REALTIME RAYTRACING AND WATERTIGHTNESS IN MOBILE GPU ARCHITECTURES

A Dissertation

Submitted In partial Fulfilment of the Requirements for The Award of the Degree of

Master of Technology (2018-20)
In
VLSI and Embedded Systems

Submitted by:

Jasmeet Singh (2K18/VLS/05)

Under the Supervision of

Dr. Malti Bansal
Assistant Professor
Department of Electronics & Communication Engineering
Delhi Technological University (DTU)

Department of Electronics & Communication Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

2018-20

Department of Electronics & Communication Engineering
Delhi Technological University
(Formerly Delhi College of Engineering) Bawana Road, Delhi-110042

Candidate's Declaration

I **Jasmeet Singh**, Roll No. 2K18/VLS/05 Student of MTech (VLSI and Embedded Systems), hereby declare that the project Dissertation titled "**Realtime Ray Tracing and Watertightness in Mobile GPU Architectures**" which is being submitted by me to the Department of Electronics & Communication Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology in VLSI and Embedded Systems, is original and bonafide record without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: New Delhi **JASMEET SINGH**
Date: **(2K18/VLS/05)**

Department of Electronics & Communication Engineering
Delhi Technological University
(Formerly Delhi College of Engineering) Bawana Road, Delhi-110042

Certificate

I hereby certify that the Project Dissertation titled "Realtime Ray Tracing and Watertightness in Mobile GPU Architectures" which is being submitted by JASMEET SINGH, Roll No (2K18/VLS/05) Electronics & Communication Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a bonafide record of the project work carried out by the student under my supervision. To the best of my knowledge, this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: New Delhi

Date:

Dr.Malti Bansal
Assistant Professor
Department of Electronics &Communication Engineering
Delhi Technological University

Acknowledgement

Someone wise had once said, “If the destination is beautiful, don’t ask about the pains of the journey, and if the journey is beautiful don’t ask to which destination it will lead.” And I have a lot of people to thank who have made my study journey and destination both beautiful. This dissertation is the result of work of almost two years, whereby I have been accompanied and supported by many people, to whom I would like to express my gratitude. I would like to express my deep gratitude to my supervisor, **Dr**. **Malti Bansal** Assistant professor Department of Electronics and Communication Engineering, DTU Delhi who has provided me with guidelines for my work and supported me with valuable advice throughout my studies. I would like to take this opportunity to express my appreciation to all my friends and colleagues in Electronics and Communication Engineering Department, Delhi Technological University. The two people to whom I believe I owe all and saying just ‘Thanks’ will be insufficient, are my parents. I would like to thank them for believing in me and supporting me.

JASMEET SINGH

(2K18/VLS/05)

Abstract

The folks who are in computer graphics and have dedicated their valuable time and resources to this industry are quite aware of the term Ray Tracing and recent proliferation in realtime raytracing . Ray tracing delivers visual effects that are second to none except reality itself and imitates Realtime visual graphics. ray tracer provides platform and tools for developers on “how a particular 3-Dimensional scene is created”. Traditionally a strategy named Rasterization was used in rending images, but it lacked level of realism such as shadows and reflections and was computationally cheaper as well. Imagine a scene having different vehicles in a war zone scenario of a gameplay. If we render this frame using traditional rasterization, we would be observing that the detailing such as fire raging in between the vehicles and reflections of fire from adjacent vehicles and details such as broken glasses would be not that accurate in rasterization. On contrary if we render this scene with Ray tracing the level of detailing will be presented realistically on the screen. The reason behind this is that the Ray tracing algorithm mimics the actual properties of light rays such as reflections and refractions. Its main purpose is not just to make prettier images but to have a fundamental impact on game play. Example is there are enemies approaching and are Main character of the game due to ray tracing surrounding our refectory when the player can easily identify them and intercept them. Recent trends have shown major research and development in computer graphics and most of it is associated with proliferation of Ray tracing algorithms in terms of accurate Ray primitive intersections and ray box intersections. So, we will be discussing different Ray triangle algorithms in our thesis work, Raytracing in modern day graphics pipeline, different strategies to make raytracing quicker (acceleration structures) and its close Association with the hardware aspects of VLSI design flow from algorithm level(design specifications) to the transistor level .

List of Published Papers

1. **Malti Bansal, Jasmeet Singh, “Qualitative Analysis of CMOS Logic Full Adder and GDI Logic Full Adder using 18 nm FinFET Technology”, 3rd International Conference on Recent Developments in Control, Automation & Power Engineering (RDCAPE) 2019 Oct10 (pp. 404-407). IEEE.**

2. **Malti Bansal, Jasmeet Singh, “Comparative Analysis of 4-bit CMOS Vedic Multiplier and GDI Vedic Multiplier using 18nm FinFET Technology” 2020 International Conference on Smart Electronics and Communication (ICOSEC), 10-12 Sept. 2020, ISBN:978-1-7281-5462-6**

3. **Malti Bansal, Jasmeet Singh, “Qualitative Analysis of 2-bit CMOS Magnitude Comparator and GDI Magnitude Comparator using 18nm FinFET Technology”, 2020 International Conference on Smart Electronics and Communication (ICOSEC), 10-12 Sept. 2020, ISBN:978-1-7281-5462-6**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

<u>LIST OF ABBREVIATIONS</u>

VLSI - Very Large-Scale Integration

LVS - layout vs Schematic

CPU - Central Processing Unit

GPU – Graphics Processing Unit

TPU – Turing Processing Unit

IP – Intellectual Property

DRC - Design Rule Check

SV – System Verilog

GUI - Graphical User Interface

ERC - Electrical Rule Check

SoC - System On Chip

IC - Integrated Circuit

LAT- Latency

BW - Bandwidth

RTL- Register Transfer Level

HDL- Hardware Description Language

LAL- Lower Abstraction Level

CAD- Computer Aided Design

CUT- Circuit Under Test

IC - Integrated Circuits

MOSFETs - Metal Oxide Semiconductor Field

MAC – Multiply and Accumulate

FET - Field Effect Transistors

FPGA - Field Programmable Gate Array

DSP - Digital Signal Processor

RT-	RayTracer
AI-	Artficial Intelligence
ALU -	Arithmetic Logic Unit
BVH –	Bounding Volume Hierarchy
BB –	Bounding Box
MT –	Moller Trumbore
DXR –	Direct X
MUXs –	Multiplexer
I/O –	Input/Ouput

CHAPTER-1

INTRODUCTION

1.1. BACKGROUND

Before looking at the Ray tracing and its several constraints ,first we'll need to understand the basic graphics -pipeline or rendering pipeline and use of traditional rasterization process in it. The focus in graphics pipeline is the mesh-mesh interconnections of several triangles or co-ordinates that are intercepted in order to form bigger scene or artifacts . Graphics pipeline is a conceptual model that depicts different steps required to render a 3D scene on to 2D screen. Graphics pipeline in a layman language perform the task of displaying what we want to see on the computer screen by preprocessing the 3D scene itself for instance in a video game or in animation graphics we create a three dimensional model and then we turn that 3D model into what computer displays. Or in other words we can say that graphics pipeline represents crisp set of steps that are required to convert set of coordinates in a sample space to final rendered image or frame.

What are Meshes?

Meshes are collection or interconnection of small polygons (such as triangle) to construct a bigger complex artifact. As we increase the mesh density the level of detailing increases gradually as we can shade polygons individually to increase detailing.

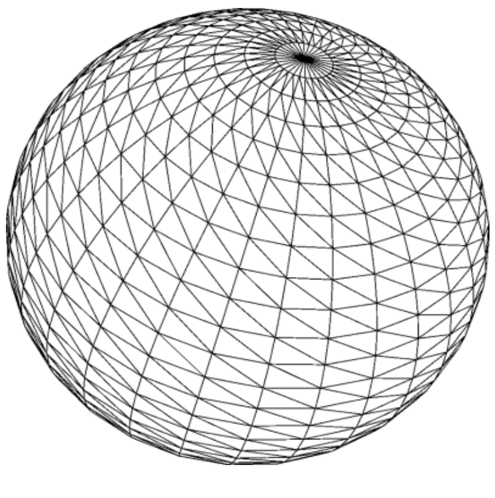


Fig 1.1 Demonstration of Meshes using 3D sphere

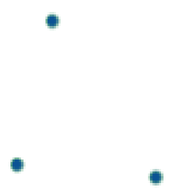
1.2. Rasterization

Rasterization is simply a technique where 3D objects are created using a mesh of triangles(polygons). [1]These are then mapped onto a 2D plane and then shaded and textured. It is the strategy of converting image described in shapes(vector format) to raster image. This shading operation is carried out by shader-core(Block in GPU). Although rasterization is a computationally cheaper technique, but it lacks the level of realism as shader core itself determines how each pixel should look without taking natural lighting/shadows into considerations. it is used in realtime 3D engines as it is swift in nature. Here the main problem statement is to turn triangle(mesh) data into pixels, for this there are different processing methods :

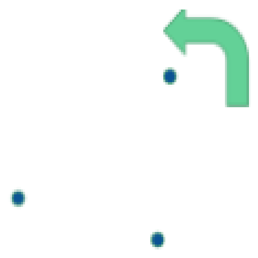
- Triangles being projected onto the screen know as geometry processing
- Identifying the pixels covered by the polygon(triangle) on the screen space : Rasterization
- Color assigning to each pixel which is referred to as pixel processing or(fragment shading)

Rasterization flow:

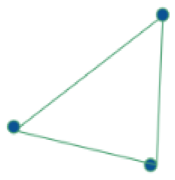
1. Initially , for the geometry to be constructed we only have 3D point coordinates or vertex data.



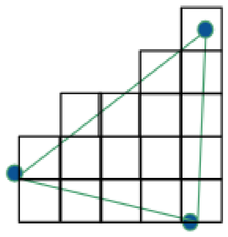
2.Vertex Shading takes place , that is vertices are being projected on screen space(2D)after transformation from 3D.



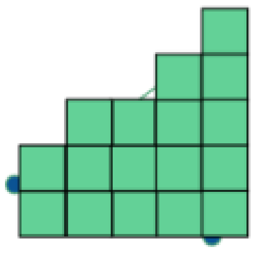
3.Transformed vertices are then assembled into triangles(also known as primitives).



4. Spotting pixels that are being touched by primitives. Also referred to as fragments.



5. Fragment Shading: at the end Assignment of color to each of the fragments is carried out.



1.3 GPU's Anatomy:

Graphics processing units (GPUs) are electronic circuits that are designed to intensify the creation of frames (image at a particular instant) in a frame buffer intended for output to screen space. They are programmed in a way such that they can vary and manipulate memory to accelerate frame creation. They can be used for algorithms that process insanely large parallel-data-blocks as they exhibit more performance than general-purpose central processing units (CPUs) due to their highly parallel structure. GPUs are used in embedded systems, gaming consoles, server stations, personal computers, mobile phones and workstations. Modern GPUs exhibits great performance matrices for advance image processing and next generation computer graphics. In a personal computer, a GPU can be embedded on the motherboard or present on a video card . In certain CPUs, they are embedded on the CPU die.

Modern day GPUs perform their big chunk of processing related to 3D computer graphics by using most of its transistors(switch-level-processing). In addition to the 3D hardware, today's GPUs include basic 2D acceleration and framebuffer capabilities (usually with a VGA compatibility mode). In the beginning, GPUs were used in computer graphics to accelerate memory-hungry processes like rendering polygons and texture mapping. later adding units to accelerate geometric operations such as the translation and rotation of vertices into different coordinate systems. Recent GPUs support programmable shaders which can manipulate vertices and textures. techniques such as Interpolation and oversampling to mitigate aliasing, and very high-precision color spaces. since majority of these complex calculations involve matrix and vector operations such as cross product, dot product ,addition etc. engineers and scientists have increasingly studied the use of GPUs for non-graphical calculations; they are exceedingly suitable for complex parallel problems.

GPU features:

- Modern day GPU’s contain nearly 16 shader cores(shown in Fig.1.13) , with 16 Arithmetic Logic Units(ALU) each.
- Electronic circuit that exhibit extreme Parallel architecture, as it has thousands of threads and can process them in a flight and different processes don’t communicate with each other.
- From over 16 different instruction streams it processes 256 operations in parallel.
- Generally, they are clocked at 256 Gigaflops
- Program counter is shared by multiple threads.
- Excellent performance as it has capability of executing a lot of independent operations at same instant.
- Extreme memory bandwidth (around Hundreds of GB/s) .but high latency(around thousands of cycles).

CPU’s aid to GPU- Commands are issued by the CPU to the GPU such as :

- Using vertices and vertex data create geometric drawings.
- Setting the viewport for the screen

- Toggling state of the GPU
- while the GPU is rendering the current frame the CPU typically prepare command for the next frame
- GPU parse the commands that are written into the command buffer by the CPU.
- CPU assists GPU in double buffering.

1.4 Graphics pipeline

In computer graphics, there is a need to define a conceptual model for a graphics system to render a 3D scene on to a "what computer displays" and these set of steps are collaboratively referred to as graphics pipeline or rendering pipeline , Once a 3Dimensional model is generated, graphics pipeline is the set of process of turning that 3D scene into what the computer displays on 2D screen. Moreover, there is no as such individual graphics pipeline that would perform successfully for variety of cases, since the steps required for these operational cases are software and hardware dependent and it is also associated with the display requirements of end user as well. However, in order to combine similar conceptual models, we require aid of graphics application programming interfaces (APIs) such as OpenGL, DirectX and Vulkan. These API's were designed for a given hardware accelerator to regulate the rendering pipeline and unifying similar steps. Graphics pipeline contains various abstraction layers that are both programmable and non-programmable (operated via shaders). as we can see in Fig.1.2[1], it is defined in a logical manner to provide end-user with "what they desire to see on the screen!".

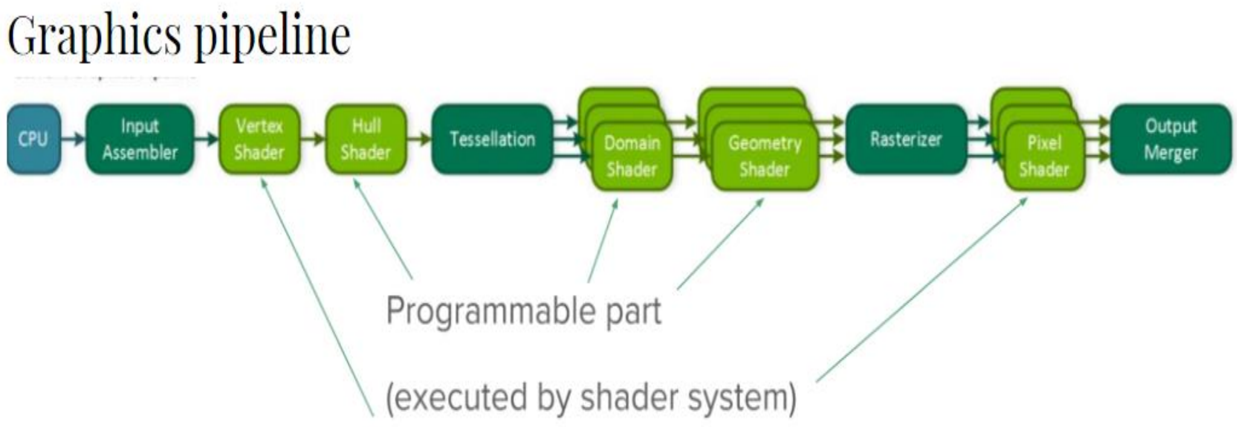


Fig.1.2 Graphics Pipeline[1]

Geometric processes: as it can be seen in Fig.1.2 geometry stage comprise of input assembly, Vertex shading and Primitive assembly. Plus non-compulsory stuff such as Domain shader Tessellation shader, Geometry shader and output merger.

1.4.1. Features of Command Buffer parser :

- A point of contact where GPU and CPU are synchronized
- It deconstructs or parses command buffer (which contains set of commands)
- commands are been directed by command buffer parser to down the rendering pipeline .
- it stays idle till all rasterization finishes it's all draw commands (or draw calls) and then bind it as texture.

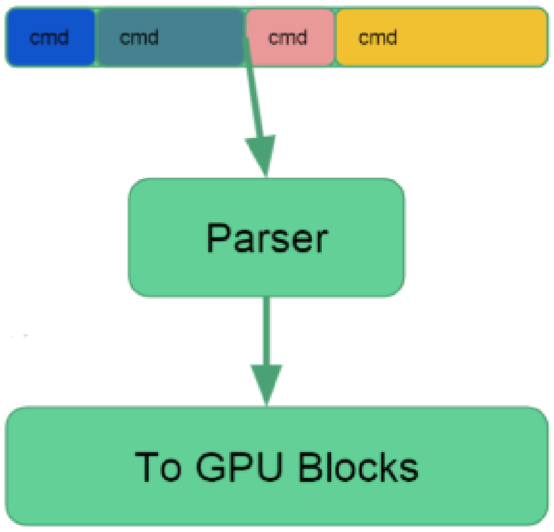


Fig.1.3 Command buffer parser via GPU

1.4.2. Input assembler:

It is a component of input assembly unit Fetches and it is directly coupled with Central processing Unit(CPU) . being one of the initial stages of rendering pipeline they are directly linked with main memory(ROM). Generally GPU confines multiple input assembler unit . it performs following operations:

- From main memory vertices/indices are retrieved by input assembler.

- It possesses a reuse cache for vertices. It is likely to have a hit as Different primitives(triangles) share vertices.in case of cache miss vertices are sent for transform into the shader system.
- Usually GPU is embedded with more than one input assembly unit which accelerates load sharing. Generally load sharing is done at draw call level. For example, individual AI units are directed to process 128 indices.

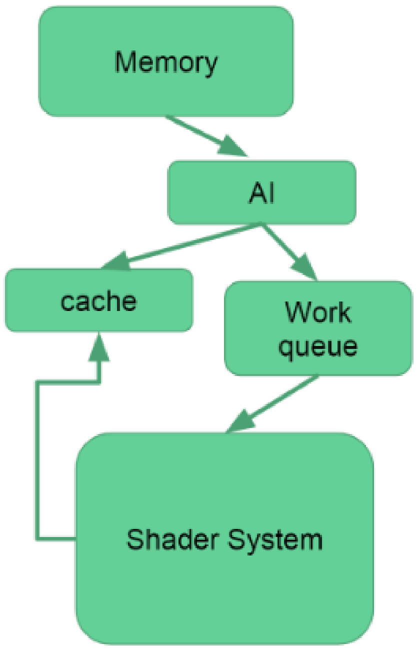


Fig1.4 Input Assembler

1.4.3. Vertex shading:

It is the first stage in rendering pipeline which is completely programmable. Its operation is executed by shader core(will be discussing later) .It has various cache allotments such as positional cache , which is used to store primitive data at input assembler stage. Others attributes such as texture and color mapping are stored in separate cache units(attribute cache) which are in need only at pixel shading stage. being the one of the initial programmable stages in pipeline vertex shading is programmed using APIs such as Vulcan, DirectX etc.

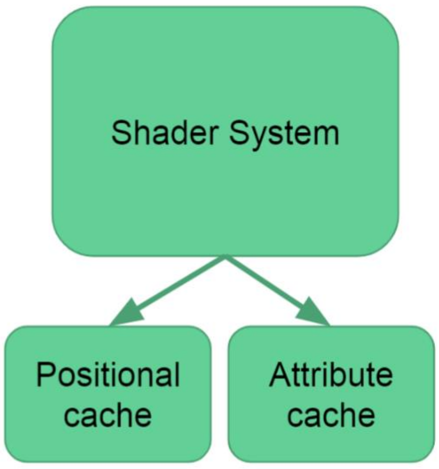


Fig.1.5 Vertex Shader

1.4.4. Primitive assembly Unit:

In this unit, primitive data is retrieved from positional cache. the translation and rotation is carried out after prospective projection (dividing by w) followed by scraping operations such as back face culling, front face culling .

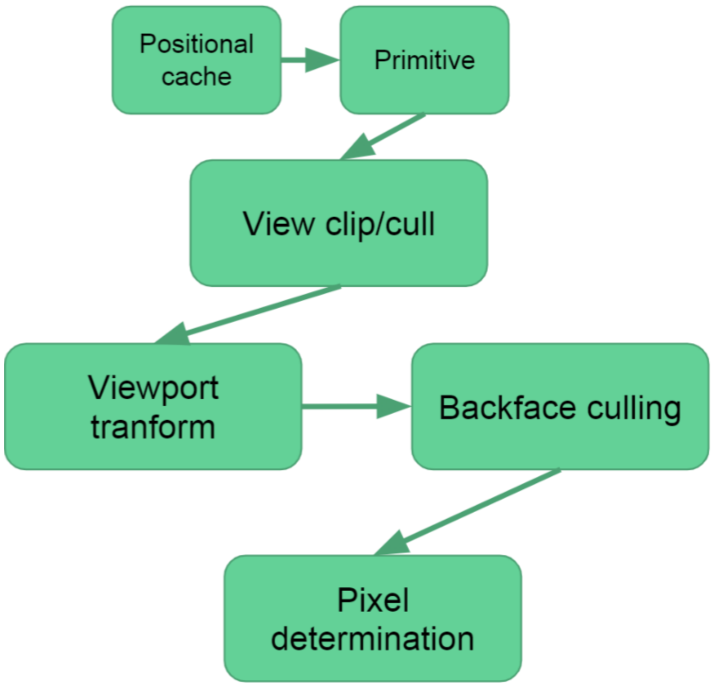


Fig.1.6 Primitive Assembly unit

1.4.5.Rasterization Unit -It performs following operations and tasks:

- Determines which triangle is covered by which set of pixels.
- It operates in hierarchical approach, that is in layered meshes(mesh inside mesh). There should be at least two layers.
- Performs coarse rasterization(discussed in next section)
- Multiple rasterizers are embedded in GPU dedicated towards different portion of the screen.
- for efficient culling/discarding of primitives it performs hierarchical Z and early Z(discussed later).
- quad pixels (2 x 2) are assembled/accumulated by rasterization unit
- job is sent to the shader system when sufficient quads are assembled.

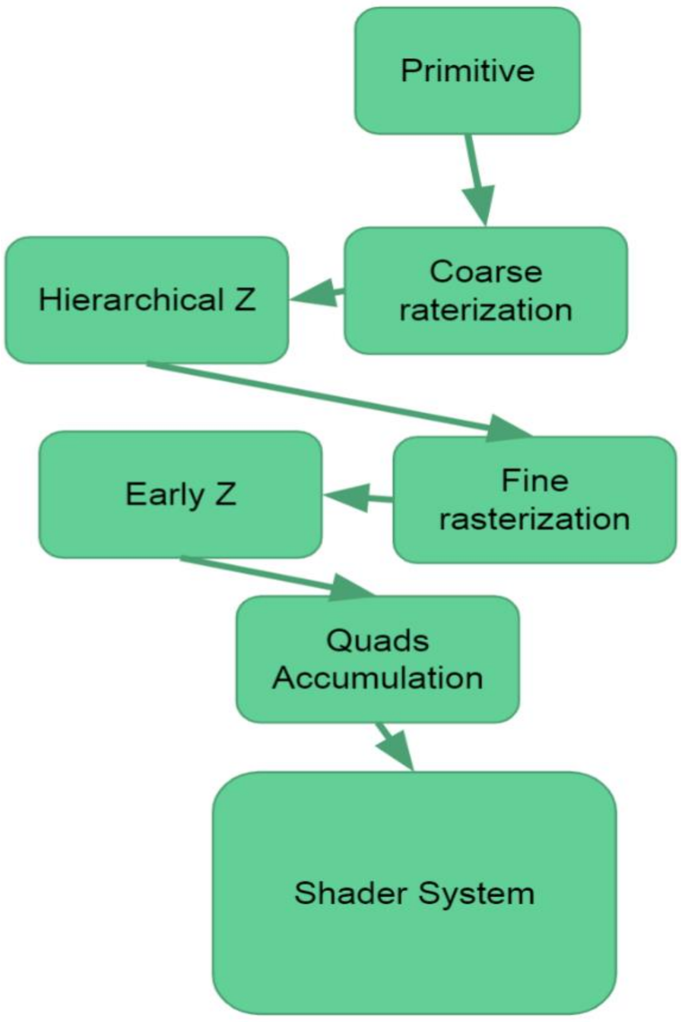


Fig.1.7 Rasterization unit

1.4.5.1 Coarse rasterization: it involves following steps:

- Dividing screen to form numerous "tiles"
- Testing of triangles(primitives) against tiles obtained.
- Here unnecessary tests cases are not processed if the triangle doesn't hit the tile.

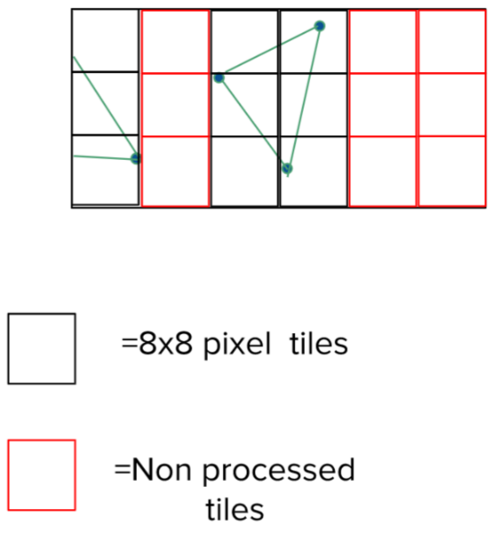


Fig.1.8 Coarse Rasterization

1.4.5.2 Hierarchical Z unit: performing early rejections of triangles is its key functionality. It keeps record for min and max z for each tile. Rejection is carried out if triangle's min z is greater than tile's max z. it also handles fast clearing of Z-buffer.

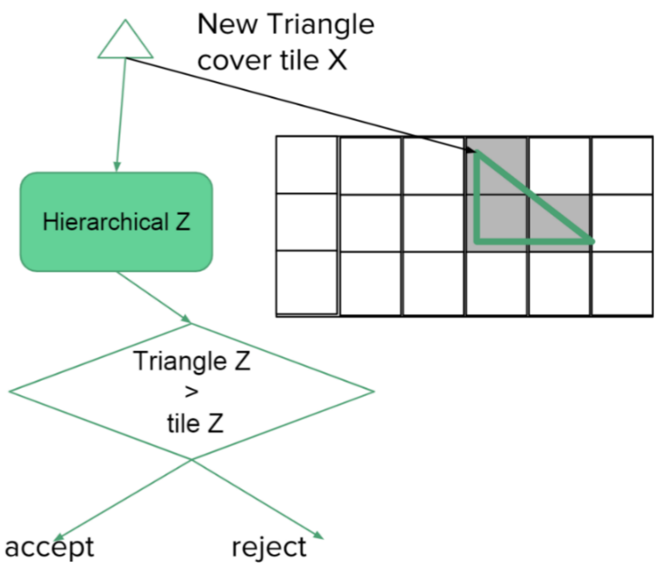


Fig.1.9 Hierarchical Z unit

1.4.5.3 Early Z: this unit is similar to that of Hierarchal-Z but it is carried out at sample level. Moreover before Depicting pixel's color it's depth is computed on the first place. After checking the test cases individually and if they pass then only the pixel shader is executed

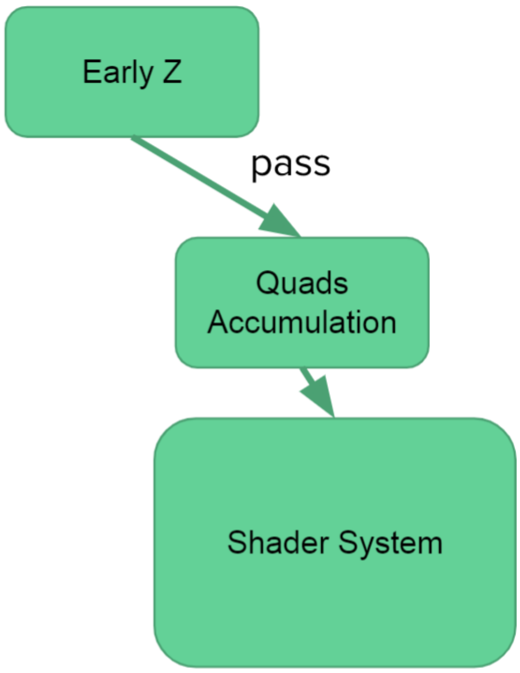


Fig.1.10 Early Z unit

1.4.6 Fragment shading:

- Fragment shader comes into the picture once the rasterizer loaded with sufficient quads(2x2).
- the size of quad wave is usually architecture dependent and normally 16 or 8 quads are transmitted together. The transmission of quad wave is carried out by shader system.
- for better visual quality and performance MIPS(Millions Instruction Per Second) level is required , which is estimated using varying rate(derivative) of the quads.
- GPU's can manipulate and process thousands of instructions on a single go.

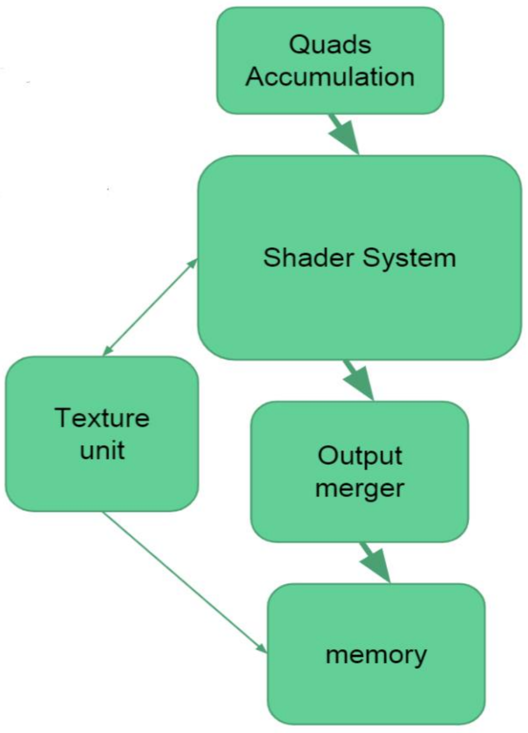


Fig.1.11 Fragment Shading [1]

1.4.7 Pixel shading: in this stage shading o pixels is carried out.in this section we'll discuss about its bottlenecks:

- Only a single primitive may be enclosed by a quad.
- Ghost pixels are formed In event where primitive do not touch a quad , and are generally created along the edges.
- for derivative and MIPS level calculations we essentially need ghost pixel .
- ghost pixels are not a problem if primitives are big enough.
- A lot of unintended overshading is done if such needle-shaped and tiny triangles are created.
- Although Thousands of pixel shading operation can be completed in a flight at an instant.
- Tons of threads are created due to these small triangles which is a bottleneck.

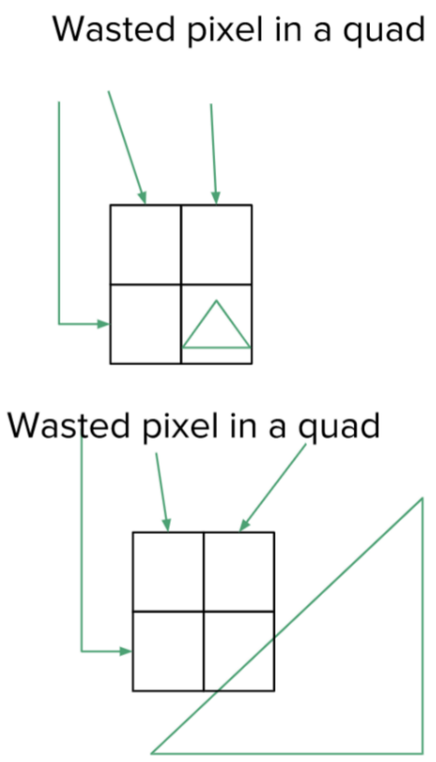


Fig.1.12 constraints in pixel shading [1]

1.4.8 Texture unit:

- Textures are generally retrieved by shaders.
- Texture unit performs the requests to access texture.
- Multiple shader cores can be served by single texture unit.
- Texture unit performs Decompression and texture interpolation.
- If the cache doesn't possess required texture, then:
    - The texture would be fetched from main memory.
    - Typically, very high latency thousands of cycles
    - Suppose you want to cook something
    - each time you need an ingredient   you would be going to a shop a mile away .

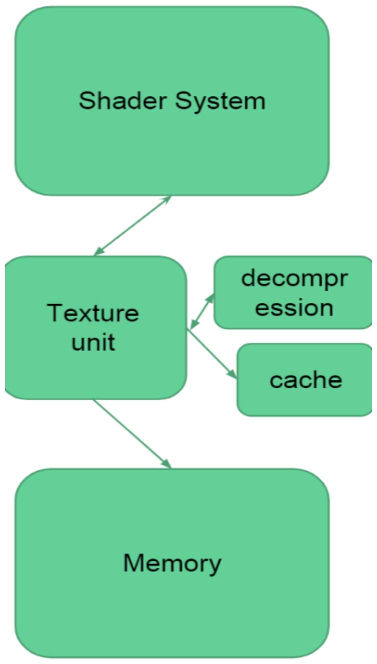


Fig.1.13 texture unit

1.4.9. Output merger:

- it is the last stage of rendering pipeline and is also referred to as Raster ops
- to render the target pixel color is exported
- after this stage the final output is set to be positioned main memory
- Also perform blend operations
- Limited number of pixel operation per clock

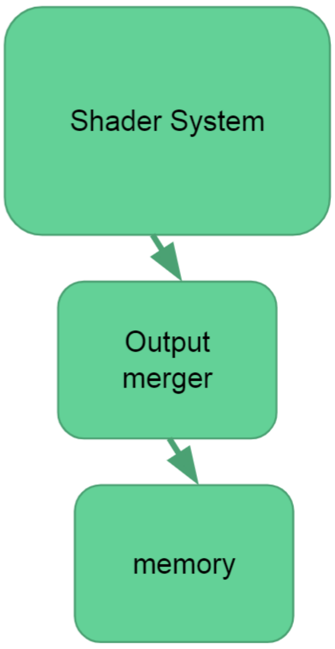


Fig.1.14 Output Merger

1.4.10 Shader Core:

- Part of GPU which is programmable
- GPU contains multiple shader core.
    - In parallel fashion a lot of load sharing and workload can be done.
    - Example: Geforce RTX 2080 has 2944 cude cores
- Typical Shader core can execute 16 instructions in parallel, as it has 16 scalar ALUs.
- In one particular cycle each ALU can execute one Floating point(32-bit) instruction.
- Single program counter(PC)
- Comparing with CPU, shader cores are less complex
    - Execution is carried in ordered fashion
    - It supports multithreading
    - Excellent in cloaking latency
    - No conjecture
    - No branch predictions
    - But very fast at context switching
- Program counter is shaded by multiple ALU's
- In order to perform context switching and to work in parallel for 16 ALUs the Register file is big enough .
- for individual threads 16 slots are there in each register.

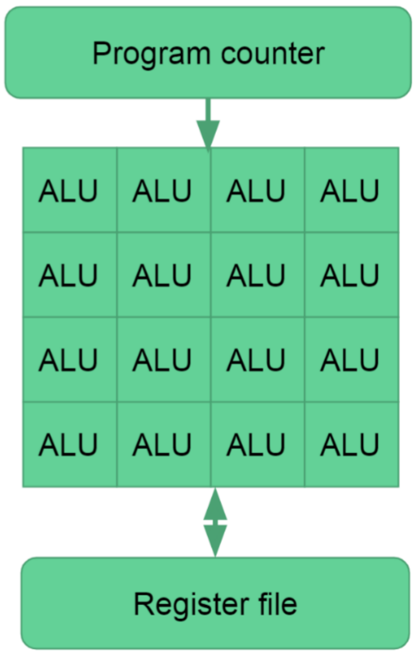


Fig.1.15 Shader Core [1]

CHAPTER-2

LITERATURE SURVEY : RAYTRACING

2.1 Ray tracing Background:

Ray tracing was firstly developed in 16 century, but for computer graphics it came into picture in 1979 “Compleat Angler” was the first movie that incorporated Ray tracing. In the first decade of 2000s large giants such as Global illumination were using Ray tracing and it was also used in Disney pixars movie ,Monsters University in 2013 it was pixars first movie where they featured shading and lighting using Ray tracer. For rendering a frame individually using Ray tracing it took about 17 to 18 hours , thus we can say that Ray tracing is computationally expensive as compared to the traditional algorithms.

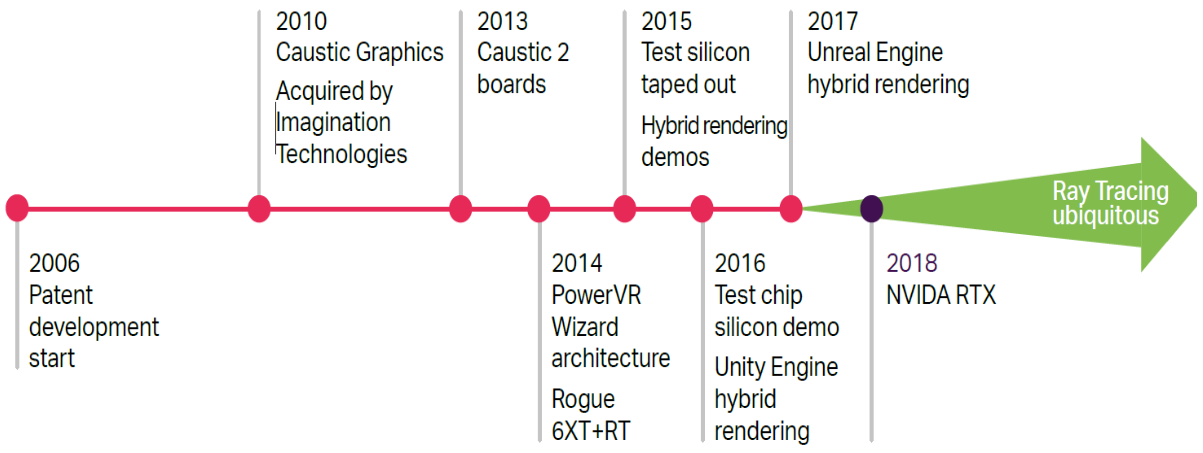


Fig.2.1 Ray tracing development[2]

Starting in 2006, where the patent development of a basic Ray tracer was under progress to the modern-day Ray tracer. Before 2013 big giant like Nvidia was using rasterization for rendering frames in their upcoming gaming GPU .later at the time of 2015 there were several test planned for hybrid based rendering, in which both rasterization and Ray tracing were incorporated which was groundbreaking at that time as in terms of power and performance.

as rasterization requires less computational cost and we can and hence elevate level of detailing by rendering fewer fragments with Ray tracing .In 2016 more advanced hybrid base rendering were established and in recent years(in 2018) the Nvidia RTX was developed which was entirely based on Ray tracing as we can see the advancement in Ray tracing for virtual realism is increasing. Ray tracing is a technique in which we back-trace reflected ray from camera coordinates (point of vision) to the point of intersection and all the way up to the light source or in other words we trace shadow ray. As depicted in Fig.2.2 the basic raytracer operation. Ray tracing provides far more superior results in terms of realism. This technique solved two major issues faced by graphic designers that were Shadows and Reflections.

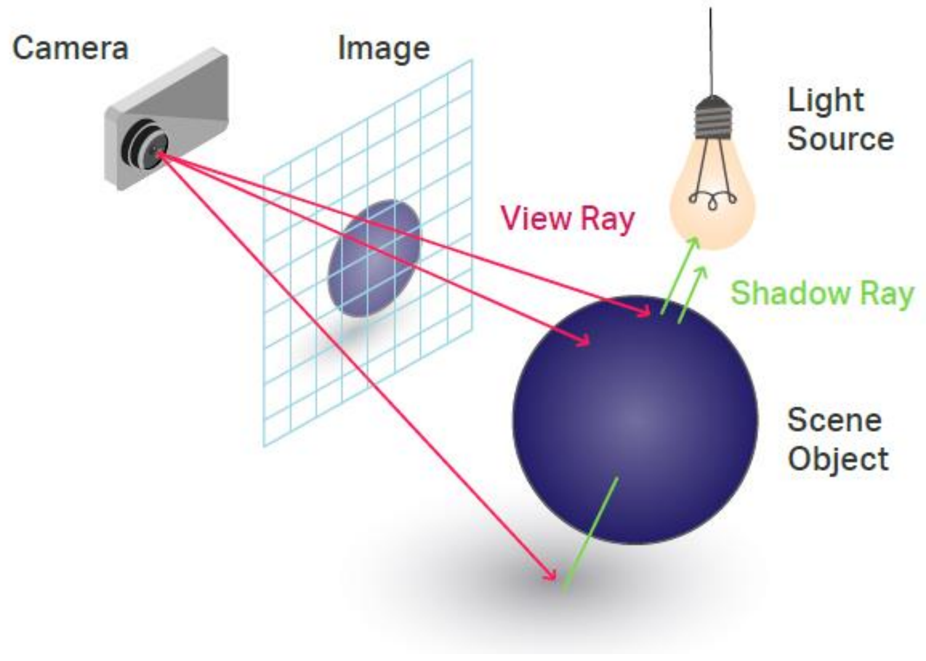


Fig.2.2 Ray tracer's operation [2]

It may be seen as contradictory to elementary physics, but when we are back tracing light ray from viewers eye to the light source the reason behind that is : not majority of light rays come back to viewers eye after hitting the artifact. So, this back-tracing [2] strategy is efficient for GPU's power and performance as we don't need to fire immense number of light rays thus we'll be minimizing computational cost. For all subset objects in the scene are striked with individual rays and are tested for the incoming light at the point of intersection. then the color of the pixel is determined by properties such as material, reflection and texture. Certain scenes that have extensive refectory or refractory surfaces need to be rendered with extensively large number of Rays in order to make scene more Realistic. in 3D Computer Graphics environment visual images are being produced by incorporating optical Ray tracing models which are extensive in photo realism. their operation works as : from an imaginary eye a path is traced to each pixel on a virtual screen and according to lighting and occulation the color of the pixel is determined A visual artist or a programmer mathematically describes a particular three dimensional scene via advanced tools such as directX and Vulcan.

2.2 Bounding volume Hierarchy (BVH):

As the details tend to increase in a scene mesh density also becomes extensive and so is the time to render frame. Therefore, in order to mitigate such this timing trade-off, we deploy acceleration structures(such as BVH) in our raytracing algorithm. so, in other words we can say that acceleration structures are those which help to render a frame with minimum time complexity. we can use acceleration structures such as BVH where several bounding boxes are constructed in a hierarchical manner straight from bigger bounding box to primitive level. So here as we can see in Fig.2.3 the ray is striking the head of the rabbit which is enclosed by a bounding box and if we want further details we traverse Deep down the bounding boxes and at the end we obtain Ray-triangle intersection ,although without bounding boxes these intersections were possible but it used to increase an extensive overload on GPU and shaders itself [3].BVH tend to accelerate the rendering time of the frame and are likely to be perfect candidate for real time Ray tracing.

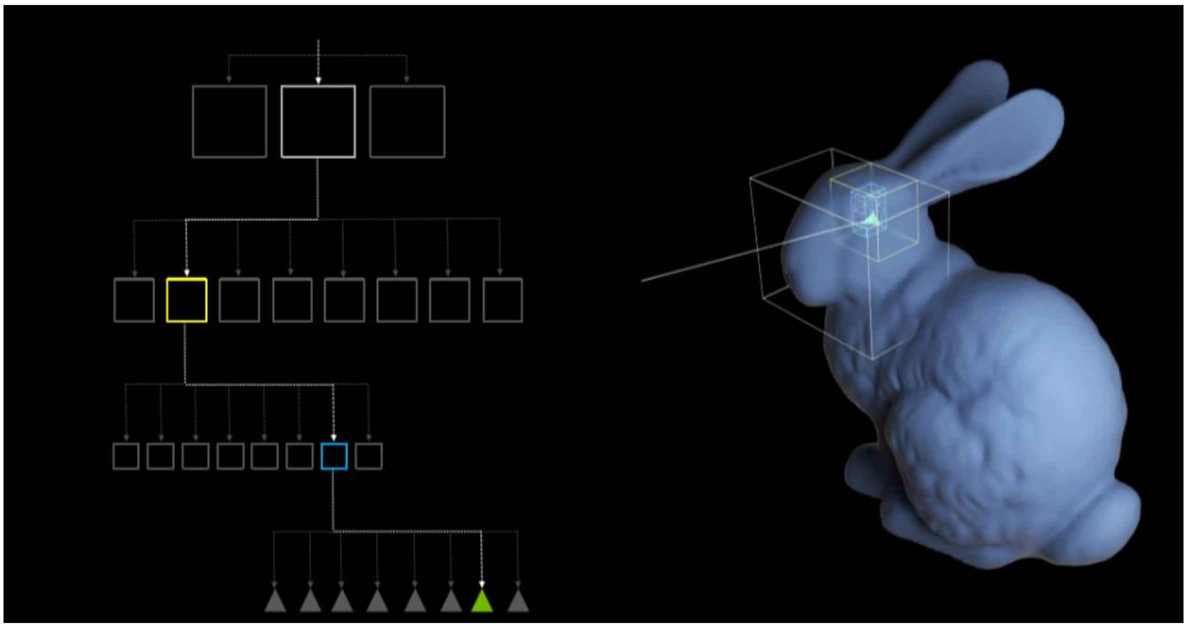


Fig.2.3 Bounding Boxes for Rabbit [3]

For a set of different geometric objects BVH acts as a tree structure. All bounding volumes tend to confine all geometric objects right away from the top level to the leaf nodes of the tree , this is done in order to reduce time complexity and number of test performed .with this strategy the hidden leaf nodes or children need are not traversed if their parents are not intercepted by the ray. Initially we perform BVH building before rendering a particular scene. the BVH building create BVH structure. if the next frame that must be rendered is significantly different from the previous frame then we construct a new BVH structure for that particular scene as well. All these BVH instructions are carried out to fetch Ray-primitive co-ordinates quickly ,due to this BVHs are referred to as acceleration structures. For real time rendering there is a strategy named BVH-refitting in which if there are only minute changes present in the next frame then we update the existing BVH structure which further reduces computational complexity.

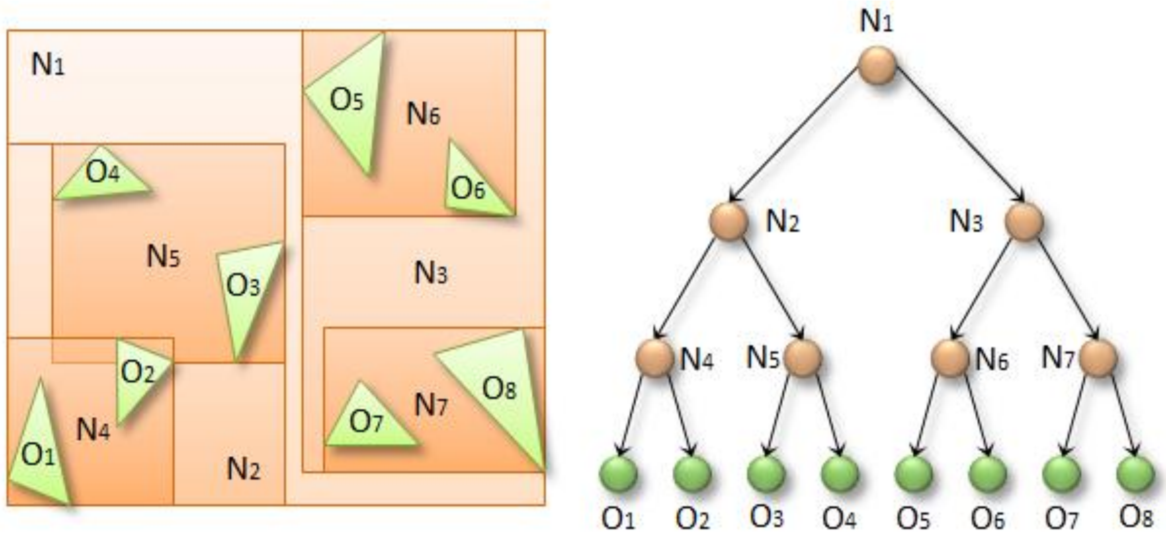


Fig.2.4 Depiction of Bounding Boxes inside Bounding Boxes and triangles inside Bounding boxes

Generally, there are two types of BVH building techniques:

- Top-down: in this approach a parent bounding box is divided into several smaller bounding boxes all the way down to the primitive level. in this a parent is generally divided into two or more branches depending upon the geometric structure.

- Bottom-up: approach the bounding boxes are created from scratch All the Way from primitive level to the parent level and generally these also have two or more parent branches or we can say the construction of bottom up approach is quite complex

Let’s imagine a scenario in which Ray is striking a particular artifact so initially, the ray bounding box intersection takes place and after that Ray travels down the hierarchy all the way to the primitive level in order to give a hit. so, in this way bounding boxes tend to hit primitives forming a structure.

2.3 Different Raytracing Architectures in the industry

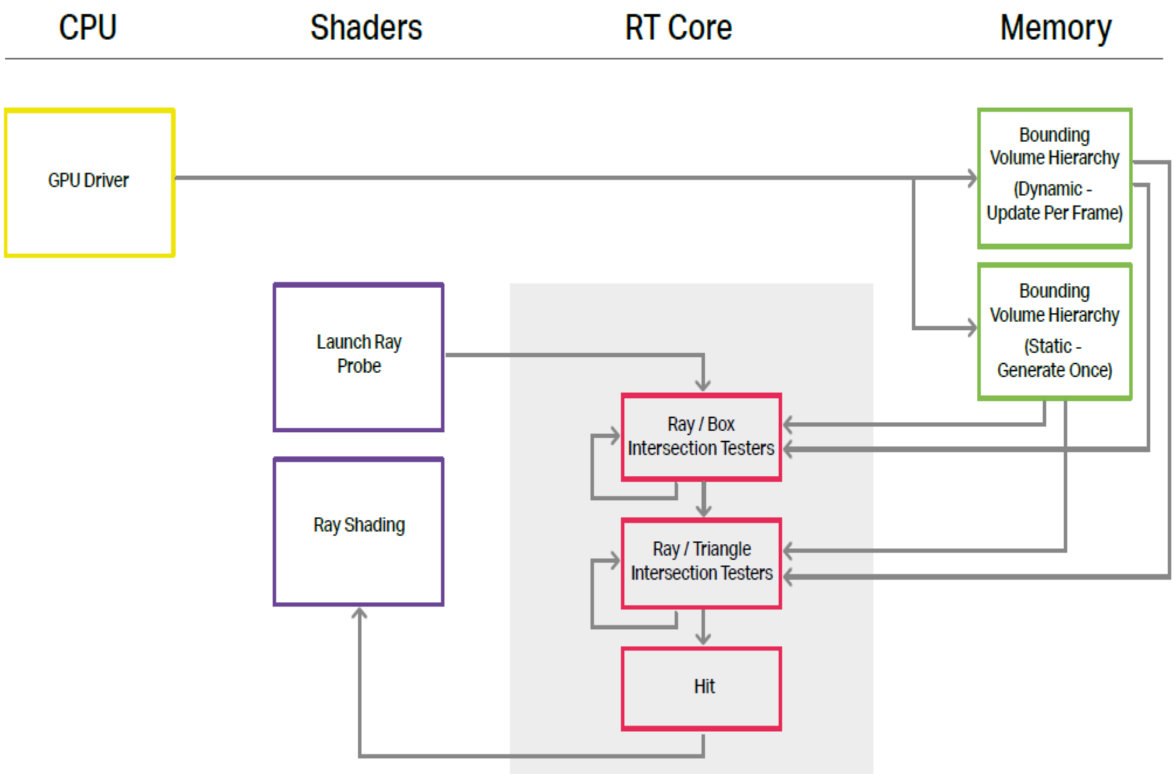

CPU
Shaders
RT Core
Memory
GPU Driver
Bounding Volume Hierarchy (Dynamic - Update Per Frame)
Bounding Volume Hierarchy (Static - Generate Once)
Launch Ray Probe
Ray Shading
Ray / Box Intersection Testers
Ray / Triangle Intersection Testers
Hit

Fig.2.5(a)Raytracing Architectural Flow in current Nvidia RTX [2]

As we can see in Fig.2.5(a) the Nvidia’s RTX architecture a separate RT core is used for all the ray box and Ray triangle intersections which increases its performance although it might require an additional hardware for the GPU itself whereas AMD(Fig.2.5(b)) currently do not have any dedicated unit for Ray tracing computations and all the computations are performed in shaders and thus it has lower performance mattresses

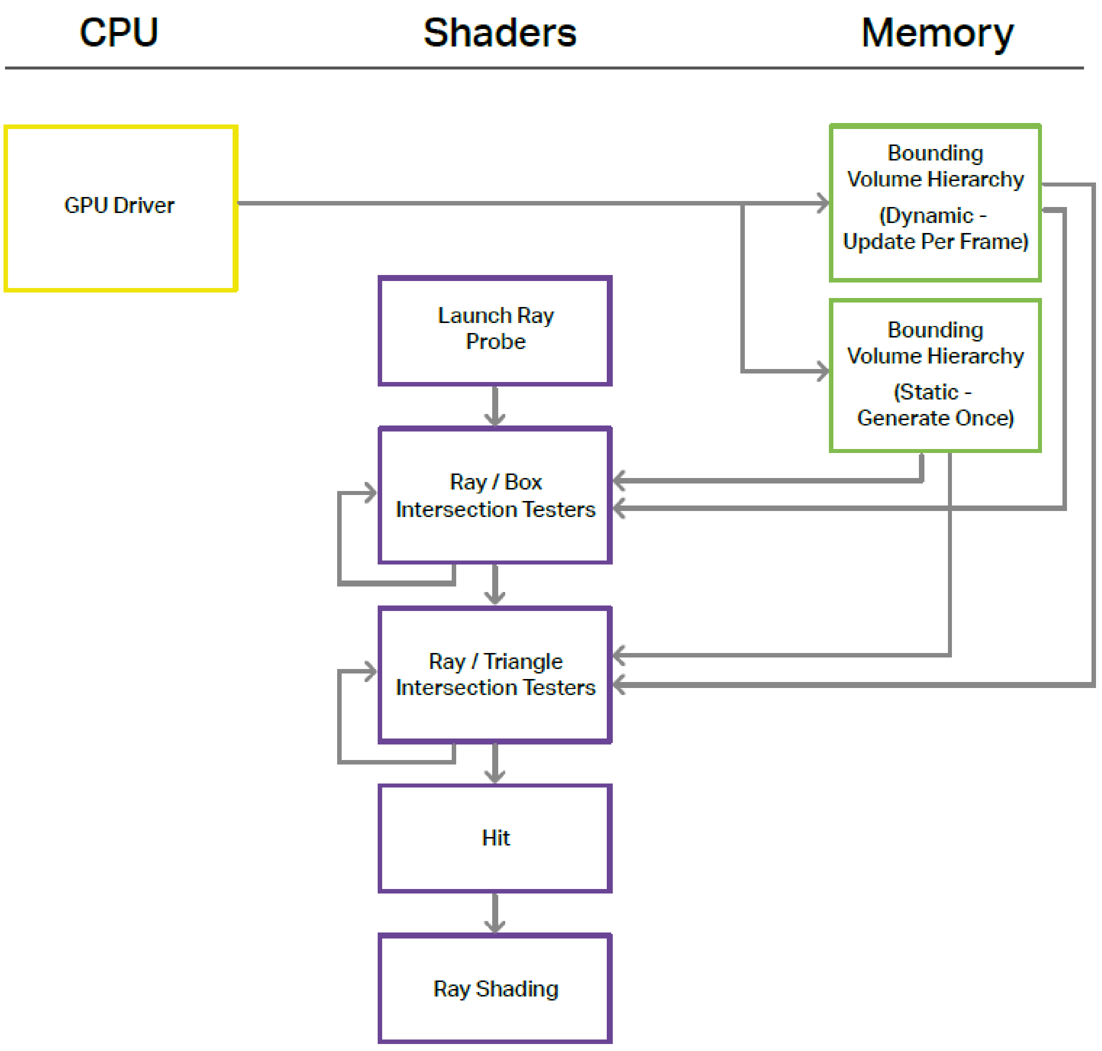

CPU
Shaders
Memory
GPU Driver
Bounding Volume Hierarchy (Dynamic - Update Per Frame)
Bounding Volume Hierarchy (Static - Generate Once)
Launch Ray Probe
Ray / Box Intersection Testers
Ray / Triangle Intersection Testers
Hit
Ray Shading

Fig.2.5(b)Raytracing Architectural Flow in AMD[2]

2.4 What is Watertightness?

The term watertightness is inferred from analogy of a "wall" that is put together by using bricks and plaster as an adhesive. If our wall is constructed and uncertain climatic circumstances such as rain occurs in that case leakages might appear on the other side of the wall so we can say that our wall lacks water tightness, if a good quality wall is established then it'll not compromise its water tightness aspect .Similarly in computer graphics water tightness is referred as creation of unnecessary holes/ false misses or false hits that is some an unintended is rendered at the final image. In other words, we can say that water tightness is one of the main concern modern day graphics enthusiast are working upon [4]. in layman language we can say that “Ray tracings is the cause and water tightness is the consequence”. In a desire of next to reality graphics ,the mesh density/detailing increases in a scene which results in thinner/needle shaped triangles and eventually when we perform ray-triangle intersection (after ray-box intersection) test unwanted false-misses/false hits appear in our rendered image.

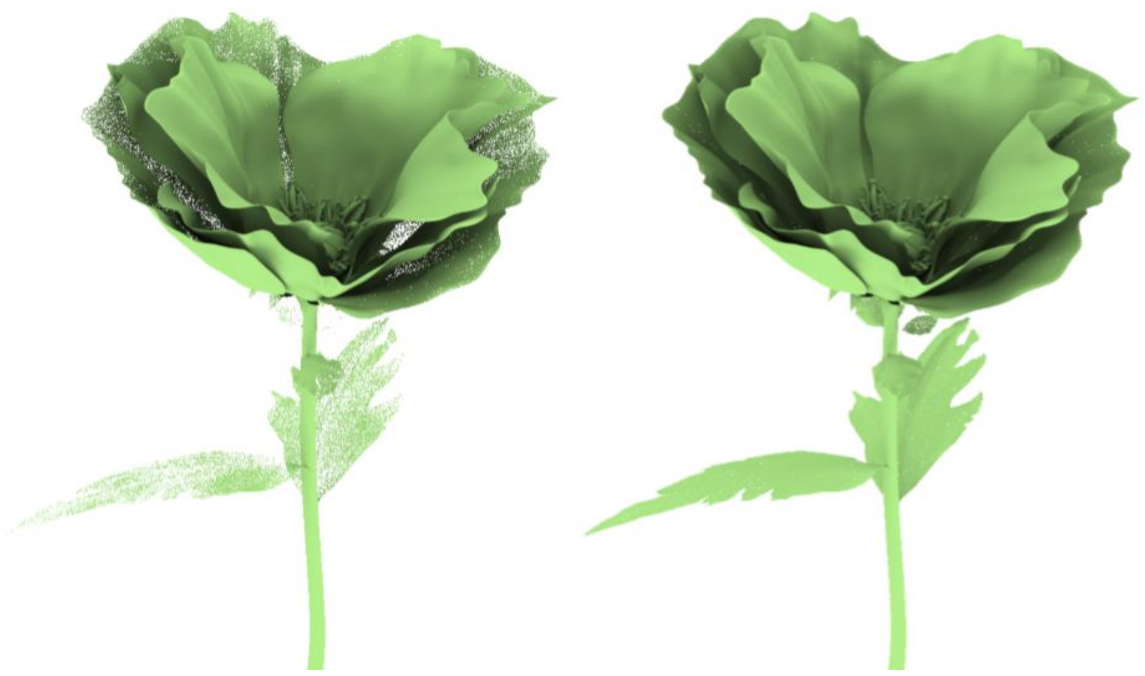


Fig.2.6 Non-Watertight vs Watertight Artifact [3]

One of the key factors for such holes/false misses is improper Ray triangle intersection calculation. Primarily due to incorrect calculation of ray triangle intersection point as a result of floating-point errors of the compiler for a Ray-triangle as well as Ray box-intersection. Water tightness sacrifices as we keep on increasing number rays per primitive, so in other words if we want our rendered output to be more realistic than watertightness is compromised hand in hand. In order to work on this issue various algorithms have been introduced in computer graphics recently where different geometric transformations and translations are steered. And different strategies such as using double point Precision(64-bit) instead of single point(32-bit) to calculate Ray triangle intersections. Generally majority of false misses arises in the case of shared vertices or edges of the primitive.

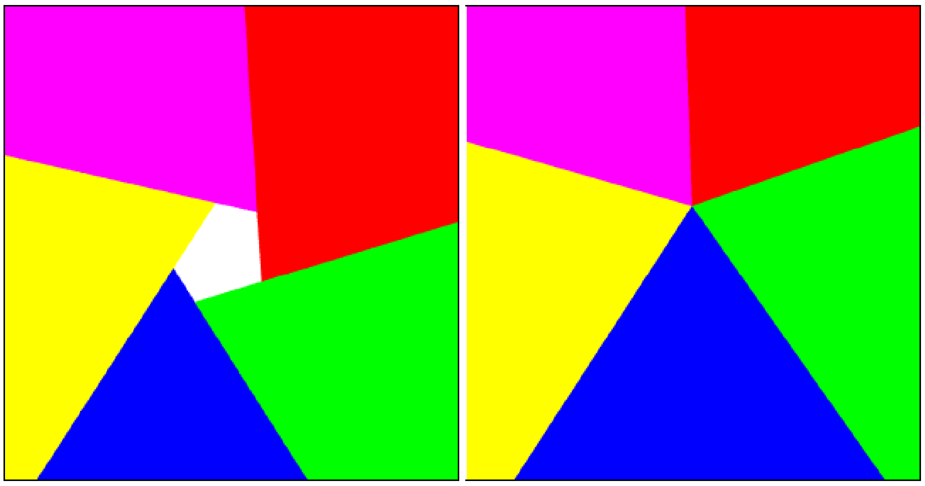


Fig.2.7 Non-Watertight vs Watertight [5]

It can be inferred from Figure 2.6 that triangles/polygons (that is blue, yellow, pink, red and green) share a common edge at the center. when the rays strike at this center point ,false misses occur in rendered output. Since floating point calculation for a such tiny region requires higher precision ,so 32-bit precision(normal) might get fail. We will be discussing how to mitigate these false misses in order to create extra ordinary real-time experience for the end user during the gameplay or any other animations. there are different algorithms that have been in the vision of graphics industry that work on this water tightness failure. Moreover, the main scope of our thesis revolves around determining that out of all these algorithms which one can be preferred for real time Ray tracing and mitigate water tightness Simultaneously . These algorithms will further be under light in terms of performance and rendering time in the next section.

2.5 Watertight Ray-triangle Intersection algorithms:

Before getting directly towards different algorithms, first we'll need to understand "**what is a ray**?" with respect to algorithms. In 3D a ray is an important computational construct for Ray tracing. both in mathematics and Ray tracing a ray is denoted as a three-dimensional half-line. for the better understanding we will be using different construct in ray such as rays-origin rays-direction. considering a ray *R(t)* with origin *O* and normalized direction *D* is defined as:

$$R(t)= O + tD \qquad (1)$$

And as all of us know that a triangle is constructed using its three vertices, so let three vertices be V_0 , V_1 and V_2 .since all these vertices are placed in three dimensional cartesian space so they will have their respective x, y and z coordinates[4]. The algorithms discussed below will be using **barycentric coordinates** as key ingredient to determine whether the ray have intersected the triangle or not. Barycentric coordinates are just like weighted averages. By observing barycentric coordinates we can tell whether a specific coordinate lies inside or outside the Triangle.

2.5.1 Moller-Trumbore intersection algorithm:

In this algorithm firstly we translate triangle with vertices to V_0 , V_1 and V_2 towards the origin (0,0,0) and transform it in order to form a unit Triangle which is placed in y-z plane and direction of ray is aligned with x-direction. the transformation is done so that if a triangle is placed in any arbitrary space(coordinates) and ray with a particular origin(O) is fired towards the triangle with certain directional vector D[4], then we can figure out whether it is rendered or not depending upon a hit or a miss.

Mathematically:

A point, $T(u,v)$, on a triangle is given by

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2, \qquad (2)$$

Where *T(u,v)* is the point of intersection and *(u,v)* are the barycentric coordinates. Which satisfies the condition $u \geq 0$, $v \geq 0$ and $u + v \leq 1$.

Since the Ray and Triangle would be meeting at point *T(u,v)* , then we can infer that, R*(t)*= *T(u,v)* (Equating (1) and (2)).

$$O + tD = (1-u-v)V_0 + uV_1 + vV_2 \qquad (3)$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \qquad (4)$$

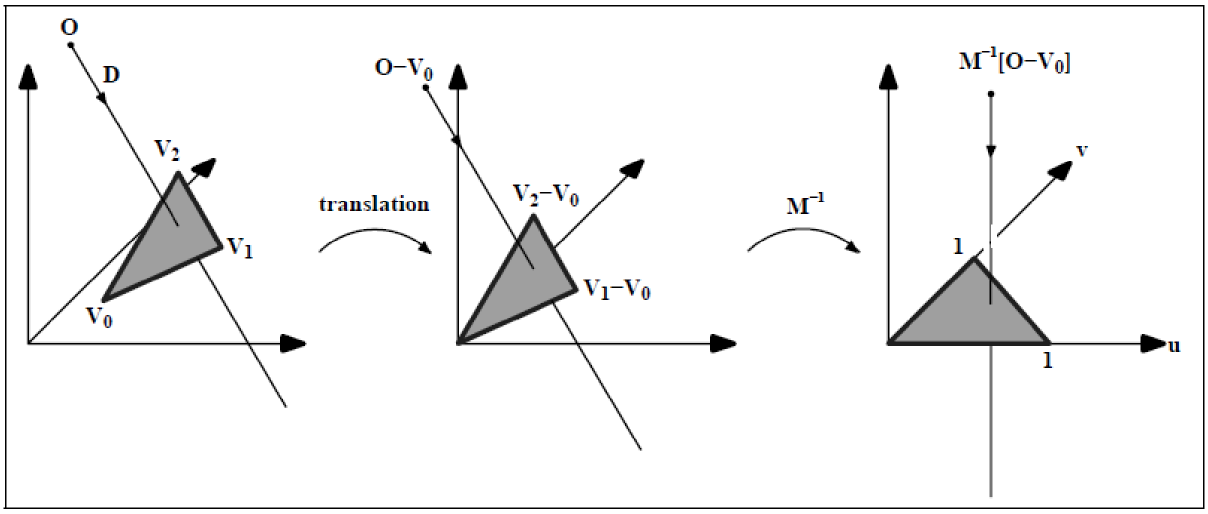


Fig.2.8 Translation and change of base of the ray origin[4]

in Fig. 2.8 we can see translation of triangle to origin and rotation. Here *M= [-D, V_1 -V_0 , V_2 -V_0]* this transformation is the key ingredient in Moller-Trumbore ray triangle intersection algorithm.

Denoting $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ and $T = O - V_0$, the solution to equation (4) is obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{bmatrix} |\,T,\ E_1,\ E_2\,| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{bmatrix} \qquad (5)$$

¿From linear algebra, we know that $|A,\ B,\ C| = -(A \times C) \cdot B = -(C \times B) \cdot A$. Equation (5) could therefore be rewritten as

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \qquad (6)$$

where $P = (D \times E_2)$ and $Q = T \times E_1$. In our implementation we reuse these factors to speed up the computations.

Implementation:

In C++ code a function is defined that takes ray parameters such as ray origin(orig) ,direction(dir) and triangle vertices(tri),minimum distance(tmin) and maximum distance (tmax)and returns "T_hit". the Boolean expressions front_face_culling,culls front faces(for the primitve) and cull_back_face are for culling back facing triangles. This culling is done in order to eleminate unwanted triangles that might cause over shading in the final render image. Moreover, various functions such as sub() ,cross() and dot() are explained in Appendix-I . Epsilon eps is taken as the machines epsilon which depends on on precision of the compiler

Code referenced from [4]

```
 float MT(const vec3f orig, const vec3f dir, float tmin, float tmax,
                      const triangle tri,
                      float& bary0, float& bary1, bool& isFrontFacing,
                      bool cullFrontFaces, bool cullBackFaces )
{
  const vec3f e1 = sub(tri.vtx[1], tri.vtx[0]); //translation of V₁ and V₀ for origin
  const vec3f e2 = sub(tri.vtx[2], tri.vtx[0]); //translation of V₂ and V₀ for origin
  const vec3f s1 = cross(dir, e2);
  float denom = dot(s1, e1);
  float const eps = 1e-8f;

  if (cullBackFaces && denom < eps)
      return tmax;
  else if (cullFrontFaces && denom > eps)
      return tmax;
  else if (denom < eps && denom > -eps)
      return tmax;

  float const invd = 1.f / denom;
  const vec3f d = sub(orig, tri.vtx[0]);
  float const b1 = dot(d, s1) * invd;
  const vec3f s2 = cross(d, e1);
  float const b2 = dot(dir, s2) * invd;
  float const tHit = dot(e2, s2) * invd;

  if (b1 < 0.f || b1 > 1.f || b2 < 0.f || b1 + b2 > 1.f || T_hit < tmin || T_hit >
tmax) {
         return tmax; //this if block is executed when point lies on any vertex or
                                                                     edge //or outside
  } else {
      bary0 = b1;
      bary1 = b2;
      isFrontFacing = denom > 0.0f; // if denom > 0 then front facing else backfacing

      return T_hit; //returning hit distance
  }
```

barycentric coordinates b1,b2(or u,v) are checked and if they satisfy the conditions as:

$0 \leq b1 \leq 1$, $0 \leq b2 \leq 1$, $b1 + b2 \leq 1$

Then this distance(T_hit) is calculated , and if not then we simply don't render that triangle

2.5.2 Woop’s Algorithm

Moller-trumbore algorithm lagged water tightness and gave false misses in cases such as shared edges and shared vertices of the triangle so in order to mitigate false misses or holes we will be moving to another algorithm That is woop’s algorithm, in this algorithm there is a provision of fall back to double precision in which instead of 32 bit, 64 bit double calculations are carried whenever there is a shared edge or a shared vertex.

Steps:

1. In the first stage, in order to simplify intersection problem, ray parameters(such as origin and direction) and triangle vertices undergo an affine transformation. As affine transformation, we choose a transformation that simplifies the ray R to the unit ray R0 with origin P0 = (0,0,0) and direction D’ = (0,0,1)[5]. Here we take the Z-component of the ray direction D has the largest absolute value. Here at time of ray generation matrix *M* is precalculated as and for all successive ray-triangle intersection it is reused.

$$M = \begin{pmatrix} 1 & 0 & -D_x/D_z \\ 0 & 1 & -D_y/D_z \\ 0 & 0 & 1/D_z \end{pmatrix}$$

2. Taking origin P=(0,0,0) as relative point and using transformation *M*, we’ll be calculating transformed triangle vertices .

$$A' = M \cdot (A - P),$$
$$B' = M \cdot (B - P),$$
$$C' = M \cdot (C - P).$$

3. Using Benthin[2006] we will calculate scaled barycentric coordinates

$$U = D' \cdot (C' \times B'),$$
$$V = D' \cdot (A' \times C'),$$
$$W = D' \cdot (B' \times A').$$

4. After intersection with unit ray D’ = (0,0,1),the above formulas for *U,V,W* gets simplified for 2D as:

$$U = C'_x \cdot B'_y - C'_y \cdot B'_x,$$
$$V = A'_x \cdot C'_y - A'_y \cdot C'_x,$$
$$W = B'_x \cdot A'_y - B'_y \cdot A'_x.$$

5. Ray doesn't hit the triangle, if $U < 0, V < 0, or\ W < 0$. Then we calculate the determinant of the system of equations as $det = U + V + W$. If this determinant is zero, the ray is co-planar to the triangle (that is parallel to plane in which triangle is placed) and we adopt it as a miss.

6. Moreover, by dividing U,V,W by det we can calculate actual barycentric coordinates (u,v,w) .if conditions mentioned in (5) don't fail , then we interpolate Z-values for transformed varices in order to compute scaled hit distance T as :

$$T = U \cdot A'_z + V \cdot B'_z + W \cdot C'_z.$$

7. The actual distance(tHit) can be calculated by division of T(scaled distance) by det
8. If test cases fails for shared edge or shared vertex conditions then we'll do a fallback to double precision(64-bit).

Code:

Code Referenced from [5]

```
float WDP
(
const vec3f origin, const vec3f dirtn, float tMin, float tMax, const triangle tri,
float& bary0 ,float& bary1,bool& isFrontFacing,bool cullFrontFaces, bool cullBackFaces)
{

float hitT;

// Calculate dimension where the ray direction is maximal
int kz = max_dim(abs(dirtn));
int kx = kz + 1; if (kx == 3) kx = 0;
int ky = kx + 1; if (ky == 3) ky = 0;

// Swap kx and ky dimension to preserve winding direction of triangles
if (dirtn[kz] < 0.0f) {
   // Swap kx and ky
   int ii = kx;
   kx = ky;
   ky = ii;
}

// Calculate shear constants
float Sx = dirtn[kx] / dirtn[kz];
float Sy = dirtn[ky] / dirtn[kz];
float Sz = 1.0f / dirtn[kz];  // Calculate vertices relative to ray origin
vec3f A = tri.vtx [0] - origin;
vec3f B = tri.vtx [1] - origin;
vec3f C = tri.vtx [3] - origin;


// Perform shear and scale of vertices
```

```
const float Ax = A[kx] - Sx*A[kz];
const float Ay = A[ky] - Sy*A[kz];
const float Bx = B[kx] - Sx*B[kz];
const float By = B[ky] - Sy*B[kz];
const float Cx = C[kx] - Sx*C[kz];
const float Cy = C[ky] - Sy*C[kz];

// Calculate scaled barycentric coordinates
float U = Cx*By - Cy*Bx;
float V = Ax*Cy - Ay*Cx;
float W = Bx*Ay - By*Ax;

// Fallback to test against edges using double precision
if (U == 0.0f || V == 0.0f || W == 0.0f) {
          double CxBy = (double)Cx*(double)By;
          double CyBx = (double)Cy*(double)Bx;
          U = (float)(CxBy - CyBx);
          double AxCy = (double)Ax*(double)Cy;
          double AyCx = (double)Ay*(double)Cx;
          V = (float)(AxCy - AyCx);
          double BxAy = (double)Bx*(double)Ay;
          double ByAx = (double)By*(double)Ax;
          W = (float)(BxAy - ByAx);
   }


// Perform edge tests. Moving this test before and at the end of the previous conditional
gives higher performance.
if (cullBackFaces) {
   if (U < 0.0f || V < 0.0f || W < 0.0f) {
          return tMax;
   }
}
else if (cullFrontFaces) {
   if (U > 0.0f || V > 0.0f || W > 0.0f) {
          return tMax;
   }
}
else {
   if ((U < 0.0f || V < 0.0f || W < 0.0f) && (U > 0.0f || V > 0.0f || W > 0.0f)) {
          return tMax;
   }
}

// Calculate determinant
float det = U + V + W;

if (det == 0.0f) {
   return tMax;
}

// Calculate scaled z coordinates of vertices and use them to calculate the hit
distance.
const float Az = Sz*A[kz];
const float Bz = Sz*B[kz];
const float Cz = Sz*C[kz];
float T = U*Az + V*Bz + W*Cz;
```

```
if (cullBackFaces) {
    if ((T < tMin * det) || (T > tMax * det)) {
            return tMax;
    }
}
else if (cullFrontFaces) {
    if ((T > tMin * det) || (T < tMax * det)) {
            return tMax;
    }
}
else {
    int det_sign = sign_mask(det);
    if ((xorf(T, det_sign) < tMin * xorf(det, det_sign)) || (xorf(T, det_sign) > tMax
* xorf(det, det_sign))) {
            return tMax;
    }
}

//Faceness is only meaningfull for triangles that are hit
isFrontFacing = (det > 0.0f) ? 1 : 0;


// Normalize U, V, W, and T
const float recpDet = 1.0f / det;

bary0 = V*recpDet;
bary1 = W*recpDet;
//hitBary.x = U*rcpDet;

hitT = T*rcpDet;
return hitT;
}
```

2.5.3 Microsoft (DXR) Ray triangle intersection algorithm

In previous algorithms that is Moller-Trumbore and Woop’s algorithm. The woop’s one was quite superior in terms of mitigating false misses, but it too had few misses in the rendered image ,so in order to eradicate even those minute misses we will be using DXR algorithm. it is a algorithm in which shared edges as well as shared vertices are prioritize according to a rule-base, and surprisingly we have observed that it had solved significant corner cases.

i) The first case is one of the barycentric coordinates u,v,w is zero . in this case a ray hits an edge . if that edge is shared by two triangles then ray will hit only one Triangle .there are two cases for edges . one is top edge the next is left edge.

- in the top edge case the triangle that has its top edge shared will be rendered .

- in the left edge case the triangle having its left edge shared will be chosen

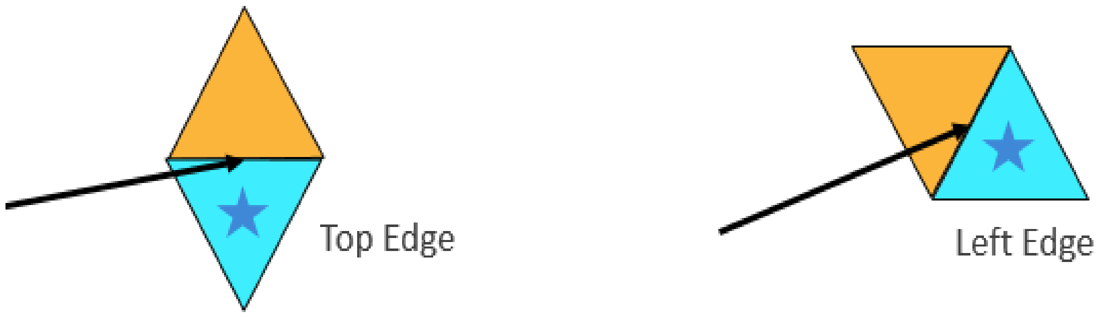


Fig.2.9(a) Triangle with shared edge

Here blue triangle wins , if any of the above two edge cases are arises.

ii) second case is two of the barycentric coordinates are 0 this is a case when ray hit a shared vertex assume there is a closed fan. the requirement is to hit only one Triangle for this scenario there are two rule bases.

- First one is left North edge : the left and North edge of a triangle in a closed fan is rendered in this case
- north direction :the triangle whose edge is facing the north get selected

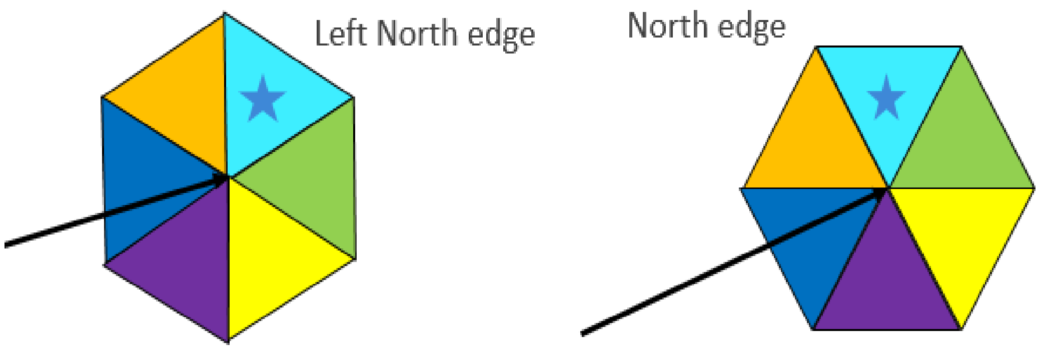


Fig.2.9(b) Fan of triangles with shared edge

CHAPTER-3

DESIGN FLOW AND MODELS

3.1. Design Flow:

Firstly, in order to understand the VLSI design flow, one should understand that an IC or Chip cannot be fabricated in a flight ,it requires numerous steps and processes in order to produce high-performance bagged circuit . it is just like planning an architectural to construction flow for a house or building, but it is relatively a challenging thing for designers.

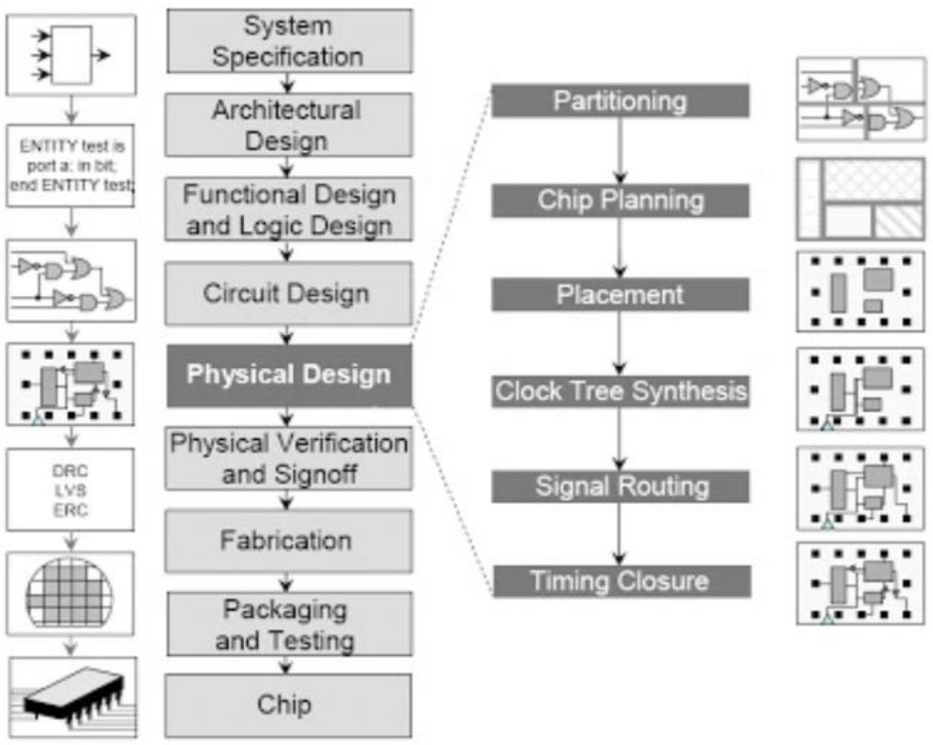


Fig 3.1 VLSI Design Flow

- **System Specification:** In this the overall goal and high-level requirements of the system are defined by circuit designers and library and layout designers . the system specification such as how several blocks or how a IP should operate. in other words, concerned mostly about “what would be the basic algorithm behind the product that we are designing?”
- **Architectural design:** here we work on architectural perspective of design under vision. activities like different components such as Memories, Multimedia, Processing

units(CPUs, GPUs) ,Input-Output ports, Power management systems and other integrated components are interacting. here a chief architect states and works on functionalities such as physical dimensions, performance(operating voltage, frequencies) ,process technology(whether 14nm MOS or 10nm FinFET) .some of the popular architectural constraints are :

a) Number of cores and Digital signal processors
b) Power management and requirements
c) Memory latency and requirements
d) Signal integrity for analog and mixed signals
e) Ports connectivity for end product

it's just like creating a blueprint of a house prior to construction of the house.

- **Functional and Logic Design:**
  a) In functional design the functionality is specified by the designer in HDLs such as Verilog, VHDL. This is also referred to as RTL(Register transfer Logic) level
  b) in logic design process we convert our high level HDLs to lower level(Gate level) netlists , this process is also referred to as synthesis . after this step we'll be seeing familiar hardware blocks such as Flipflops , gates, MUXs etc.
- **Circuit design**: In the circuit design process number of crucial low-level elements are designed at transistor level using SPICE simulations. We perform circuit design for certain crucial blocks such as the analog blocks, I/O blocks ,Multipliers etc.
- **Physical Design**

  physical design is referred to as backend for VLSI design flow and all the steps that were discussed earlier came under Front-end flow .in Physical Design the standard cell placement is carried out that is cells(eg. AND gate) are instantiated form libraries. In physical design There are several sub steps such as :

  a) floor-planning
  b) placement and routing
  c) clock tree synthesis ,
  d) signal integrity
  e) routing of Signals
  f) timing closure

- **Physical Verification** : in this step the layout that was designed in the physical design process is verified to ensure correct logical and electrical functionality .some faults and bottlenecks can be identifies during this process. DRC, LVS, ERC are done under this scheme

- **Fabrication:** After Physical verification the GDS-II format is sent to the foundry and it is given for manufacturing process . in VLSI industry it is labelled as tapeout. There are majorly two foundries :Global foundry, TSMC.
- **Packaging and Testing:** at last final ICs after fabrication are sent for pin connections and solder bumps. the final testing is carried out to check for further faults or flaws or loss in functionality.

3.2Models:

In order to plan a intellectual property(IP)there are different type of models in the industry at different level of abstractions and in order to design or simulate or test these models different modelling platforms and languages(coding) are used .

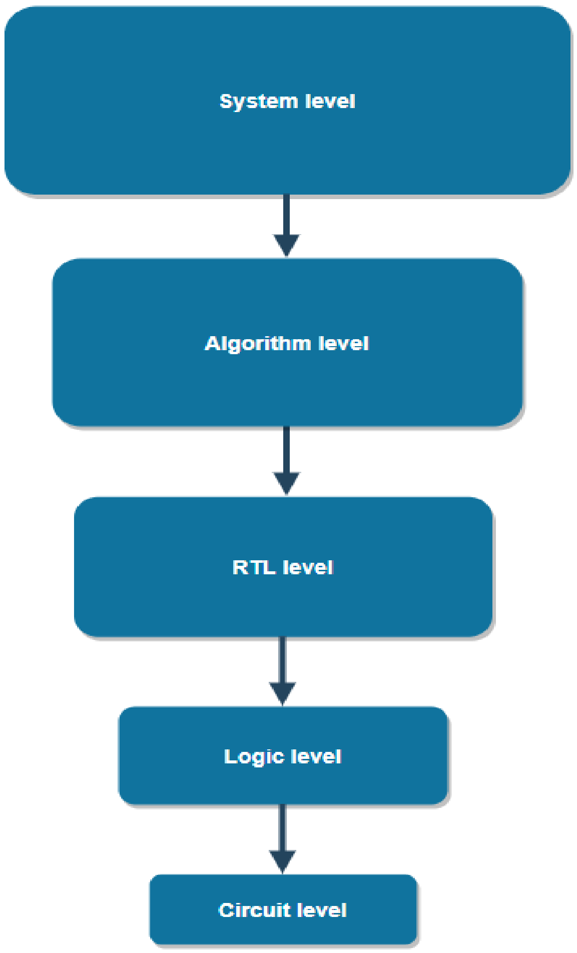
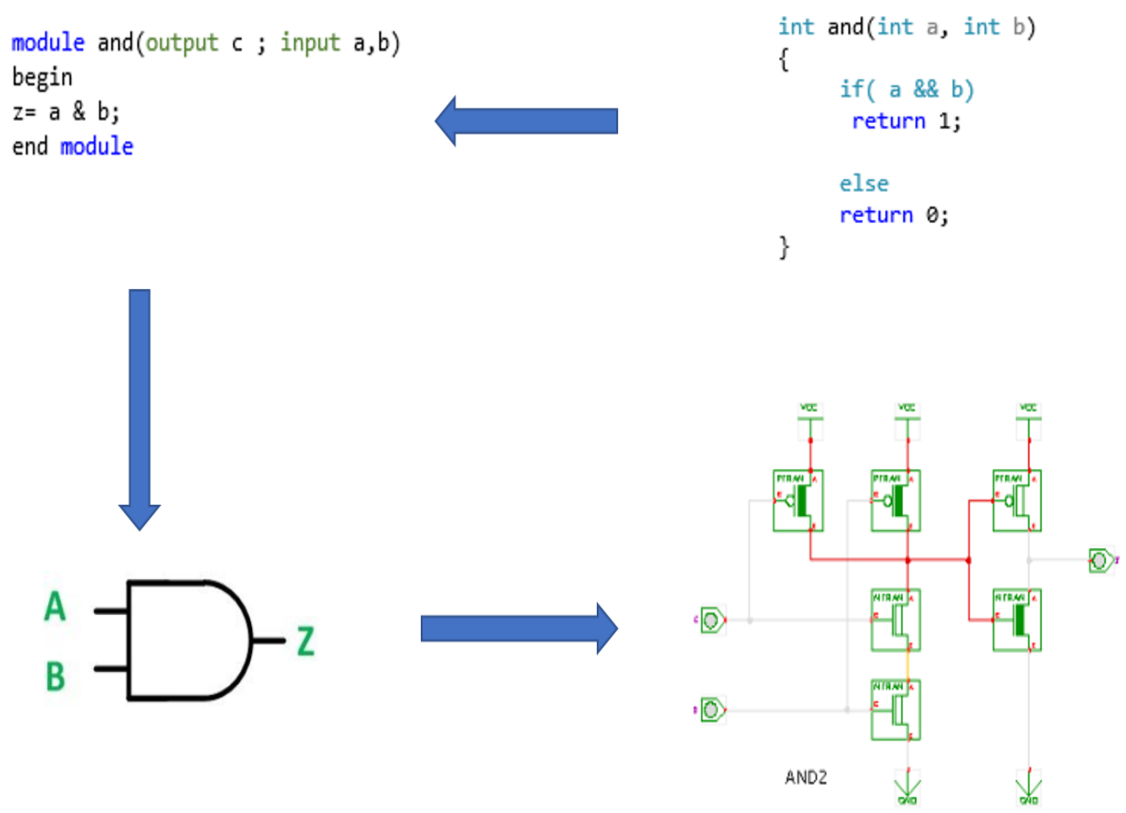


Fig 3.2 Models with Descending Abstraction level

- **System model** : as it’s at highest level of abstraction we are at the system level that is we are visualizing the system as a black box and we are performing different operations that are intended from our system. for this we use platforms such as C++ and python.
- **Algorithm level:** we define the algorithm for the tasks and operations ,this model depicts the behavioral description of intended design were we state our system the type of instructions need to be executed .we can use System C, System Verilog for their modelling. Here at this level timing is not a constraint.
- **RTL level:** going downward to the lower level of abstraction which is the RTL level .the communication or data fetching/retrieving is done by registers. for this we use different HDLs such as System Verilog, Verilog or VHDL. Here our design is represented in the form of netlist .here we generate cycle accurate timings.
- **Logic Level:** here the above constructed RTL is synthesized into Logic gates. And from now on we tend to see logic circuitries such as MUXs, Flipflops, Gates etc.
- **Circuit level:** After converting RTL to Boolean expressions, we now convert to circuit level . at this level we can now see transistors. And we can use different libraries such as MOSFETs or FinFETS for creating schematic.it is the exact implementation of the design



```
module and(output c ; input a,b)
begin
z= a & b;
end module
```

```
int and(int a, int b)
{
     if( a && b)
      return 1;

     else
     return 0;
}
```

Fig 3.3 Flow of system level to circuit-level model

From figure 3.3 it can be concluded that if we implement a code performing desired operations at system level model then it can be bring down to lower abstraction levels . in the previous sections we have studied different algorithms such as Woop's,Moller-Trumbhore and DXR, all these algorithms are at algorithm / system level and as we can see from figure 3.3 that all system-level model are somehow crisp down to the transistor level model .so we can say that when we move from subsequent level of abstraction to transistor level we can even model those algorithms as well. in my MTech curriculum I have implemented several standard cells such as adders[7], multipliers[8] and comparators[9] and all these standard cells can be instantiated in upcoming future designs and their algorithmic implementation.

.

CHAPTER-4
SIMULATION AND RESULTS

The simulation and results that are depicted in this section are obtained by conducting an experimental analysis in which random rays are being striked at a unit sphere with different Ray triangle intersection algorithms:

- Moller-trumbhore
- Woop's
- DXR

the camera point is acting as light source and we can also vary the camera distance accordingly. We have observed that as we keep on increasing the distance between the light source and the sphere, the number of false misses tend to increase. Moreover, if we increase number of rays per sample then we are getting a trivial increase in false misses. Here sphere that needs to be rendered is represented by blue background in the images. square image is obtained by transforming sphere onto the flat screen.

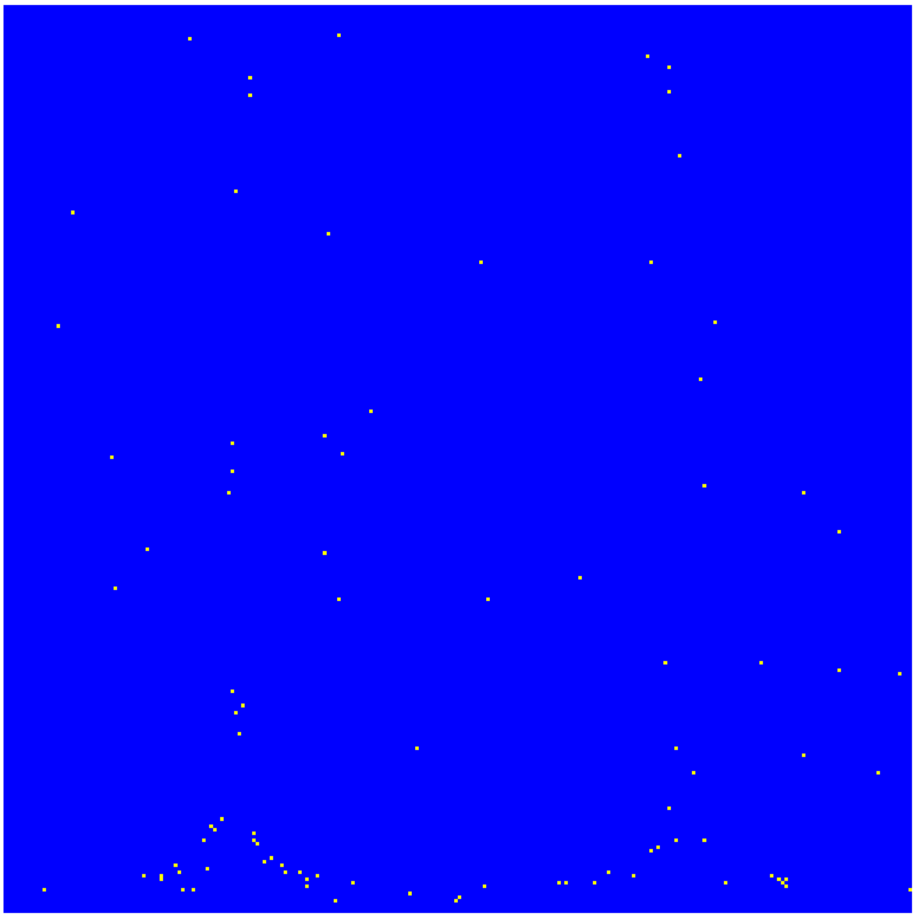


Fig4.1(a) Moller trumbore (Camera coordinates: (65536.96937, 65536.62,65536.42353245)

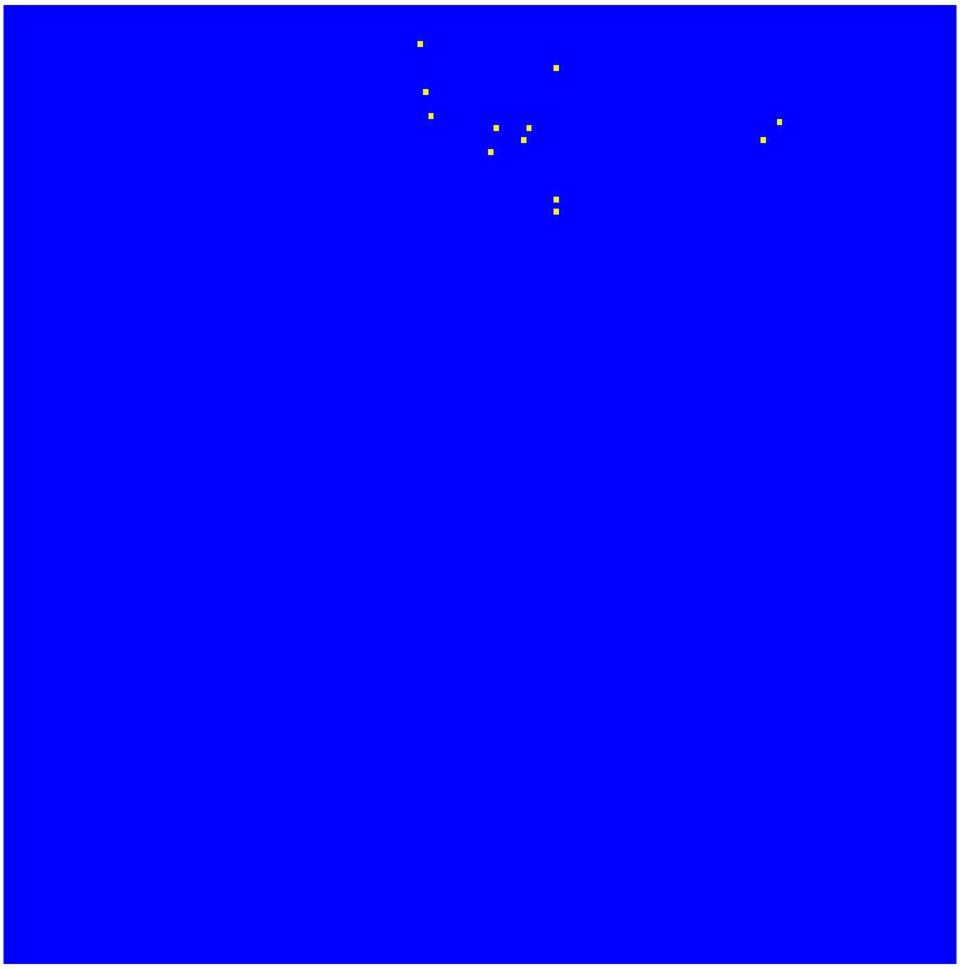


Fig4.1(b) Woop's Algorithm

(Camera coordinates: (65536.96937, 65536.62,65536.42353245)

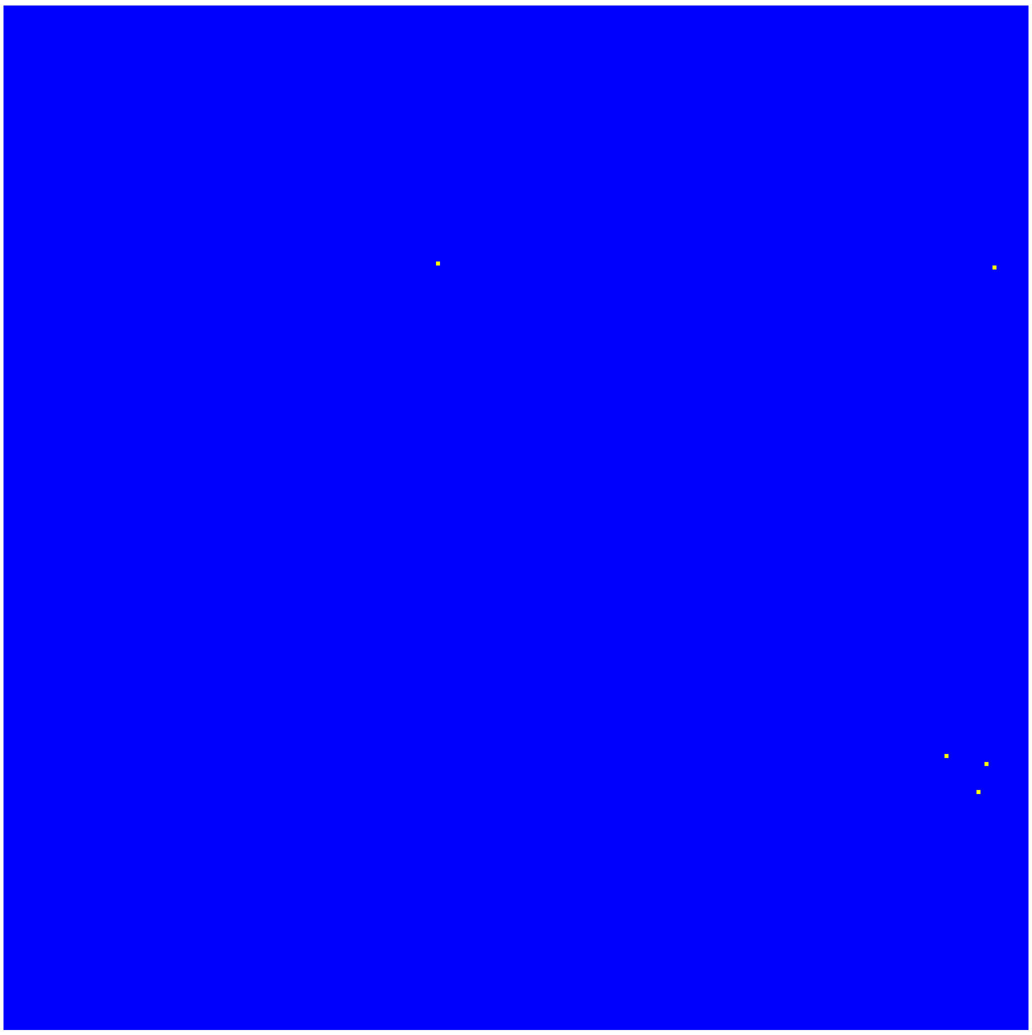


Fig4.1(c) DXR Algorithm

(Camera coordinates: (65536.96937, 65536.62,65536.42353245)

Algo	**Misses**	**Rendering time**
Moller Trumbore	111	Low
Woop’s	12	Moderate
DXR	5	High

Table-4.1 Comparison of algorithms

As we can see from table 4.1(a) that molar-Trumbhore algorithm exhibit extensive number of false misses although it's rendering time is low but it imposes certain challenges when we want to implement it for Ray tracing architecture .on the other hand woop’s algorithm exhibits less number of misses and the rendering time is also moderate. on contrary in DXR algorithm misses are extensively dropped but we see rendering time and time complexity shooting which is acting as a tradeoff. Coming back to Woop’s algorithm it guarantees water tightness as it is implemented on double Precision fallback ,the purpose we are stating so is that implementation of floating point (64-bit) on a hardware is complicated task . moreover, Designers are also suggested they should try to incorporate single precision then double because synthesis of double point is problematic, that is from C++ model to transistor level model it is quite challenging.

Conclusion and Future Scope

We have discussed “basic Ray tracing architecture” till now ., which is a portion of rendering pipeline and it has different sub blocks in it . we have also cultured two ray tracing architectures , one was from Nvidia and another one from AMD .Nvidia one had dedicated RT core in it where as AMD one performs RayTracing using its Shaders itself as they didn't had dedicated hardware to support Ray tracing, due to this they lacked performance as compared to Nvidia , Moreover these RT cores or shaders perform several computational operations generally Matrix operations such as cross-product, dot-product, inversion, translation and rotation .what I believe as a VLSI enthusiast is that in order for a GPU to perform these complex computational tasks several sub blocks or clusters are needed. Although these clusters are not that much under the vision if we see from a higher level of abstraction but these blocks/clusters play a crucial role for example if I have written a C++ code for addition of two numbers then my compiler will be using ALU and subsequently Adder will be will be initialized from ALU block. similarly, for Ray tracing at lower level of abstractions we have seen that we need different hardware blocks such as adders, subtractors or multipliers. although I have completed my internship at system level of ray tracing but the “VLSI-bug” inside me crawled all the way from top level system to the bottom level blocks such as standard cells . being a VLSI enthusiast and seeking to put one step forward in this industry I would like to say that the circuits that were implemented using GDI technique in our MTech curriculum can be used as standard cells for many of the extensive processing units such as CPU,GPU, TPU etc. In our MTech curriculum we did our projects on gate level simulations on virtuso and designed several standard cells such as adder, multiplier and comparator and these cells(circuits) can be in instantiated in a bigger block/circuit as they exhibit less power and more area Optimization . so, they could be future for upcoming processing architectures .

coming back to ray tracing the developments or proliferations in Ray tracing architecture has led to different issues one of the issues is watertightness. Which is formation of false misses or holes due to light rays not intersecting primitive properly so in order to eliminate this issue we used distinctive algorithms. The first one was Moller-Trumbore which was typically a two-edge test as one edge was translated to the origin and two other edges need to be manipulated . the second one is the woop’s algorithm it has a special feature of fall back to double Precision whenever watertightness comprises, such as shared vertices and shared edges. the downside of this algorithm is that the synthesis of double floating point computational hardware on a chip or SoC is quite problematic for the designers , although the number of false misses or holes in woop’s algorithm are less then Moller-trumbore but its construction is a bit tricky, Moreover the shearing and translation operation in woop’s algorithm introduces small rounding errors .lastly coming to DXR algorithm which is a rule-based algorithm that works on a set of particular rules in order to prioritize our major primitive from a set of primitives, by set we mean a pair of triangles or a fan of Mesh. This algorithm guarantees watertightness as it is even more superior than woop’s algorithm but with great efficiency come greater trade off as well ,so the rendering time is huge enough ,at last what I would like to conclude DXR is the algorithm that will be the future for coming ray-tracing architectures and it can be developed even more but we need continous efforts upon upgrading and improving our GPUs hardware as well in order to increase its performance in terms of power area and speed.

Appendix

Reference Codes:

Vector Subtraction

```
vec3f sub(vec3f a, vec3f b)
{
    vec3f res;
    res.x = a.x - b.x;
    res.y = a.y - b.y;
    res.z = a.z - b.z;
    return res;
}
```

Vector Dot-Product

```
vec3d dot(vec3d a, vec3d b)
{
    vec3d res;
    res.x = a.x * b.x;
    res.y = a.y * b.y;
    res.z = a.z * b.z;
    return res;
}
```

Vector Cross-Product

```
vec3f cross(vec3f a, vec3f b)
{
    vec3f res;
    res.x = a.y * b.z - a.z * b
    res.y = a.z * b.x - a.x * b
    res.z = a.x * b.y - a.y * b
    return res;
}
```

Here Vec3f is floating point vector class

References

[1] https://www.huangwm.com/wp/archives/2935#VGPRSGPR

[2] Shining a light on ray tracing ,Imagination Technologies, May 2019

[3] Nvidia Turing GPU Architecture , Turing Advanced Shading Technologies,

[4] Möller.T., and Trumbore, B. 1997. Fast, minimum storage ray/triangle intersection. Journal of graphics tools (jgt) 2, 1, 21–28. 67, 74, 75, 76

[5] Watertight Ray/Triangle Intersection S.Woop, C.Benthin and I.Wald Intel Corporation, Journal of Computer Graphics Techniques, Vol. 2, No. 1, 2013

[6] Gribble, C., Naveros, A., and Kerzner, E. Multi-Hit Ray Traversal. Journal of Computer Graphics Techniques 3, 1 (2014), 1–17.

[7] Malti Bansal, Jasmeet Singh, “Qualitative Analysis of CMOS Logic Full Adder and GDI Logic Full Adder using 18 nm FinFET Technology” , 3rd International Conference on Recent Developments in Control, Automation & Power Engineering (RDCAPE) 2019 Oct10 (pp. 404-407). IEEE.

[8] Malti Bansal, Jasmeet Singh, “Comparative Analysis of 4-bit CMOS Vedic Multiplier and GDI Vedic Multiplier using 18nm FinFET Technology” 2020 International Conference on Smart Electronics and Communication (ICOSEC), 10-12 Sept. 2020 , ISBN:978-1-7281-5462-6

[9] Malti Bansal, Jasmeet Singh, “Qualitative Analysis of 2-bit CMOS Magnitude Comparator and GDI Magnitude Comparator using 18nm FinFET Technology”, 2020 International Conference on Smart Electronics and Communication (ICOSEC), 10-12 Sept. 2020, ISBN:978-1-7281-5462-6