Light-Field Rendering on Mobile Devices

Ricardo Fonseca
ricardopfonseca@tecnico.ulisboa.pt

Instituto Superior Técnico

October 2021

Abstract— Light-Field Rendering is quite an old concept, as most pillars of Computer Graphics. Ahead of its time, it was overlooked by many, as, even though it was a great concept, there was a lack of processing power to implement it. Nowadays, with the increasing capability of computers, the concept is being revisited with eyes set on VR. And one particular type of device that is being picked up are the mobile devices with its ever-growing processing power. This thesis focuses on implementing Light Field Rendering algorithms on a mobile device. An Android application was developed for this, following two approaches to the problem, a naive one and a more optimized one, to be able to check feasibility, analyse the effects of optimization and what are the bottlenecks in present time. The results of this work show that this technique is usable in real-time on mobile devices. The performance of the algorithm mainly depends on the memory size that the input lightfields occupy and how aperture effects are processed. With those issues addressed, the bottleneck relies on the device's capacity of loading big image datasets into memory.

Keywords— light-field, computer graphics, real-time rendering, OpenGL ES, Android

I. Introduction

The ever-increasing mobile market has opened exciting research opportunities. With the developments in VR in recent years and with mobile performance getting better and better, it is only logical to take advantage of the best of both worlds. In VR the strive for photo realism is big, but even for less detailed worlds, it is still a somewhat technical challenge to get everything running stably at the desired 90 FPS, even on modern desktop hardware. However, with the processing power as it stands today, we can look back in time to rather ambitious algorithms that can now be run interactively and which might help overcome this hurdle. One of them is Light-Field Rendering (LFR), an algorithm that enables photo realism usually only possible with ray tracing methods, in real time rendering. Disney and Google have been researching and developing new technology in this field for the past few years, with the latter having implemented one specifically for mobile VR called Seurat [1]. Wanting to be at the forefront of the Computer Graphics field and more specifically in the VR field, Samsung UK is aligned with our excitement about this technique and with that a partnership was established with them to help on the development of this thesis.

Image based rendering is a set of techniques that makes it possible to visualize 3D objects and scenes in a realistic way without actually reconstructing a full 3D geometric model. It does this *by interpolating through discrete input images or re-projecting pixels in input images.* [2]

The range of this set goes from rendering techniques with no geometry, to some with implicit geometry ending with explicit geometry. This work, as the title entails, focuses on the no geometry end of the spectrum with Light Field Rendering at its core. In the beginning of this thesis a study of what was done and was being done in Light Field Rendering was done to assess where to start working.

A few papers and projects were analysed, including the initial proposal for this technique, but also recent research projects by Disney and projects like Seurat [1] by Google, all to be introduced in the next chapter.

Seeing that these were quite complex approaches and over-engineered to follow for the purpose of this thesis, we opted for a simpler approach and went with porting two existing open-source light field viewers, similar to one another, where one is a naiver approach and the other an optimized one.

It's clear from articles and projects like Seurat that real time Light Field Rendering applications or at least interactive ones are possible to implement in mobile devices.

With this in mind, the main focus of this thesis became validating, optimizing and studying its effects on this specific device and see what the current bottlenecks are.

The research was conducted in partnership with Samsung Research UK who provided the hardware to test the outcomes of this work, which led it focusing exclusively on Android devices.

II. Related Work

A. Image-Based Rendering methods

1) Texture-Based Volume Rendering

It starts by loading the data volumes onto the CPU, creating the transfer function lookup tables and fragment shaders. Afterwards it enters an Update phase where every time there is a change in the viewing parameters, the proxy geometry is computed and stored in vertex arrays [3]. A change of rendering mode and/or transfer function parameters will trigger an update of textures.

Moving on to the Draw phase, first there's the setup of the rendering state which *typically includes disabling lighting and culling and setting up alpha blending* [3]. The rendering state is then restored *after the slices are drawn in sorted order* [3].

2) Texture-Mapped Models

Texture mapping rendering has been around for many years now. The initial work is commonly attributed to Catmull in 1974. It's one great example of IBR and probably the most commonly used in present time. This technique can be seen being used in billboard rendering, when rendering 2D clouds in 3D games, 3D views of satellite imagery in applications such as Google Earth, or many different types of mapping for image fidelity such as ambient occlusion maps, normal maps, bump maps, etc.

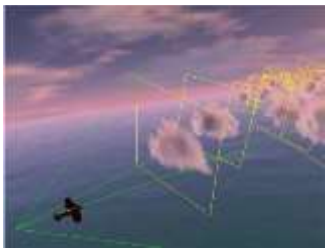

Figure 1 – Billboard Rendering [4]

Figure 1 - Effects of normal maps [5]

Although visual results have been improving throughout the years while using such techniques, they are still a long way from achieving what can be achieved through offline rendering.

3) Concentric Mosaics

The Concentric Mosaics approach is a 3D parameterization of the plenoptic function proposed by Shum and He [6]. As the name implies, *the camera motion is constrained along concentric circles on a plane* [7]. They are simple to capture, much like a traditional panorama, but do require more images

However, concentric mosaics allow the user to observe significant parallax and lighting changes, since the user is able to move freely in a circular region, providing a much better user experience.

This technique does present vertical distortions in rendered images, but these can be smoothed out using depth correction.

It offers good space and computational efficiency and it's very easy to capture, offering a smaller file size than LFR since it only constructs a 3D plenoptic function.

4) Transfer Methods

Coming from a term used in the photogrammetric community, it uses a small number of images by applying geometric constraints to *reproject image pixels appropriately at a given virtual camera viewpoint.* [7]

These constraints can be known depth values at each pixel, epipolar constrains between pairs of images, or trifocal/trilinear tensors that link correspondences between triplets of images.

5) 3D Warping

These techniques can be used to render viewpoints that are a short distance between themselves if depth data exists for every point in a set of images. When looking from any nearby viewpoint, an image can be generated by projecting the pixels of the original image to their respective 3D locations and re-projecting them onto the new picture.

This technique has a common issue of existing holes in the warped image, due to difference of sampling resolution between input and output images, and because of parts of the scene which are seen by the output image and not by the input one. To easily fix this, the common method is to stretch a pixel from the input image to match a size of several pixels on the output image.

6) View-Dependent Texture Maps

This technique was brought to light as texture-mapped models of real environments being generated through a 3D scanner or application of computer vision techniques to captured images aren't as accurate as desired, with the addition of the difficulty of capturing highlights, reflections, and other visual effects through the use of a single texture-mapped model. This is particularly useful for architectural environments.

Debevec et al. [8] proposed an approach where, using a 3D model of the real building and photographs of it from various viewpoints, the images are projected onto the model and merged between them, creating a composite rendering by taking into account the corresponding pixels in the rendered views.

7) Multiple Viewpoint Rendering

The Multiple Viewpoint Rendering (MVR) [9] technique bridges the gap between light fields and 3D scene geometry. Arranging the images from each camera by viewpoint location it is possible to form an image volume called the spatio-perspective volume. This rendering method takes advantage of the coherence and regular structure Epipolar Plane Image (EPI) representation, which is a slice through said volume, to efficiently render the scene.

Comparatively to Single Viewpoint Rendering (SVR) algorithms, MVR reduces the cost of computing the perspective image sequence as it can perform as many calculations as possible once per sequence instead of once per view.

8) View Interpolation

View interpolation is the process of creating a sequence of synthetic images that, taken together, represent a smooth transition from one view of a scene to another view. [10]

This technique was first introduced by Chen and Williams in 1993 [11]. It proposes using dense optical flow between two input images from a scene to interpolate arbitrary viewpoints [11]. It works really well if each two consecutive input images are close enough for there to exist overlapping between them. These view changes can also be improved by selecting an appropriate warping function, such as a cubic or quadratic interpolation, with each equation degree increasing the amount of computation rates needed to calculate said results.

9) View Morphing

View Morphing builds upon the work done for View Interpolation. It is better suited for larger camera angle changes and for non-linear camera paths. This is possible through the addition of Prewarping and Postwarping phases. The first one aligns the image planes without changing the optical centres of the camera with the latter yielding the image.

B. Light Field Rendering Projects

1) Seurat

This project was developed by Google and works as a plugin for major game engines. It is a *scene simplification technology designed to process very complex 3D scenes into*

a representation that renders efficiently on mobile 6DoF VR systems. [12]

It processes the scenes by generating data for a single headbox. It starts by generating RGBD input images of the scene, which should then be ran through the existing pipeline to generate the output geometry and RGBA texture atlas which can later be imported into the engine of choice.

The scene captures are organized into view groups, also called cube maps, which consist of a set of views, containing a camera and the RGBD information. The most common setup is render 32 groups from random positions inside the headbox[1]. These images are then used as input in the pipeline and the outcome is a textured mesh, according to a configurable number of triangles, texture size and fill rate.

2) Real-Time Rendering with Compressed Animated Light Fields

Disney Research proposed a real-time rendering approach using compressed animated light fields [13]. The outcome is an end-to-end solution for presenting movie quality animated graphics to the user while still allowing the sense of presence afforded by free viewpoint head motion. It mainly targets VR applications, using the tracking in real-time of the head pose to display an immersive representation of movie content that was previously offline rendered.

Contrary to immersive 360º videos, this approach enables motion parallax, as the input capture doesn't assume a fixed location. This in turn contributes to better immersion as the content doesn't seem flat and the user has free movement around the scene.

The proposed solution works by generating, for each frame, using a set of 360º cubemap cameras positioned near potential viewer locations, a set of cubemap images per frame containing colour and depth information. Using an optimization process, the cameras are positioned in such a way that maximises coverage while producing minimal redundancy. The computed dataset then runs through a compression step. The colour and depth data are compressed separately, using schemes perfected for use on GPUs for VR applications.

The algorithm for real-time rendering uses ray marching to reconstruct the scene *from a given camera using data for a set of viewpoints (locations and color/depth textures) and camera parameters*. [13] It first calculates the intersection with geometry by marching along the ray, and then calculates the colour contribution from all views.

To aid performance, the authors developed the compression methods to support a view-dependent decoding mechanism, enabling the decoding of only the parts of the video that are visible to viewers, reducing the per-frame bandwidth necessary to update viewpoint texture data. They also applied view-selection heuristics to prioritize set of viewpoints for each calculation given that not all cameras can give useful data in every situation.

C. Light Field Rendering

1) What is a Light Field

The concept of a light field comes from a proposition by Michael Faraday in an 1846 lecture, stating that *light should be interpreted as a field, much like the magnetic fields*[2]. The actual coining of the expression was by Andrey Gershun in 1936 in a paper about the radiometric properties of light in 3D space.

A light field is a vector function that describes the amount of light flowing in every direction through every point in space[2]. A 5D plenoptic function gives us the space of all possible light rays, with Radiance giving us the magnitude of each of the light rays.

The rendering technique is later proposed in 1999, by Levoy and Hanrahan, and proposes that the light field can be represented as *radiance as a function of position and direction, in regions of space free of occluders (free space)* [14].

This proposed restriction makes the five-dimensional function contain redundant information as the radiance along a ray remains constant as there are now obstacles to hit. Since this information is exactly one dimension, we can drop it, getting a 4D light field in the process.

2) Radiance

Radiance can be defined as the amount of light traveling along a ray, commonly represented in graphs by L and its unit of measurement is watts per steradian per meter squared.

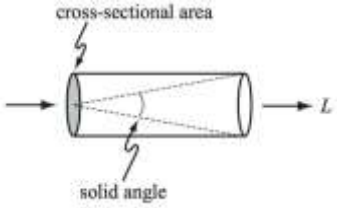

Figure 2 - Radiance can be thought of as the amount of light traveling along all possible straight lines through a tube whose size is determined by its solid angle and cross-sectional area. [15]

Steradian is the measure of a solid angle [16] and meter squared appears as is used as a measure of the cross-sectional area.

3) 5D Plenoptic Function

The plenoptic function describes the intensity of each light ray in the world as a function of visual angle, wavelength, time, and viewing position.

Adelson in 1991 [17] defined the plenoptic function as the *radiance along all such rays in a region of three-dimensional space illuminated by an unchanging arrangement of lights*[2]. This function is particularly useful in computer vision and computer graphics to define an image of a scene from any possible viewing position and angle at any point in time. It is five dimensional as rays in space can be parameterized by three coordinates and two angles.

[1] Stated in: https://github.com/googlevr/seurat
[2] Stated in: https://en.wikipedia.org/wiki/Light_field

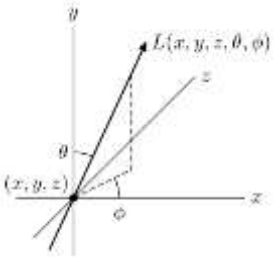

Figure 3 - 5-dimensional function [15]

4) 4D Light Field

As previously mentioned, we get a four-dimensional light field once we restrict ourselves to the locations outside the convex hull of an object, which defines it as radiance along rays in empty space.

A big difficulty with its representation is how to parameterize it as there are several issues to take into account, mainly efficient calculation, as the calculation of the position of a line from its parameters should be fast [14], control over the set of lines, since only a finite subset of line space is ever needed from the infinite space of all lines [14], and uniform sampling, where the number of lines in intervals between samples should be constant everywhere [14].

The most common way to parameterize a light field, proposed in Levoy's paper, is to represent lines by their intersection with two planes in arbitrary positions [14]. These lines represent rays of light hitting these planes. This representation is called a light slab:

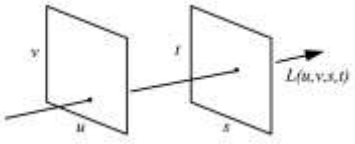

Figure 4 - Light slab representation [14]

Making the connection between this and the concept of IBR, an image corresponds to a 2D slice of the 4D light field making that, to create a light field from a set of images is the same as inserting each 2D slice in the 4D representation.
This representation enables the placing of one of the planes at infinity which in turn makes it possible to define lines by a point and a direction. That enables constructing light fields from orthographic images or images with a fixed field of view.

D. Mobile Architecture

Mobile devices nowadays feature some similarities to computers. Some even call them computers themselves. Which they are, as they feature most of the characteristics that define a computer: a screen, memory, storage, and a power source.

If component-wise they are quite similar, at least superficially, the way everything works is quite different. Starting at the size of each component, in the smartphone everything is quite a bit smaller, as these are handheld devices, meant to be carried in everyone's pockets. Then weight, power supply and heat dissipation are also major issues when talking about a computer that fits on the palm of your hand. And all of this means that processing power is also much more reduced than the one found in personal computers.

1) Unified Memory Architecture

UMA may also be known as Integrated graphics processing unit (IGPU)[3], and the difference to a dedicated graphics card is that it uses a portion of a computer's system Random Access Memory (RAM) rather than dedicated graphics memory. While on desktop it is used something called Immediate Mode as graphics pipeline, on mobile this isn't feasible. This happens because desktop GPU's have a dedicated memory interface with fast RAM optimised for GPU usage.

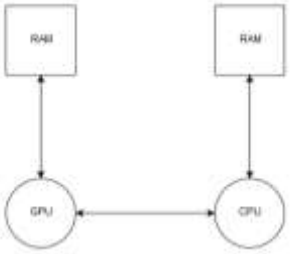

Figure 5 - Desktop memory layout

However, on mobile, communication between GPU and RAM is quite expensive, performance wise, with the devices not having a dedicated graphics memory but using a shared memory interface instead. So, the Tiled based method takes care of this issue.

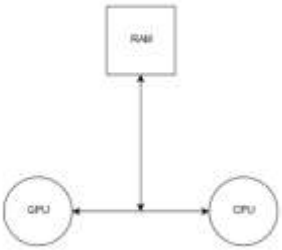

Figure 6 - Shared memory interface on mobile

2) Tile Based Rendering

While on desktop it is used something called Immediate Mode as graphics pipeline, on mobile this isn't feasible. This happens because of the non-existent GPU with a dedicated memory interface with fast RAM optimised for GPU usage, as mentioned in the previous section. However, on mobile, communication between GPU and RAM is quite expensive, performance wise, with the devices not having a dedicated graphics memory but using a shared memory interface instead. So, the Tiled based method takes care of this issue. In an Immediate Mode renderer, as each triangle is submitted its data enters the GPU pipeline, and all the pixels for this triangle (and its Z buffer values) pop out of the other end of the pipeline.

[3] https://en.wikipedia.org/wiki/Graphics_processing_unit#Integrated_graphics

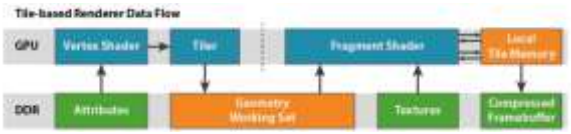

Figure 7 - Tile-based Renderer [18]

Which means that each geometric primitive is rendered one at a time, while storing information in colour, depth, and stencil buffers in the RAM. This takes up a lot of memory and makes it necessary for the GPU to communicate with the RAM quite a lot which is unmanageable on mobile devices as previously told. Enters the tiled based renderer. On this approach fragments are processed in tiles, as the name implies. More specifically, the ARM Mali GPUs process fragments in 16x16 tiles.

The first step to achieve this is to use a concept called Tiled Memory where we start reducing memory bandwidth by treating each cache line a two-dimensional rectangular area, in simpler terms a "tile", in memory. This will lead to less transfers to memory as more rendering happens within the cache because we are using square cache areas that are the same size as a linear cache. It works given that usually, *triangles that are near to each other in space are often submitted near each other in time*, resulting in more cache hits. [19]

Rendering then is broken into two phases:

- Binning phase (writes to memory)
- Rasterization (reads the bin contents)

In the first phase every triangle is checked to see whether it does or doesn't touch a tile. The renderer then goes through each tile and draws the triangles that were identified as touching that same tile.

With this phase ended, we get to the tile-based rasterization, where the rasterizer processes the scene one bin at a time, *writing only to local tile memory until processing of the tile is finished*. [19] Given that the renderer is only rendering the area corresponding to that tile, it can be processed in fast on-GPU memory. With this, main memory is only touched once as the tile is written out to it only when rendering is finished. [20]

When comparing this form of rendering with immediate-mode rendering, it is to note that it introduces latency as the last phase *cannot begin until all the geometry has been processed* [19], but the reduction in bandwidth, in turn, increases the speed of this phase. A few more complete details about this type of rasterization can be found in [19].

III. Implementation

As the title of this thesis implies, the work focused on the mobile environment and given the partnership with Samsung Research UK, Android was the underlying operating system for the application that was developed to test the algorithm for Light-field Rendering.

Stemming from this partnership, a smartphone was provided for development and testing purposes, more specifically a Samsung Galaxy Note 9 (Model SM-N960U) which makes use of a Qualcomm Snapdragon 845 System on-chip (SoC) with an octa-core CPU and a Qualcomm Adreno 630 GPU that takes care of the graphics processing.

Contrary to usual Android development, which is done in Java using Android's Software Development Kit (SDK) since the operating system runs on a Java Virtual Machine (JVM), the application was developed in C++ by using the Native Development Kit (NDK), which links Java and C/C++ through the Java Native Interface (JNI), and OpenGL ES which Android includes. For this application, OpenGL ES 3.3 was used.

A. Base Algorithm

The algorithm implemented follows what is stated in the original paper that first introduced the concept of Light Field Rendering.

The idea behind it is quite simple. Using either some rendering software (in this case Blender was used, as an open-source plugin was available to capture the light fields using any scene one would want to test) or a camera, one must shoot a scene from multiple consecutive angles. How much ground the camera captures images go left, right, up, and down will determine how much of the scene will be seen in the application. It is important too that the interval between camera angles isn't severe, to ensure that no jarring transitions occur when interacting with the app.

Having this set of rendered images, the application then introduces the set in a texture atlas, with the images ordered from left to right and then top to bottom.

The shader will receive the camera position, which is calculated using the motion values gathered by the gyroscope, which will then be used in conjunction with the aperture and focus values to select the right sub image from the texture atlas.

The selected image is then set to the texture that is being shown in the screen and mimics the way we would move through a scene, limited by boundaries which are defined when rendering the set of images offline.

The algorithm is quite simple, delivering great results, with great quality, resembling a flipbook animation.

The shader receives the camera position and checks the images positioned in the texture atlas around said vector position, with the search being limited by the aperture size, and if all parameters check out, the colour from for the current pixel is a combination of the adjacent ones.

B. Optimized Algorithm

When starting to implement this optimized approach, it was found that a small open source WebGL project was available on GitHub[4].

Given the lack of information on light field rendering implementation, the time constraints on this thesis, and with the major focus of this work being more on the study of the performance of this type of rendering in a mobile environment, it was decided to take inspiration from it and work on a form of a port of it in C++ and as an Android app. Taking advantage of memory features of C++, many of the variables we stored as pointers to reduce copies of variables and with that conserve memory. The user input gathering was changed to support gyroscope information as the original application was created to be interacted using a mouse and by clicking and dragging to mover around the scene. The shaders were also optimized to cache some calculations.

The original project not only optimizes the algorithm, but also compresses the input dataset. It does so by dividing it into *intra* and *predicted* frames.

The algorithm basis its processing on two shader passes while having four textures. Two are virtual textures, more

[4] https://github.com/mpk/lightfield

commonly known as texture atlas, one for the intra frames and another for the predicted ones. The third texture works as a lookup table, called Page Table in this case. It does so by storing in each pixel one two colours: black or blue. When calculating the fragment colour in the shader, it will sample from the intra atlas if black or from the predicted if blue.

The last is a Render Target which will be the output image on the last shader pass.

In each render cycle, a set of frames are selected for bilinear interpolation based on the current viewpoint and aperture. The frames are adjacent to the current viewpoint both on the y axis and x axis, and the range in which these are selected is bounded by the aperture value.

With the selection finished, it updates the respective texture atlas according to the type of frame. When updating the virtual textures, it also updates the Page Table with the new information regarding each new frame.

The first shader pass generates the composite image that is stored in the render target. The framebuffer is changed from the default one to point to the render target texture. Any calculations done in the current bound shader will be stored in said texture.

It uses the current viewpoint, the intra and predicted virtual textures and the lookup texture as input for calculating the composite image to be shown at the end of the current render cycle.

Using frame offsets of a maximum of one unit to each axis, and adding that to the current viewpoint, on the vertex shader pixel entries are selected from the page table. Four entries are selected and passed onto the fragment shader.

On the fragment calculation, these entries are used to either sample from the intra texture atlas or the predicted one, as stated previously. The final fragment colour is calculated by mixing the four calculated colours. This mix of sampled colour information enables the smooth interpolation between existing image frames.

The implementation described here was all ported into C++ and an Android app. It was able to display images and transition between frames, but this transition never got to be as smooth as the original application. A tremendous debugging effort was made to try and solve this issue, but with the delivery deadline approaching, development had to stop.

IV. Evaluation Methodology

A. Metrics

1) Frame rate

Frame rate is *the frequency at which consecutive images called frames appear on a display*[5]. The reason for it being the most important metric comes down to a concept called Flicker Fusion Threshold, also known as Flicker Fusion Rate. It is a concept in the psychophysics of vision and is related to the persistence of vision[6], and it is defined as *the frequency at which an intermittent light stimulus appears to be completely steady to the average human observer*[7]. If the frame rate falls below this threshold, the flicker will become apparent to observer, the so called "jerky" movement. This threshold varies with the viewing conditions.

In this case, where we want real-time rendering for the purpose of interactivity with the scene displayed by the application, ideally the values should be from 50Hz upwards, as studies show this value is the start of what most participants call stable images. However, we accept this value to drop down to 30Hz as it is still deemed acceptable by most users (many console video games are run at this frame rate).

The way this is calculated in this application is we first store the current time using the respective window management method for that purpose and then calculate the delta between the current time and the previous stored current time. We also store the number of frames, increasing by one every time the method is called. The delta is always checked as an if clause, and when equalling or surpassing 1.0, the application runs the code associated with this condition, where the frames per second value is calculated by dividing the number of frames by the calculated delta. Afterwards we display the frame rate value (on desktop it is displayed in the window title bar and on mobile is displayed as an overlay) and then reset the number of frames to zero and store the current time in the variable for the previous current time, which is used on the delta calculation.

2) Memory Usage

Although processing and displaying the image is somewhat inexpensive, the main drawback of this approach is needing to store every image of the rendered set, which, to have good results is synonym of at least 169 images, and with devices nowadays supporting ever increasing resolutions, this quickly adds up.

To measure all of this, we use the Snapdragon Profiler. This tool *allows developers to analyse CPU, GPU, DSP, memory, power, thermal, and network data, so they can find and fix performance bottlenecks* [21] by connecting with *Android devices powered by Qualcomm® Snapdragon™ processors over USB* [21]. We analysed these values by exporting them into a Comma Separated Values (CSV) file which in turn enables the data processing and creation of data visualizations. With such visualisations it was possible to confirm most of the expected outcomes that were conceived when developing the present work.

B. Test Scenes and Testing Methodology

Three scenes were created in Blender[8] using an open source lightfield generator plugin[9]. The idea behind the creation of the scenes was to make them progressively more complex.

Figure 8 - Simple Scene

[5] Stated in: https://en.wikipedia.org/wiki/Frame_rate
[6] https://en.wikipedia.org/wiki/Persistence_of_vision
[7] Stated in: https://en.wikipedia.org/wiki/Flicker_fusion_threshold
[8] https://www.blender.org
[9] https://github.com/lightfield-analysis/blender-addon

Figure 9 - Monkey Scene

Figure 10 - Dragon Scene

The first scene contains simple geometry consisting of a few cubes of different sizes and proportions. It was inspired by the Cornell Box scene and features a small cube surrounded by two cubes modified to resemble walls and another below to resemble a floor and features no texture mapping. The second scene features one complex object, in this case Blender's mascot Suzanne, surrounded by a few cubes of different sizes and proportions. The third one is a simple scene comprised of a very complex object with a high poly count, in this case the famous Stanford Dragon [22].

The scenes were structured in such way to gradually increase the processual effort needed to obtain a photo realistic render when rendering offline in order to compare the time taken to complete such task versus displaying such scene interactively through the proposed work in this dissertation.

To keep the conditions consistent the camera grid defined in Blender has the same configurations for every scene, with 13 by 13 cameras at a 10 cm distance between them, and the light source is of the sun type, positioned to provide the best visual appeal.

V. Results

Visual tests were done first to ensure the optimal parameters were chosen. This first step was important for two reasons: one because, given that the purpose of using this algorithm is to enable interactive apps using photorealism, human perception of how smooth the scene movement really is becomes key to the success of the application of such technique; and two because, given that there was going to be two test runs per scene, it was important to have chosen the right parameters to deliver the best results, as multiple configurations would quickly add up in terms of data to analyse.

The testing phase moved into the second step which was comprised of two test runs, a static and a dynamic. The first occurred with the device resting on top of a table without any interaction whatsoever. The latter, as the name suggests, involved movement with the device being hold straight and steadily about the same height every run and once the application finished loading a series of movements were performed to move around the scene. It is worth noting that the set of movements was always the same as well as the timestamps used to gather the data in between.

A. *Visual Tests*

For the focus plane variable tests revealed no impact on framerate so it was decided to just adapt it to each scene, since the objects are not all at the same distance from the camera. However, that is not the case for the aperture value. Analysing the fragment shader that calculates what image to display at any given camera position what was clear that this particular value would have an impact on the performance. Starting with the default value, where everything is in focus in the chosen focus plane, kept the framerate stable at 60 FPS. This situation was no longer true as soon as the aperture value started decreasing, introducing the famous blurring outside the focal area, the so called bokeh[10]. The framerate started having a noticeable impact, gradually dropping to the lowest measured value of 9 FPS, clearly not a suitable value for an interactive real-time application. Throughout this aperture range however, it is still possible to have some bokeh effect at an acceptable 30FPS, if keeping the value closer to the default setting.

Influencing the ability of loading the app was also the size of the dataset being used and the resolution of each image that it comprises. Initially the app was developed using Stanford's Light Field Archive[11] datasets, mainly one of a 17x17 grid with image size of 1024x1024 pixels. Developing the first custom lightfields it was decided to follow the same grid structure but with 1920x1080 pixels. This revealed to be an issue as the app would crash even before displaying any image, running out of memory. Bringing down the grid structure to 13x13 cameras helped launch the application, but it still took a long time loading everything into memory and displaying the first image. It also would have an impact on generating the datasets, as an image with that size would take almost an hour to render, if not more, which would hinder the timeline of the project. A middle ground was found, by choosing to keep the 13x13 grid but instead which each image having 1280 pixels of width and 720 of height, keeping the 16:9 ratio. Although the smartphone had a bigger resolution, keeping the ratio helped maintain a decent visual quality. Possible solutions to get around this issue can be found on section 6.2. It is also worth noting that, alongside the stable 60 FPS framerate, transition between viewpoints offered no jarring effects.

B. *Performance Tests*

One key measurement of these tests is the percentage of CPU utilization versus GPU:

Figure 11 - CPU Utilization (%)

From Figure 11, it is possible to conclude that the application relies very little on the CPU and in turn is GPU intensive, as expected. It is also worth noting the variation that exists between each static and dynamic test. This is due

[10] https://en.wikipedia.org/wiki/Bokeh

[11] http://lightfield.stanford.edu/

to extra work processing the camera position from the always changing gyroscope values.

The next Figure 12 helps support the previous statement about most processing happening mainly in the GPU. Although a small variance is present, it is mostly negligible, and the values stay rather constant.

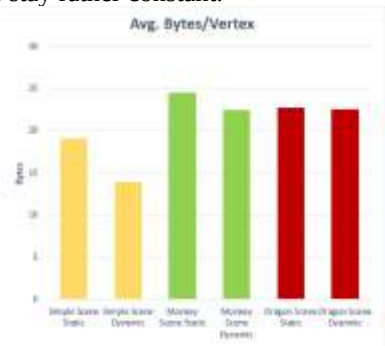

Figure 12 - Average Bytes loaded from main memory per vertex

Figure 13 shows a bigger discrepancy in the measurements suggesting more operations on the GPU side. Given the Monkey scene being the most visually complex one it is understandable that the number of bytes being loaded is higher than on the others.

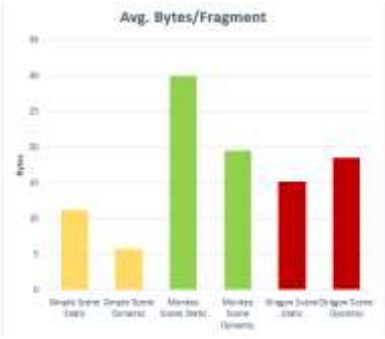

Figure 13 - Average Bytes loaded from main memory per fragment

Supporting the affirmations above, we have figures Figure 14 and Figure 15. Looking at the values presented, we can see that the values stay fairly consistent between both measurements, meaning that most if not all information being transferred from memory is mainly texture data. There is a small deviation in the case of the Simple Scene Dynamic test, possibly due to some misreading of the data from the profiler.

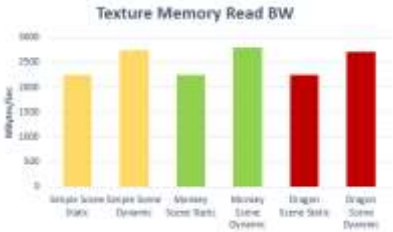

Figure 14 - MB of texture data read from memory per second

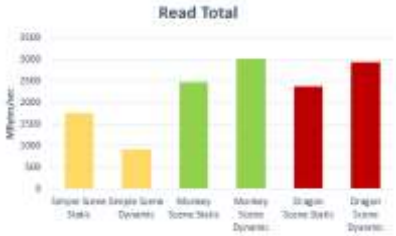

Figure 15 - Total num. of MB read from memory per second

Figures Figure 16 and Figure 17 support this claiming even further as practically 100% of the running time is spent shading fragments and not vertices:

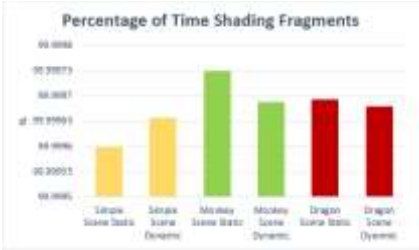

Figure 16 - Percentage of time spent shading fragments

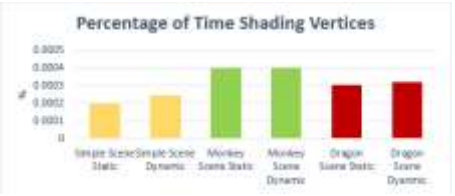

Figure 17 - Percentage of time spent shading vertices

This matches what was expected when developing the application given that the algorithm resembles the behaviour of a flipbook animation, displaying every render cycle a texture corresponding to a given viewpoint, nothing more, nothing less.

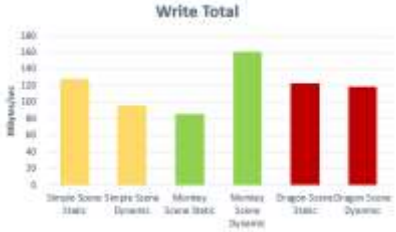

Figure 18 - Total num. of MB written to main memory per second

Further analysing the GPU's work, and how the algorithm functions, figures Figure 19 through Figure 21 show that, as expected, not much work happens at the vertex level, as it is only displaying a texture and no more geometry. In fact, in terms of Elementary Function Unit (EFU) instructions, there is a total of zero operations per vertex. In contrast, close to a thousand Arithmetic Logic Unit (ALU) instructions happen per fragment.

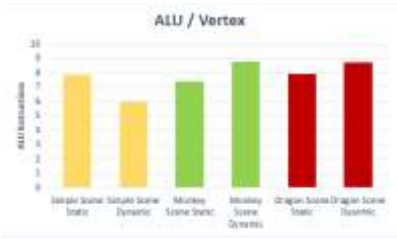

Figure 19 - Average number of ALU instructions per vertex

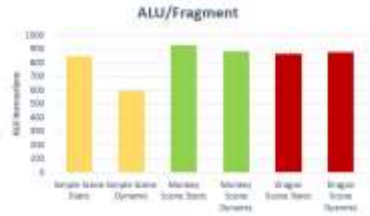

Figure 20 - Average number of ALU instructions per fragment

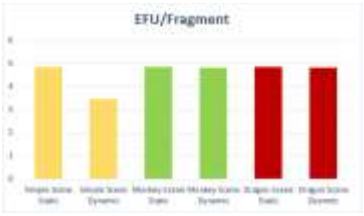

Figure 21 - Average number of EFU instructions per fragment

In terms of texture loading, given that all input images are loaded onto a texture atlas right at the beginning, and the application therefore only uses this structure, it was expected that not much stall would happen when the GPU tried to load new information. Although loading data from memory is really heavy on mobile devices, this technique has this major advantage of having everything needed preloaded at the expense of more memory occupied throughout the entire application lifetime.

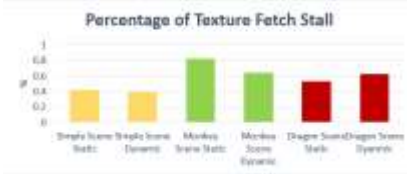

Figure 22 - Percentage of clock cycles where no more requests for texture data are possible

However, given that there is no way of blocking the gyroscope action as even really small variations are detected, the GPU is always calculating new information, making it difficult to have successful cache requests. Figures Figure 23 and Figure 24 show this phenomenon, with high percentages of missed cache requests.

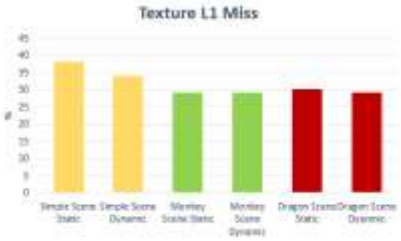

Figure 23 - Percentage of failed L1 texture cache requests

Figure 24 - Percentage of failed L2 texture cache requests

As for System Memory Usage, Figure 25 shows that memory consumption stays fairly stable throughout different scenes, only slightly increasing in the dynamic tests, which matches the results found in Figure 11, with the extra processing also having an influence here.

Figure 25 - MB of System Memory used

C. Summary

From these results it was possible to confirm many of the ideas had in the beginning about how the application would perform:

- It is possible to run interactive real-time applications using light field rendering;
- The application is GPU intensive;
- The main bottleneck is the amount of storage needed for the datasets;
- Low overhead for fetching textures as these are preloaded into memory at startup.

VI. Conclusions and Future work

The seed for this thesis was to test the feasibility of doing light field rendering on mobile devices, since photorealism and mobile graphics don't usually go hand in hand. Since this had already been done in some way, and since we were partnering with Samsung UK, we decided to focus on studying how a Samsung smartphone would cope with such rendering approach and what we could do to achieve the best results.

A. Achievements

With this work we were able to explain the topic of Light Field Rendering in a more simplified way and successfully develop a mobile app which implements the algorithm proposed in the original paper, bringing LFR to mobile

graphics and exposing its potential. The initial concerns about the main bottleneck being the amount of memory needed to store the datasets, were quickly confirmed the moment we started rendering the custom scenes. For the best results we needed to generate 169 images (13x13), with at least 1280 x 720 pixels, and even with that the megabyte count quickly went up. Since the algorithm works as a flip book animation, selecting the right view from the camera position, the main issue is having to have a big texture atlas containing all the views generated previously. As the dataset grows, the burden on memory grows too.

B. *Future work*

There is room for optimization. As previously stated, the original plan was to implement a "naive" approach, which is the one presented in section 3.2, and then optimize the algorithm in some way and compare both implementations. Work was started to achieve this, but a roadblock was hit. As stated in section 3.3, although it was possible to get the application running, displaying views from the scenes and reacting to user input, the transition between novel views was jittery, which is clearly not desirable on an interactive real-time application. Graphics debuggers such as RenderDoc[12] were used to a great extent in order to try and fix such problems. Although some bugs were found and quickly corrected, none were the solution for the jittery transition.

With this, one that wishes to continue this study could pick up on this and debug the application even further as time constraints dictated those efforts had to come to a halt.

Alongside this, one could also follow other approaches that could have an impact on the performance. The fragment shader could be refactored to work better without so much computational effort and the data sets could have their images further compressed and even be optimized by removing similar lightfields, as this WebGL viewer showed that these can be predicted through the neighbouring viewpoints. It would be interesting too to try tackling the memory bottleneck by exploring scene streaming possibilities so no dataset preloading would be necessary, enabling even bigger scenes to be used which in turn could improve interactivity and open new possibilities for this technique. Pairing this with the emergent 5G technology could prove to be really valuable.

VII. Bibliography

[1] Google, "Seurat," [Online]. Available: https://developers.google.com/vr/discover/seurat.

[2] Y. Chang and W. Guo-Ping, "A review on image-based rendering".*Virtual Reality & Intelligent Hardware.*

[3] M. Ikits, J. Kniss, A. Lefohn and C. Hansen, "Texture-Based Volume Rendering," Nvidia, [Online]. Available: https://developer.nvidia.com/sites/all/modules/custom/gpugems/books/GPUGems/gpugems_ch39.html.

[4] M. Harris and A. Lastra, "Real-Time Cloud Rendering," *Computer Graphics Forum*, 2001.

[5] J. D. Vries, "Learn OpenGL," [Online]. Available: https://learnopengl.com/Advanced-Lighting/Normal-Mapping.

[6] L.-w. He and H.-Y. Shum, "Rendering with Concentric Mosaics," *Association for Computing Machinery, Inc.*, 1999.

[7] H.-Y. Shum and S. B. Kang, "A Review of Image-based Rendering Techniques," Microsoft Research.

[8] P. Debevec, C. Taylor and J. Malik, "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach," in *SIGGRAPH*, 1996.

[9] M. Halle, "Multiple Viewpoint Rendering for Three-Dimensional Displays," Massachusetts Institute of Technology.

[10] M. Russell, "View Interpolation," [Online]. Available: http://pages.cs.wisc.edu/~rmanning/homepage/research/view.interpolation.html.

[11] S. E. Chen and L. Williams, "View interpolation for image synthesis".*Proceedings of the 20th annual conference on Computer graphics and interactive techniques.*

[12] Google, "Seurat," [Online]. Available: https://developers.google.com/vr/discover/seurat. [Accessed October 2021].

[13] B. Koniaris, M. Kosek, D. Sinclair and K. Mitchell, "Real-time Rendering with Compressed Animated Light Fields," *Graphics Interface*, 2017.

[14] M. Levoy and P. Hanrahan, "Light Field Rendering," *ACM SIGGRAPH*, 1996.

[15] M. Levoy, "Light-Field Sensing," [Online]. Available: https://graphics.stanford.edu/talks/lightfields-uncc-10jun08-public.pdf.

[16] S. P. Parker, "Steradian," in *McGraw-Hill Dictionary of Scientific and Technical Terms*, McGraw-Hill, 1997.

[17] E. B. J. Adelson, " The Plenoptic Function and the Elements of Early Vision," *Computation Models of Visual Processing*, 1991.

[18] P. Harris, "The Mali GPU: An Abstract Machine, Part 2 - Tile-based Rendering," [Online]. Available: https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/the-mali-gpu-an-abstract-machine-part-2---tile-based-rendering.

[19] "GPU Framebuffer Memory: Understanding Tiling," Samsung, [Online]. Available: https://developer.samsung.com/galaxy-gamedev/resources/articles/gpu-framebuffer.html#Limitations-of-tile-based-rendering.

[20] A. Garrad, *Moving Mobile Graphics: Mobile Graphics 101*, SIGGRAPH, 2018.

[21] Qualcomm, "Snapdragon Profiler," [Online]. Available: https://developer.qualcomm.com/software/snapdragon-profiler.

[22] S. C. G. Laboratory, "The Stanford 3D Scanning Repository," [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/.

[23] O. Kreylos, "Interactive Volume Rendering Using 3D Texture-Mapping Hardware," [Online]. Available: https://web.cs.ucdavis.edu/~okreylos/PhDStudies/Winter2000/TextureMapping.html.

[24] C. Lindsay, "CS563 Advanced Topics in Computer Graphics: Introduction to Image Based Rendering," [Online]. Available: http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/cliffl/cliffl_ibr_intro.html.

[25] ARM, "Tile-Based Rendering," [Online]. Available: https://developer.arm.com/documentation/102662/0100/Tile-based-GPUs?lang=en.

[26] H.-Y. Shum and S. B. Kang, "A Review of Image-Based Rendering Techniques," *A Review of Image-Based Rendering Techniques*, 2000.

[27] E. Catmull and E. E., "A subdivision algorithm for computer display of curved surfaces," 1974.

[28] S. Chan, "Plenoptic Function," *Computer Vision*, 2014.

[29] P. Debevec, Y. Yu and G. Boshokov, "Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping," University of California at Berkeley, 1998.

[12] https://renderdoc.org/