Ray Tracing Acceleration Structures on Mobile Environments

Nuno Guilherme Sousa
nuno.g.sousa@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2018

Abstract

Mobile devices are quickly getting more powerful and more capable of performing computationally expensive tasks. One of such tasks is Ray Tracing. A lot of research has been done both into the process of accelerating Ray Tracing using Acceleration Structures, and into the computational power of GPUs. However, with mobile GPUs having different architectures and limitations, not much research has been done into the usage of such GPUs in the process of Ray Tracing. Our work tries to fill this void by providing a performance analysis not only of Ray Tracing as a whole, but also the specific performance of different Acceleration Structures and rendering techniques. As such, we implemented an Android Application using OpenGL ES capable of Ray Tracing different scenes, applying several Acceleration Structures and rendering techniques. This application was then used to gather results, using measurements both provided by the application itself and third party tools. The gathered results led us to conclude that, currently, mobile environments seem to favour Acceleration Structures that use reduced memory bandwidth and ALU usage. We also concluded that using a Hybrid Rendering Technique seems to be the best approach for current hardware.

Keywords: Ray Tracing, Acceleration Structures, Mobile Environment, Android, OpenGL ES

1. Introduction

Mobile devices have been subject a lot of development over the last years. There are, however, limitations, and developers are always searching for optimizations that allows them to make the best out of the available hardware. With this in mind, nowadays, a mobile device is capable of rendering graphically intensive applications with reasonable quality and with a decent framerate.

One of such graphically intensive applications is Ray Tracing. Several methods have been proposed, to optimize Ray Tracing, to the point where it becomes feasible to use in a production environment.

One of the most used methods for accelerating Ray Tracing is the usage of acceleration structures. Acceleration structures help by reducing the number of intersection tests that have to be performed during rendering. The performance of such structures, both by themselves and when compared to each other, has been extensively studied. However, with the rise of powerful mobile devices, we are now at a turning point where such devices have become powerful enough to run Ray Tracing applications. But with all the differences between desktop and mobile architectures, all the research that was made into the performance of such structures is possibly invalid when considering mobile environments.

With little to no research existing in this particular area, our main objective is to try provide a baseline regarding Ray Tracing performance in a mobile environment, as well as measuring and comparing the performance of different acceleration structures when used in such environments. With this we hope to give insight to the current performance of the different structures and traversal methods, along with an overview of their limitations and shortcomings.

With new technologies, like PowerVR and DirectX DXR, Ray Tracing research is increasingly relevant and we hope to further improve it.

2. Previous Work

The idea of using ray shooting for the generation of images was first introduced by Arthur Appel in 1968 [1]. In his work, Appel presents an alternative for solving the "hidden surface" problem when rendering solid objects. Rays are shot from the virtual camera, through each pixel, and into the scene. Each ray is then tested for collisions against objects, and the closest object is determined. The colour that is returned depends on the properties of the object the ray hits. Appel also hints at the usage of ray shooting to shade the pixel. Shadows can be computed by shooting a "shadow ray" in the direction of light sources in the scene to determine if the corresponding light source is occluded. This became the simplest form of ray tracing. Several
other techniques have been developed since that provide much higher visual fidelity, by simulating visual effects like reflections[8], soft shadows, depth-of-field[2] and even techniques that include global illumination. This, however, is outside the scope of our work because our focus is the performance of Acceleration Structures and not the visual fidelity achieved with ray tracing.

2.1. Acceleration Structures

In Ray Tracing, ray-primitive intersection testing takes most of the computational time. As such minimizing the number of intersection tests is paramount to achieving a good performance and that can be achieved through the usage of acceleration structures. For our research we chose to focus on the performance of KD-Trees and Bounding Volume Hierarchies (BVHs) as these are considered state of the art in ray tracing. Both these structures, when going through the construction phase, make use of the Surface Area Heuristic (SAH) [3], which is the most commonly used heuristic for ray tracing performance, to determine what the best splitting point is for each node that is being subdivided. The SAH assumes that rays are uniformly distributed in space and are not blocked by the objects in the scene. Under these assumptions and assuming that both the costs of traversing a node and of the ray-triangle intersection are known, then the total cost of a split can be calculated. The cost is based on the conditional probability of a ray hitting a primitive inside a node while knowing that the ray intersects the node. By using this heuristic to subdivide space recursively, we maximize the efficiency of the resulting structure. The SAH also provides a termination criteria. When the calculated cost of the cheapest split is still greater than the cost of the undivided parent node, then there is no more need to continue subdividing.

2.1.1 Bounding Volume Hierarchies

Bounding Volume Hierarchies are based on bounding volumes. A Bounding Volume Hierarchy (BVH) is a tree in which the root consists of a bounding volume that encloses the whole scene. Each internal node is a bounding volume of a subset of objects of its parent node. The leaves contain actual geometry to test against. This binary tree can then be used to accelerate ray tracing.

The optimal subdivision scheme can be calculated in a number of ways. One way is to use the SAH. At each node, the algorithm considers all potential splitting planes and tries to choose the one that minimizes the SAH cost of the node. If the algorithm finds a split that minimizes the cost, the node is split and children are created, otherwise, the node is made a leaf and processing continues upwards. This results in a BVH that minimizes the overall SAH cost.

Regarding traversal of BVHs there have been several algorithms proposed. Two algorithms are of special interest to us:

- Stack-less Parent-Link Traversal - a link-based algorithm that tries to provide the same traversal order of the stack based algorithm while being stack-less. To achieve this, it uses extra information associated with each node, a parent link, that points to the parent node of the current node. It then uses the normal child links to traverse from parent to child, and the new parent links, to traverse from child to parent.

- Restart Trail Traversal - This algorithm tries to adapt the KD-restart algorithm, used to traverse KD-trees, to be used with BVHs. The algorithm cannot be used directly because different nodes of the BVH may overlap. Instead, Laine [4] proposed using what he called a "trail", which is a 32 or 64-bit code with information of what nodes have already been traversed. This way, every time a restart happens, the algorithm uses the trail to assure that no node is visited twice.

2.1.2 KD-Trees

First introduced as a methods for fast-searching of point in k-dimensional space, KD-Trees were adapted to serve as an acceleration structure for ray tracing. KD-Trees are a specific case of Binary Space Partitioning (BSP) trees, that work by subdividing space itself, as opposed to the BVHs which subdivide primitives, in a recursive manner. This means that a primitive can belong to more than one child node, if it straddles a splitting plane. These trees are usually built using a top-down approach in which, each node, is recursively split creating two child nodes.

Just like BVHs this subdivision can be based on the SAH. A possible implementation for construction in $O(N \log N)$ was presented by Wald and Havran [7] in 2006 using an ordered event list with special list splitting rules. A similar process can
be used for BVHs where each event in the list is simply a primitive centroid and where splitting the list is trivial because each primitive always belongs to only one node. Several algorithms have been proposed for the traversal of KD-trees. These algorithms can be divided into two groups:

- Basic algorithms, developed for sequential computing (Central Processing unit (CPU))
- Algorithms based on the sequential algorithms, optimized for Graphics Processing Unit (GPU)

Given our focus on GPU based traversal, the second group is of special interest to us, more specifically these two algorithms:

- Kd-Push-Down Traversal - The Kd-Push-Down algorithm expands Kd-Restart, which works by moving a point along the ray and finding the leaf where the point is located. When this happens the ray is intersected with all the objects contained in the leaf. If an intersection is found, the algorithm terminates. Otherwise, the exit point of the leaf node is calculated and the search point is advanced to the outside of the leaf bounding box and the cycle begins again. By keeping the lowest depth-wise node that contains the interval of intersection in its entirety this node can then be used instead of the root node when restarting the search.

- Kd-Backtrack Traversal - This algorithm adds to each node the corresponding bounding box and a pointer to the parent node. With this added information the algorithm can avoid restarting the search from the root node. Instead, every time the point is moved forward along the ray path, the algorithm traverses upwards the tree until the point lies inside the bounding box of the node being traversed. Then, the search proceeds downwards as usual.

2.2. Mobile Environment

Mobile architectures were created with several constraints in mind like power draw, heat dissipation and size. This led to the creation of an architecture that although powerful makes some compromises. These compromises include reduced memory bandwidth and limited shader resources. There are, however, some things unique to mobile architecture, like tile rendering, that provide new opportunities for optimization.

3. Implementation

Considering that the focus of this research is on mobile environments, and that it is done in partnership with Samsung UK we chose to focus exclusively on Android. Given our previous experience with development of graphical applications using C++ and OpenGL we chose to implement the core of our solution using Native Code and OpenGL ES.

The application was developed and optimized for on a Samsung Galaxy S8 (Model SM-G950U) with a 64bit - Qualcomm Snapdragon 835 CPU and a Qualcomm Adreno 540 GPU provided by Samsung Research, UK.

While in an initial phase we implemented a CPU based version of ray tracing, our focus quickly shifted towards GPU based methods. Therefore the CPU based implementation is not as optimized as the GPU based one and serves more as a baseline for the worst case performance of traditional, poorly optimized CPU based ray tracing.

3.1. Ray Tracing Implementation

Considering that the focus of this research is the performance of Acceleration Structures and not the quality of Ray Tracing we chose to implement the simplest of Ray Tracing methods. We simply cast a primary and shadow rays for each pixel and use this to shade the pixel.

Regarding ray intersection testing for ray-triangle intersections we used the Möller–Trumbore algorithm[5] and for Axis Aligned Bounding Box (AABB)-ray intersections we used the Ray-Box Intersection Algorithm by Amy Williams[9].

Along the development process we also implemented several approaches to the way primitives are stored in Shader Storage Buffer Objects (SSBOs) for the rendering process. Our initial approach was to store each triangle vertex and each normal as a vec3 with an extra float added as padding to prevent issues. Our second approach was an attempt to try and minimize the size used per primitive. To do so we use the paddings of the three vertices to store the first normal. Although this reduces the memory needed for each primitive it also introduces extra instructions to retrieve the information. Our last approach was based on the fact that, while doing intersection testing, normals are not used. Therefore, all bandwidth used by the normals is wasted. With this in mind our third approach consists of separating the vertices and normals into two separate SSBOs.

3.2. Acceleration Structures Implementation

For the CPU rendering methods we implemented KD-Tree, BVHs and Regular Grids, with one traversal algorithm per Acceleration Structure with traversal being single-threaded.

For the GPU rendering methods we chose to implement only KD-Trees and BVHs. We chose not to implement Regular Grids given time constraints and the fact that it’s been shown that they are consistently outperformed by BVHs and KD-Trees except in very specific situations.[6]
3.2.1 KD-Tree Implementation

The construction of KD-Trees in our implementation is done using the SAH algorithm by Wald and Havran described in the Previous Work section. Our current implementation allows for the creation of empty leaf nodes. However it does not perform triangle clipping. Regarding the SAH $C_{\text{Traversal}}$ and $C_{\text{Intersection}}$ values, by experimenting with several values we concluded that mobile architectures tend to favor wider, shallower trees. As such, the $C_{\text{Traversal}}$ and $C_{\text{Intersection}}$ values that we found to be efficient all around are 3.0 and 1.5 respectively.

The memory layout for KD-Tree nodes varies according to which traversal algorithm is being used.

![KD-Pushdown node layout](a.png)
![KD-Backtrack node layout](b.png)

Figure 2: Layout of KD-Tree nodes. vMin and vMax represent the node bounding box.

Another difference between trees for the two traversal methods is that building the tree for the KD-Backtrack traversal method, we do not allow perfectly flat nodes. This is done to avoid precision related issues when traversing the tree. For traversal we chose to implement the KD-Pushdown and KD-Backtrack algorithms. For the KD-Pushdown algorithm we did some changes to try and minimize branching when doing Near-Far classification. For the KD-Backtrack algorithm there were no such changes considering that the algorithm itself does no Near-Far classification.

3.2.2 BVH Implementation

In our implementation, the construction of BVHs is very similar to that of the KD-Trees as was described in the Previous Work section. Like with KD-Trees, after experimenting with several values, we came to the conclusion that, again, wider, shallower trees tend to perform best. As such the values chosen for $C_{\text{Traversal}}$ and $C_{\text{Intersection}}$ were, again, 3.0 and 1.5 respectively.

The memory layout for BVH nodes also varies according to which traversal method is being used. The possible layouts are shown in Figure 3.

![BVH Trail traversal node layout](a.png)
![BVH Parent traversal node layout](b.png)

Figure 3: Layout of BVH nodes. vMin and vMax represent the node bounding box.

For GPU traversal we implemented the Trail traversal along with the Parent-Link traversal algorithm.

3.3. GPU Rendering Methods

In the current implementation several rendering processes are available. Regardless of what rendering process is chosen, the application starts by constructing the selected acceleration structure along with the auxiliary structures for primitive storage. These structures are then copied to GPU memory as SSBOs. The application also uploads creates Vertex Array Object (VAO) containing a full-screen quad that is then used for the every rendering process. The drawing process, however, changes according to which rendering process is selected:

- **Using Fragment Shaders** - the application renders a full-screen quad using a very simple vertex shader. The fragment shader is then responsible for ray tracing the corresponding pixel. In this case all the code for ray tracing and structure traversal is contained in the Fragment Shader.

- **Using Compute Shaders** - the application performs a two step process. In the first step the application dispatches the necessary compute workgroups so that each thread processes a pixel of the final image. The result of this first step is stored in an Image Buffer which is then utilized on the second step as a texture. The second pass simply draws a full-screen quad, using the texture generated in the first step.

- **Using a Hybrid Shading Technique** - the application not only creates a VAO containing the full-screen quad, but also a second VAO containing the entire geometry for the scene being rendered. This second VAO is used in the first phase of the rendering process, where the application issues a drawcall to draw all the primitives using rasterization. This first phase outputs the calculated normals to the color attachment. This is done by encoding
the normal using the \((normal \ast 0.5 + 1)\) formula. From this first step a depth buffer is also generated. These two buffers are then used on the second phase of the drawing process where the full-screen quad is rendered. By using two shader extensions the application is capable of retrieving the values that were created in the first pass from the framebuffer. These values are then used to create and cast the shadow ray which then triggers a structure traversal. For this rendering method all the ray tracing logic is in the Fragment shader of the second pass.

4. Evaluation
For the evaluation of the implemented algorithms, we used both metrics gathered by our own application, as well as metrics gathered by external applications. All test were performed using a Samsung Galaxy S8 (Model SM-G950U) with a 64bit - Qualcomm Snapdragon 835 CPU and a Qualcomm Adreno 540 GPU provided by Samsung. While testing, the device was kept at high performance settings with a display resolution of 2220x1080 pixels.

4.1. Application Metrics
While rendering using ray tracing, the application provides several measures related to the rendering process. These measurements are displayed below the rendering window and are as follows:

- **Framerate** - expressed in Frames-per-Second (FPS) is one of the most used metrics for measuring graphical performance. It measures how many times the image that is being presented by the graphics engines updates every second. Higher numbers of frames per second equate to an image with smoother, more fluid movements.

- **Rays per Second** - expressed in millions of rays per second. While FPS tend to be dependent on several factors like rendering resolution and early traversal exits Rays-per-Second (RPS) more accurately describes the raw ray tracing power of the underlying hardware.

- **Structure Size** - expressed in KBytes. Represents the overall size of the generated acceleration structure.

The application also provides the option to output a heatmap representing the number of node traversals for each pixel, instead of the standard output. This was done through the usage of OpenGL ES Atomic Counters. By implementing a counter inside the shader that is incremented every time a node is traversed, we can easily obtain the number of traversed nodes. As an effort to keep results consistent, we consider that a node is traversed when it is fetched from the acceleration structure Shader Storage Buffer Object (SSBO). The application then outputs an interpolation between several colors according to the number of traversed nodes.

4.2. External Tools
The only external tool used to profile the application was the Qualcomm Snapdragon Profiler, an application developed by Qualcomm that allows developers to profile Android devices with Snapdragon processors. The application provides several metrics of interest to our work in real time:

- **SP Memory Read** - Number of bytes read from memory by the Shader Processors per second.

- **% Shader ALU Capacity Utilized** - % of maximum shader ALU capacity that is being utilized.

- **% Time ALUs Working** - % of time the ALUs are working while the shaders are busy.

- **ALU/Fragment** - Average number of ALU instructions performed per fragment.

4.3. Test Scenes and Test Methodology
All test scenes used in our test are scenes that have been extensively used to benchmark ray tracing. The scenes that we decided to use are shown in Figure 4.

![Scene Selection](a) Cornell Box  (b) Buddha  (c) Fairy Forest  (d) Crytek Sponza

Figure 4: Scenes selected for testing.

The reasoning behind the choice of these scenes was to try provide a wide array of scenes with different characteristics. The Cornell Box scene is a simple one, with very few primitives, while the Cornell Buddha is a more complex scene with a higher number of primitives, but still reasonable. The Fairy...
Table 1: My caption

<table>
<thead>
<tr>
<th></th>
<th>Cornell Box</th>
<th>Buddha</th>
<th>Fairy Forest</th>
<th>Crytek Sponza</th>
</tr>
</thead>
<tbody>
<tr>
<td>Struct Size</td>
<td># Nodes</td>
<td>Struct Size</td>
<td># Nodes</td>
<td>Struct Size</td>
</tr>
<tr>
<td>KD-Backtrack</td>
<td>0.61 Kb</td>
<td>13</td>
<td>6.5Mb</td>
<td>139k</td>
</tr>
<tr>
<td>KD-Pushdown</td>
<td>0.20 Kb</td>
<td>13</td>
<td>2.2Mb</td>
<td>139k</td>
</tr>
<tr>
<td>BVH Parent-Link</td>
<td>0.52 Kb</td>
<td>11</td>
<td>3.1Mb</td>
<td>66k</td>
</tr>
<tr>
<td>BVH Trail</td>
<td>0.34 Kb</td>
<td>11</td>
<td>2.1Mb</td>
<td>66k</td>
</tr>
</tbody>
</table>

Forest scene was chosen as it is a prime example of the "Teapot in a Stadium" problem where a high polygon count object is inside a large area with low polygon count. Lastly the Crytek Sponza Scene was chosen for it’s high complexity to try and test the limits of our application.

All the test performed use the exact scenes shown above. The application always renders a 1024x1024 image. As such all the results that will be presented are for 1024x1024 resolution. We also restart the application between every test.

5. Results
5.1. Acceleration Structure Comparison
5.1.1 Structure Size

One of the most important things regarding the usage of Acceleration Structures is the size they occupy. This is especially important on mobile environments where memory, and memory bandwidth is not as available as in desktop architectures. As such, it is important to provide an overview of the size of each Acceleration Structure.

In Table ?? we show the structure size for each of the traversal methods for each of the test scenes as well as the size of the underlying structures in total number of nodes. The difference in the number of node for both traversal methods for KD-Trees is due to the limitation previously mentioned regarding perfectly flat nodes.

As we can see BVHs have a considerable advantage regarding the total number of nodes generated. This, however, does not necessarily translate to a smaller structure size as node layout also plays an important role. And, although size as an impact on performance, it is not the only deciding factor, as we will see in the following section.

5.1.2 Performance

To test the performance of different acceleration structures we performed a series of tests using a combination of all the traversal methods and all the scenes. For these particular tests we used only the Fragment Shader Renderer along with a normal primitive layout. This was done to reduce the number of tests and to provide a greater focus on what is being compared, by removing unnecessary variables.

Figure 5 shows the results obtained from all the test runs. Although the KD-Pushdown traversal excels at scenes with lower number of primitives it quickly falls off when the scene gets more complex. On the other hand, the BVH Trail traversal has the best results with more complex scenes.

To understand the performance differences between each traversal methods we began by creating traversal heatmaps for each traversal method and each scene. In Figure 6 a subset of the heatmaps generated is shown. The Fairy Forest scene was chosen for this comparison as it better showcases the differences between traversal methods.
As the figure shows, the KD-Pushdown algorithm is the worst in terms of traversed nodes. The best is the BVH Trail, with KD-Backtrack and BVH Parent-Link showing similar results. This supports the results obtained previously and seems to indicate that the number of traversed nodes is heavily correlated to the performance of an algorithm. Considering that the main performance impact of traversing a node is out the memory bandwidth usage, the next step was to try and verify what these results meant in terms of memory bandwidth. The results gathered can be seen in Figure 7.

Figure 7: SP Memory Read values for each structure.

The results show that algorithms based on KD-Trees utilize much more memory bandwidth than those based on BVHs. One possible explanation for this is the increased number of nodes in KD-Trees. With increased node numbers it is much more probable that, for each new node traversed, there is a need to execute a memory fetch. This is because each memory fetch to local memory is less probable to contain the next node that needs to be traversed next.

However, with memory bandwidth not fully explaining the deltas in performance between methods, we also analysed the impact of each method on the ALU. The results can be seen in Figure 8.

Figure 8: ALU instructions per fragment.

As the results show the KD-Backtrack algorithm has the least overall amount of ALU instructions. This is possibly due to the fact that it does no Near-Far classification, and, consequently, does fewer instructions per node traversed than other traversal methods.

On the opposite end, the traversal method with most ALU instructions per fragment is the KD-Pushdown. This is possibly derived from the extra traversal steps that the algorithm takes because of the increased number of nodes in the underlying KD-Tree. This along with the high memory bandwidth usage makes for the worst performance of all the traversal methods.

These results seem to indicate that traversal methods with fewer ALU instructions per fragment have better performance. There is, however, the exception of the KD-Backtrack traversal. This is possibly due to the memory bandwidth usage we saw earlier. Despite having fewer ALU instructions per fragment, the traversal is slowed down by the higher memory requirements. It seems that the current performance of the traversal algorithms seems to be limited by a combination of ALU processing power, number of memory accesses and bandwidth limitations. However, the strongest limitation appears to be the number of traversed nodes, more specifically the number of accesses to the SSBOs that contain the acceleration structure.

5.2. Primitive Layout Comparison

To analyse the differences in performance for the different primitive layouts in the shader a series of tests were ran using a combination of different scenes, different traversal methods and different primitive layouts. In Figure 9 we can see the results obtained when varying the type of layout for the primitives.

The results show that the usage of the compact layout not always equates to better performance. This is possibly due to the fact that the compact layout needs a few extra instructions to retrieve the stored normal. The split layout on the other hand consistently provides either similar or better results. To better understand what the impact of these changes might be on a memory level the next step was to perform a memory bandwidth analysis. To do so, a series of tests using the Snapdragon Profiler was ran. For these tests only the Cornell Box and Buddha scenes being used, each representing simpler and more complex scenes. The results are shown in Figures 10 and 11.

The results show that there is a clear improvement in memory bandwidth usage from using the compact and split layouts with split layout having the best results. Having vertices and normals separated means that no bandwidth is wasted on normals that are not being used. This maximizes the
number of vertices fetched at a time in turn requiring less fetches for more vertices.

These results also show that these improvements seem to be more pronounced in scenes with higher geometric complexity.

With scenes having more geometry, and consequently more accesses to the structures containing the scene geometry, optimizations on primitive layout become more important and have a bigger impact in memory bandwidth usage.

### 5.3. SAH Costs Comparison

One of the ways to optimize KD-Trees and BVHs built using the SAH is to tweak the values for traversal and intersection cost given to the construction function. Usually, having the intersection cost higher than the traversal yields better results. To test this we performed several test runs using a combination of all the scenes, all the traversal methods and four different scenarios for the cost values.

The results of our test runs are condensed in the graph shown in Figure 12.

As shown by the results, having an intersection
cost lower than the traversal cost provides the best results overall. The results also show that, overall, it is best to keep the traversal cost only slightly higher than the intersection cost. To understand what kind of impact changing these values has on the traversal process we did a heatmap analysis, again, using the Fairy Forest scene as it better emphasises the differences between traversal methods.

5.4. Rendering Techniques Comparison
The developed application has three different approaches to rendering to choose from. To compare the performance for each different approach we tested a combination of all traversal methods with all the scenes using all three rendering approaches. Figure 14 shows the results we obtained.

As we can see, rendering using Fragment Shaders and rendering using Compute Shaders is very similar. This is possibly due to the process of rendering being extremely similar with Compute Shaders just adding the extra step of rendering the full screen quad which, considering the sizes used for our renderings, and the current frame rates achieved, as minimal impact in FPS.

While Fragment and Compute rendering processes had very similar performance, the Hybrid Rendering process distinguished itself by having a significantly better performance than other rendering methods. This was to be expected given that the ray tracing process is much more computationally expensive than the simple rasterization, with essentially no fragment shading, needed for Hybrid rendering.

Overall our results seem to indicate that using Hybrid Rendering is the best approach when implementing Ray Tracing on mobile. However, more complex types of ray tracing may benefit from using Fragment and Compute Shader based rendering.

6. Conclusions
With existing papers focusing on the development of entirely new mobile architectures focused on ray tracing there is little to no research exploring the ca-
Figure 14: Graph comparing the performance of different rendering methods.

pabilities of current mobile architectures. Our work tries to fill this void by providing an overall analysis of the performance of what is considered state of the art when implemented in mobile environments. Our main goal was to establish a basis for future research into the potential of mobile environments. There is a lot of possible future work resulting from or research. We also have the intention to publish a paper detailing our research and what our findings were.

Acknowledgements
We would like to thank David Cena, Nikolaos Papadopoulos and everyone over at Samsung Research UK for their invaluable help and for sending us the necessary equipment for development and testing.

References


