays/sec (RPS) e frames/sec (FPS), para cada cena

Processador	**Cena Padrão**		**Tea Pot**		**Bunny**		**Cube**	
	RPS	FPS	RPS	FPS	RPS	FPS	RPS	FPS
Controle	38K	0,78	30	2,9E-3	3	3,1E-4	37K	3,79
Controle_FPU	143K	2,94	79	7,9E-3	6	5,5E-4	114K	11,41
Equilibrado	224K	4,61	141	1,4E-2	10	9,9E-4	181K	18,15
MaxDesempenho	239K	4,90	151	1,5E-2	11	1E-3	193K	19,30
Intel Core i7-9750H	1M	21,3	788	7,82E-2	43	4,3E-3	640K	64,0

Fonte: Autor.

Após essa análise foi possível realizar o cálculo da energia gasta em cada processador, de forma aproximada, realizando a multiplicação do tempo de execução pela potência do processador. Os resultados obtidos são apresentados na Tabela 9.

Tabela 9 – Energia utilizada por cada processador em cada cena

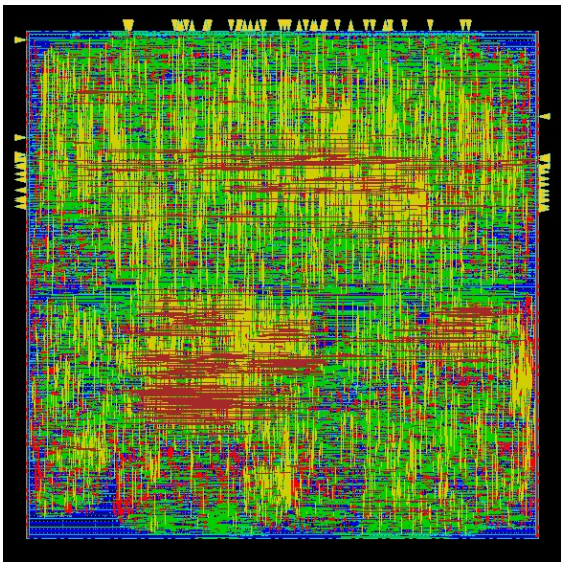
Processador	**Energia (mJ)**			
	Cena Padrão	Tea Pot	Bunny	Cube
Controle	52,61	13997	130740	10,83
Controle_FPU	14,14	5299	75869	3,64
Equilibrado	12,48	4101	58405	3,17
MaxDesempenho	15,15	4963	70628	3,85
Intel Core i7-9750H[1]	2109	575156	10477969	703

Fonte: Autor.

5.1.1 Resultados da Implementação VHDL

Para a Implementação VHDL, foram realizados as etapas básicas de um fluxo ASIC: especificação, projeto hdl, simulação e validação, sínteses lógica e física do circuito e, posteriormente a simulação *pós-layout*. Na Figura 21 temos o *layout* resultante gerado através do *Encounter Digital Implementation* (EDI).

Figura 21 – *Layout* do circuito gerado pelo EDI



Fonte: Autor.

Por não ser uma implementação completa do algoritmo do *ray tracing*, e sim somente do método de interseção raio-triângulo, não é possível determinar um resultado para o circuito de forma equiparável com os da Implementação Xtensa. Dessa forma vamos assumir que existe um hardware perfeito que realiza todas as outras operações necessárias para o *ray tracing* de forma extremamente rápida, e que o gargalo de todo o sistema é a Implementação VHDL, ou seja, o bloco de interseção raio-triângulo. Com isso, definimos que a quantidade de interseções raio-triângulo/sec que o nosso circuito calcula, será a quantidade de rays/sec que o suposto hardware teórico iria gerar, assim o nosso circuito seria o gargalo de todo o sistema. Como visto em alguns trabalhos da academia e nas simulações, o circuito apresenta um pior caso (quando colide) e um melhor caso (quando não colide).

Para o melhor caso registrou-se um tempo de 1,86μs para concluir, assim resultando em uma taxa máxima teórica de até 537,8 KiloRays/sec. Para o pior caso, o circuito demora 3,18μs para concluir, assim resultando em uma taxa máxima teórica de até 314,1 KiloRays/sec. Como resultados gerais temos, seguindo a tecnologia de 180nm, um circuito que alcança uma **frequência de 250 MHz**, com uma **área de 0,67 mm² (44.389 *gates* equivalente)**. No melhor caso, o circuito teve um **desempenho máximo de 537,8 KiloRays/sec**, com um gasto de **potência de 129,73 mW**.

Considerando o caso em que existem colisão, temos um total de 796 cilcos de clock, dos quais 262 são do *cros_product* (2 execuções de 131 ciclos cada), 288 do *dot_product* (4 execuções de 72 ciclos cada) e 105 do *sub_vector* (3 execuções de 35 ciclos cada), sendo o restante do circuito comportamental o *top_control*, para sua máquina de estados e alguns usos da FPU, como cálculos de multiplicação, divisão, adição, subtração e comparação de escalares.

5.1.2 Comparação dos valores de FoM

Com todos os dados necessários foi possível realizar o cálculo do Figure-of-Merit (FoM), aplicando a Fórmula 4.1, no qual foi utilizado o número de *gates* equivalentes para a área e o tempo em ms, seguindo a lógica da fórmula. Na Tabela 10 temos os dados da FoM para cada processador, normalizados em comparação com a configuração Controle em cada cena.

Tabela 10 – Métrica FoM, normalizada em relação a configuração Controle.

Processador	**Figure-of-Merit**			
	Cena Padrão	Tea Pot	Bunny	Cube
Controle	1,0	1,0	1,0	1,0
Controle_FPU	10,7	5,4	2,3	6,8
Equilibrado	18,4	12,0	5,2	12,1
MaxDesempenho	14,9	9,8	4,2	9,8
Intel Core i7-9750H[1]	6,47E-5	6,16E-5	1,62E-5	2,47E-5
Implementação VHDL[2]	1,89E+2	3,14E+2	2,74E+2	1,91E+2

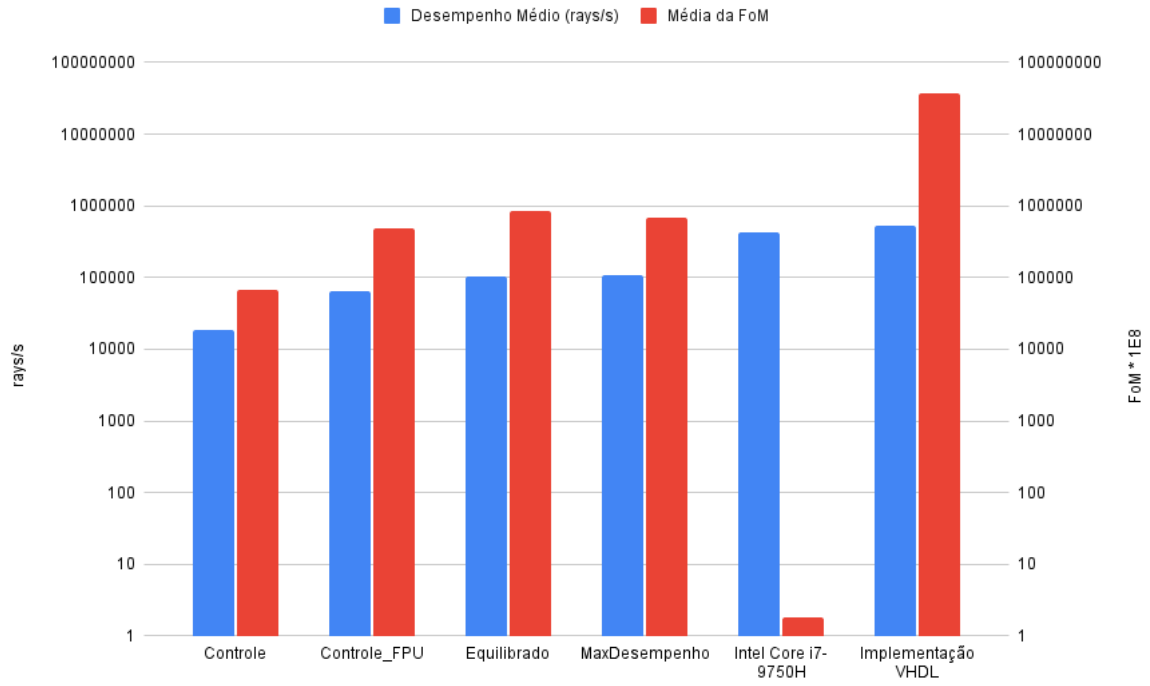
Fonte: Autor.

Para termos uma comparação mais visual foi montado um gráfico, em escala logarítmica, comparando o desempenho e a FoM média de todas as cenas para cada processador. Na Figura 22 este gráfico é apresentado. Fica claro que, mesmo tendo o melhor desempenho o processador Intel Core i7-9750H apresenta uma FoM muito inferior, cinco ordens de magnitude abaixo. Isso pode ser explicado pela alto consumo energético, mas principalmente pela elevada área do processador que é quase mais de 7 mil vezes maior. Entre os processadores Cadence® Xtensa® LX, podemos ver que se atinge um pico no *Equilibrado*, apresentando a melhor FoM. No *MaxDesempenho* já vemos uma leve queda em todas as cenas. Isso se deve ao fato que o ganho de desempenho no tempo de execução (-6%) foi marginal em relação ao aumento da área (8%) e da potência (29%) do processador, em comparação com o anterior.

Já na Implementação VHDL, considerou-se o seu máximo teórico, aplicado para a quantidade de interseções raio-triângulos de cada cena, computados pelo

softwares nas outras implementações. Fica claro que é a implementação que apresenta a melhor FoM. Isso muito por conta de sua alta performance e baixa potência, sendo essa baixa potência explicada pelo fato do circuito não implementar hardware suficiente para executar todo o *ray tracing*.

Figura 22 – Comparação, em base logarítmica, do desempenho e FoM média de todas as cenas para cada processador



Fonte: Autor.

[1]Para o processador Intel Core i7-9750H foram considerados dados obtidos no site do fabricante (INTEL, 2019) e repositório público de informações de processadores como o (WIKICHIP, 2021) e o (CHIPGUIDER, 2022), realizando o cálculo de número de *gates* estimado considerando o número de transistores divido por 4, por ser a quantidade em uma porta NAND.

[2]Considerando o máximo teórico de 537,8 KiloRays/sec aplicado em cada cena, considerando a quantidade de interseções raio-triângulos computados pelo softwares nas outras implementações.

5.2 COMPARAÇÃO DOS RESULTADOS COM TRABALHOS DA LITERATURA

Como apresentado anteriormente, uma boa métrica comumente usada é a do raios por segundo (*rays/sec* ou RPS). Alguns fatores fazem com que o RPS varie significativamente entre as simulações. Os principais são a cena e a implementação em si do algoritmo de *ray tracing*. A cena interfere, pois dependendo da quantidade e da maneira pela qual os triângulos se agrupam, o cálculo de cada raio pode vir a ser mais complexo, em especial se existirem muitos reflexos. Mesmo com cenas idênticas, pode-se sofrer variações, pois como são objetos 3D, dependendo do ponto de observação, resultados diferentes serão auferidos. Já a implementação do algoritmo possibilita que se otimizem ou não determinados pontos, podendo tal otimização ser algo recursivo, uma repetição etc.

Como o objetivo desse trabalho é fazer uma comparação simples e determinar a viabilidade técnica de se empregar os processadores Cadence® Tensilica® Xtensa® LX para o *ray tracing*, não será levada em consideração a rigorosidade de comparar uma mesma imagem sob diferentes ângulos ou perspectivas, entre outras características. O foco está em validar e ter a ordem de magnitude que as configurações criadas e o algoritmo usado obtêm em comparação com outros trabalhos e a indústria. Seguem abaixo os trabalhos que serão usados na comparação:

1. *A Hardware Acceleration Engine for Ray Tracing* (GAO et al., 2014);
2. NVIDIA RTX (NVIDIA, 2018c) (Burgess, John, 2019);
3. *An FPGA Implementation of Whitted-style Ray Tracing Accelerator* (PARK et al., 2008);
4. *Estimating Performance of a Ray-Tracing ASIC Design* (WOOP; SLUSALLEK, 2006);
5. *RT Engine: An Efficient Hardware Architecture for Ray Tracing* (YAN et al., 2022).

Em cada trabalho foi realizado a extração dos dados de desempenho com o objetivo de compor um único número. Para alguns trabalhos foi aplicado a média das diversas cenas apresentadas pelos autores, obtidos a partir de especificações técnicas divulgadas publicamente e, em caso de só conter a métrica de FPS, foi realizada a conversão em raios por segundo. Na Tabela 11 estão reunidos os trabalhos, seus desempenhos e comentários pertinentes.

Ao confrontar com os outros trabalhos, podemos ver que estamos uma ordem de magnitude abaixo, o "GAO, item 4"apresenta um desempenho médio 403 vezes maior, o "PARK, item 5"apresenta 12,6x mais desempenho, já o "RT Engine, item 3"apresenta 858x mais desempenho. Em comparação com o melhor candidato "NVIDIA, item 1"o mesmo apresenta um desempenho de quase 100 mil vezes maior.

Tabela 11 – Comparação do desempenho médio[0] , em *rays/sec*, e a *Figure-of-Merit* dos trabalhos.

Item	Trabalho	Desempenho	FoM [6]
1	NVIDIA RTX (NVIDIA, 2018c)[1]	10G	1,35E-4
2	*Estimating Per...* (WOOP; SLUSALLEK, 2006)[2]	227M	-
3	*RT Engine* (YAN et al., 2022)[3]	92,7M	-
4	*A Hardware Accel...* (GAO et al., 2014)[4]	43,5M	3,96E-4
5	*An FPGA Implemen...* (PARK et al., 2008)[5]	1,36M	-
6	Implementação VHDL (Autor)	538K	1,58E-5
7	Intel Core i7-9750H (Autor)	420K	3,93E-17
8	*Max-Desempenho* (Autor)	108K	1,04E-12
9	*Equilibrado* (Autor)	102K	2,52E-11

Fonte: Autor.

Ao analisar detalhadamente os trabalhos citados, além de somente a performance, chamam a atenção dois principais fatores: a área dos circuitos e o algoritmo usado.

Em relação à área, podemos notar no trabalho *RT Engine* (YAN et al., 2022), que é a mediana dos resultados (item 3 da Tabela 11), que o mesmo possuí uma área de 0,48 mm^2, sendo mais que o dobro em comparação ao *Max-Desempenho* (item 8). Já a *NVIDIA RTX* (NVIDIA, 2018c), temos uma área de 754 mm^2 e mais de 18 bilhões de transistores, sendo mais de mil vezes maior do que o *Max-Desempenho*.

Algo que também chama a atenção é na FoM, na qual a *NVIDIA RTX* (NVIDIA, 2018c) (item 1) ficou com a melhor FoM de todos os trabalhos, mesmo apresetando a maior área e uma potência de 260W (TDP). Isso é explicado pelo excelente resultado de performance da mesma.

Já em relação ao algoritmo do *ray tracing*, temos que a principal diferença entre os trabalhos são as otimizações aplicadas no processo de determinar a intersecção do raio-triângulo e como reduzir a quantidade de vezes que essa operação precisa ocorrer para cada cena. Quanto ao cálculo da intersecção, já usamos um método que está bem próximo do mais otimizado que temos na indústria. Todos os algoritmos de

[0] Cada trabalho/proposta teve o seu número extraído de um local específico e muitos foram feitos a partir da média de algumas cenas.

[1] Para o item 1 foi considerado o valor divulgado pela fabricante.

[2] Para o item 2 foi considerado a média dos valores apresentadas na Tabela 3 para a *DRPU8 ASIC*. O valor obtido parte da multiplicação dos FPS pelo número de raios nas cenas.

[3] Para o item 3 foi considerado o valor apresentado na Tabela 7.

[4] Para o item 4 foi considerado o *Best Case* apontado na Tabela 1.

[5] Para o item 5 foi considerado a médias dos valores apresentado na Tabela 1 do trabalho.

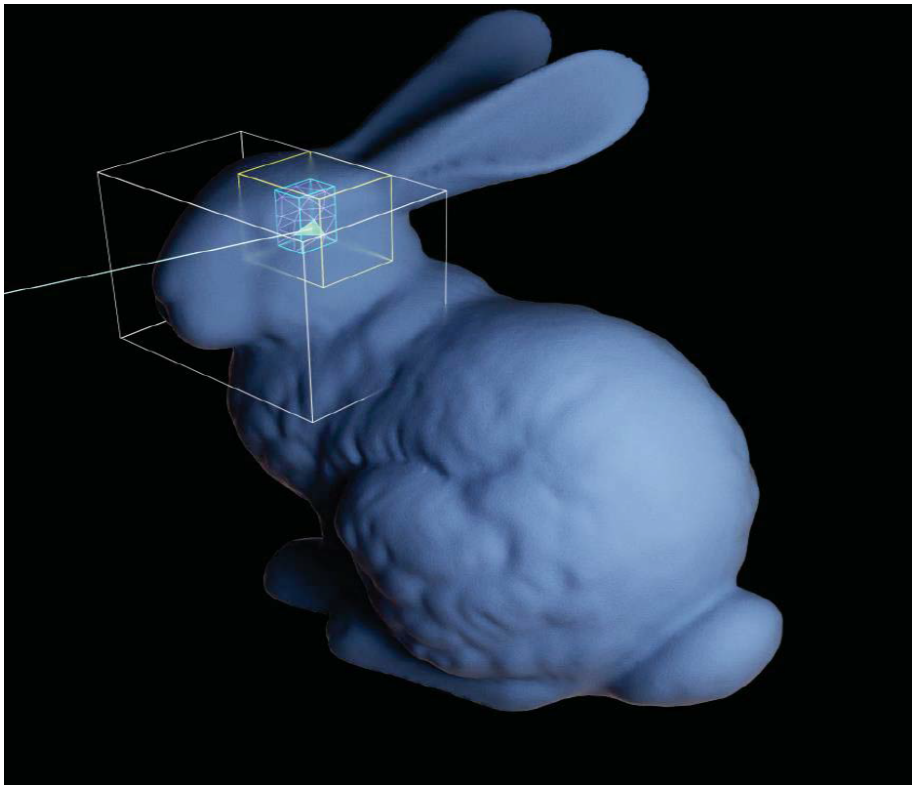
[6] Para os trabalhos que não são do do autor foram utilizados os dados de *rays/sec* médio e aplicado a quantidade de raios calculado nas cenas, assim chegando no tempo de execução e área sendo em mm^2. Para os casos que não foram deixados vazios, foi pela falta de dados de potência dos circuitos nos trabalhos.

ray tracing acabam tendo de repetir essa conta, porém a quantidade de vezes que essa conta é feita pode ser otimizada. Para uma determinada cena, exitem inúmeros triângulos que podemos determinar previamente que não serão atingidos por um raio, sendo alguns exemplos os triângulos que não estão na direção do raio ou os que estão atrás de outros triângulos. O algorítimo aplicado nesse trabalho não faz diferenciação nenhuma quanto a isso, não tendo nenhuma otimização para reduzir a quantidade de vezes que o cálculo de intersecção raio-triângulo é executado, realizando o cálculo iterativamente para todos os triângulos. Já em outros trabalhos e na indústria, são aplicadas as otimizações. O trabalho *Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques* (DENG et al., 2017) compila os principais avanços na área, ao mesmo tempo que apresenta as duas técnicas mais usadas em outros trabalhos, sendo elas:

1. Kd-Tree: É uma estrutura de dados para indexar pontos de um ambiente de K dimensões, seguindo a lógica de uma árvore binária. Realiza a divisão da cena em duas metades, de forma recursiva. Cada nó final fica com 2 triângulos, que estarão próximos, mas podem estar alinhados em qualquer eixo;

2. Bounding Volume Hierarchy (BVH): Também é uma árvore, porém agrupando os triângulos de forma um pouco diferentes, em cubos/caixas de forma a serem alinhadas em um eixo específico ou *axis-aligned bounding box* (AABB). A diferença é que dessa forma os triângulos serão agrupados, por proximidade, mas seguindo um alinhamento específico. Os triângulos estarão dispersos no ambiente em qualquer posição, mas os seus agrupadores, as caixas, estarão alinhados em um eixo específico e podem se sobrepor. Este algoritmo base e suas variantes são usadas como padrão pela indústria. A Figura 23 apresenta um exemplo prático na cena Bunny dessa técnica.

Figura 23 – Demonstração do algoritmo BVH aplicado a cena Bunny.



Fonte: (Burgess, John, 2019)

6 CONCLUSÃO

A renderização utilizando o *ray tracing* vem cada vez mais sendo utilizada, permitindo uma simulação real do comportamento da luz e uma experiência visual extraordinária. Porém, continua sendo um processo computacionalmente intensivo requerendo componentes de hardware específicos e de alta capacidade.

Neste trabalho foram abordadas duas soluções distintas para a renderização, um circuito específico (Implementação VHDL) e um circuito de propósito geral (Implementação Xtensa). Foram obtidos os resultados, respectivamente, de 538 KiloRays/-sec e 108 KiloRays/sec. Ambos os resultados não sendo suficientemente aceitáveis para utilizar em aplicações *realtime*, mas compatíveis para o uso em simulação e aplicações específicas.

Fica notório que com a tecnologia atual, não parece promissora a aplicação dessa técnica em sistemas que demandam por baixo consumo de potência ou computação de borda, principalmente para aplicações *realtime* e em circuitos de propósito geral. Exemplos como óculos de Realidade Virtual (com hardware integrado) e *smartphones* ainda não possuem a capacidade computacional para executar, em tempo real, aplicações que utilizam essa técnica. Ao comparar com outros trabalhos e produtos comerciais, vimos que o caminho está na aplicação de circuitos específicos, altamente paralelizados. Por sua vez, esses circuitos acabam ocupando uma grande área e, tendo um consumo elevado de energia, porém sendo necessários, visto a alta quantidade de intersecções raio-triângulo que devem ser calculadas em um curto período de tempo para tornar a aplicação *realtime*.

Trabalhos futuros podem ser explorados nessa área com os processadores Cadence® Tensilica® Xtensa® LX, aplicando as otimizações de software mencionadas no Capítulo 5.2, avaliando pontos críticos para as estruturas, como o BHV, o desenvolvimento de instruções específicas para elas e a interseção raio-triângulo. Com estas melhorias certamente teremos ganhos, o que pode permitir a utilização do *ray tracing* em algumas aplicações que não são em tempo real ou algum sistema embarcado para simulação breve de cobertura/dispersão de sinal, mas pouco provável para renderização de imagens em tempo real.

REFERÊNCIAS BIBLIOGRÁFICAS

APPEL, A. Some techniques for shading machine renderings of solids. **AFIPS '68 (Spring) Conference**, Atlantic City, New Jersey, USA, p. 37–45, 1968. Acesso em 24 nov. 2020. Disponível em: <doi.org/10.1145/1468075.1468082>.

BALASUBRAMANIAN, P.; NARAYANA, M. L.; CHINNADURAI, R. Analysis of non-adjacency in k-maps and its impact on power consumption reduction in non-regenerative cmos circuits. p. 708–711 Vol. 1, 2005. Acesso em 06 nov. 2022. Disponível em: <ieeexplore.ieee.org/document/1594199>.

Bart Veldhuizen. **19 Billion Triangles Render**. 2012. Acesso em 13 nov. 2020. Disponível em: <blendernation.com/2012/03/23/19-billion-triangles-render/>.

Burgess, John. **RTX ON – The NVIDIA TURING GPU**. 2019 IEEE Hot Chips 31 Symposium (HCS), 2019. 1-27 p. Disponível em: <ieeexplore.ieee.org/document/8875651, note={Acessoem31out.202}.>

Cadence. **Optimizing Cadence Tensilica Processors for your application**. 2016. Acessado em 03 dez. 2020. Disponível em: <https://www.youtube.com/watch?v=7iupXZkiQog>.

Cadence Inc. **Tensilica Datasheet**: Xtensa lx7 processor. 2018. Acessado em 03 dez. 2020. Disponível em: <ip.cadence.com/uploads/1099/TIP_PB_Xtensa_lx7_FINAL-pdf>.

____. **Tensilica Processor**. 2020. Acessado em 03 dez. 2020. Disponível em: <ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable>.

CHIPGUIDER. Intel core i7-9750h specs. 2022. Acesso em 06 nov. 2022. Disponível em: <chipguider.com/?proc=intel-core-i7-9750h>.

Chris Williams. **Microsoft's HoloLens secret sauce: A 28nm customized 24-core DSP engine built by TSMC**: How to make your own virtual reality brain. 2016. Acessado em 03 dez. 2020. Disponível em: <theregister.com/2016/08/22/microsoft_hololens_hpu/>.

CHRISTENSEN, P. H. et al. Ray tracing for the movie 'cars'. **IEEE Symposium on Interactive Ray Tracing**, Salt Lake City, UT, p. 1–6, 2006. Acesso em 19 out. 2020. Disponível em: <ieeexplore.ieee.org/document/4061539/>.

COLEMAN, P. et al. Into the voyd: Teleportation of light transport in incredibles 2. Vancouver, 2018. Acesso em 19 out. 2020. Disponível em: <graphics.pixar.com/library/PortalsIncredibles2/>.

CULAU, E. C. et al. An efficient single core flexible processor architecture for 4096-bit montgomery modular multiplication and exponentiation. In: **2018 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2018. p. 1–5. ISSN 2379-447X.

CURLESS, B. Ray-triangle intersection. 2006. Acesso em 02 dez. 2020. Disponível em: <courses.cs.washington.edu/courses/csep557/10au/lectures/triangle_intersection.pdf>.

DENG, Y. et al. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. **ACM Computing Surveys**, p. 1–41, 2017. Acesso em 10 out. 2022. Disponível em: <researchgate.net/publication/319442281_Toward_Real-Time_Ray_Tracing_A_Survey_on_Hardware_Acceleration_and_Microarchitecture_Techniques>.

Dibya Chakravorty. **The Most Common 3D File Formats**. 2019. Acesso em 15 nov. 2020. Disponível em: <all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/\#stl>.

GAO, Y. et al. A hardware acceleration engine for ray tracing. **2014 IEEE 17th International Conference on Computational Science and Engineering**, Chengdu, China, p. 505–508, 2014. Acesso em 19 out. 2020. Disponível em: <ieeexplore.ieee.org/document/7023628/>.

GLASSNER, A. S. **An Introduction to RAY TRACING**. Palo ALto CA 94304, EUA: Elsevier, 1989. 286 p.

Hanika, J.; Keller, A. Towards hardware ray tracing using fixed point arithmetic. In: **2007 IEEE Symposium on Interactive Ray Tracing**. [S.l.: s.n.], 2007. p. 119–128.

Harold Serrano. **What is a mesh in OpenGL?** 2015. Acesso em 13 nov. 2020. Disponível em: <quora.com/What-is-a-mesh-in-OpenGL>.

HERY, C.; VILLEMIN, R.; HECHT, F. Towards bidirectional path tracing at pixar. 2016. Acesso em 19 out. 2020. Disponível em: <graphics.pixar.com/library/BiDir/>.

HUGHES, J. et al. **Computer Graphics: Principles and Practice - Thrid edition**. One Lake Street, Upper Saddle River, New Jersey 07458: Pearson Education, Inc, 2014. 61, 286 p.

IBM. **CMOS 7RF (CMRF7SF) 1.8V (12-TRACK) Standart Cell Databook Foundry IP (Standard cells da IBM 180nm)**. 2010. Acesso em 09 dez. 2018. Disponível em: </home/tools/design_kits/ibm180/IBM_PDK/cmrf7sf/V2.0.0.2AM/ibm_cmos7rf_std_cell_20111130/std_cell/v.20111130/doc/CMOS7RF18V_db_00_pub_022210.pdf>.

INTEL. Intel® core™ i7-9750h processor. 2019. Acesso em 31 out. 2022. Disponível em: <ark.intel.com/content/www/us/en/ark/products/191045/intel-core-i79750h-processor-12m-cache-up-to-4-50-ghz.html>.

Jidan Al-Eryani. **FPU**. Vienna, Austria: Open Cores, 2017. Acesso em 19 ago. 2020. Disponível em: <opencores.org/projects/fpu100>.

Library of Congress. **Wavefront OBJ File Format**. 2020. Acesso em 15 nov. 2020. Disponível em: <loc.gov/preservation/digital/formats/fdd/fdd000507.shtml\#useful>.

Martin Newell. **Teapot**. 1975. Acessado em 28 jun. 2021. Disponível em: <casual-effects.com/g3d/data10/index.html\#mesh35>.

Min-wuk Lee et al. A fixed-point 3d graphics library with energy-efficient cache architecture for mobile multimedia systems. In: **2005 IEEE International Symposium on Circuits and Systems**. [S.l.: s.n.], 2005. p. 4602–4605 Vol. 5.

MöLLER, T.; TRUMBORE, B. Fast, minimum storage ray-triangle intersection. **J. Graph. Tools**, A. K. Peters, Ltd., USA, p. 21–28, out. 1997. Disponível em: <doi.org/10.1080/10867651.1997.10487468>.

Morgan McGuire. **Cube**. 2005. Acessado em 28 jun. 2021. Disponível em: <casual-effects.com/g3d/data10/index.html\#mesh9>.

NVIDIA. **NVIDIA RTX and GameWorks Ray Tracing Technology Demonstration**. NVIDIA GEFORCE, 2018. Acesso em 19 ago. 2020. Disponível em: <youtube.com/watch?v=tjf-1BxpR9c>.

____. **NVIDIA Unveils GeForce RTX, World's First Real-Time Ray Tracing GPUs**. NVIDIA, 2018. Acesso em 19 out. 2020. Disponível em: <blogs.nvidia.com/blog/2018/08/20/gamescom-rtx-turing-real-time-ray-tracing/>.

____. **NVIDIA Unveils Quadro RTX, World's First Ray-Tracing GPU**. NVIDIA, 2018. Acesso em 19 out. 2020. Disponível em: <nvidianews.nvidia.com/news/nvidia-unveils-quadro-rtx-worlds-first-ray-tracing-gpu>.

____. **Project Sol: A Real-Time Cinematic Scene Powered by NVIDIA RTX**. NVIDIA, 2018. Acesso em 19 ago. 2020. Disponível em: <youtube.com/watch?v=KJRZTkttgLw>.

____. **NVIDIA RTX A6000 Graphics Card**. NVIDIA, 2020. Acesso em 28 nov. 2022. Disponível em: <nvidia.com/en-us/design-visualization/rtx-a6000/>.

PARK, W.-C. et al. An fpga implementation of whitted-style ray tracing accelerator. **2008 IEEE Symposium on Interactive Ray Tracing**, Los Angeles, CA, USA, p. 187–187, 2008. Acesso em 18 ago. 2022. Disponível em: <ieeexplore.ieee.org/document/4634650>.

PLATO. **Timaeus of Locri**. [s.n.], 360 a.C. seção 45b e 46b s. Plato in Twelve Volumes, Vol. 9 Traduzido por W.R.M. Lamb. Cambridge, MA, Harvard University Press; London, William Heinemann Ltd. 1925. Acesso em 06 nov. 2020. Disponível em: <shorturl.at/bjzD7>.

Research and Markets. **Global Animation, VFX & Video Games Industry: Strategies, Trends & Opportunities (2020-2025)**. Research and Markets, 2020. Acesso em 19 ago. 2020. Disponível em: <researchandmarkets.com/reports/4900485/global-animation-vfx-and-video-games-industry>.

ROSSANT, C. **Very simple ray tracing engine in (almost) pure Python**. Paris, França: Git Hub, 2017. Acesso em 19 ago. 2020. Disponível em: <gist.github.com/rossant/6046463>.

Rudolf Usselmann. **Floating Point Unit**. Tailândia: Open Cores, 2014. Acesso em 19 ago. 2020. Disponível em: <opencores.org/projects/fpu>.

Sicoe, O.; Popa, M. Comparison metrics for the output of a mixed fixed point and floating point graphics pipeline. In: **2017 25th Telecommunication Forum (TELFOR)**. [S.l.: s.n.], 2017. p. 1–4.

Stanford University. **Stanford Bunny**. 1996. Acessado em 28 jun. 2021. Disponível em: <casual-effects.com/g3d/data10/index.html\#mesh4>.

WANG, L. et al. An accelerated algorithm for ray tracing simulation based on high-performance computation. **2016 11th International Symposium on Antennas, Propagation and EM Theory (ISAPE)**, Guilin, China, p. 512–515, 2016. Acesso em 19 out. 2020. Disponível em: <ieeexplore.ieee.org/document/7834001/>.

Wavefront. **Appendix B1. Object Files (.obj), Advanced Visualizer Manual**. 1992. Acesso em 15 nov. 2022. Disponível em: <fegemo.github.io/cefet-cg/attachments/obj-spec.pdf>.

WHITTED, T. An improved illumination model for shaded display. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v. 13, p. 14, 1979. Acesso em 24 nov. 2020. Disponível em: <doi.org/10.1145/965103.807419>.

WIKICHIP. Core i7-9750h - intel. 2021. Acesso em 31 out. 2022. Disponível em: <en.wikichip.org/wiki/intel/core_i7/i7-9750hs>.

WIKIPEDIA. **Ray tracing (graphics)**. Wikipedia, Ray Tracing 2020. Acesso em 20 out. 2020. Disponível em: <en.wikipedia.org/wiki/Ray_tracing_\(graphics\)>.

WOOP, E. B. S.; SLUSALLEK, P. Estimating performance of a ray-tracing asic design. **2006 IEEE Symposium on Interactive Ray Tracing**, Salt Lake City, UT, USA, p. 7–14, 2006. Acesso em 18 ago. 2022. Disponível em: <ieeexplore.ieee.org/document/4061540>.

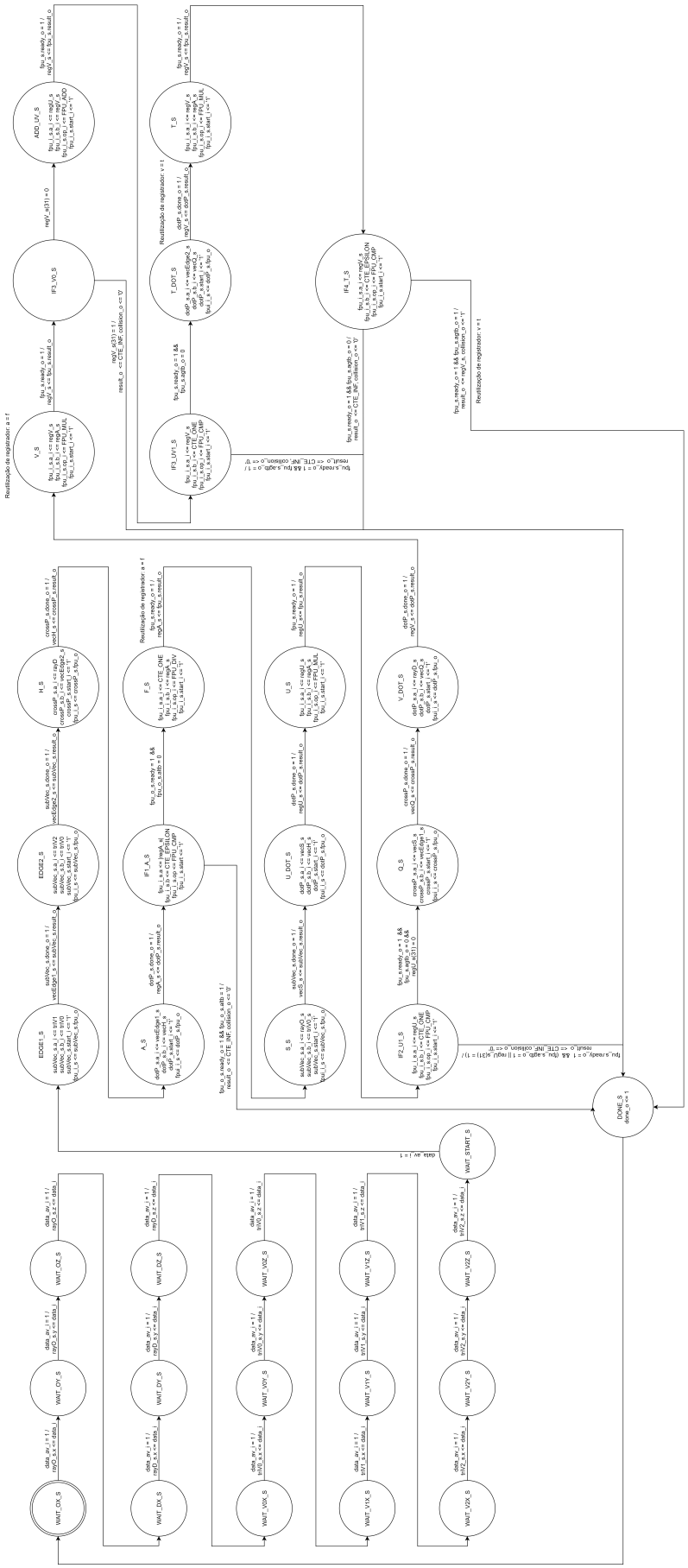
YAN, R. et al. Rt engine: An efficient hardware architecture for ray tracing. **Applied Sciences**, p. 9599, 2022. Acesso em 10 out. 2022. Disponível em: <researchgate.net/publication/363850259_RT_Engine_An_Efficient_Hardware_Architecture_for_Ray_Tracing>.

YUN, Z.; LIM, S.; ISKANDER, M. F. An integrated method of ray tracing and genetic algorithm for optimizing coverage in indoor wireless networks. **IEEE Antennas and Wireless Propagation Letters**, v. 7, p. 145–148, 2008. Acesso em 19 out. 2020. Disponível em: <ieeexplore.ieee.org/document/4456057/>.

ANEXO A – FSM COMPLETA

[PÁGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Figura 24 – FSM completa do controle do circuito feito em VHDL



Fonte: Autor.