Kaunas University of Technology

Faculty of Mathematics and Natural Science

Development of Methodology for 3D Model Reconstruction from CT Images

Master’s Final Degree Project

Vishnu Manoj

Project author

Dr. Benas Gabrielis Urbonavicius

Supervisor

Kaunas, 2021

Kaunas University of Technology

Name of the faculty

Development of Methodology for 3D Model Reconstruction from CT Images

Master's Final Degree Project

Vishnu Manoj
Project author

Dr. Benas Gabrielis Urbonavicius
Supervisor

Arvaidas Galdikas
Reviewer

Kaunas, 2021

Kaunas University of Technology

Faculty of Mathematics and Natural Sciences

Vishnu Manoj

Development of Methodology for 3D Model Reconstruction from CT Images

Declaration of Academic Integrity

ktu
1922

Kaunas University of Technology

Faculty of Mathematics and Natural Sciences

Development of Methodology for 3D Model Reconstruction from CT Images

Topic of the project	**Development of Methodology for 3D Model Reconstruction from CT Images**
Requirements and conditions (title can be clarified, if needed)	
Supervisor	Dr. Benas Gabrielis Urbonavicius
	(position, name, surname, signature of the supervisor) (date)

Manoj, Vishnu. Reconstruction and Export Software for 3D Printing. Master's Final Degree Project / / supervisor lect. dr. Benas Gabrielis Urbonavičius; Faculty of Mathematics and Natural Sciences, Kaunas University of Technology.

Medical Physics (6213GX001), Faculty of Mathematics and Natural Sciences

Keywords: 3D Rendering, Vulkan, Surface Rendering, ITK, VTK, glTF

Kaunas, 2021. 68 pages

Summary

The space of 3D modelling is an everchanging landscape where technology is constantly innovating and as such programs dedicated to 3-dimensional rendering along with the associated firmware that they are built to operate upon have very short half-lives.

This research is dedicated to explore more recent technological innovations that have occurred in the space of 3-dimensional graphics rendering and to integrate it into the development of next generation programs that can develop photo-realistic 3-dimensional models generated from data obtained via a CT modality. The goal of this project is to develop an open source program that can render 3D models from CT datasets.

Rendering programme is designed using Vulkan rendering engine which is a low-overhead platform agnostic Application Programming Interface. These programs serve as an alternative to open source programs that exist in the market today such as Slicer 3D which are built upon the older OpenGL standard. Programs built on Vulkan reduce computational time, increase energy efficiency and reduce overload. These programs also allow for the implementation of more advanced computational workflows such as neural networks or cinematic rendering which implements photorealistic detail to the model that is rendered.

A working Vulcan based engine was developed and compared with other opensource tools . Due to the nature of Vulcan engine, performance benchmarks were proven to be in favor of the latter. Also an option of exporting models to glTF allowed for a multiplatform compatibility, including mobile devices.

Manoj, Vishnu. Trimačių modelių rekonstrukcijos algoritmo iš KT vaizdų kūrimas. Magistro baigiamasis projektas / vadovas lekt. dr. Benas Gabrielis Urbonavičius; Kauno technologijos universitetas, Mateamtikos ir gamtos mokslų fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Sveikatos mokslai, Medicinos technologijos

Reikšminiai žodžiai: 3D modeliavimas, Vulkan grafinis variklis, Paviršiaus atkūrimas, ITK, VTK, glTF

Kaunas, 2021. 68 p.

Santrauka

3D modeliavimo sritis yra nuolat besikeičianti, kurioje nuolat diegiamos naujos technologijos, o programos, skirtos trimačiam modelių atvaizdavimui, kartu su susijusia programine aparatine įranga, turi labai trumpą gyvavimo periodą ir yra keičiamos naujomis.

Šis tyrimas skirtas ištirti ir pritaikyti technologines naujoves, atsiradusias erdvinės grafikos perteikime, ir integruoti jas į naujos kartos programą, galinčią sukurti fotorealistinius trimačius modelius, iš duomenų gaunamų kompiuterinės tomografijos metu. Šio projekto tikslas yra sukurti atviro kodo programą, kuri galėtų atvaizduoti 3D modelius iš KT duomenų rinkinių.

Atvaizdavimo programa yra sukurta naudojant „Vulkan“ grafikos variklį, kuris yra efektyvus ir taikytinas kaip aplikacijų programavimo sąsaja. Ši programa galėtų būti alternatyva šiandien rinkoje egzistuojančioms atvirojo kodo programoms, tokioms kaip „Slicer 3D“, kurios yra sukurtos remiantis senesniu „OpenGL“ standartu. „Vulkan“ pagrindu sukurtos programos sumažina skaičiavimo laiką, padidina energijos vartojimo efektyvumą ir sumažina techninius reikalavimus kompiuterinėms sistemoms. Vulkan taip pat leidžia įgyvendinti pažangesnes skaičiavimo darbo eigas, tokias kaip neuroniniai tinklai ar kinematografinis perteikimas, kuris yra ypatingai svarbus fotorealistiniam modelių pateikimui.

Buvo sukurtas veikiantis „Vulcan“ grafikos variklis, kuris buvo lyginamas su kitais atvirojo kodo įrankiais. Įrodyta, kad dėl „Vulcan“ variklio charakteristikų efektyvumo rodikliai yra naudingi pastarajam. Taip pat galimybė eksportuoti modelius į „glTF“ formatą, leidžia naudoti gautus modelius įvairiose platformose, įskaitant ir mobiliuosius įrenginius.

Table of contents

Contents

List of figures

List of Tables

List of abbreviations and terms

Abbreviations:

CT- Computed Tomography

HU- Hounsfield Units

DICOM- Digital Imaging and Communications in Medicine- a format used to store medical imaging data and context of clinical environment in stated procedure for which data was acquired.

SOP- Service Object Pair Class

IOD- Information Object Definition

UID- Unique Identifier

API- Application Programming Interface

glTF- Graphics Language Transmission Format- Format used to encode 3D data. Can be transmitted across the web.

SPIR-V- Standard Portable Intermediate Representation- Bytecode format used to encode shader code information into a binary. This makes it easier for compilers to process and output during application building time.

EEPROM- Electrically Erasable Programmable Read-Only Memory

EPROM- Erasable Programmable Read-Only Memory

SRAM- Static Random Access Memory

ITK- Insight Segmentation and Registration Toolkit- C++ API used for image processing applications

VTK- Visualization Toolkit- C++ API used also for image processing but mostly 3D model data processing and visualization.

JSON- JavaScript Object Notation- data interchange format utilized to categorize and store information to be queried by application.

Introduction

Hospitals around the world today utilize vendor specific solutions to address their imaging needs which many a times are highly specialized for particular tasks and are not extensible in nature. Any future updates that may be required will require an overhaul of the entire infrastructure and a renewal of the license that will cost an additional fee. As a result, hospitals often conserve their budget and manage with antediluvian infrastructure that has not been upgraded for several years. This can most predominantly be seen in hospitals that employ a PACS-based infrastructure where these systems do not encompass enterprise wide networks being accessible only to a few select departments that utilize imaging modalities such as cardiology. This leads to models being accessible only via select workstations within those departments and not via other client devices. These systems are also of limited extensibility as additional software components cannot be incorporated into these devices due to vendor lock-in. This is also the case for newer hospital infrastructure solutions such as Enterprise Imaging where even though data is stored in a vendor neutral archive and is accessible across all departments and client devices, the software solutions are still proprietary in nature making it resistant to customizability tailored for the end user’s needs or for future updates as technology progresses with time. These proprietary solutions naturally have short half lives and quickly get outdated with time.

The only open-source solutions available in the market today operate on older graphics libraries such as OpenGL which has been an industry standard for several years [1]. In a review conducted on 28 DICOM viewing solutions, it found that OsiriX was found to be the best solution available in the market today as open-source, platform independent solution in the market [2]. The study noted that 3D rendering quality of the images as lacking especially the web-based rendering modality. All these viewers also utilize OpenGL graphics library which is gradually being replaced by libraries that offer more energy efficient designs such as Vulkan. With the advent of GPU computing where programs are being developed to run on GPUs which offer many advantages such as parallelization of the workload, large memory bandwidth and double precision arithmetic operations [3]. DICOM viewer solutions do not optimize the GPU heavily and as a result large data driven workloads such as 3D rendering suffer in performance. With the advent of machine learning protocols that can embed added functionality into the program, DICOM viewer solutions would have to revamp their architectures to be more GPU than CPU driven as generating data driven workflows at a higher fidelity will require maximum optimization of computational resources allocated. This is the problem that this thesis plans to address by developing a 3D rendering solution that operates on the Vulkan firmware.

Aim: To develop an open source program that can render 3D models from CT datasets for multiplatform use

Tasks:

1. Select a best suited approach for a rendering engine designed for multiplatform use
2. Design and debug a rendering engine with CT modality data.
3. Test the rendering engine with CT modality data on multiple platforms and evaluate its performace.

2. Literature Review

2.1 Introduction to CT/Overview of thesis project

CT stands for Computed Tomography and it was developed by Godfrey N. Hounsfield and Allan M. Cormack [4,5]. It is an imaging modality that is executed on a series of devices that share a common system design. This system design consists of an apparatus shaped like a toroid which is referred to as a gantry. The gantry consists of a series of components such as a X-ray generator that rotates around a target immobilized on a table along with detectors which are positioned opposite to the generator. These detectors then detect the incoming X-rays that pass through the target via a scintillation mechanism where these x-rays generate visible light that is then amplified/converted into an electrical signal which can be digitally stored. These devices deliver X-rays to the patient and generate cross-sectional images termed as "slices" which represent the subsection of the imaging plane of a patient's body [4,5]. These images are then separately combined and rendered into a 3-dimensional model. There are various image acquisition profiles available but most of them involve the rotation of the gantry around the target. This technique is non-invasive which is beneficial as no physical examination has to be conducted that might end up inflicting permanent blemishes on a patient. Each pixel of the image corresponds to the attenuation of X-rays by the tissue in question. The image is developed through the use of the Radon transform and the Fourier Slice Theorem [4]. The Radon Transform is utilized to transform the information obtained from the CT imaging modality into an image where it is obtained in the shape of the sinogram which can be attributed to the rotational manner in which image acquisition takes place.

$$p(\theta, \mathrm{s}) = \int_{-\infty}^{\infty} \mathrm{f}(\mathrm{x}, \mathrm{y})\delta(\mathrm{x}\cos(\theta) + \mathrm{y}\sin(\theta) - \mathrm{s})\, dxdy \qquad (1.1)$$

The Radon transform is used to reflect the mapping transformation that exists between values that exist on the Cartesian coordinate system and the polar co-ordinate system with every image being represented by a sinogram with a relative thickness with the points located within the sinogram containing the necessary information required to generate the image. This mapping relationship is capitalized upon in a technique termed as filtered back projection where the inverse fourier transform is taken of the Radon transform from 0 to pi generating the original attenuation function f(x,y) which is the image of the cross-sectional slice that the light rays traversed across the target tissue sample before hitting the detectors on the gantry. There are other methodologies available alongside filtered backprojection but this is outside the scope of this thesis.

The attenuation function expresses the attenuation coefficient of tissues at a particular coordinate of the tissue cross section and this is expressed in Hounsfield units where a Hounsfield unit is defined as the linear attenuation coefficient of a particular region of tissue with respect to the radiodensity of distilled water under standard conditions of temperature and pressure (273.15 K , 10^5 Pa). It is measured by the given formula-

$$\mathrm{HU} = 1000 \times \frac{\mu - \mu_{water}}{\mu_{water} - \mu_{air}} \qquad (1.2)$$

The Hounsfield unit is utilized for CT because of the intrinsic composition of life where water is a major chemical component that constitutes the biological structures of all living organisms on the planet. This makes the Hounsfield unit relevant for CT scans [6]. The images generated via a CT modality are greyscale images where regions found to have a lower density than water are signified as darker regions on the image and regions with a higher density than water are brighter. The human eye however cannot distinguish finer shades of grey where this issue can potentially be addressed by 3D models. It does this by automatically assigning colors to regions of tissues that correspond to different Hounsfield ranges via segmentation algorithms [6].

In order to build a program that can take as input a dataset of CT images and output a 3D model, there are several stages that need to be incorporated into it. These stages are image processing, model creation and viewing operations. These stages individually would have several more substages that comprise each individual stage but it can be divided clearly into 3 easy stages.

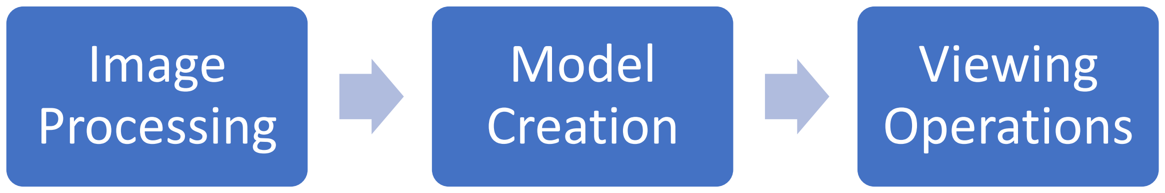


Fig.1. A flowchart representing the crucial stages in the 3D rendering program.

The Image Processing stage will utilize the ITK library that is written in C++ to generate the necessary object file that stores the necessary data to render the model in image space. The Model Creation stage utilizes the glTF format where gltf stands for graphics Language Transmission format. This format is a relatively new format that is developed to address the problem of rendering 3D models that can be shared across a network. This format is lightweight and utilizes JSON data interchange format to store images that serve as textures to paste over the model as well as mesh data required to reconstruct the model. This format allows renderings to be done on the web due to its light size. This could be incredibly beneficial in the clinical environment especially for consultation where 3D models could be shared over the web with low latency. The model is loaded from a glTF file onto a program module utilizing the Vulkan API which constitutes the architecture upon which the program is built. This module is specialized for loading the model and also accommodating viewing operations that allow the model to be translated or panned in the image space.

2.2 Medical applications of 3D rendering

There are various medical applications that can arise with improved 3D modelling technologies. One example of this is the use of next generation CT methodology for the representation of the pulmonary vasculature which has never been done before. This new CT modality is termed as threshold-based 3D CT volumetry where this modality is capable of extracting the pulmonary vasculature of the lungs and utilizing it to generate a 3D model [7]. The 3D model that is generated has found clinical translation in preoperative planning for cyanotic congenital heart disease where they utilize the 3D model of the pulmonary vasculature to determine arterial pressure non-invasively [7]. The parameters of pulmonary vasculature such as volume and arterial pressure is shown by this study to be accurately characterized [7]. Characterizing the pulmonary vasculature is considered to be important for the study of diseases with an unknown etiology such as pulmonary hypertension where this disease has no clinical data because no modality has been developed extensively enough that is efficient to measure the local blood pressure of the pulmonary vasculature and thus quantify pulmonary stiffness [7]. This can be extremely useful especially in the characterization of pulmonary hypertension with extensive detail as determining which pulmonary arteries play a contributing factor to the development of the disease can influence therapy planning. This technology could potentially contribute to personalized medicine where pulmonary hypertension is a chronic disease that is idiosyncratic to the individual involved and thus doctors would need a personalized diagnostic profile of the patient before developing a treatment plan for the disease [7].

3D reconstruction algorithms especially help in Computer Vision. Computer Vision is known to be helpful for minimally invasive vascular interventions with the use of a 3D model generated from data obtained via CT modality. These 3D models are utilized for minimally invasive procedures which require a superposition of the virtual model over the live patient [7]. The 3D model is utilized for a variety of purposes such as registering the co-ordinates of the surgical instrument employed for the procedure. Here, 2 3D models are generated where one model corresponds to a native or control CT dataset which is not contrast enhanced and the other corresponds to a model generated from contrast- enhanced CT datasets [7]. These 2 3D models are then superimposed onto each other to generate a digitally reconstructed radiograph (DRR) where this is then utilized to determine the location of the delicate surgical instrument in 3D space. Another radiograph is generated for determining the positional information of the instrument with respect to the X-Ray source and these coordinates of the instrument are encoded to the 3D model that is generated [7]. Using 3D reconstructed models for these minimally invasive vascular procedures reduced the incision size increasing the accuracy and precision of the procedure. This can lead to drastic reductions in complications that could occur where these surgical procedures are usually conducted using 2D images as reference. Using 2D images can be disadvantageous because moving the surgical instrument perpendicularly to the image plane displaces the instrument from the recognized point of location on the 2D image where this would not happen with a 3D model where all planes of reference are considered. This can lead to surgical complications as the surgeon now is conducting the procedure blind [7]. Thus, the use of 3D models to supplement surgical procedures is extremely beneficial because on top of reducing the risk of surgical complications, it can improve the accuracy of the surgically invasive procedure making minimally sized perforations and thus reducing the length of hospital stay for the patient [7]. The advent of 3D models in computer vision can greatly benefit surgeons as this can be applied to a wide variety of surgical procedures ranging from internal medicine, orthopedics, pediatrics and many others [7].

3-dimensional reconstruction of the kidney was deemed helpful for postoperative surgical planning with regards to renal cell carcinoma. MDCT was considered valuable in depicting the renal arteries and veins that threaded throughout the kidney and this was used to determine if a tumor had potentially metastasized by ascertaining the presence of a blood supply near a tumor lesion. Presence of a blood supply would indicate tumor angiogenesis, one of the hallmarks of cancer and thus would be a strong indicator of metastasis. With the help of 3D reconstruction, it becomes much easier for doctors to visualize the relationship of the tumor with the renal surface of the kidney as they can view the tumor from multiple orientations [8]. 3D reconstruction is also helpful in determining blood disorders such as blood clots or thrombus of the inferior vena cava where it is found that visualizing the thrombus from multiple planes allows for more accurate localization of it in the inferior vena cava. Utilizing 3D models allow representation of diagnostic markers in a more palatable fashion where tumor metastasis can be represented through the intervening fat tissue. This mode of metastasis is termed as perinephric standing where it represents interstitial fluid buildup near the fatty tissue that surround the kidneys. Visualizing perinephric standing is aided with 3D reconstruction programs and segmentation algorithms as the relationship between these tissues and that of multiple organs can be visualized appropriately in 3-dimensional space allowing the user to switch from different vantage points of the model with ease [8].

For diseases that afflict the colon such as acute mesenteric ischemia, 3D reconstruction techniques are instrumental for generating models that could depict clinical signs which correspond to certain pathologies easier. One study reported the use of a 3D model for enhancing the visualization of clinical signs for acute mesenteric ischemia such as loop dilatations and ileal thickening. These clinical signs were visualized more easily through the use of a 3D model where a technique termed as tissue transition projection was used. This technique involved utilizing the transitions from the content within the intestines to the surrounding bowel tissues to differentiate between bowel tissue and the chyme that is being digested. Tissues corresponding to the bowel wall are rendered transparent. This allows for the visualization of arterial dilatation which is a clinical sign for acute intestinal ischemia [9]. There are also studies utilizing machine learning algorithms to segment organs in the abdominal cavity which make identification of characteristic features such as lesions easier. One study utilized a convolutional neural network, termed as a Markov Chain Monte Carlo guided Convolutional Neural Network, which was used to identify the pancreas by segmentation and isolated to create a 3-dimensional model of the pancreas [9].

Another study utilized a contrast agent composed of nano particles for micro-CT where 3D models were generated to study the caudate lobe size where this serves as a diagnostic marker of liver pathologies such as liver cirrhosis, a chronic disease symptomatic of large scale liver damage [10]. 3D models were utilized to also automate segmentation of visceral and subcutaneous fat in the body where Dice scores are utilized to determine the level of similarity between automated and manual segmentation models with a similarity level of 99.6% being reported. The algorithm is thus able to outcompete human readers achieving the same result within a shorter duration [10][11]. The 3D algorithm performs better than 2D analysis of visceral and subcutaneous fat because 3D models account for intestinal peristaltic movement which 2D slices do not. The peristaltic movement of intestinal organs can alter calculation of visceral adipose tissue fractions where this is not altered with 3D segmentation models [11].

All in all, several studies show the benefit of utilizing 3D reconstruction models in modern medicine where it is found to increase diagnostic accuracy by improving visualization, reduce diagnostic downtime which can be automated through the use of segmentation algorithms, models that display more information that cannot be gained from simple inspection of cross-sectional 2D slices and even complement surgical procedures minimizing surgical complications.

2.3 Classical 3D reconstruction techniques

One of the earliest algorithms utilized for 3D reconstruction involved the implementation of the Marching Cubes or Marching tetrahedra algorithm where the idea behind this algorithm can be observed with a square. Here, a square can be intersected in 16 different ways by a 2-dimensional contour where the number 16 is obtained from the number of planes of symmetry that a square possesses which is 4. Each plane of symmetry can be signified by a vertex where each vertex can either be in the on or off position which signifies whether it was intersected by a 2-dimensional contour or not. This leads to $2^4 = 16$ unique modes in which a square can be intersected by a 2D contour. The scaled-up version of this concept is the Marching Cubes algorithm where this concept is applied to the 3rd dimension where instead of squares and lines being used, cubes and surfaces are studied instead [12-14].

In the marching cubes algorithm, the surface is generated by a series of voxels where a voxel is a representation of a unit of object space signified by a cube. Each voxel generated has two faces on consecutive CT slices [12,13]. The purpose of generating a voxel like that is for the voxel to intersect 2 consecutive CT image slices where 4 vertices in one faces of the voxel lie in a particular CT image slice and the 4 vertices of the opposite face lie in the consecutive CT slice. Each of these 8 vertices have a binary value where 1 corresponds to a vertex that is inside the surface and 0 corresponds to them being positioned outside the surface (the portion of the voxel located outside the surface). These leads to 2 different possible states that each vertex on a cube could be in- either 0 or 1. Since there are 8 different vertices and 2 possible states- this leads to $2^8 = 256$ ways a surface can intersect a cube [12,13]. The manner in which the binary state of the vertex is decided by attenuation density gradient at the vertices of the voxel where this information can be encoded within an attribute within the vertex container that stores data features of the vertices that form the model.

The 256 possible combinations were then further reduced to 14 unique and different possible intersections generated by examining 2 different symmetries of the cube where the 1st symmetry represented edge intersections on complementary surfaces which due to symmetry led to repetition of these edges [12,13]. An example of this type of symmetry can be depicted through the use of complimentary edges of the voxel where swapping edge intersection and thus the binary state across these vertices on complimentary sides of the cubes result in the same intersection leads to both intersections being classified under the same category- this reduces the number of possible intersections from 256 to 128 [12,13]. Using rotation as the second symmetry, the cases were further narrowed down from 128 to 14 possible cases which is given in figure 2 below-

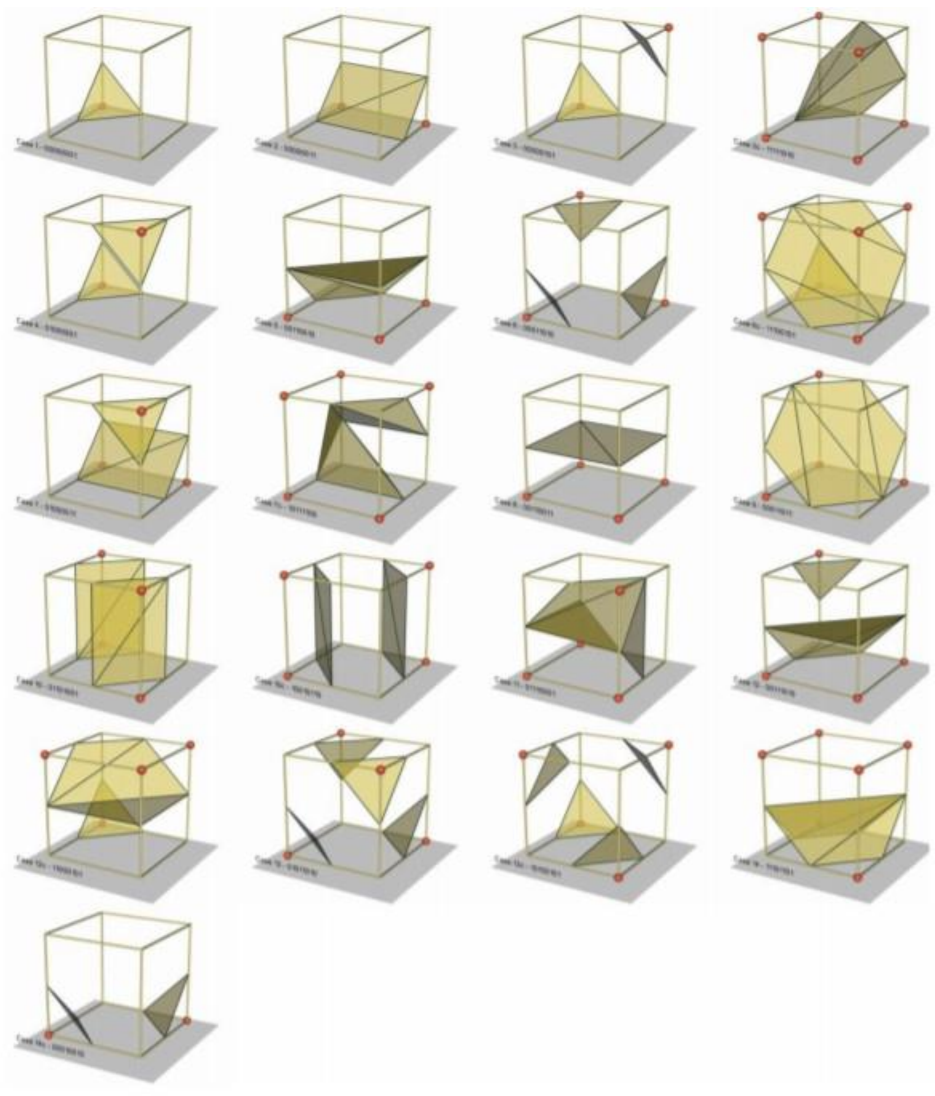


Fig.2. A representation of the number of possible permutations in which a 3 dimensional surface can intersect a voxel where there are 14 different cases. Additional cases arise as a result of complementarity where these cases need to be treated differently [12].

The 1st case, labelled 0, is the one where the majority of the cubes that are rendered are most likely to be where most of the cubes are going to be completely ensconced by the structure or lie completely outside of it [12,13]. The ones that lie at the boundary lines of the structures are the ones that are utilized to render the surface. All these surfaces are indexed based on the permutation of the binary states that all the vertices of the logical cube fall into which leads to a generation of a surface that can fall into any of the 14 different categories [12,13]. These categories are identified by the indexes that they are labelled with where these indices represent the combination of vertices within the voxel in their binary state. Once these indices are identified, a surface that is bounded by one logical cube is rendered where each of these surfaces are termed as Gourard-shaded surfaces which are surfaces that are composed of polygons. In this case, all the surfaces are composed of the simplest polygon, the triangle [12,13].

Once the surface is rendered, the vertex of intersection between the surface and the cube is determined. This is done by determining the density function at each vertex that constitutes the voxel where the gradient of the density function across the vertices of the voxel is utilized to generate a unit normal to the rendered surface within the voxel [12,13]. The gradient of the density function is assessed across all edges of the voxel at which the surface intersects [12,13]. Through linear interpolation, it is determined what the gradient at the vertex where the intersection occurs is which is given by the following equations-

$$G_x(I,j,k) = \frac{D(i+1,j,k)-D(i-1,j,k)}{\Delta x}$$

$$G_y(I,j,k) = \frac{D(i,j+1,k)-D(i,j-1,k)}{\Delta y}$$

$$G_z(I,j,k) = \frac{D(i,j,k+1)-D(i,j,k-1)}{\Delta z} \qquad (1.3)$$

Where $G_x(I, j, k)$ represents the gradient density function at each vertex in a given voxel and D(I, j, k) represents the density attribute of each individual vertex within the voxel. The gradient is calculated in order to determine the unit normal of the vertices of the voxel which in turn is utilized to calculated the unit normal of the surface that intersects the voxel in question where the unit normal is required for generating Gourard-shaded surfaces of the 3D model in question. These surfaces are referred to as textures and the process of shading these surfaces is known as texture mapping which is covered in greater detail in the shader section. The Gourard shading technique utilized here continues to be used today for shader programs as they implement instructions on the GPU [14]. Gourard shading is covered in greater detail in the shader section of this review-

2.4 Technical Considerations

Most of the techniques elucidated in the previous section have been conducted on CPU. The structure of the GPU is such that it supports multithreading where GPU architecture is built to support single instruction multiple thread architecture that delivers more computational power required for rendering more advanced models. CPU are considered to be low latency low throughput processors whereas GPUs are considered to be high latency high throughput processors. This means that CPUs utilize short term memory storages such as caches to deliver high speed performance with minimal lag times whereas GPUs are responsible for conducting large scale operations such as processing millions of fragments in a model for a single frame that is presented to a screen but respond with much higher latencies. As models become more advanced and a demand arises for more intricate renderings, there lead to a shift from allocating the processing load from the CPU onto the GPU as the GPU can support larger throughput as it replaces the space generally allocated for caches in the CPU for compute units which allows several thousand threads to be run in parallel [14]. Memory is incredibly vital for graphics and compute applications especially those allocated for 3D rendering and as such, it is important to know how memory is organized in micro-controllers.

Memory is accessible as a single heap by the host where system memory is integrated with the GPU otherwise it is bundled as discrete subsets that need to be queried separately.

Memory is stored in a pyramid scheme where it can be organized into 2 categories- volatile and non-volatile memory where volatile memory refers to registers, caches and SRAM. Volatile memory is defined as memory that is lost when the power is disconnected from the device whereas non-volatile memory is unaffected by loss of power. Hard drives, tape-based storage, Read-Only Memory which can be categorized into various types such as EEPROM (Electrically Erasable Programmable Read-Only Memory), EPROM (Erasable Programmable Read-Only Memory) where the memory could be erased through the exposure of UV light and Flash. Non-volatile memory unlike volatile systems have a finite lifespan or endurance as to the number of times read write operations can be conducted before the memory becomes obsolete. For ROM, it is programmable only once before the memory is stored permanently within the system and for later systems such as EPROM and EEPROM, it could be done several times. Flash is the latest generation of non-volatile memory technology where the memory here can undergo write-erase cycles ranging from 10,000 to 100,000 times [15-16].

For programs running on embedded systems, memory usage can be categorized into 3 different sections based on how it is used by the program- runtime memory, data memory and code memory. The linker script is responsible for sectioning and allocating components of the program from the object file into addresses in disparate regions of memory that constitute the microcontroller. Features that characterize code memory are non-volatility whereas data memory is more dynamic in nature due to data changing over time as it is processed by the program. This is handled by the SRAM and flash memory respectively [15-16].

SRAM memory ranges from 4 KB to 32 KB whereas flash ranges from 32 KB to 256 KB where in embedded systems, flash is utilized to store memory associated with the program code and

SRAM is utilized to store memory associated with data that is operated on by the program. Programs typically tend to be larger than the data that they operate with and thus flash is allocated larger memory heaps and it is static in nature whereas data is overwritten many times during several program iterations. SRAM and flash memory need a controller that can interface and exchange information with a CPU regulating the read/write operations in a synchronized and reliable manner. These controllers are built into the microcontroller. In case of additional memory requirements, external ROM such as EEPROM can be attached to the microcontroller via a Serial Peripheral Interface. This memory can serve as a reserve to store both code and data memory for microprocessor if data is overloaded on the system [15-16].

There are also additional memory components termed as Caches whose function serves to lower latency where latency is defined as the time it takes for a particular memory motif to respond to a read/write request where reading and writing assembly level instructions to memory can take a certain period of time. Microcontrollers are built with a certain level of pipelining where Caches lie at the intermediate level of the pyramid chart built to handle read/write operations and reduce the downtime that would accrue if operations were assigned to lowe- level memory systems [15-16].

2.5 Concepts of 3D Rendering

Shaders are compact programs that take in as input a certain type of processing element such as a vertex container for a series of calculations which then outputs an attribute that is presented to the screen. These shader programs are written either in assembly-level language or higher-level languages such as C [17]. For 3D rendering, input attributes are always vertices that are sequentially sampled from a buffer into the shader which is then submitted to a graphics queue belonging to a GPU. These vertices undergo certain transformations after which these vertices are then rendered as triangles that retain all the attributes of the vertices that compose them. These triangles are referred to as primitives and a collection of primitives defines the entire surface that can then be visualized on the screen. These transformations can be of a variety of types such as texture mapping where functions known as textures that define the properties of the surface to be rendered are mapped to the surface. This is a mapping that occurs from the texture space to the object space where special co-ordinates are assigned to each primitive that defines how the texture should be mapped to the primitive. These coordinates are referred to as texture coordinates. This process is known as parametrization where surfaces of polygons are parametrically defined either through the usage of texture coordinates or through the use of a normal vector that defines the plane or surface of the polygon from which it emerges where all points within a particular plane have the same normal [18].

The shaders stream their input to the GPU but however, they are subdivided into two separate sections which are streamed to the GPU independently. They are termed as vertex and fragment shaders respectively. The vertex shader is responsible for managing the model-view transformations that arise as a result of manipulating the orientation and position of the model as displayed on the screen. They are also responsible for defining variables responsible for subsequent transformations in the fragment shader. They process vertices one at a time during the render stage [17]. They rely on data that is already stored in the vertex buffers or caches and cannot generate new vertices on their own. They also do not operate on primitives. Vertex shaders have access to 4 different clusters of memory that allow it to implement the required functionality. These memory clusters can be defined as input-vertex registers, constant memory, temporary registers and output vertex registers. The input-vertex registers store the data required to define a single vertex. A single vertex is defined by a container that stores various attributes required to define the properties of the vertex that shape how the model is viewed on the screen. The vertex shader is mostly independent of the program that is compiled to run on the CPU where it functions in a closed system enabled by the memory registers that forms a part of the system. Every graphics pipeline can only have one vertex shader active at a particular time and thus only 1 shader can carry out the encoded instructions at a time. All memory locations store vectors that are 4-dimensional and each input is a 32-bit floating point value which can be both positive and negative in value. These vectors are utilized to represent the value of each individual attribute where a collection of these attributes is used to define the vertex. Examples of these attributes can be the position of the vertex, color of vertex, texture co-ordinates which is later utilized by the fragment shader, value of the normal vector for shading and ray-tracing calculations that is conducted by the fragment buffer. Constant memory is important because it is required for encoding transformations brought upon the 3D model in screen space where uniform buffers come into motion here where constant memory stores global data of the scene and objects that are known to change with each frame. The frame is the buffer onto which the scene is rendered and every monitor screen has a particular frame rate that is utilized for screen presentation- this will be covered in the methodology section in greater detail. Uniform buffers are responsible for keeping track of data that changes with each frame and this data is stored in the constant memory register of the vertex shader [17-18]. Temporary registers are utilized for storing intermediate data during vertex calculations where there are 12 general purpose registers according to NVIDIA where each register is a 4-dimensional vector. Output registers are utilized to store the position of a vertex on the screen space of the client system where the vertices are subject to a transformation termed as a view frustrum where all vertices are transformed to points that lie between 0 and 1 with points lying outside this range being clipped. These transformed vertices are termed as clip-space vertices where these are the final vertices that are projected onto the screen.

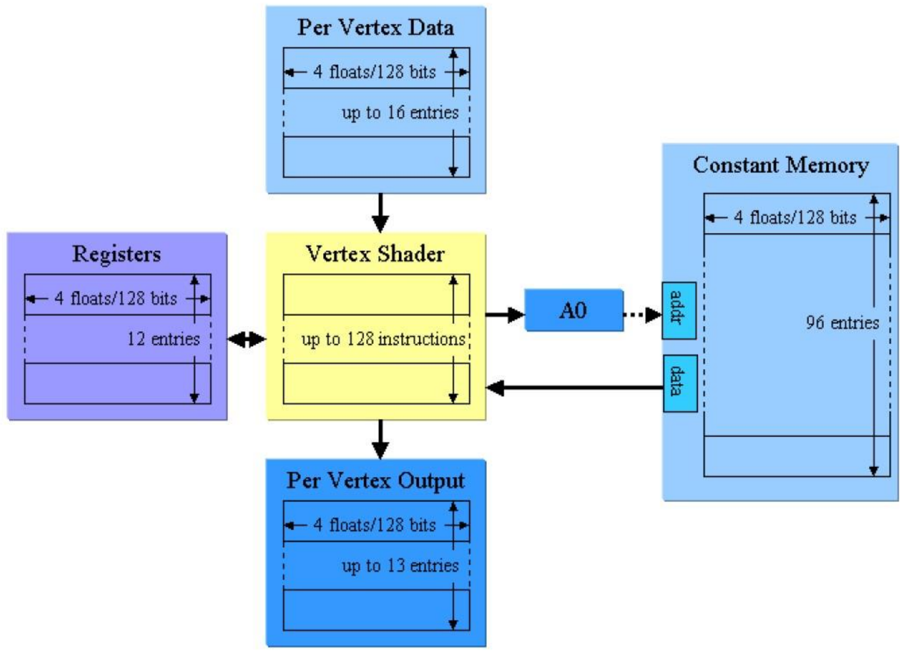


Fig.3. A representation of the major components required by a vertex shader to execute instructions. These 4 components are memory heaps that support various stages of the shader pipeline as it executes an instruction set [17].

The fragment shader is the penultimate shader where it is responsible for shading the assembled primitives that arise out of the vertex shader [17]. It is primarily responsible for rasterization of individual pixels according to the information it receives as output from the vertex shader. It is executed once for every pixel allocated to the buffer on which the model is rendered. It takes longer to process as compared to the vertex shader because of the sheer number of pixels that a 3D model needs to be rasterized to on the screen [19]. The fragment shader is also responsible for assigning textures to primitives required for advanced illumination. Each fragment is referred to as a pixel that contains scene-specific information such as color or texture co-ordinates. An example of one type of texture mapping is Gourard shading.

Gourard shading was one of the 1st shading algorithms developed to generate smooth polygonal surfaces where it was introduced in 1971 by Henri Gourard. This was introduced to combat the problem of Mach bands, edge effect artefacts, that resulted in the appearance of polygons within the rendered surfaces. Gourard shading resolved this problem through linear interpolation where these edges are monitored by means of the scan line. The scan line refers to the row of pixels that compose one particular section of the screen that the model is to be rasterized onto. The scan line intersects 2 edges at 2 separate points and each edge is connected to 2 vertices leading to a total of 4 vertices. Each vertex has a density or shading attribute used to define the shading at the surface. There is a myriad of ways to do this where the attribute can represent the color of the vertex or the texture which influences surface properties. The shading values of these pairs of vertices are implemented in a parametric equation termed as linear interpolation where the shading value of each vertex of a particular edge, AB become parameters that influence the shading across this edge. The shading value of the edge E which represents the density of a particular attribute is influenced by the equation

$$S_E = S_A * (1 - \alpha) + S_B * \alpha \tag{1.4}$$

Here, in 1.4, α represents the position of the intersection point E along the edge AB and this value ranges from 0 to 1 [18-20]. Thus, clipped vertex co-ordinates which are the co-ordinates of vertices rendered on the screen are those in which co-ordinates can only range from 0 to 1 and this is because of linear interpolation. 1.4 is repeated for both edges and the shading values obtained for these respective edges are inputted once again into 1.4 to obtain the shading value of any point on the polygon surface. This is utilized in fragment shaders when rasterizing the surface of the screen for the 3D model to be presented on where the scene is rasterized on a line-by-line basis. The normal of the polygon is utilized here to identify the gourard shaded surfaces that underlie these polygons where a vertex can be a part of multiple planes. The vertex normal is utilized to identify the set of vertices that defines a particular plane after which the attributes that define the shading value of each individual vertex is then utilized in the linear interpolation formula given in 1.4 [19].

In modern day GPU architectures, shading is now conducted via texture mapping where textures are functions that determine the property of the surface [20]. One of the more widely used techniques of texture mapping is illumination mapping where the shading values of each vertex would correspond to direction of reflected virtual rays of light. Texture mapping is utilized as it can give added freedom to the number of ways that surfaces can be generated where surfaces can be generated with a variety of properties where it can be modelled to reflect light in a specular or diffuse manner leading to glossiness of the surface, bump mapping via normal vector perturbation of the surface to give the impression of surface roughness and illumination mapping used to imitate surface behavior with light. Texture mapping does not utilize linear interpolation but instead utilizes texture coordinates where multiple scanning techniques are utilizes to map the texture onto the surface of the primitive in the 3D model. Texture mapping can complement gourard-shaded surfaces especially for surfaces that require more realistic illumination details but this would be computationally more intensive and thus would be executed on a GPU where each scanline of the shader might even have to invoke multiple textures such as in the case for ray casting [18-20].

When manipulating the model on the client surface, various mathematical transformations are applied to the model to generate the model view required. The transformations that are encoded into the simple 3D reconstruction program facilitate rotation and panning only. However, even these relatively simple transformations require a series of mathematical tools to and motifs to be employed before such transformations can be enacted. This subtopic will cover these concepts in great detail-

The view frustrum is defined by the use of a matrix where it defines a volume of space that is visible to an observer in a 3-Dimensional frame of reference. The frustrum is delineated by 6 planes with four of the planes being top, bottom, left and right planes respectively. The final two planes are the near and far planes which determines the minimum and maximum distances of the view frustrum that is within the field of view of the observer. Any points beyond this range are culled from the surface. The view frustrum of the camera corresponds to the camera space which is the space within which the entire scene which comprises the model appears to a pin-hole camera. There are 2 types of spaces, global and local spaces where the camera space constitutes the local space. These spaces correspond to frames of reference of a scene with the global space being the default frame of reference to observe a scene in. In order to change the observer's frame of view, it requires a transformation from the global to the local space where each transformation is represented by a matrix. [21]

The view frustrum shown in the figure below-

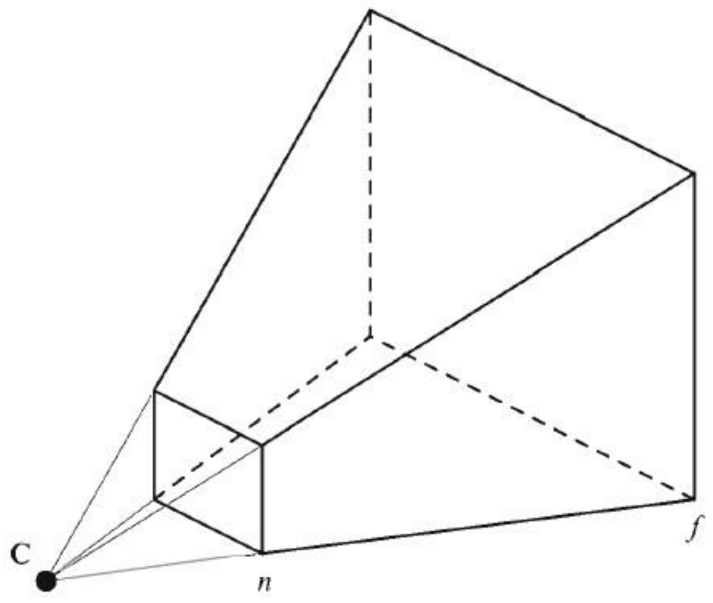


Fig.4. A schematic representation of the view frustrum which is utilized to transform the global space of the scene into the local space of the pin-hole camera [21].

There are generally two different types of projections, perspective and orthographic projections which do not take perspective into account. The view frustrum depicted above takes the position of the camera with respect to the scene into account and maps it onto the plane that the observer is viewing the scene from. Objects that are located farther from the plane will appear smaller while those that are located ahead of this plane will appear larger. All points located within the volume of the frustrum that is projected by the camera into the global space will then be transformed into the camera's local space where if applied to objects such as a cube, the topology would be distorted with respect to the distance of the points intrinsic to the cube from the camera. In the local space generated by the view frustrum, the x-axis points to the right while the y-axis points upwards. The z-axis points in the opposite direction towards the camera, this is done commonly via OpenGL and this direction will not change in Vulkan which utilizes OpenGL shaders to compute geometries sent to the shader [21]. The transformed image is transferred to the plane of projection which is located at a focal length e from the camera and is subtended at an angle termed as the field of view. This angle is generated by the intersection of the left and right planes that form projection plane. The field of view is utilized to determine how much of the scene can be viewed on the projection plane after transformation where larger field of views give you larger volumes of the scene but also corresponds to shorter focal lengths which is characterized by the distance e of the camera located from the projection plane in image space. The view frustrum is also defined by its aspect ratio where the aspect ratio represents the ratio of the height of the screen over its width. The aspect ratio α is utilized to define the vertical field of view formed from the angle subtended by the top and bottom planes where these two planes intersect the projection plane at ± α from the focal axis on which the camera is placed.

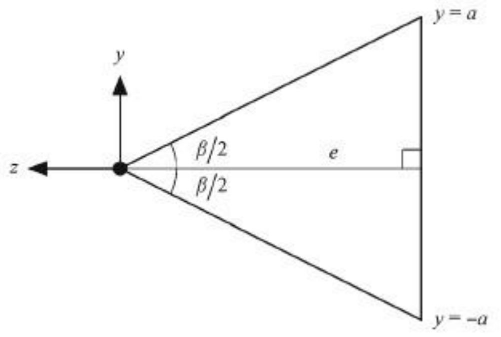


Fig.5. A graphical depiction of the vertical field of view (FOV) of the projection plane at a focal distance e. The angle β represents the vertical FOV [21].

The planes of the view frustrum are defined using normal and an additional coordinate termed as the homogenous coordinate which is utilized for depth interpolation of the vertices displayed in screen space. This is represented using bracket notation as depicted here-

$\mathbf{P} = < \boldsymbol{N}, \boldsymbol{D} >$ (1.5)

where P represents a point in camera space, N represents the normal vector associated with the surface that the point resides on and D represents the homogenous coordinate which will be used in further calculations to represent vertex depth when the point is mapped to screen space or homogenous clip space [21]. The Perspective projection is implemented via the view frustrum transformation where the x-coordinate of points should lie within the range of the left and right planes depicted by the left and right edges of the left and right planes respectively. The y-coordinate should not exceed the top and bottom edge which is formed from the intersection of the top and bottom planes with the y-axis of the global space. The z-coordinate should lie within the intersection of the near and far planes with the z-axis. The homogenous co-ordinate is often initialized to the z-coordinate. The view frustrum needs to be transformed from the global space to the local camera space which is a cube where here it will be referred to as the homogenous clip space. In the homogenous clip space, x, y and z coordinates need to lie within the range of -1 to 1.

The transformation that the points within the view frustrum would have to undergo in order to normalize these points in the homogenous clip space is given with the following equations-

$$x' = \frac{2n}{r-l}\frac{-P_x}{P_z} - \frac{r+l}{r-l} \quad (1.6)$$

$$y' = \frac{2n}{t-b}\frac{-P_y}{P_z} - \frac{t+b}{t-b} \quad (1.7)$$

The equations 1.6 and 1.7, represent the x and y coordinates in homogenous clip space that is projected onto the monitor screen. It is termed as the clip space because the entire model cannot be displayed on the screen and thus certain points within the model that is outside the view of the frame to be presented on the screen are clipped. Here, for equation 1.6, l and r represent the edges arising from the intersection of the left and right planes whereas n stands for the edge of intersection of the near plane with the z axis which is the space where the scene is mapped to the screen as an image. In Equation 1.7, t and b stand for the top and bottom planes respectively which form edges via the intersection with the y axis. The reason these equations have complicated coefficients is in order to normalize the coordinates such that the new clip coordinates x' and y' lie within range between -1 to 1 [21]. For z coordinates, the calculations are relatively more complicated where it is given by equation 1.8.

$$z' = -\frac{2nf}{f-n}\frac{-1}{P_z} + \frac{f+n}{f-n} \quad (1.8)$$

For the z-coordinate, since it is responsible for interpolating depth information about the model on homogenous clip space. Here, the model is rasterized by the reciprocal of the z-coordinate and where f and n stand for the edges of the far and near planes respectively. The x', y' and z' coordinates on the homogenous clip space which represents the plane of projection is represented in 4D space by a 4-dimensional vector with the 4th dimension of this vector representing the homogenous coordinate. This homogenous coordinate is responsible for delivering the perspective required for the projection in the plane with the w coordinate being $-P_z$. Thus, the size of all objects in the scene is scaled by its position along the z-axis and this is mathematically represented by the transformation of object space to a 4-dimensional camera space represented by the matrix, $M_{frustrum}$. The newly transformed 4-dimensional space is then converged onto a unit-normalized cube by dividing the vector by the homogenous coordinate- this is akin to projecting a 3D space onto a plane for every line that intersects the plane converges onto a singular point [21].

This is all combined to generate a matrix that represents the view frustrum transformation from camera space to homogenous clip space-

$$\mathbf{P'} = \mathbf{M_{frustrum}P} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \quad (1.9)$$

The vector generated from **P'** would need to be projected to the unit cube that represents the homogenous clip space and the vector for **P'** is given as **P'** = [$-x'P_x$, $-y'P_y$, $-z'P_z$, $-P_z$]. These vectors are in the 4 dimensional camera space which is then projected onto the 3D "plane" where the plane in this case is a unit normalized cube [21]. Once this is done, the perspective transformation is complete.

These transformations form part of a subset of transformations termed as the Model Projection View matrices where the view and projection transformations were elucidated in greater detail. The model transformation involves mapping the model from model space to global space whereas the view matrix transforms the global space into the local camera space. Both of these matrices are often combined into a single transformation termed as the model-view matrix. The $\mathbf{M}_{\text{frustrum}}$ represents the projection matrix that transforms the camera space into the homogenous clip space which is normalized via perspective division by the homogenous coordinate and the resulting scene is then mapped to the window/screen space of the screen where the scene is squished into the space ranging from -1 to 1. This is done during the rasterization stage of the graphics pipeline.

In the shaders, vertices are rasterized sequentially in a line-by-line manner where the screen can be represented as a discrete number of pixels. The screen space is represented as a series of scan lines where rasterization of the scene occurs. This mode of rasterization is based on linear interpolation as explained earlier using the Gourard-shading technique which was implemented on older graphics cards [21]. Modern day GPUs utilize non-linear interpolation techniques to account for depth values The depth co-ordinate is important because it is used to take the perspective of the camera into account where the z-axis points in the negative direction towards the camera [22]. This means that objects with smaller z-values are closer to the camera and thus will be transformed to appear larger and occupy more screen space whereas larger z-values occupy less screen space. This comes under the responsibility of the rasterizer which utilizes the perspective transform to interpolate the number of pixels taken up by the scene after perspective correction. All this is automatically encoded into the shaders as this is a fixed stage of the graphics pipeline that is explained in greater detail under the methodology section.

There are a wide variety of DICOM viewers that are available in the market and they range from open-source to proprietary solutions. These DICOM viewers are divided into 3 use cases that can be categorized into the following sections- Central Viewing, Decentral Viewing and Advanced Viewing. Central viewing does not require 3D rendering and only involves the extraction of DICOM data from PACS interfaces to view 2D cross-sectional slices. Decentral Viewing solutions requires web-based solutions where data needs to be transferred via long distances and is especially utilized in multi-center clinical trials where large cohorts of patient data needs to be integrated. Decentral Viewing currently also does not support advanced 3D rendering technology which is offered only by advanced viewing solutions where the entire focus is on developing sophisticated 3D rendering technology and improving analysis of volumetric data. Extensive visualization is obtained by the combination of image segmentation tools followed by powerful graphical rendering techniques.

One study surveyed 28 DICOM viewing programs which varied in a number of features and use cases. The most versatile solution offered in the market was OsiriX which was deemed to be suitable for a wide variety of cases but this was relegated to only cases that sufficed for 2D visualization. Web-based browsers for DICOM images were only available for 2D image visualization. The foremost of them was considered to be a program titled Weasis where this program is found to be optimal for 2D image visualization for web-based viewing where it covers the Decentralized viewing case which utilizes WADO or Web Access to DICOM Objects functionality. This is a DICOM feature that allows for the transmission of DICOM data over the internet. However, this is only applicable for 2D DICOM data and not supported to implement complex 3-dimensional rendering over the web. OsiriX does not support web-based protocols and is mostly implemented offline. ImageJ and OsiriX Lite were found to be the most advanced in terms of 3D rendering functionality however ImageJ utilized Javascript library for generating all of its 3D rendering capabilities where this library is not as competitive with the industry standard OpenGL which is not as competitive as Vulkan. Image J utilizes the Java 3D library to handle all its render functionality where the problem arises with additional driver overhead which can lead to unpredictable behavior for high-end graphics applications which requires a lot of computational power and thus minimal driver overhead. The Java 3D library also interfaces with OpenGL which can be skipped entirely by utilizing OpenGL directly. ImageJ was found to utilize the marching cubes algorithm to execute surface rendering of the 3D model. OsiriX is more advanced in this scenario because it utilizes the VTK library which was developed by the US National Institute of Health which employs the OpenGL library for graphics rendering capabilities. It is written in C++ which is a low-level programming language that allows for memory allocation however OpenGL comes with its fair share of problems such as lack of developer freedom when it comes to designing the memory allocation workflow or closed off from access to more intricate components of the microcontroller hardware such as the GPU. This leads to unpredictable performance at higher clock rates when memory is not properly allocated as this comes under the jurisdiction of the OpenGL driver which can allocate memory erroneously [23-26]. Developing an efficient program that synchronizes across the CPU and GPU more effectively leads to programs that are capable of executing advanced rendering techniques such as global illumination modelling or raytracing [27]. This cannot be conducted as efficiently as programs built on openGL.

ITK is an image segmentation library that was developed by the US National Library of Medicine in collaboration with a consortium of other institutes and this is a library that is responsible for conducting a wide variety of image segmentation protocols. OsiriX utilizes ITK for medical image analysis whereas ImageJ utilizes the Java 3D library which is written in JavaScript. ITK libraries are preferred because it was specifically designed to analyze medical image data where it was developed for the Visible Human Project whose main aim is to analyze the anatomy of the head and neck via data obtained through non-invasive imaging techniques, one example being the CT modality. The library is open-source and supports a wide variety of image analysis techniques such as filtering, processing, segmentation and incorporates mathematical concepts such as partial differential equations, level set mathematics, finite element models amongst others [28].

For image processing and data extraction, the itk and vtk toolkits were utilized where itk was developed specifically for handling medical image data that is encoded within DICOM files. The ITK toolkit is written in C++ and utilizes generic programming in its API. It is primarily utilized for tackling volume segmentation and image registration problems. The generic programming architecture allows the ITK toolkit to be utilized for the analysis of n-dimensional image data. ITK toolkit is utilized to extract the volumetric data from the DICOM image series which can then be acted upon by using the VTK toolkit [28].

VTK is utilized for advanced visualization being built upon the OpenGL library however here it will be utilized for filtering the volumetric dataset which will then be displayed on the custom-designed Vulkan engine. The VTK toolkit has a wide variety of filters that can be employed to remove artefacts and smoothen the data for enhanced visualization. Example of such filters are gaussian and median smoothing filters. VTK was chosen for the image processing steps due to the ease of use of the API however more advanced processing algorithms are offered by the ITK toolkit [29].

Vulkan is a high performance graphics and compute Application Programming Interface (API) that was developed by the Khronos Group which is an international consortium of companies that create open standard APIs for accelerating and authoring high-end graphical media. Vulkan is the successor of the OpenGL standard which has been widely used by industry for the development of high end graphics applications for years. However, with the advent of machine learning and computer vision, there is an increased demand for more compute-intensive tasks that can be multithreaded onto the GPU reducing the overhead on the CPU [30]. OpenGL was found to be ineffectual in this manner as it did not accommodate multithreading and complex drivers carry much overhead as they deal with low level tasks such as automatically allocating memory and managing errors that arise from the firmware, all of which is not under the control of the developer [30]. Vulkan circumvents these issues wherein it facilitates multithreading with multiple threads being generated by the CPU being allocated in a synchronized manner to the GPU. It also utilizes an intermediate shader language termed as SPIR-V whose full form stands for Standard Portable Intermediate Representation. This language is crucial in that it improves shader module portability allowing it to be easily incorporated into any program and reduces driver complexity which reduces overhead. It also allows developers to debug their code without any impact on performance which was not possible in openGL [30]. Debug information is transmitted through validation layers which are automatically embedded into the API which can be accessed interactively. SPIR-V is also unique as it is able to incorporate compute and graphics functionality together through one API which was previously not possible where compute functionality was delivered only by the OpenCL language whereas OpenGL delivered high-end graphics performance but with limited customizability of the graphics pipeline to run self-designed algorithms on shaders. This allows for the first time to integrate high end compute capabilities that lie within the realm of machine learning and artificial intelligence for graphical applications such as model segmentation or the possibilities of introducing novel rendering techniques such as cinematic rendering [27].

With Vulkan also comes a supported file format which is the glTF format where this format is paving the way forward in the manner in which 3D files are stored and transmitted across different clients. This format is unique because it was specifically designed to support 3D rendering on a web browser. It is quickly becoming the standardized format to store 3D data with companies such as Google, Facebook and Microsoft supporting the implementation of this format at a wider scale with augmented reality applications choosing glTF as the standard format to store scenes and objects in. The reason this format is steadily becoming increasingly popular is because of its ergonomic design where it is lightweight and offers multi-functional support. The way in which it does this has to do with the manner in which it stores 3D data where the data interchange format JSON comes into play. Here, JSON is utilized to categorize the 3D object data also referred to as a scene into several different compartments and these compartments serve as references to the data which is stored in a binary format. The scene is composed of nodes which are subcomponents of the scene. Each node is embedded with multi-functional support and stores intrinsic data such as the geometry of objects within the scene. Geometric data is stored as meshes which themselves are composed of a series of smaller geometric objects termed as primitives which are basically a series of triangles used to define the mesh. These meshes are stored within buffers which serve as silos that store mesh data. These buffers are accessible through JSON file which serves as an intermediary delivering references that point to said buffers when requested. Materials are another object that determine what surface rendering technique is employed to generate the model where this once again goes back to texture mapping where textures are utilized by materials to determine how the surface should be rendered. One example can be the glossiness of the material which utilizes illumination mapping where textures can be utilized to deliver information about the normal vectors used to define primitive surfaces that compose the model where surfaces with misaligned normal vectors generate diffuse reflection giving the appearance of roughness whereas surfaces with aligned normal vectors generate specular reflection giving the appearance of glossiness characteristic of metal surfaces. This is similar to the Bling-Phong algorithms utilized to generate the glossiness of surfaces [31]. Each node contains geometric data represented by the mesh as well as functional data represented by objects such as materials and texture which determine how surfaces should be rendered. The glTF format also supports animations which is another feature that is stored in the node used to render a particular object within the scene where an object such as a skin is utilized to generate required animations. Because of how extensively the architecture is designed to compile and maintain the structural integrity of the data required to generate a 3D model, no conversions are needed from glTF to another file format and data can directly be streamed by the applications written using the Vulkan API [30-32].

2.6 DICOM file format-

The file format DICOM which stands for Digital Imaging and Communications in Medicine is the benchmark medical standard when it comes to storing and transferring patient information obtained via various radiological imaging modalities digitally. The DICOM format delineates the exact criteria according to which obtained imaging data must be encoded and stored. This file type is designed to be universal in nature where it is independent of the modality or vendor make that is utilized in obtaining the data. This allows data to be digitally exchanged across different firmware regardless of the vendor make but has initially been developed solely to be viewed on specialized workstations which naturally with time, as technology progressed, expanded to the domain of commercial client hardware such as regular laptops and desktops that is accessible to the regular person. The DICOM file format was developed by the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) where a joined committee of the aforementioned respective bodies was created in 1983 to promulgate their efforts into the development of a file format that meets the clinical requirements of radiology and keeps pace with the technological advancements of the time where the transfer of digital data was becoming increasingly widespread at that point in time. It was originally termed as the ACR-NEMA standard to reflect the governing bodies responsible for the standard but has since been revised in 1993 to become the current standard as we know today [33]. The current standard has a plethora of functionalities with the core functionality being the ability to facilitate an interface across which data can be exchanged across different hardware in a multi-vendor environment. This led to the development of PACS systems which were discussed earlier where data could be successfully retrieved from the imaging modalities and stored offline in servers. The DICOM format is incredibly versatile being able to link across a gamut of devices such as workstations, printers, networking servers that uphold the PACS infrastructure and the imaging modalities. The DICOM file format cannot be directly viewed on a personal desktop requiring specialized software termed as DICOM viewers to be able to open the file for viewing [34].

The basic layout of a DICOM file consists of 2 components- the header and associated image data acquired from the imaging modality. The image data is stored as a long string of binary data that is categorized under the tag 7FE0 [33-34]. This binary data encodes pixel information which requires the use of information stored within the header file to process the binary pixel data into an output image. The binary data is encoded in a data container termed as a data attribute where each DICOM file contains only a single data attribute relegated to the pixel data itself. However, multiple frames can be stored within this single data attribute. The data attribute also has an arbitrary bit depth that can be adjusted based on the image data obtained.

The header compiles information of the patient and associated parameters of the treatment such as patient demographic information, acquisition parameters of the imaging modality, image dimensions and referrer or practitioner responsible for treatment oversight [35]. The information in the header can be subdivided into 4 categories- information specific to the patient, the study itself, the series of images obtained within the study and the nature of the image obtained itself [35].

The patient information here corresponds to personal details that can be used to identify the patient such as his/her name, sex, weight, birthdate and so on. The study information expands upon the details regarding the study itself such as the practitioner involved, examination report, etc. This information is stored in an object termed as a unique identifier or UID [34]. UID is also utilized to define all aspects of the image series obtained such as the modality utilized to acquire the image series, date and time the image series was acquired, body part that was the target to the image acquisition procedure among others. The image specific information is concerned with the resolution and size of the image that is obtained. This information about the image that is encoded within the header is crucial for the client device in displaying the image on the monitor. All this information is stored in a motif termed as the DICOM Information Model which serves as a template that records all the attributes required to define the dataset of images retrieved from the patient. These attributes can be utilized to define the nature of the images that are retrieved, details of the performed procedures elucidated in the structure of a report and patient-specific data [36]. The DICOM Information Model is stored in the form of discrete data structures termed as Information Object Definitions (IODs) which are stored in the header file and is required to parse the data encoded within the pixel image data into the necessary format. These IODs are crucial in filing data into the subcategories mentioned earlier where each IOD represent a class of information objects that correspond to the real-world objects in the data that they are trying to encapsulate. Instantiating each IOD results in a DICOM data file that can be transcribed into an image output but IODs in itself without any data specific to the patient is just an empty template [34-36]. The data obtained from the patient is acquired in what is termed as a Service Episode which is a series of events that is aggregated and bounded within an interval delineated by a particular start and stop time. The service episode defines the environmental context surrounding the treatment/acquisition of image data of the patient where this can take into account the healthcare organization involved with the treatment (the authorized physician, radiologist and other healthcare providers legally authorized by administrative entities representing the organization), the physical location or department at which the treatment was conducted and the diagnosis of the patient at the time of admission and discharge [34-36]. A visit which can be classified as a subset of the service episode, representing a single event that occurred within the vicinity of a single healthcare organization whereas a service episode represents a series of events that can be categorized as the entirety of a particular treatment for the patient where prolonged service episodes have multiple visits such as the duration of a pregnancy or a rigorous oncology regimen. A single visit would encode more specific information about the context of the clinical environment as described above [34-36].

Associated with the visit are a series of information objects that starts with the Imaging Service Request which carries more detailed information about the particular procedure that is to be conducted. It is brokered between an imaging service requester and an imaging service provider. The imaging service request provides information for the particular procedure with respect to the visit or service episode and can be identified through the use of an Accession number [34-36]. Each accession number consists of 2 tags with these tags representing the group and element tags respectively. The Imaging Service Request represents an example of an Information Object Definition. IODs are utilized to represent various types of clinical data ranging from image data to patient data, medical records, reports and technical information about the equipment and modality used for procedures. These data members are often represented as data attributes and consist of accession numbers that represent them. An example of this would be the accession number (0008, 0090) which gives the name of the referring physician and is one of the attributes that collectively form a part of the Imaging Service Request IOD [34-36]. The Imaging Service Request is utilized to populate another IOD termed as the requested procedure which represents a clinical procedure to be employed with a given modality. The requested procedure is an information object which serves to bridge the connection gap that exists between the imaging service request and various other procedure types. The procedure type is the final specification of the exact procedure where it is designated as an element in a catalog. The procedure type is requested by the imaging service requester and is specified by the imaging service provider in the form of a requested procedure. A requested procedure is an instantiation of the procedure type which is populated with specific information required within the clinical context applicable to the patient. Here, according to NEMA, a requested procedure is the smallest unit of service that can be requested and billed for. Here, each requested procedure corresponds to one procedure plan and can specify one or more imaging modality equipment required for the necessary procedure [34-36]. The requested procedure is itself a canopy of information objects termed as the Modality scheduled procedure step. All the information object definitions are recorded in a flowchart given in figure 6 below.

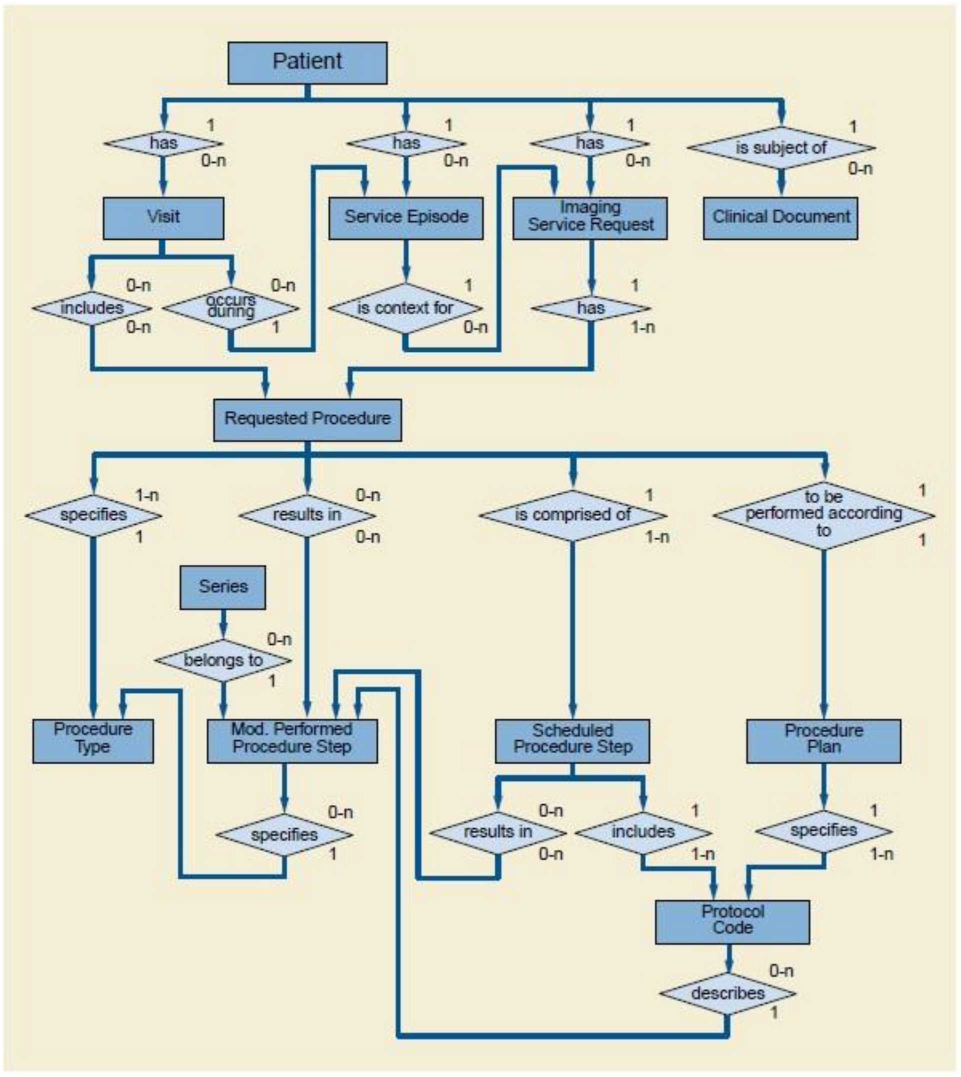


Fig.6. A flowchart diagram representing a model of the real-world steps that would be taken to record a procedure where each step is encoded as an Information Object Definition (IOD). The purpose of this

model is to organize and collate all the information regarding the context of the clinical environment in which the procedure was performed. This model is required to normalize communication protocols between cross-platform equipment by utilizing agreed-upon definitions within the model to barter information exchange [36].

The Modality scheduled procedure step is defined as a unit of service, a series of which are specified by the Procedure plan to make up the requested procedure. Each step elucidates a certain protocol specified by the procedure plan. These set of protocols are termed as the Defined Procedure Protocol and are referenced to via assorted protocol codes. The defined procedure protocol is what forms the core of the procedure plan that is utilized by the imaging service provider to develop the requested procedure. This is done via informing the scheduled procedure step. Note that the protocols within the scheduled procedure step can differ to those elucidated within the procedure plan. The scheduled procedure step or protocol entails specific information with regards to the clinical context of the patient where variables such as human resources, start and stop time alongside duration, imaging modality and other clinical equipment paramount for the protocol, location and consumable supplies are specified in more detail. The imaging modality is specified but the exact calibration parameters may not which is considered to be outside the scope of the standard. However, it is made explicitly clear that only one imaging modality equipment be used within any scheduled step.

The modality performed procedure step is one that specifies the protocol that was actually conducted after the procedure is performed. Any image series that is generated is referenced by the modality performed procedure step and this step contains detailed information as to the parameters of the treatment and how it was performed. However, a crucial note to consider is that the modality performed procedure step is independent of the storage semantics employed while encoding the data into the designated storage space. The modality performed procedure step is utilized for the creation of SOP classes where SOP stands for Service-Object Pair. These classes are responsible for encoding the imaging data obtained from the modality into a format that can be stored. If a DICOM file needs to be transferred across a network to another device or stored with full fidelity, SOP class would have to be instantiated and the modality performed procedure step is responsible for generating the data required for a SOP instance. All this information is stored in the header file [37].

IODs however are all not of the same category where they can be subdivided into 2 different types- composite and normalized IODs. Composite IODs represent data that is acquired from multiple real-world objects whereas normalized IODs represent data corresponding to a single real-world object. Composite IODs are utilized to represent imaging data acquired from modalities and are designed to be objects that cannot be altered by subsequent users. This preserves the fidelity of the medical records and data. Composite IODs were originally designed to enable backward compatibility with predecessors of the DICOM standard and were designed to omit any update protocol that could alter the existing imaging data that was established in the file. Normalized IODs can be altered and represent more simplified data attributes often representing properties of single real-world objects [37-38]. They were designed to reflect more flexible information management protocols and support basic operations such as update (N-SET), retrieve (N-GET) and create (N-CREATE). Composite IODs operations are more complex with them having to do around dealing with the management and transfer of image data with examples being store (C-STORE), retrieval (C-GET) and query (C-FIND). They however do not support update operations. Both the composite and normalized services are handled by SOP classes which represent an amalgamation of 2 components of the DICOM Information Model which are the IODs and another object termed as the DICOM Message Service Element (DIMSE) group or media storage services. DIMSE is another structural motif that forms a part of the DICOM Information model along with Information Object Definitions (IODs) as shown in figure. This motif is responsible for managing communication protocols that lead to the transmission of data across networks and point-to-point interfaces between consumer applications termed as application entities [37-38]. DIMSE utilizes SOP classes for these communication protocols where instantiating SOP classes will result in a SOP instance that is brokered between two entities, a storage class provider and a storage class user. In order for a communication terminal to be successfully brokered between the two bodies, a function agreement must be come to where this function agreement specifies that the service such as image transmission and the type of modality or equipment used for the service have to be the same in order for data transmission to be successful. SOP instances are required to ensure that this function agreement is satisfied. This is important because occasionally there can be workstations that do not support a given functionality such as not supporting the display of angiographic images and SOP instances are required to validate whether communication channels can be opened or not between two stations [37-38]. Once this agreement is set and a communication channel is opened up, data can be transferred where data is transferred in the direction from the Storage Class User to the Storage Class Provider where for the example of image acquisition through a CT modality, the CT modality is the Storage Class User and the workstation that receives the acquired image series becomes the Storage Class Provider. In each SOP Instance, the data is structured through the use of a Unique Service Class Identifier (UID) which is utilized for querying specific data attributes such as date/time information of an image series or patient/equipment information regarding the clinical context of the procedure that was performed. UID is important for identifying where the series dataset is stored and extracting it for image processing. It is also used by the server tasked with allocating the required memory needed to store the information object in question. The UID is important as it will be used by the program to retrieve the image series in question for volume construction of the 3D model [37-38].

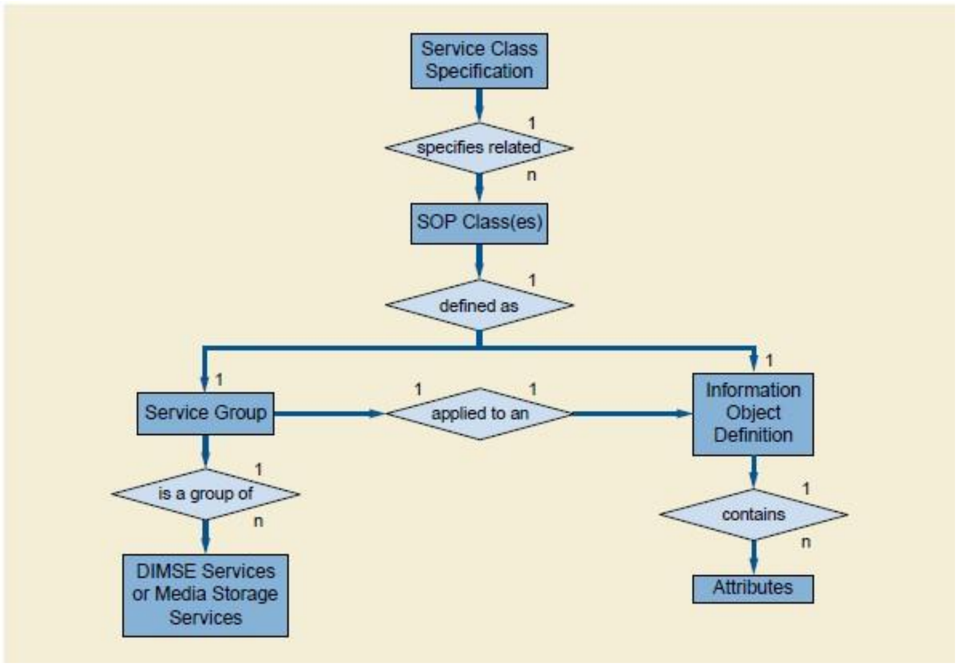


Fig.7. A flowchart diagram representing the relationship between an Information Object Definition object and the associated Service-Object Pair class. The SOP class is responsible for defining the operations required to be enacted on the Information Object Definition that serves as the target of these operations [36].

3. Research Object and Methods

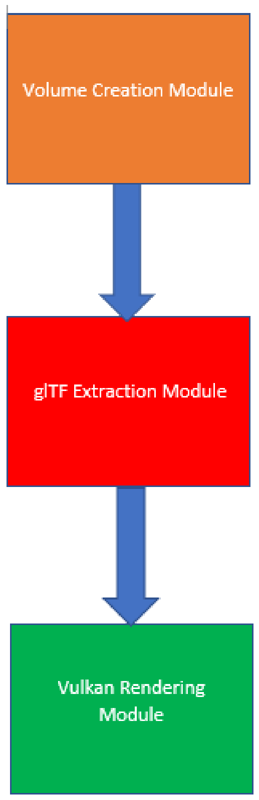


Fig.8. A flowchart/model representing the various components of the Vulkan DICOM Viewer.

The program has 3 modules which are linked together to generate a working program. It begins with the volume creation module which processes a set of DICOM images in the '.dcm' format where it utilizes the functionality offered by the itk toolkit to search and extract the data when the input offered is just the directory where the DICOM dataset is located. The volume creation module then generates a .vtk file that represents the volume file that can then be used for visualization. The volume file is then sent to the processing portion of the volume creation module where this is responsible for identifying artefacts and removing them as well as smoothening the model for presentation. The processed data which is now a model is then converted into the appropriate file types that can be utilized by the rendering engine for visualization which is the (.stl) and (.glTF) format where this is conducted by the file format glTF extraction module. The data that is extracted is then channeled through into the Vulkan Rendering Module which serves as the graphics rendering engine which then projects the 3-dimensional model onto the display screen of the client desktop/laptop device.

3.1 Volume Creation/Processing Module

The volume creation/processing module was created using the vtk/itk libraries where the vtk library is utilized in the model processing and file format conversion step whereas the itk libraries are utilized in the volume creation step where the images obtained are superimposed upon each other to generate a file in the (.vtk) format which carries the necessary information to render a volumetric file in 3D. The itk library is utilized to obtain images from a directory which is the only input that needs to be specified. The directory is then processed according to the Unique Series Identifier (UID) that is specified. Generally, 2 UIDs are present for a DICOM series where only one of them requires to be specified in the volume module as it specifies the required set of DICOM images utilized to comprise the volumetric dataset of the model which generates the volumetric file in the (.vtk) format.

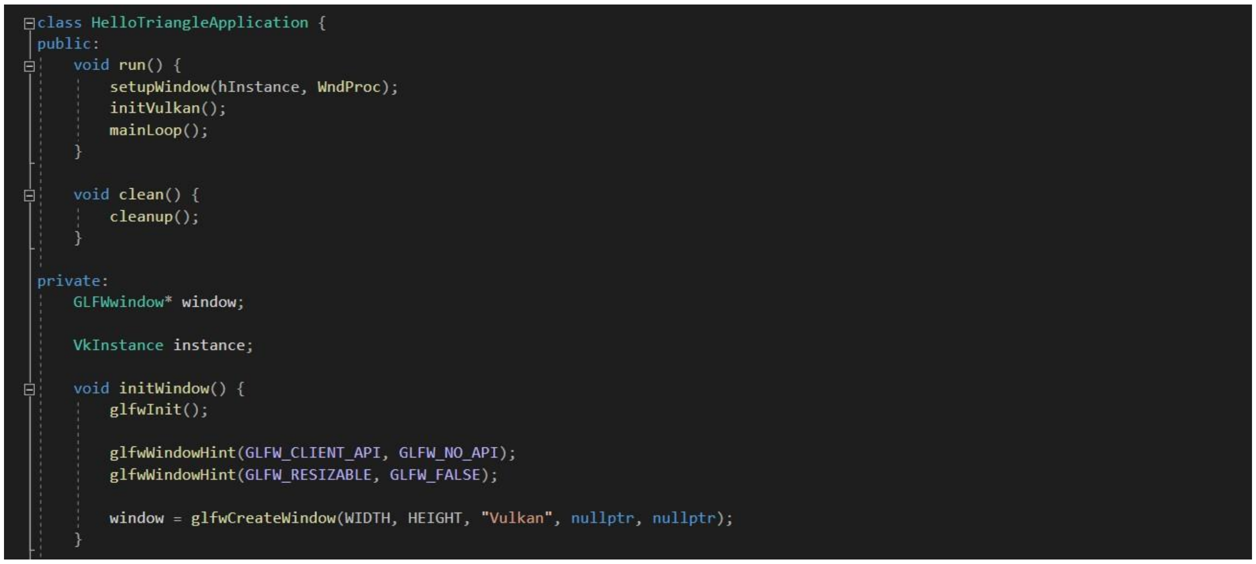
This volumetric file is then siphoned through a variety of filters generated via the use of the vtk library where the role of the filters is to smoothen the data to remove surface roughness that generates noise that can be viewed in the application. It is also required to threshold the data where models are generated from those vertices within the dataset that consist of the same Hounsfield values. This allows segmentation of different organ systems within the body such as blood, bone or soft tissue based on the threshold value that is inputted into the system. The vtk library supports this type of threshold-based sectioning of the model where a subset of points corresponding to the same Hounsfield values are extracted for the purpose of rendering a model for the specified organ type. In this case, the skeletal system is required to be extracted from the model dataset and thus the threshold values specified for this dataset is set to 500 that was arrived to via trials of experimentation. Determining the threshold value that corresponds to the associated organ system of interest is something that the user would have to experiment with the program to find out. The vtk core of the module is responsible for creating the model in 3 different file formats- the default format that the 3D geometry data of the model is generally stored in which is the (.vtk) format, the (.stl) format which can be utilized for 3D printing the models of interest and the (.gltf) file format which is the primary format that is utilized by the rendering engine module to display the model on the screen. The contents of the (.gltf) file can also be viewed on any internet browser application window if the need arises giving the user the freedom of choice.

3.2 Vulkan Rendering Module

Microsoft Foundation Classes and Base framework of the application-

The basic framework for the Vulkan Rendering Module was built using Microsoft foundation classes where this is represented by the header classes <windows.h>, <fcntl.h>, <ShellScapingAPI.h> and <io.h>. These header classes encapsulate a variety of methods that are important for establishing an application framework that can operate on the windows operating system such as setting up buffers that can stream to windows on the screen and sending operations to the windows kernel which is responsible for orchestrating multiple threads to run simultaneously. Each thread is responsible for running a particular application where the windows kernel is responsible for allowing the windows operating system to run multiple programs on their interface. The function that allows a program to interface with the windows kernel is titled as WinMain which has 4 arguments. These arguments don't need to be filled out for the purpose of this program. The WinMain function is instantiated with the type WINAPI which is required to declare to the windows kernel that memory needs to be allocated for another thread for a new application to operate upon. The WinMain function is required to generate an instance which is a data object that represents the application that is scheduled to be run on the windows kernel as a thread [39].

There is also the establishment of a callback function WNDPROC which is required to send messages to the instantiated window and a function that allocates screen space for setting up a window termed as the setupWindow function. Another function termed as mainLoop is utilized to keep the application running where it handles and executes the messages delivered by WNDPROC [30]. The mainLoop uses the WINAPI commands TranslateMessage and DispatchMessage to execute these commands where these messages are dispatched to the windows kernel to execute the required windows procedures. These commands are encapsulated within a while loop which is required to keep running until it receives the WM_QUIT message at which point the application logs out of system memory and shuts down. This is instantiated by clicking on the close button of the window [30]. The application functions on a frame by frame basis where each frame that is presented to the screen of the desktop is operated upon by the application and the WM_QUIT message triggers the application to quit upon entry to the next frame. The mainLoop function is established within a data structure termed as a class with the name 'HelloTriangleApplication' where this class consists of both public and private members which are data containers termed as access specifiers [30]. Classes are utilized to aggregate and sort multiple objects and methods in a systematic manner that simplifies organization of the code and smoothens communication of data across different objects. All the objects and methods are directly accessible from the class through the use of reference pointers which are symbolic representations of memory addresses of the client device. The public member of the class reference methods or functions that are directly accessible by methods that exist outside the class within the program whereas the private member contains methods that are not accessible except within the frame of reference of other methods that are instantiated from within the class itself [30]. This is shown in figure a1 below-

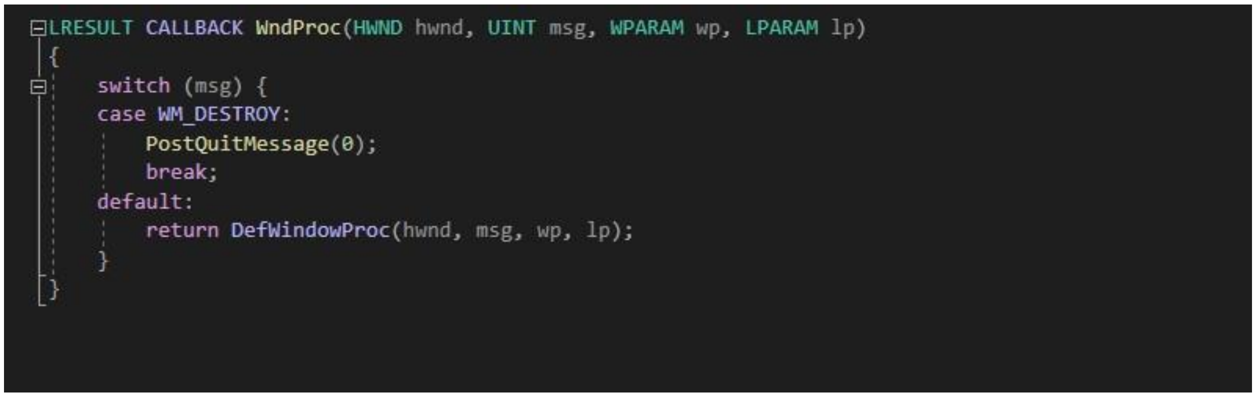


```
class HelloTriangleApplication {
public:
    void run() {
        setupWindow(hInstance, WndProc);
        initVulkan();
        mainLoop();
    }

    void clean() {
        cleanup();
    }

private:
    GLFWwindow* window;

    VkInstance instance;

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    }
```

Fig.9. A representation of the HelloTriangleApplication class with the public and private member structs termed as access specifiers with the individual methods contained in each struct.

The method ‘run’ executes the entire program and is called from within the main function of the application. The run function is a public method and as such is accessible by the main function of the application. The run method itself however contains multiple methods to be executed in a directed fashion which are private member functions that cannot be accessed from outside the frame of reference of the HelloTriangleApplication. These functions however can be accessed from the public member struct. The method ‘initVulkan’ is a function that executes the vulkan library where all functions that utilize vulkan functionality are specified within the private member struct of the HelloTriangleApplication class and these functions are accessed by initVulkan which is also part of the private member struct. initVulkan is necessary to establish the scaffolding of the program where all the necessary components needed for the rendering engine are created and integrated to deliver a working program. This function is only called once in the entirety of the program’s creation. All the functions in the private member struct do not get instantiated within the application unless it is called for by run function in the public member struct within the HelloTriangleApplication class.

The WNDPROC function is utilized to log all events that are initiated by the user examples of which are keyboard inputs or mousepad movements. These inputs are logged as messages which can be extracted from the windows kernel through the use of the WNDPROC function which is designed to catch and collect these messages from the windows kernel. This is done through the use of a switch-case statement which are utilized to determine the identity of the required messages that are captured by the application.



```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wp, LPARAM lp)
{
    switch (msg) {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd, msg, wp, lp);
    }
}
```

Fig.10. A representation of the WNDPROC Callback function that is responsible for delivering messages from the mousepad or keyboard to the application to trigger transformations of the model.

With the application framework set up, the Vulkan library can now be utilized where here all the necessary functions required for model rendering and visualization are utilized. Before the methods from the Vulkan API can be utilized, an instance must first be created where memory must be allocated within the application itself for Vulkan objects to operate without external interference of operations arising from within the application itself. This allows the application to successfully interface with the Vulkan library and employ the required methods necessary for the application. The function createInstance is established in the private member struct of the 'HelloTriangleApplication' class and can generate the instance with the required functionality.

Once this is done, the surface is created where the surface represents a type of interface that the application can utilize to stream data to the window of the screen. This is set up directly after the GPU is selected for the application to run on. One of the core features of Vulkan is that as mentioned earlier, it gives the developer the freedom to configure the necessary hardware components required for the application to run on. This is generally taken care of automatically by the driver which has pejorative implications for optimal runtime of high-throughput applications for high-end graphics. One of the manners in which this is accomplished is giving the developer the freedom to select which GPU to use for the application and the ability to configure the GPU accordingly. This is accomplished through the pickPhysicalDevice and createLogicalDevice functions. These functions select the GPU based on the functionality required by the application. Here, the GPU needs to be capable of presenting images to the screen and thus this property needs to be queried for by the pickPhysicalDevice function. The createLogicalDevice function is then utilized to create the configuration required for the GPU to support the application. Note that the entire program is executed on the CPU and only the shader which was discussed earlier is executed on the GPU which is responsible for rasterizing the pixels on the allocated space on the desktop screen. The shader program is specified separately which will be discussed in a later section-

One of the functionalities that is required of the GPU is the swapchain where the swapchain is a data motif responsible for the organization of images or frames that need to be displayed to the monitor when requested. Essentially the purpose of the swap chain is to synchronize the rate at which images are streamed by the GPU with the refresh rate of the monitor in question. This is depicted in the diagram below-

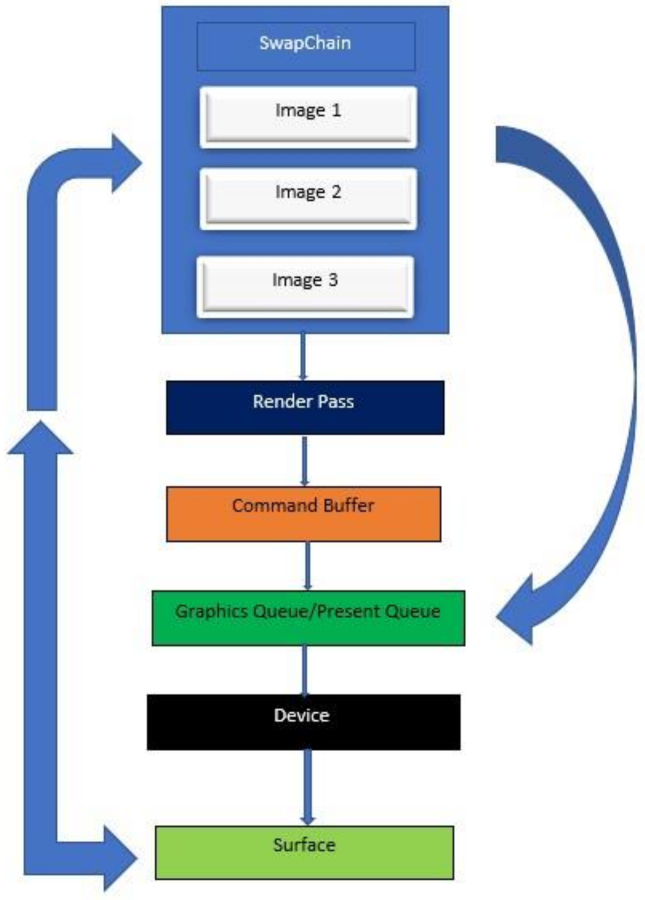
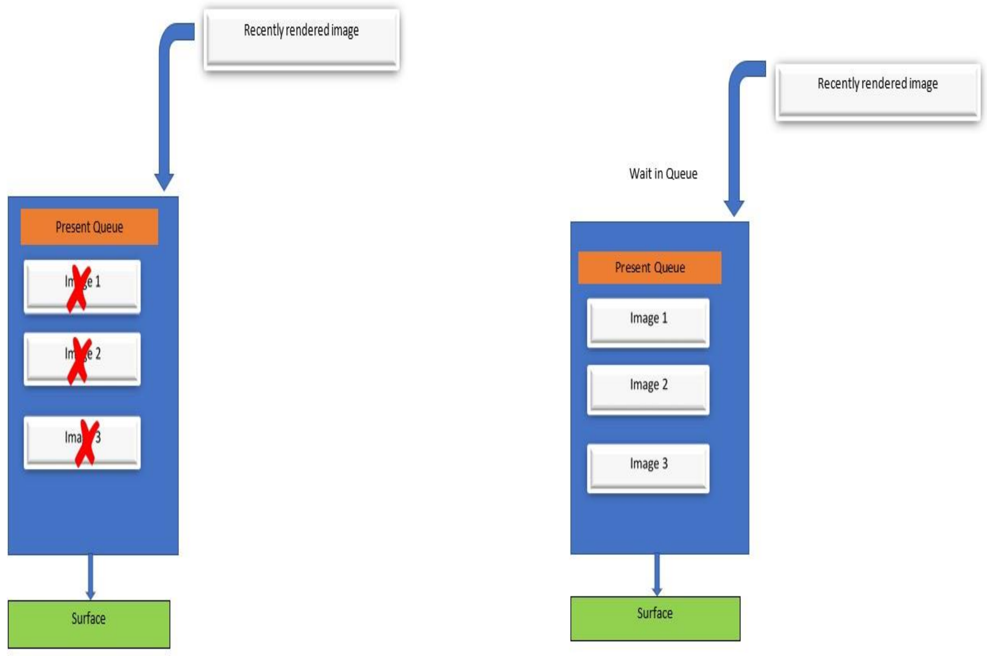


Fig.11. A representation of the image relay framework consisting of the swapchain and its relationship with various other components within the Vulkan rendering engine. Here the image is relayed from the swapchain indirectly to the graphics queue via the renderpass for graphical read/write operations and directly to the present queue for surface presentation. In this application, both the graphics and present queue are represented by the same queue.

Initiating a swapchain requires support from both the device/GPU that supplies the images and the surface which displays them. Firstly, the device needs to be queried for swapchain support which is one of the responsibilities handled by the pickPhysicalDevice function. To determine if a GPU can support a swapchain, the pickPhysicalDevice function would need to determine the system properties of the GPU mainly what kind of queues that can be supported by it. Queues and the associated queue families that queues originate from are the building blocks of the GPU with each queue family supporting a specialized operation with the queues that are derived from each queue family being identical in design and purpose. Each queue represents the most inchoate unit of functionality that can be employed by the application. There are different types of queues for various applications where here, the application is relegated to using only presentation and graphics queues. The graphics queues are required for processing geometry data and rasterizing the model on the screen whereas the presentation queues are responsible for delivering the images to the monitor to be displayed. Generally, the graphics queue families tend to have the same properties as the presentation queues so queues can belong to both families simultaneously. However, to be on the safer side, the function is designed to query whether the GPU does indeed possess both queue families. Once this is confirmed, the createLogicalDevice function is utilized to generate a data object termed as a logical device that can configure the GPU to handle the load of the application. All the necessary queues are configured through the logical device with the presentation queue being configured to interface directly with the swap chain which will be created next.

The createSwapChain function is utilized to generate the swap chain, another object that handles the references to the collection of images that will be presented to the monitor. These images represent frames that have been successfully rendered and ready to display on the screen [40]. The swapchain has a variety of properties that are important in influencing how the user can interact with the 3D model on the monitor screen where this depends on the presentation mode. The presentation mode defines how the frames will be rendered on the screen where vulkan consists of a variety of different modes based on the potential load and the latency of the application. Based on these parameters, the present mode can be configured to shield the user from the inconsistencies in application performance that can crop up from time to time when dealing with high-performance graphics. The presentation modes of the swapchain are handled by the present queue that was configured in the logical device. The two main modes supported by Vulkan are fifo mode and mailbox mode where Fifo mode works by stalling the application when the present queue on the GPU is fully occupied with images [40]. This occurs when the refresh rate is much lower than the output generated by the application. An image is presented every time the display monitor is refreshed whereas rendered frames arriving from the graphics queue are delivered to the swap chain which is then inserted at the back of the present queue. The mailbox mode is utilized to maintain synchronicity between the images that are displayed on the monitor and the rendered frames generated by the graphics queue. This is done by eliminating the images lined up on the present queue that are already out of synchronicity. These asynchronous images that are removed are then replaced with the most recent rendered frame that is generated by the graphics queue [40]. This can result in sharp and jarring transitions of the 3D model when the application is performing erroneously. For this program, the mailbox mode is configured for the swap chain. This is depicted in the diagram below-



(A) (B)

Fig.12. A representation of the different present modes that can be customized according to the preference of the application where in A) the MailBox present mode is represented where the images that have been forced to wait in the present queue due to lagging performance are instantly eliminated and replaced by the incoming rendered frame which synchronizes the rendering operations with the frames being presented on the screen. B) represents the Fifo present mode which is the default present mode that is queried when MailBox present mode is unavailable where in the case of performance errors, the recently rendered image frame will have to wait in present queue along with the other image frames-this could potentially lead to stalled applications.

The swap chain needs to contain images and the dimensions of the image as well as the type of image that will be used can be specified in the createSwapChain function. The function then instantiates a swapchain which will allocate memory from the application for the creation of images. The swapchain in this application is allocated 3 images to handle which will be collectively recycled across the graphics and present queue of the GPU [40]. These 3 images are the only images that will ever be displayed on the screen as the swapchain recycles these images and thus only 3 frames are rendered to during the application's runtime [40].

After the swapchain is set up, other methods within the private member struct of the HelloTriangleApplication class need to be able to access the images that have been instantiated for various post-processing operations. This access is enabled by generating 2 objects- image view and the associated framebuffer. The image view is responsible for providing access to the image especially the individual channels responsible for encoding color data for each pixel of an image. Images for 3D rendering consist of 4 channels- red, green, blue and alpha where red green and blue represent color information and alpha represents depth information which is useful for depicting 3D models. Each channel encodes 8 bits of information which leads to a total of 32 bits of information per image [40]. This comes under image formatting and has already been set up and specified when instantiating the swap chain. The image view allows other functions to modify and manipulate the properties of the image as deemed appropriate for the requirements of the application. The framebuffer behaves as a wrapper for the image and as well as the interface where other components within the application can interact with the image. The framebuffer utilizes the image view to manipulate the properties of the image as deemed appropriate and if any function requires an image to process, the image is accessed via its framebuffer. Each of the 3 images within the swap chain are linked to an image view and associated framebuffer [40].

The framebuffers that are instantiated are then incorporated into the render pass. The render pass is an important object where it serves to condense all the information contained by framebuffers into one accessible resource. The render pass is utilized along with another object known as the graphics pipeline by command buffers which are responsible for executing the program. The graphics pipeline is responsible for executing operations (such as rendering) contained within the vertex and fragment shaders onto the image frame held onto by the swap chain. The graphics pipeline employs the graphics queue within the selected GPU device to do this [30]. The graphics pipeline cannot function on its own and requires the command buffer which works as a sort of switch that activates the graphics pipeline when needed. Not only does the command buffer initiate the execution of the graphics pipeline but it is also responsible for configuring it accordingly. The render pass is responsible for supervising the management of multiple framebuffers and synchronizing them with the execution of the graphics pipeline. The render pass is important when scheduling large compute operations that can take up memory and thus be more likely to display erratic behavior [40]. Render passes come in handy when rendering to an image depend on multiple textures that are also encoded as image data. Organizing these textures and scheduling image transitions between states (such as read and write state) as well as making sure that the image is in the proper state to become a target for the graphics pipeline are all steps that the render pass is responsible for [40].

The Graphics pipeline is the main core of the application which is solely responsible for all operations related to geometry of the 3D model and rendering. It contains a series of stages some of which are modifiable by the developer and others are left static [30, 40]. These stages are classified as either fixed function or programmable. This is shown in the figure below-

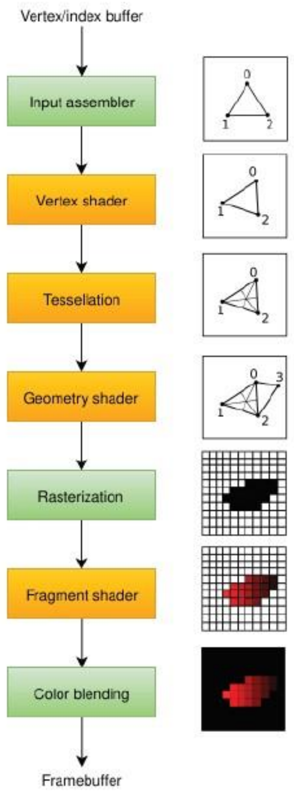


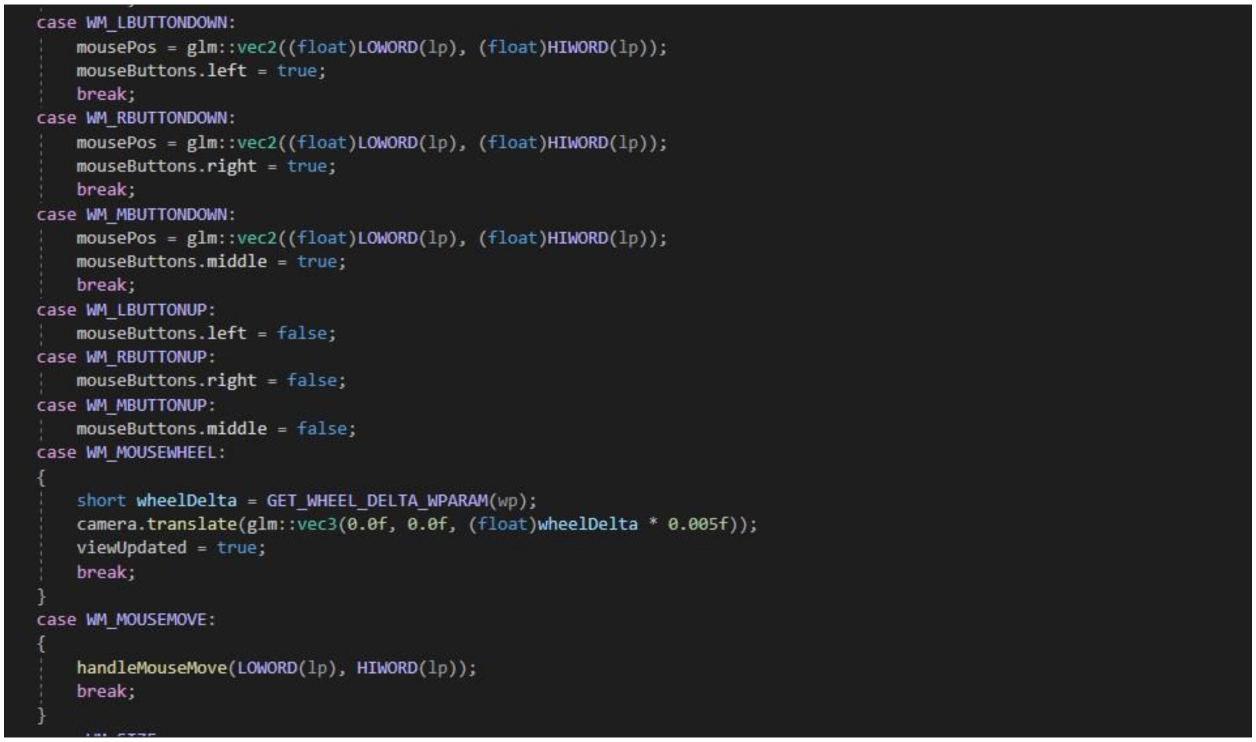
Fig.13. Depiction of the graphics pipeline which represent a total of 7 stages that are required for the execution of the graphics pipeline. The stages in yellow are programmable and can be altered by the developer whereas the stages in green are fixed function and thus cannot be altered. The graphics pipeline is responsible for rendering geometry data to target image frames.

The graphics pipeline is responsible for rendering geometry data to target image frames and is executed entirely on the GPU where the graphics queue selected earlier using the pickPhysicalDevice and createLogicalDevice functions are utilized. Only 1 graphics queue is required to execute the graphics pipeline and this is done via command buffers. The geometry data more specifically refers to the vertices and textures that represent the surface of the model that are then translated to pixels on the screen during rasterization [30,40]. The graphics pipeline is capable of surface rendering where the 3-dimensional surface that constitutes the model can be rendered on the display screen. For more advanced rendering such as volumetric rendering, a different pipeline would have to be utilized which is termed as a compute pipeline which is outside the scope of this thesis [40].

This process of rendering the geometry of the model on the screen can be subdivided into 7 stages starting from the input assembler stage all the way down to the color blending stage. The input assembler stage is responsible for handling the input from the vertex buffer which is a reference to the geometry data of the model that needs to be rendered onto the screen [40]. The geometry data represents raw vertex data that is referenced by the vertex buffer. There is also an index buffer that specifies how each of the vertices are to be connected when rendering the mesh which will later go on to form the framework that generates the surface that is displayed on the screen. The index buffer specifies the formation of primitives where primitives are simple shapes generated by the connection of vertices where here the index buffer assembles vertices in triplets to generate triangle primitives. The manner in which these triangle primitives are assembled to generate the mesh superstructure is also handled by the input assembler where the triangle primitives can be connected in different configurations to generate different meshes. All this functionality is performed by the input assembler [41].

After the input assembler stage is the vertex shader stage which consists of a shader program that executes transformations on every vertex that constitutes the mesh preassembled by the input assembler. The vertex shader is connected very closely to another object termed as the descriptor set where the descriptor set is responsible for specifying the type of transformations that need to be applied to the vertices which is then executed through the vertex shader [40]. As explained earlier in the literature review section of the thesis, the vertex shader is responsible for transforming the position of vertices in the model from model space to screen space where the positions of the vertices are represented as a fraction that lies between -1 and 1 on both the x and y axes. The z-axis is represented with the use of a depth buffer which is utilized to encode depth values that can be utilized by the shader for perspective projection. The values for the z buffer however only range from 0 to 1 and not -1 as that would generate erroneous perspective projection values that can distort the model completely when being rendered on the screen [40]. Each vertex is a container defining a specific set of attributes that defines the properties of the surface where the position of the vertex can be defined as one such attribute. The vertex shader is also responsible for streaming vertex data down the graphics pipeline.

Descriptor sets are objects that serve as a bridge that connect the vertex shader to components within the program that are responsible for implementing transformations. One component would be the uniform buffer where this particular object is important for encoding geometric transformations of the model including real-time transformations. Real-time transformations correspond to transformations invoked by the user [30, 40]. This is done by encoding the mousepad inputs as intended transformations of the 3D model. The transformations utilized for the 3D model are rotation and panning where the user can utilize the mouse to rotate the model around the x and y axis and allows panning across the z axis. These inputs are encoded in the WNDPROC function as shown in the figure below-



```
case WM_LBUTTONDOWN:
    mousePos = glm::vec2((float)LOWORD(lp), (float)HIWORD(lp));
    mouseButtons.left = true;
    break;
case WM_RBUTTONDOWN:
    mousePos = glm::vec2((float)LOWORD(lp), (float)HIWORD(lp));
    mouseButtons.right = true;
    break;
case WM_MBUTTONDOWN:
    mousePos = glm::vec2((float)LOWORD(lp), (float)HIWORD(lp));
    mouseButtons.middle = true;
    break;
case WM_LBUTTONUP:
    mouseButtons.left = false;
case WM_RBUTTONUP:
    mouseButtons.right = false;
case WM_MBUTTONUP:
    mouseButtons.middle = false;
case WM_MOUSEWHEEL:
{
    short wheelDelta = GET_WHEEL_DELTA_WPARAM(wp);
    camera.translate(glm::vec3(0.0f, 0.0f, (float)wheelDelta * 0.005f));
    viewUpdated = true;
    break;
}
case WM_MOUSEMOVE:
{
    handleMouseMove(LOWORD(lp), HIWORD(lp));
    break;
}
```

Fig.14. A depiction of the messages that are generated by the mousepad of the client device where messages that start with the prefix WM_ correspond to the left, right and center keys along with the mouse wheel components of the mousepad. These messages feed into parameters that are monitored by the method handleMouseMove given under the case WM_MOUSE_MOVE which is responsible for assigning the respective transformations based on the key that was pressed on the mousepad. The left key corresponds to rotations while the right key corresponds to zooming/panning transformation of the model.

These transformations are then fed into another data object termed as the camera. The camera represent the Model Projection View matrices discussed in the literature review earlier where the handleMouseMove function is responsible for converting the new positional information delivered by the cursor as it darts across the window into their respective transformations based on what button of the mouse pad was pressed [30, 40]. Only the left and right mouse keys have any functionality. This is fed as a new positional vector into the camera which then converts this vector into its respective transformation. The transformation data is then stored in the respective view and perspective matrices where all the data for the final transformations are stored. This data is then fed into the uniform buffers which represents an object from the descriptor set class. Uniform Buffers allow cross-talk between the application and the vertex shader where the new transformation data is transmitted from the application directly into the vertex shader which is implemented upon every vertex of the model via the graphics queue [30, 40]. This allows the users to see real-time transformations of the 3D model when manipulated with the mouse pad. The uniform buffers are enabled by instantiating descriptor Set objects which are attached to the graphics pipeline upon instantiation. This is what allows the uniform buffers to communicate directly with the vertex shader. The uniform buffers are generally attached at the end of the pipeline. The vertex shader stage is a programmable stage which means that developers can generate indigenous code that can serve the role of the vertex shader. The code is written using

GLSL language which is converted into the bytecode SPIR-V format that can be interpreted by the GPU. This is advantageous as shaders written in the bytecode format for Vulkan can be run on multiple different hard drives with little room for error that may arise because of variable interpretation which is the case for OpenGL platform which depends on drivers that can interpret shader code erroneously [30, 40].

There are additional post-processing steps that can be implemented upon the mesh after the vertex shader stage and this is covered by the tessellation and geometry shaders are utilized for improving mesh quality which in turn improve quality of the surfaces that are rendered on the screen. Tessellation shaders work by subdividing primitives and generating different meshes based on certain properties or rules that are implemented by the developer. Geometry shaders are responsible for generating more primitives when required or reducing them entirely. However, these stages are omitted entirely from this application as they are redundant where the mesh generated in the input assembler and vertex shader stages being sufficient [30, 40].

The rasterization stage comes next where each of the primitives that form the unit cell which compose the mesh is then translated into fragments. Fragments represent unit cells of the image frame as represented by the swap chain framebuffer where each fragment corresponds to the pixel on the display window of the screen. The rasterization stage is responsible for defining the extent or dimensionality of the window (width, height) which is termed as the viewport where any part of the model that falls outside of this viewport after perspective projection is clipped from screen space. This can be determined through the attribute information contained in each vertex mainly to do with position [30, 40]. The rasterization stage utilizes linear interpolation discussed earlier with Gourard shading where the attributes of the vertices such as position and color are interpolated across fragments after perspective projection. The interpolated fragments also take depth information into account in order to ensure that the view of the scene as seen from the perspective of a pin hole camera that is modelled by the camera object described earlier is as accurate and realistic as possible [30, 40].

The next stage is the fragment shader which is invoked for every fragment that successfully clears the rasterization stage of the graphics pipeline where each fragment is then rendered onto the appropriate target framebuffer [30, 40]. Fragments, much like vertices also contain attribute information which was passed down from vertices during interpolation. These attributes encode information such as depth or color to represent the fragment on the framebuffer. The fragment shader is responsible for giving individual pixels their properties on the screen. The color blending stage which is the final stage of the graphics pipeline also has a similar function to the fragment shader where it is responsible for identifying those fragments that map to the same pixel on the framebuffer [30, 40]. The color information of these fragments is ‘blended’ together to generate a new color that would correspond to the target pixel. The degree of color mixing at a particular pixel target can be configured based on the degree of transparency of the framebuffers that the fragments map to. The color blending stage is redundant as the basic requirements of color coding for the model are fulfilled by the fragment shader and thus will not be implemented within this model.

Finally, after the graphics pipeline is configured, command buffer objects are then instantiated which are involved in the final execution of the graphics pipeline and rendering of the 3D model on the designated window. Command Buffers are objects that store a list of commands that need to be executed in order for the application to be successful in rendering the image. Command Buffers are instantiated using the createCommandBuffers function [30,40]. The function is responsible for connecting the graphics pipeline and the render pass before the commands can be executed on the graphics queue. Once the command buffers are created, they are then summoned by the function that ties everything together- the drawFrame method. This method is responsible for rendering the target image frame possessed by the swap chain and delivering the prepared target frame to the corresponding present queue where it will then be displayed on the screen [30, 40]. The drawFrame function is responsible for summoning and initializing the command buffers which is then responsible for rendering the target image frame. The drawFrame function is also responsible for synchronizing the application and GPU cycles which would otherwise result in the program crashing. The function is invoked in the mainLoop for every frame that cycles through the swap chain where the function is responsible for both acquiring raw image frames ready to be targets for rendering and delivering already rendered image frames to the swap chain for presentation on screen. The drawframe function is called every time the screen is refreshed [30, 40].

3.3 glTF Extraction Module-

Now that the rendering framework is set up, the rendering engine is complete and all that is needed is the model data. The model will stream the necessary geometry data as input to the rendering engine that can then display the model on the display screen of the windows client device. This is done through another module termed as the scene extraction module. The scene extraction module is responsible for extracting the necessary data from the model data file and exporting it to a data object such as a vertex buffer that can be used to stream the data to the rendering engine [42].

The scene extraction module can only extract data from those files that are saved in the glTF format. glTF stands for graphics language Transmission Format where this is currently the most compact, ergonomic format for storage and transmission of 3D image data. This format is different from the other formats that existed because of the versatility in which the geometry data of the 3D model can be stored. The format has various containers to store attributes of the vertex data such as textures which can be used to encode material properties of the model surface to be rendered [42]. All the data of the model is stored through the use of a hierarchical tree structure composed of individual units termed as nodes where each node refers to a point within the tree. A node is used to reference data attributes of the model and can be utilized as a reference to succeeding nodes where the succeeding nodes here are termed as child nodes and the node they descended from further upstream is termed as the parent node. All the nodes lead up to the scene which serves as the progenitor node of the network [42]. The glTF format is also special because it is the only format that is supported for streaming of 3D geometry data across the internet. The glTF format is supported by web browsers which were recently updated to be capable of rendering 3D models for display. Exporting the model scene data as a glTF file thus gives users the freedom to display their 3D models across a range of interfaces whether it be through online web browsers or offline with the custom built vulkan rendering engine. The reason why web browsers can stream glTF file data is because glTF files are very compact with all the model data being stored in a binary format that is accessible through the JSON data interchange format. The glTF file itself is thus very compact where storing the data in a binary file reduces overhead and removes the need for data interchange when imported by different graphics APIs [42]. A schematic of the glTF diagram is given here below-

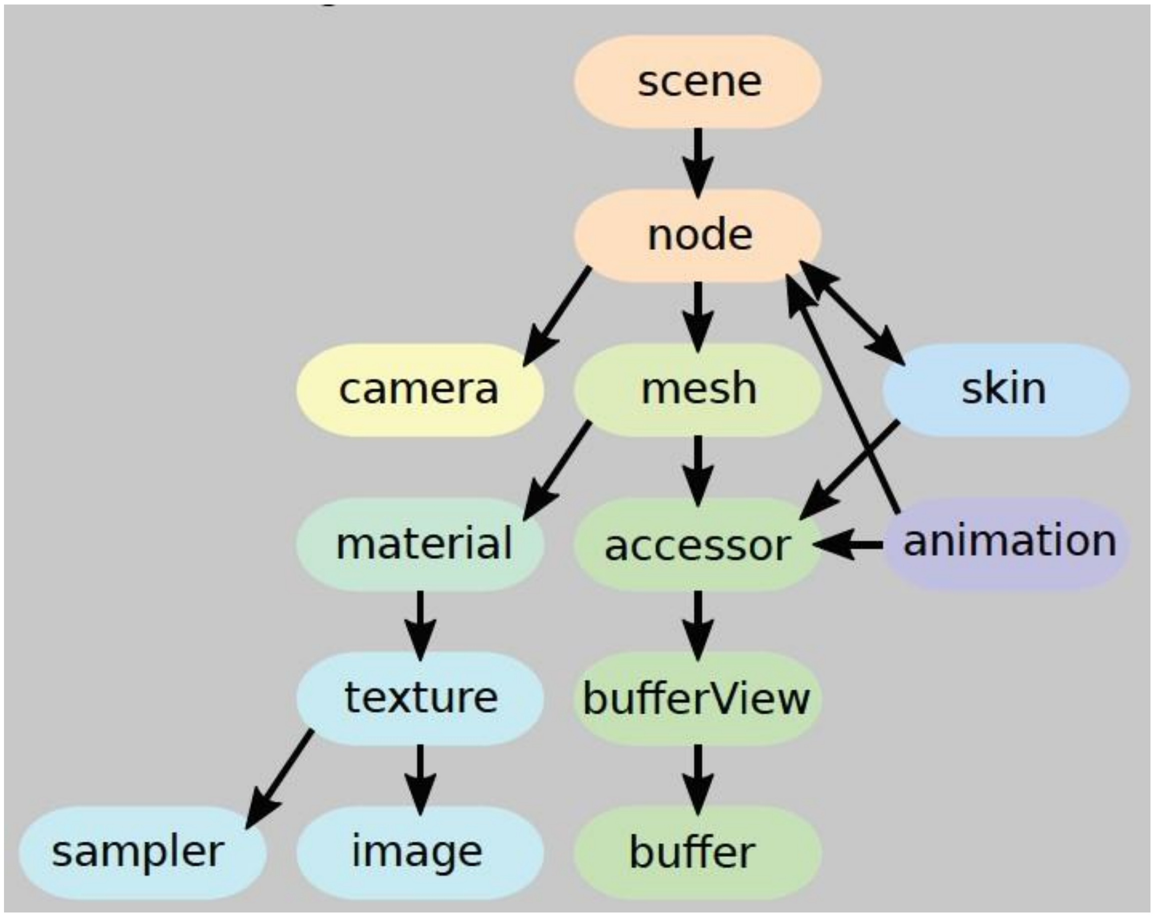


Fig.15. A schematic of the glTF format showing the organization of various components resembling the motif of a hierarchical tree structure. The most upstream component of this tree would be the scene followed by the node where the scene can be composed of several nodes. Each node has various components that it can make a reference to ranging from the buffer carrying the vertex data required to generate the object within the scene to axillary components such as the camera for viewing the scene and texture for encoding material properties of the particular object that the node references [42].

For the vulkan rendering engine, only the vertex data is uploaded from the glTF file into the rendering window for the current needs of the application [42]. The vertex data is accessible from the buffer component of the node where all the model data is stored in one node representing both the object and scene to be rendered on screen. The bufferView and accessor is utilized to access the data from the buffer where the bufferView is a wrapper that structures the data in the buffer and the accessor is utilized as a reference to access the buffer through the bufferView [42]. The scene extraction module handles all this functionality. The exact details on how this module works can be found in the appendix.

4. Results and Discussion-

Work on the project first began with the development of the rendering engine where the rendering engine was developed using the Vulkan API. Here, it takes approximately 1000 lines of code to develop a fully functional core of the rendering engine that is capable of rendering simple geometric data such as a triangle. The triangle is utilized to test the functionality of basic features which will be important as it will form the backbone of renditions of more complex 3D models. The existence of the triangle itself is utilized to validate that the fundamental components within the rendering application is working such as the render pass and the swap chain. The triangle is also utilized to test whether the application is synchronized properly because the clock rate of the CPU and GPU are most likely different leading to the propagation of delays through the chain of command that exists within the communication framework between the CPU and GPU. If synchronization between the CPU and GPU is not maintained, it will lead to the abrupt shutdown of the application. The CPU is responsible for executing the framework of C++ code that comprises the entirety of the application whereas the GPU is responsible for executing the shader code that is fed into the graphics pipeline for each vertex stored within the vertex buffer that is held by the command buffer. The synchronization of the transition of commands from the commands that lie within the command buffer that is allocated to the CPU to those commands executed by the shader on the GPU which are organized through the use of objects termed as fences. Fences are objects that are responsible for synchronizing the interchange of information between the CPU and GPU where if the clock rate of the CPU is too high, the GPU is unable to execute a single frame in that span of time leading to a buildup of frames that have not been rendered to yet. This causes the work to load up eventually stalling the application and causing it to crash. The reverse also leads to a system error where the GPU writes to frames that are not prepared and ready to be written to and this leads to the application crashing instantly. Applications that are not synchronized properly will start to exhibit some type of faulty behavior shortly within the instantiation of the model data on the screen.

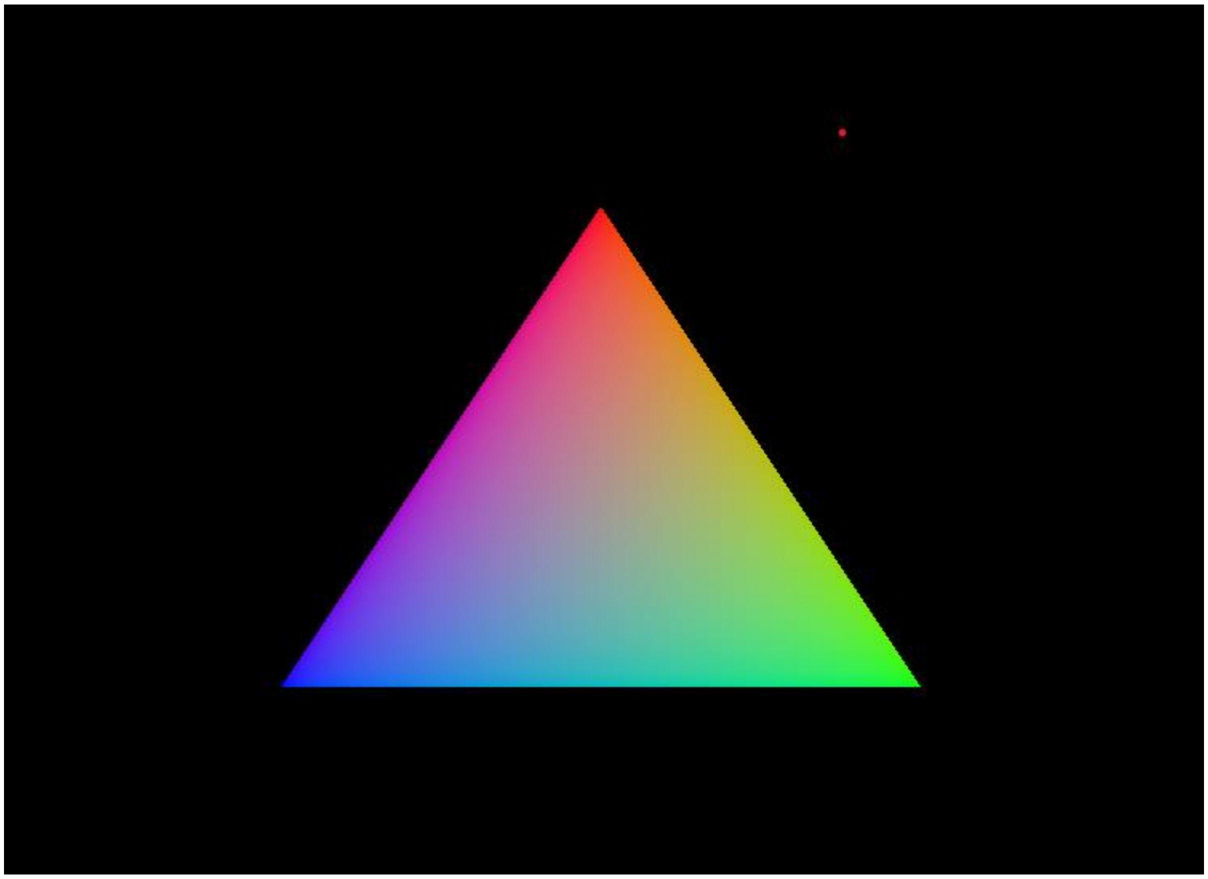


Fig.16. A representation of the rendering of a simple triangle by the Vulkan rendering engine module where this indicates that the fundamental framework has no bugs within it.

Once the triangle has been successfully rendered and no obvious faulty behavior is to be observed. Other features of the graphics pipeline can be defined where mostly in this case, it has to do with color blending where vertices can be defined with additional data attributes that can be utilized to add additional complexity in the manner in which the model is depicted on screen. In this case, color information is given to the 3 vertices of the triangle where each vertex represents either of the 3 primary colors (red, blue or green). The area that is bounded by the 3 vertices is then color coded via linear interpolation of the distance each point within the triangle is positioned at with respect to the vertices of the triangle. This leads to the blend of colors that can be viewed in figure 16 above. This is performed automatically by the fragment shader that is fed into the graphics pipeline of the application however this can be altered manually as well. The color coding scheme is also important potentially for volume rendering application where each vertex point that is bounded within the geometry dataset may be given an additional alpha value that accompanies the triumvirate of primary colors used to define the color scheme of the vertex. The alpha value is important as it is the parameter responsible for defining depth values of the vertices and thus is utilized accordingly to assign transparency values to points that overlap on a surface of a given depth [40]. However, only surface rendering features are employed in this application and thus depth stencil information will be utilized to the minimal possible effect.

Once the Vulkan framework is established and all the essential components are deemed to be functional within the framework, additional features were then incorporated into the application such as uniform buffers. Uniform buffers as explained earlier in the methods section is important for allowing the user to manipulate the model on the screen using keyboard and mouse controls. Here, figure 17 shows the uniform buffer being implemented that allow rotational transformations to be implemented upon the model. This is done using the mouse control.

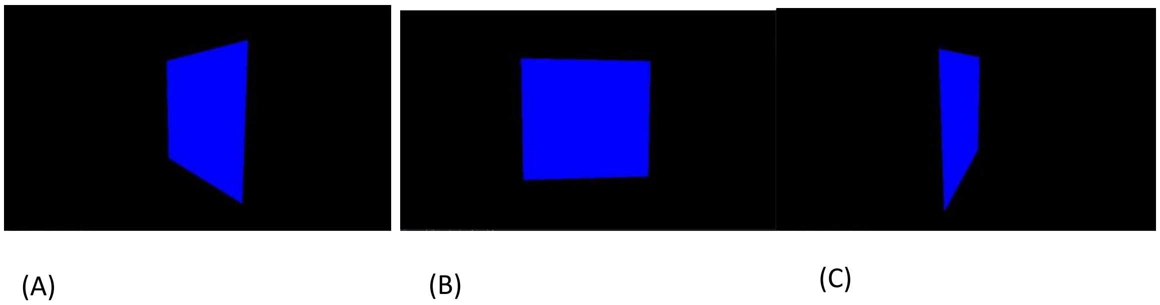


Fig.17. A representation of the rotation of a model which in this case, is a quadrilateral, 180˚ across the y axis which is oriented vertically with respect to the screen. This is done by implementing a rotational transformation on the model instantiated by clicking on the mousepad.

The transitions are found to be smooth with no lag appearing in the model as it is transformed on the screen. This further confirms that the CPU and GPU components of the application are well synchronized to at the very least support the transformation of a simple quadrilateral on the screen and thus the fences are working as required. Transformations as discussed earlier in the literature review are stated to be linearly independent of the vector space it is applied to which in this case is the position space. What this implies is that any transformation that is applied to a single vertex is equally applicable to all vertices within the geometry that constitutes the model. This is due to the linear independence of the transformation from the position space that the geometry data resides in and is crucial especially for models that have dynamically changing geometry data. Due to this property of linear independence, only a single Model-View-Projection Matrix is required to be inputted into the vertex shader where this matrix that is responsible for the transformation is equally applicable to all points within the model that is rendered [21, 43]. The design of the user command interface for the program was inspired by the programs developed by Sascha Willem [43].

Once this was confirmed, it was time to incorporate gltf reader functionality where now the vertex buffer of the rendering engine would be populated with geometry data of a sample gltf model file. Here, the model Suzanne will be utilized where Suzanne represents the face of a chimpanzee. The acquisition of this model can be attributed to the open-source model data made publicly available by Sascha Willem [43]. The code here is incorporated in a separate file termed as gltfModel.cpp which is called directly as a header file when instantiating the rendering engine. As mentioned earlier, the gltf file format is one of the most versatile file formats that exist when it comes to storing and transmitting 3D geometry data [43]. The data is packaged and organized into definite categories through the use of the human readable JSON data interchange format which can be utilized to sample the necessary data required from the file to render the 3D model. The manner upon which this extraction occurs is outside the scope of this thesis and so will not be covered here but the essential methodology incorporates the use of the JSON format to query for the required data attributes and populate these data attributes into the corresponding data buffers which can then be accessed during write time. The process of connecting the gltf reader to the rendering engine was time-consuming and resulted in several rendering errors where some of which is displayed in figure 18.

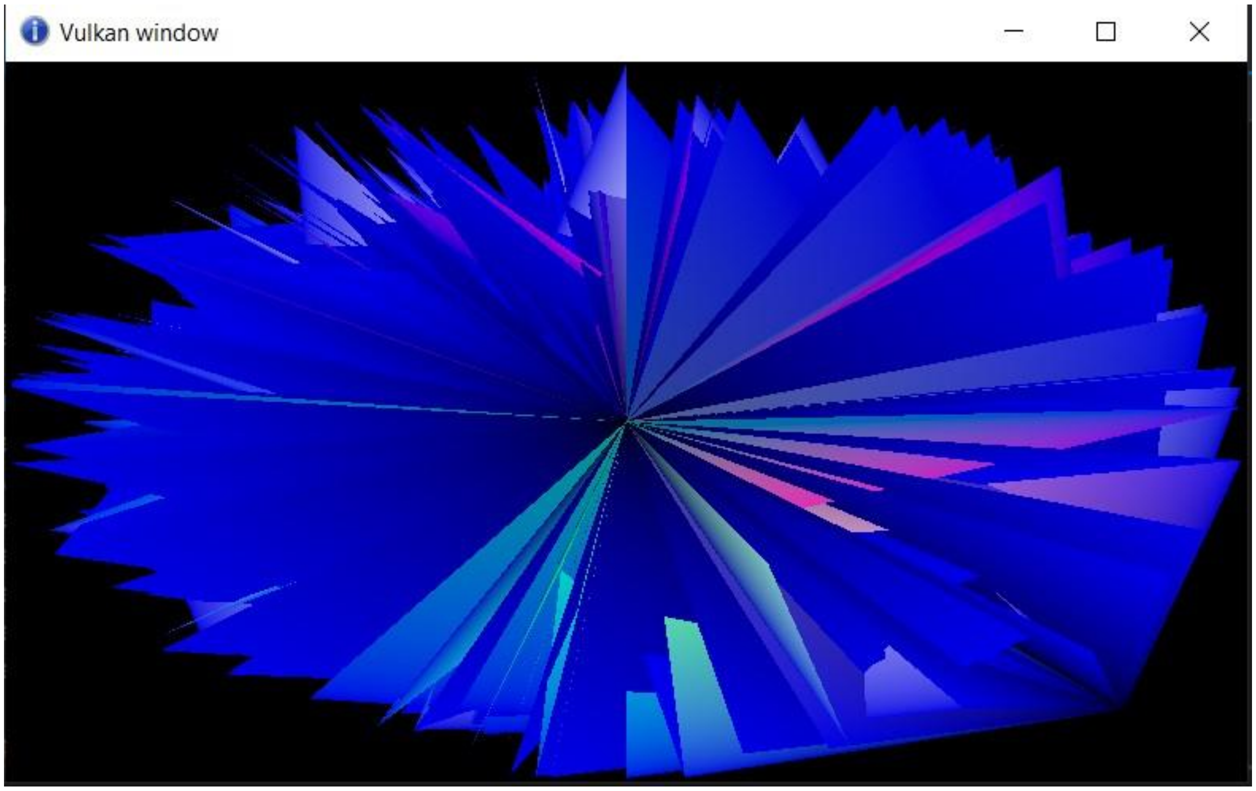


Fig.18. A depiction of one of the rendering errors that occurred upon instantiating the data pipeline utilized by the rendering engine to render the model “Suzanne” stored in the gltf file format.

This error occurs due to the misconfiguration of vertex data in the graphics pipeline where the bindings of the data attributes that correspond to the vertices do not properly align with the vertices that are outputted by the vertex buffer into the shader programs that the graphics pipeline is connected to which run on the designated GPU device. The misconfiguration of the vertex attribute and bindings leads to the distortion of the geometry data when rendered on the screen. Once this error was identified, it was easy to rectify but also gave a deeper understanding of the purpose of the vertex input attribute of the graphics pipeline where misconfiguration of vertex data attributes can lead to the splintering of the model geometry that is depicted on the screen.

Figure 18 represents how the model should appear as when the vertex input attribute of the graphics pipeline is completely rectified. The sharp contrast between the renditions exhibited by figures 17 and 18 depict the intricacy of the entire application where a slight misconfiguration of one of the components can lead to a whole gamut of errors ranging from system failure to aberrant renditions of the model in question. The error displayed in figure 17 is one of several errors that was encountered during the process of successfully building the rendering engine where figure 17 represents the more prominent of the semantic errors encountered during the construction of this application where useful insight could be gleamed as to the inner workings of the rendering engine.

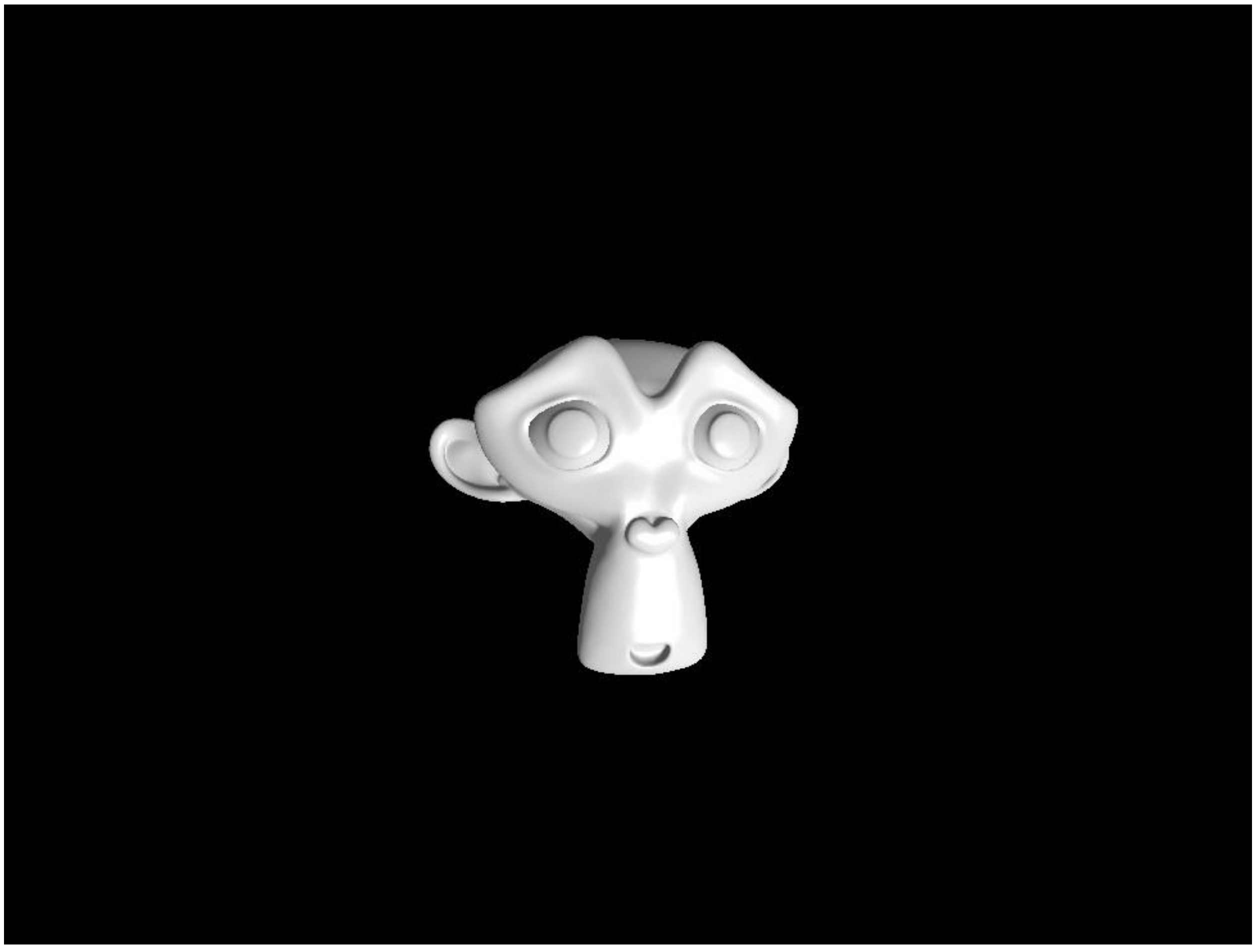


Fig.19. A depiction of the Suzanne model using the vulkan rendering engine. The model was extracted from a gltf file and rendered on screen.

Once the test model, Suzanne, is confirmed to be successfully rendered, the rendering engine is now complete where basic rendering functionality is supported. Additional features such as ray casting, volume rendering or global illumination modelling can be easily integrated into the model which is the advantage that is offered by building a rendering engine upon the Vulkan API where next generation rendering techniques that is being developed by the graphics and gaming community can be easily customized and incorporated within this rendering engine where Vulkan now serves as the gold standard for the development of state of the art graphics applications [44]. The reason why the application should be more readily customizable for algorithms developed by the gaming industry is because this industry is lucrative and thus a lot of research and development is funded to generate only the most cutting-edge algorithms for gaming applications to stay competitive in the market and the purpose of this application is to serve as the bedrock that can now allow these technologies to be translated to the clinical environment. Unfortunately, due to the limitations in time and resources, only the most bare-bones framework of the rendering engine could be developed but with the potential to incorporate more advanced visualization algorithms in the near future.

After the rendering engine is complete, the volume module was then constructed where the module in contrast to the rendering engine is relatively straightforward in code and execution with the reason being the incorporation of the itk and vtk libraries developed by Kitware Inc which are industry standard with itk libraries being designed specifically for manipulating image data derived from human anatomy [45].

The main difficulties in constructing this module lied within identifying the correct UID that represents the required series data for constructing the model. Once this was identified, the volume file generated was then passed down through vtk filters to filter out the required geometry data from the original file and generate a smoothened surface model of the target structures to be visualized. Threshold filtering is utilized to segment the target structures. In this case, the skeletal system served as the target and the corresponding threshold value that was inputted into the system to generate a 3D model of this target is 500 where these threshold values correspond to the attenuation coefficient for each vertex that lies within the geometry dataset. The processed dataset is then stored in 3 different file formats- (.vtk), (.gltf) and (.stl). Not much difficulty was encountered in building this section of the module with the vtk library being straightforward. The fidelity of the volume module was checked by examining the outputted stereolithography (stl) file where the windows operating system already comes with built-in stl viewing software that was utilized to observe the models.

After the volume module has been tested and verified to be fully operational, the rendering engine is then connected to the volume module. Here, the rendering engine is then put to the test to check whether it is capable of rendering models derived from DICOM datasets which contain significantly more complex geometry data and thus lead to larger file sizes than those that store the typical gltf model such as “Suzanne”. These tests led to the discovery of a few rendering errors that had to do with the calibration of the camera object within the scene being rendered. This is represented by the figure 20-

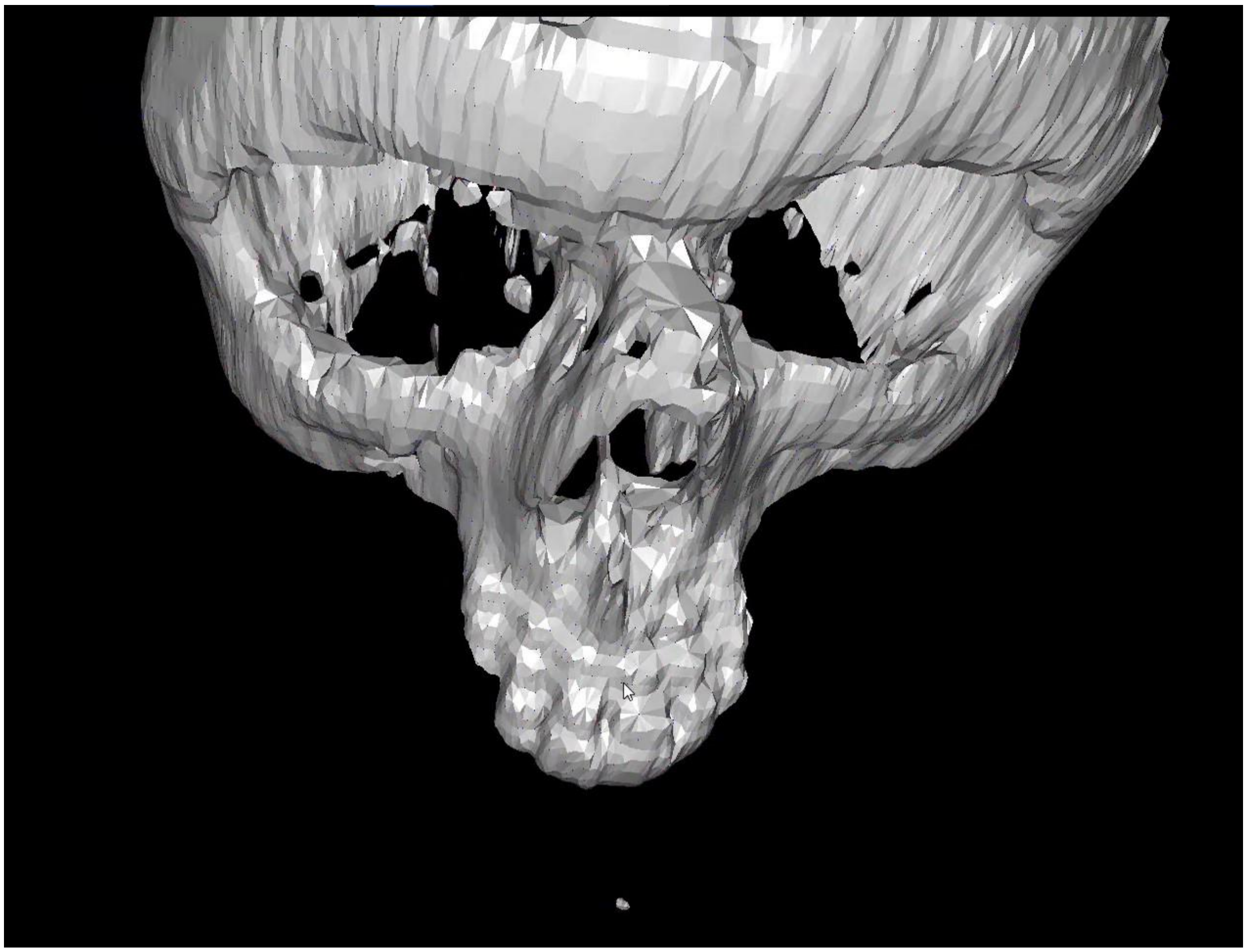
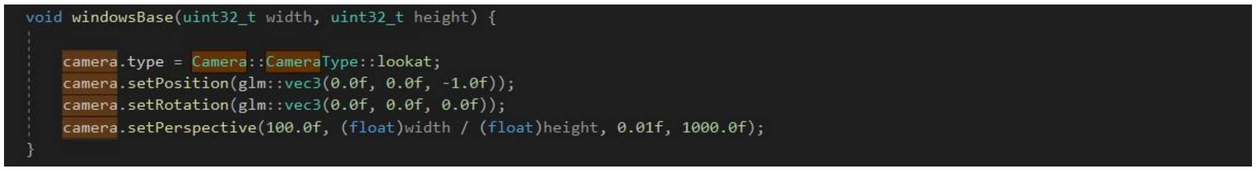


Fig.20. Rendering of the 3D model obtained from the target DICOM dataset by the Vulkan Rendering Engine. Only a portion of the model is rendered on screen with the rest being clipped off the screen.

Figure 20 represents only a part of the entire model that is to be rendered on screen with the reason being that the model derived from DICOM data is much larger in size and complexity than the previous model Suzanne. As a result, a major portion of the model is clipped off in screen space and panning away from the model does not lead to greater model exposure but rather causes the entire model to be clipped off screen space. This error arises as a result of the camera object being improperly calibrated specifically with the adjustment of the near and far planes which serve to bound how much data can be depicted on the screen. Enlarging the far plane leads to a greater allocation of the screen space for the model. Changing the near and far planes of the camera is regulated by the setPerspective method of the camera object which comes directly under the responsibility of the windowsBase function. This is depicted in figure 21.



```
void windowsBase(uint32_t width, uint32_t height) {

    camera.type = Camera::CameraType::lookat;
    camera.setPosition(glm::vec3(0.0f, 0.0f, -1.0f));
    camera.setRotation(glm::vec3(0.0f, 0.0f, 0.0f));
    camera.setPerspective(100.0f, (float)width / (float)height, 0.01f, 1000.0f);
}
```

Fig.21. A representation of the windowsBase function that is responsible for allocating the initial parameters of the model upon instantiation on the screen. These parameters are the initial position, orientation and perspective of the model as viewed by a virtual pin-hole camera. The perspective determines the fraction of the model that can be observed on the screen and this is modulated by the last 2 parameters of the setPerspective method which correspond to the near and far planes of the camera object respectively.

Once this rectification was implemented, the entire model can now be represented on screen as given by figure 22.

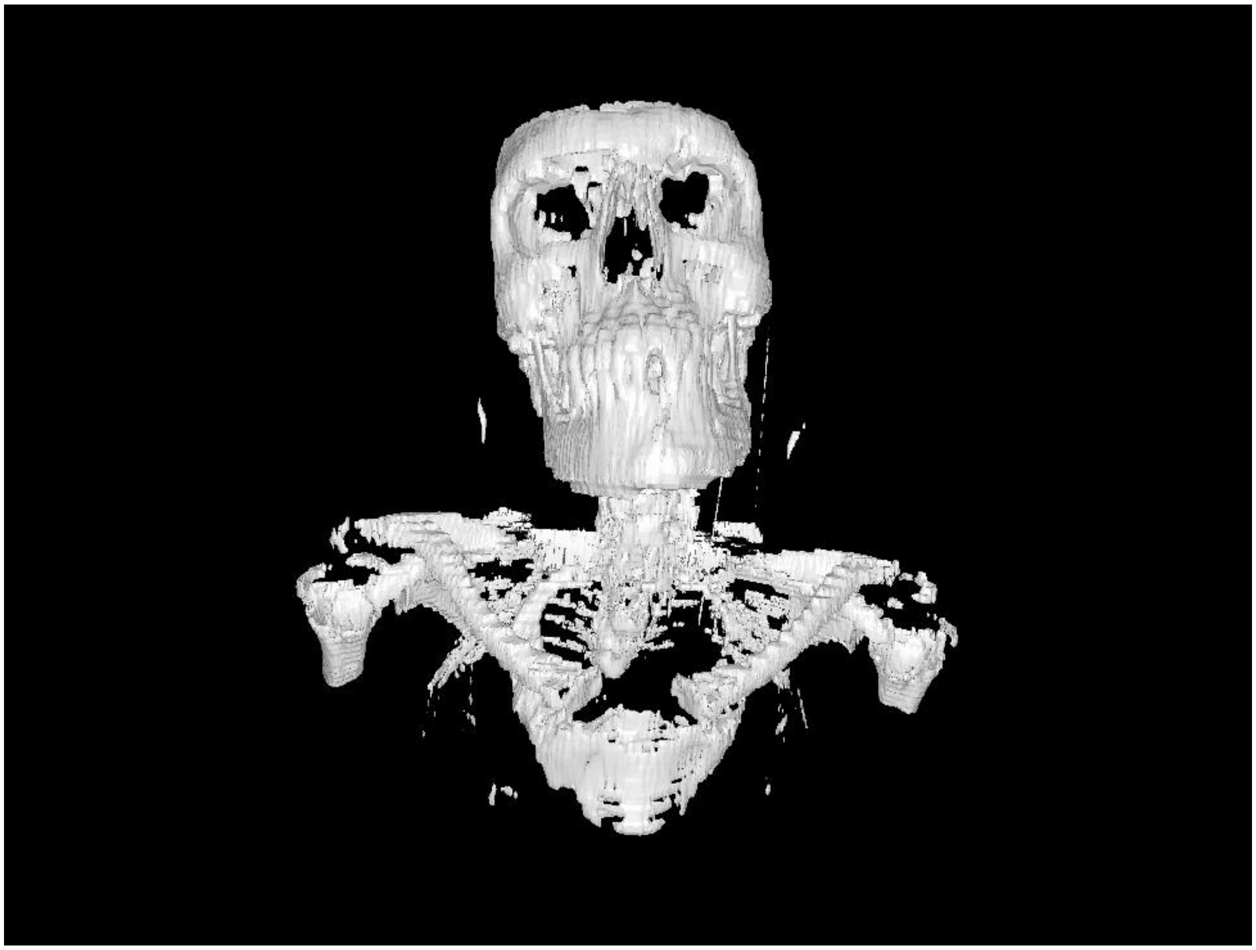


Fig.22. A fully rendered model of the skeleton extracted from the DICOM dataset as depicted utilizing the Vulkan rendering engine.

The model generated in figure 22 represents the complete model as depicted on the Vulkan rendering engine. The engine supports rotations and panning of the model in screen space where the application is able to run smoothly as the model is manipulated with the mouse pad. This indicates that the engine is capable of rendering models generated via DICOM datasets with no glitches or delays on regular laptop devices that support a NVIDIA GeForce GTX GPU.

To see how the application fares against other DICOM viewers, other open-source applications were investigated. Open source applications were selected as our current application is also open source with proprietary software being excluded from the comparison. One study found that the two most competitive DICOM viewer solutions on the market that were also open-source were Slicer 3D and Osirix. Osirix was built specifically for the Apple MacBook and Slicer 3D was built to run on the windows operating system [24]. For this reason, only Slicer 3D was used as the benchmark to compare the performance of the custom built Vulkan application.

When the two DICOM viewing solutions are compared, Slicer 3D obviously is the better option with regards to the totality of features that are offered by the program where the Slicer 3D application maintains an arsenal of image segmentation techniques that can be applied to the model in question giving the user flexibility in the choices that are employed for processing the geometry data. It also supports multi-slice views allowing simultaneous display of 2D cross-sections of the model from the coronal, sagittal and transverse plane of view. These cross-sections are the raw grey-scale CT images where the DICOM image set stores a set of 2D image cross-sectional slices for each of the three planes of view. These images are the targets of the segmentation algorithms whose output is then reflected on the 3D model after segmentation is complete. Slicer 3D not only supports segmentation but also registration of anatomical landmarks on the dataset.

However, when it comes to rendering the 3D model, the custom designed Vulkan application do show a few advantages over what Slicer 3D has to offer. The core rendering engine of Slicer 3D is built upon the OpenGL framework which is inferior to the Vulkan framework with some of the reasons being that the OpenGL framework depends on drivers to manage low-level memory allocations that can become expensive for high-performance graphics applications that run on higher clock frequencies on the respective CPU and GPU devices. This is supported by one study that compared renderings between graphics engines built on these respective 2 APIs. Here, the study reported that Vulkan was able to save considerable power without performance degradation and OpenGL is found not to be able to compete with Vulkan when power is not an underlying factor in the implementation of renderings [23]. This is due to Vulkan-applications being driver less in nature with the required low-level allocations already being configured in the source code. The superiority of the Vulkan engine can be seen when comparing the ease with which models can be manipulated between the 2 applications. The transition between transformations implemented with the mouse are much smoother for the Vulkan game engine as compared to the Slicer3D one where there is no discernible lag time apparent when manipulating the model on the Vulkan game engine. This is not the case for the Slicer 3D model where the model lags considerably when manipulated on the screen. There is also the appearance of glitches where the model loses its consistency and appears noisy between transitions. This is not the case for the model manipulated on the Vulkan application. However, the loading time is significantly lower for the Slicer 3D model as compared to the Vulkan application where the Vulkan application takes about 2 minutes for the DICOM model to load whereas the Slicer 3D model is instantiated within a second. The model generated on Slicer 3D has a smoother overall texture with better segmentation algorithms being utilized to generate the model surface in comparison to the Vulkan model where the model generated on the Vulkan application is defined by a much coarser and noisier surface where the surface is populated by polygons which are deemed as noise and can be filtered before the final rasterization of the model surface on the screen.

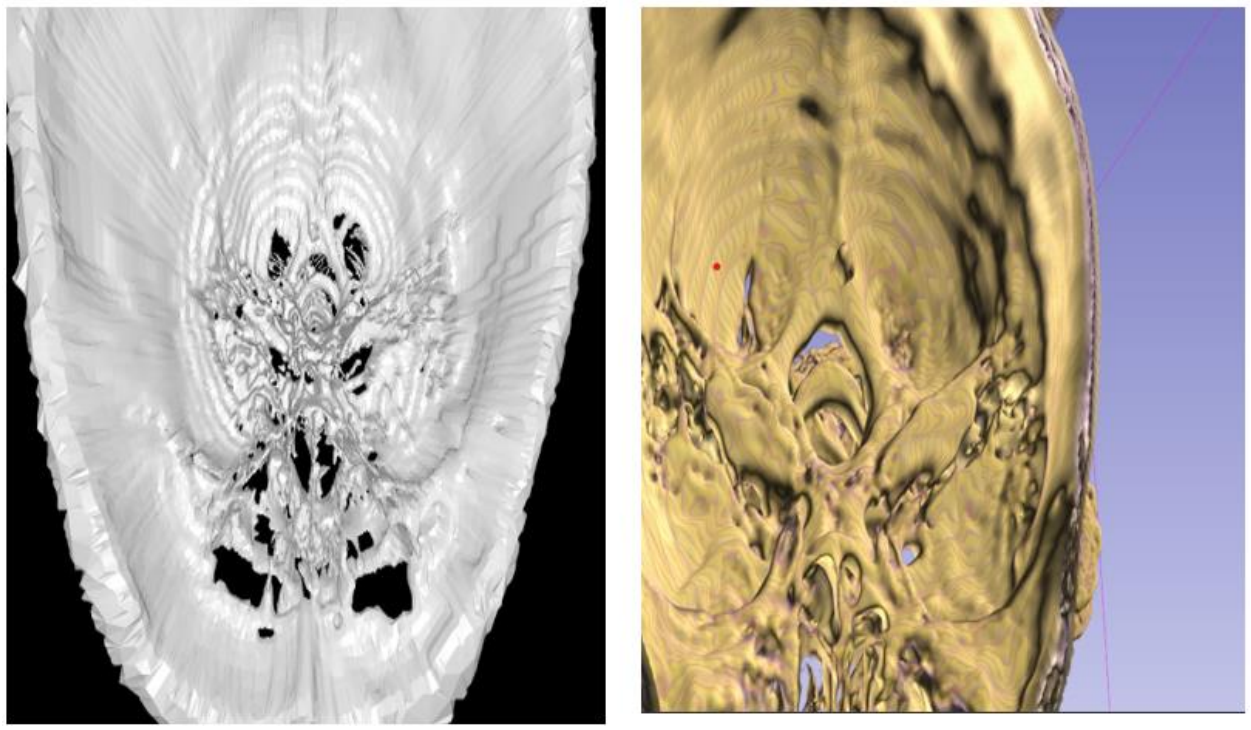


Fig.23. A representation of the same model across 2 different rendering engines where the one on the left belongs to the Vulkan rendering engine and the one on the right belongs to Slicer 3D.

It can be seen from figure 23 above that the 2 models appear very different based on the rendering approaches selected. The model here represents the interior of the skull where the model rendered with Slicer 3D represents much smoother surfaces where anatomical landmarks can be more easily distinguished on the slicer 3D model not to mention the surface has less noise. However, depth information is left more unclear in the Slicer 3D model as compared to the Vulkan model where structural depth is clearly more tangible here than it is in the Slicer 3D model not to mention more condensed detail being depicted by the Vulkan model. The Slicer 3D model also contains a certain degree of noise especially in the contours lines that traverse the inner folds of the skull where these contour lines make it increasingly difficult to determine intricate details that are ensconced within the model.

The Vulkan application also automatically stores the processed model in 3 different file formats. Something that would require additional effort on the part of the user with the Slicer 3D application. One of the formats is the gltf format which is steadily rising in popularity due to its ergonomic design and light weight. This format also gives the user the advantage to view it from a web browser if the need arises and allows transmission of 3D content across the internet. The gltf format is incredibly versatile as compared to its competitors, vtk and stl where a wide variety of attributes can be stored in the file that can be used to encode texture information, lighting and even animations. This can be used for a wide variety of applications where not only 3D but 4D clinical data can be stored in this format [42]. This is especially useful if modules incorporating artificial intelligence were to be incorporated wherein this format would prove useful in storing different types of segmentation data or even dynamically changing data as is the case with neural networks. The gltf format is currently not supported by the Slicer3D application as it is relatively new as groups are still in the process of implementing it as a de facto standard. The Vulkan application is thus cutting edge in its ability to deliver such capabilities.

The applications were implemented upon the NVIDIA GEFORCE GTX 1050 GPU hardware. GTX 1050 GPU devices operate upon commercial off the shelf computer with the Computer this application is being run on being the Lenovo Ideapad 330S [48]. The GTX 1050 GPU device is built using 14 nanometre Complementary Metal Oxide Semiconductor technology. The details of the GPU device is given in the table below-

Table 1- A tabulation of the specifications that the GEFORCE GTX 1050 GPU device has comprised. This device is the one upon which the Vulkan application is executed upon [48].

GEFORCE GTX 1050	
Manufacturer	NVIDIA
Die Size	132 mm^2
Technology	14 nm CMOS
GPU Clock Speed	1418 MHz
Texture Fill rate	56.7 GTexel/s
Transistors	330 Million
Memory Type	Graphics Double Data Rate 5 (GDDR5)
Pixel Fillrate	45.4 GigaPixels per second
Memory Size	2048 MB
Memory	1752 MHz

The memory type GDDR5 is specifically configured for supporting rendering graphics and supporting high-performance [48]. The pixel fill rate refers to the rate at which pixels can be refreshed and rewritten to per second. The transistors are responsible for storing and writing all the necessary operations required for rendering a frame onto the screen [48].

When the model acquired from the Suzanne gltf file is loaded, the frame rate of rendering is found to be 29 frames per second which does not change as the model is manipulated across the screen. The Suzanne model consists of 7958 vertices and thus the loading time is relatively low being only 9 seconds.

DICOM models however consist of more complex data such as the skeleton that was extracted from the gltf file where this model consists of 1,638,492 vertices. As a result, the loading time is significantly longer with it being 104 seconds before the model is rendered on screen. The frame rate after the model is loaded however, is consistent with the Suzanne Model where the model motion is smooth with no jarring artifacts to be observed. This indicates that the application is fully capable of handling such large datasets provided sufficient time is allow for the model to be fully processed and loaded onto the graphics pipeline from the gltf data file.

Table 2- An evaluation of graphics performance through the use of the parameters for loading time and frame rate for the vulkan rendering application. This is done for 2 different gltf files whose data is represented by the number of vertices that encompass the model.

File (gltf format)	Number of vertices	Frame rate (frames per second)	Loading Time (seconds)
Bone	1,638,492	29 frames per second	104
Suzanne	7958	29 frames per second	9

The Vulkan application has a wide scope for growth where the application developed for this project only showcases the fundamental capabilities of an engine built upon the Vulkan framework. SPIR-V allows for the incorporation of different types of shaders such as compute shaders which are responsible for executing custom built algorithms on the GPU. With this, segmentation algorithms can be incorporated into the application and this can potentially be used to connect the rendering engine to machine learning workflows that can be utilized for the diagnosis of chronic illnesses such as cancer [46]. There are already studies showing the use of Vulkan for fluid dynamics simulations which require the use of compute shaders and this could potentially be translated to the clinical environment where an example could be using 3D imaging data for surgical simulations [47]. The use of segmentation algorithms here would be more effective as it is implemented directly by the API as opposed to programs like Slicer 3D where there are several layers of architecture that separate the rendering component of the program from the image segmentation side which can lead to slower performance.

Conclusions

1. The open source DICOM viewers that are available offer limited support in 3D rendering and of those that do, the technology is slowly becoming outdated. The advent of Vulkan signaled an important milestone in the realm of high-performance graphics applications where 3D rendering applications can be built for higher performance and support more complex rendering applications especially those that support machine learning which also require algorithms that are designed to operate upon GPU devices.

2. A rendering engine based on Vulcan was thus designed to introduce features like instant 3D viewing and multiplatform support for the clinical environment where it can be used for handling 3D medical imaging data which is stored in the DICOM format.

3. A performance benchmarks were made with models running on Vulkan. Vulkan model seems to display better performance to the Slicer 3D model especially if the model is rendered on ordinary client devices such as a laptop. This along with the export of 3D imaging data in modern, ergonomic file formats such as the gltf format makes the Vulkan DICOM viewer useful for professionals in the medical workspace.

References-

1. Rosset A, Spadola L, Ratib O. OsiriX: an open-source software for navigating in multidimensional DICOM images. Journal of digital imaging. 2004 Sep 1;17(3):205-16.
2. Haak D, Page CE, Deserno TM. A survey of DICOM viewer software to integrate clinical research and medical imaging. Journal of digital imaging. 2016 Apr;29(2):206-15.
3. Pratx G, Xing L. GPU computing in medical physics: A review. Medical physics. 2011 May;38(5):2685-97.
4. Barrett JF, Keat N. Artifacts in CT: Recognition and avoidance. Vol. 24. Radiographics. 2004.
5. Feldkamp LA, Davis LC, Kress JW. Practical cone-beam algorithm. Vol. 1, J. Opt. Soc. Am. A. 1984.
6. Maier A, Steidl S, Christlein V, Hornegger J, editors. Medical imaging systems: An introductory guide.
7. Assessment of pulmonary vasculature volume with automated threshold-based 3D quantitative CT volumetry: In vitro and in vivo validation
8. Tunaci A, Yekeler E. Multidetector row CT of the kidneys. European journal of radiology. 2004 Oct 1;52(1):56-66.
9. Moschetta M, Scardapane A, Telegrafo M, Lucarelli NM, Lorusso V, Angelelli G, Ianora AA. Prognostic value of Tissue Transition Projection 3D transparent wall CT reconstructions in bowel ischemia. International Journal of Surgery. 2016 Oct 1;34:137-41.
10. Kucybała I, Tabor Z, Ciuk S, Chrzan R, Urbanik A, Wojciechowski W. A fast graph-based algorithm for automated segmentation of subcutaneous and visceral adipose tissue in 3D abdominal computed tomography images. Biocybernetics and Biomedical Engineering. 2020 Mar 29.
11. Lorensen WE, Cline HE. Marching cubes: A high resolution 3D surface construction algorithm. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987. Association for Computing Machinery, Inc; 1987. p. 163–9.
12. Khan U, Yasin AU, Abid M, Awan IS, Khan SA. A Methodological Review of 3D Reconstruction Techniques in Tomographic Imaging. Vol. 42, Journal of Medical Systems. Springer New York LLC; 2018.
13. Gouraud H. Continuous shading of curved surfaces. IEEE transactions on computers. 1971 Jun;100(6):623-9.
14. Rege A. An introduction to modern GPU architecture (Nvidia Talk).
15. Jew T. Embedded microcontroller memories: Application memory usage. In2015 IEEE International Memory Workshop (IMW) 2015 May 17 (pp. 1-4). IEEE.
16. Melear C. Integrated memory elements on microcontroller devices. InProceedings of WESCON'94 1994 Sep 27 (pp. 507-514). IEEE.
17. Moya V, Gonzalez C, Roca J, Fernandez A, Espasa R. Shader performance analysis on a modern GPU architecture. In38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05) 2005 Nov 12 (pp. 10-pp). IEEE.

18. Heckbert PS. Survey of texture mapping. IEEE computer graphics and applications. 1986 Nov;6(11):56-67.
19. Binotto AP, Comba JL, Freitas CM. Real-time volume rendering of time-varying data using a fragment-shader compression approach. InIEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. 2003 Oct 20 (pp. 69-75). IEEE.
20. Kainz B, Grabner M, Bornik A, Hauswiesner S, Muehl J, Schmalstieg D. Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs. ACM Transactions on Graphics (TOG). 2009 Dec 1;28(5):1-9.
21. Lengyel E. Mathematics for 3D game programming and computer graphics. Charles River Media, Inc.; 2003 Nov 1.
22. Ruijters D, ter Haar Romeny BM, Suetens P. Efficient GPU-based texture interpolation using uniform B-splines. Journal of Graphics Tools. 2008 Jan 1;13(4):61-9.
23. Lujan M, Baum M, Chen D, Zong Z. Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server. In2019 International Conference on Computing, Networking and Communications (ICNC) 2019 Feb 18 (pp. 777-781). IEEE.
24. Haak D, Page CE, Deserno TM. A survey of DICOM viewer software to integrate clinical research and medical imaging. Journal of digital imaging. 2016 Apr;29(2):206-15.
25. Schmid B, Schindelin J, Cardona A, Longair M, Heisenberg M. A high-level 3D visualization API for Java and ImageJ. BMC bioinformatics. 2010 Dec;11(1):1-7.
26. Ratib O, Rosset A. Open-source software in medical imaging: development of OsiriX. International Journal of Computer Assisted Radiology and Surgery. 2006 Dec;1(4):187-96.
27. Eid M, De Cecco CN, Nance Jr JW, Caruso D, Albrecht MH, Spandorfer AJ, De Santis D, Varga-Szemes A, Schoepf UJ. Cinematic rendering in CT: a novel, lifelike 3D visualization technique. American Journal of Roentgenology. 2017 Aug;209(2):370-9.
28. Yoo TS, Ackerman MJ, Lorensen WE, Schroeder W, Chalana V, Aylward S, Metaxas D, Whitaker R. Engineering and algorithm design for an image processing API: a technical report on ITK-the insight toolkit. Studies in health technology and informatics. 2002 Jan 1:586-92.
29. Schroeder WJ, Avila LS, Hoffman W. Visualizing with VTK: a tutorial. IEEE Computer graphics and applications. 2000 Sep;20(5):20-7.
30. Sellers G, Kessenich J. Vulkan programming guide: The official guide to learning vulkan. Addison-Wesley Professional; 2016 Nov 7.
31. Blinn JF. Models of light reflection for computer synthesized pictures. InProceedings of the 4th annual conference on Computer graphics and interactive techniques 1977 Jul 20 (pp. 192-198).
32. Lee GH, Choi PH, Nam JH, Han HS, Lee SH, Kwon SC. A Study on the Performance Comparison of 3D File Formats on the Web. International journal of advanced smart convergence. 2019;8(1):65-74.
33. Graham RN, Perriss RW, Scarsbrook AF. DICOM demystified: a review of digital file formats and their use in radiological practice. Clinical radiology. 2005 Nov 1;60(11):1133-40.

34. Varma DR. Managing DICOM images: Tips and tricks for the radiologist. The Indian journal of radiology & imaging. 2012 Jan;22(1):4.
35. Trivedi DN, Shah ND, Kothari AM, Thanki RM. Dicom® medical image standard. In Dental Image Processing for Human Identification 2019 (pp. 41-49). Springer, Cham.
36. Mildenberger P, Eichelberg M, Martin E. Introduction to the DICOM standard. European radiology. 2002 Apr;12(4):920-7.
37. Bidgood Jr WD, Horii SC, Prior FW, Van Syckle DE. Understanding and using DICOM, the data interchange standard for biomedical imaging. Journal of the American Medical Informatics Association. 1997 May 1;4(3):199-212.
38. Mustra M, Delac K, Grgic M. Overview of the DICOM standard. In2008 50th International Symposium ELMAR 2008 Sep 10 (Vol. 1, pp. 39-44). IEEE.
39. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess progranm. IEEE Computer Architecture Letters. 1979 Sep 1;28(09):690-1.
40. Overvoorde A. Vulkan Tutorial. Vulkan Tutorial. 2017 May.
41. Voetter RF. Opleiding Informatica.
42. Cozzi P, Hsu G. glTF [Internet]. The Khronos Group. 2021 [cited 13 May 2021]. Available from: https://www.khronos.org/gltf/
43. Willems S. Sascha Willems [Internet]. Sascha Willems. 2021 [cited 13 May 2021]. Available from: https://www.saschawillems.de/
44. Comaniciu D, Engel K, Georgescu B, Mansi T. Shaping the future through innovations: From medical imaging to precision medicine.
45. Pieper S, Lorensen B, Schroeder W, Kikinis R. The NA-MIC Kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the medical image computing community. In3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006. 2006 Apr 6 (pp. 698-701). IEEE.
46. Kourou K, Exarchos TP, Exarchos KP, Karamouzis MV, Fotiadis DI. Machine learning applications in cancer prognosis and prediction. Computational and structural biotechnology journal. 2015 Jan 1;13:8-17.
47. Gunadi SI, Yugopuspito P. Real-time gpu-based sph fluid simulation using vulkan and openGL compute shaders. In2018 4th International Conference on Science and Technology (ICST) 2018 Aug 7 (pp. 1-6). IEEE.
48. Wyrwas E. Proton testing of nvidia gtx 1050 gpu.

Appendix-

https://github.com/Vaitrix/Vulkan-DICOM-viewer