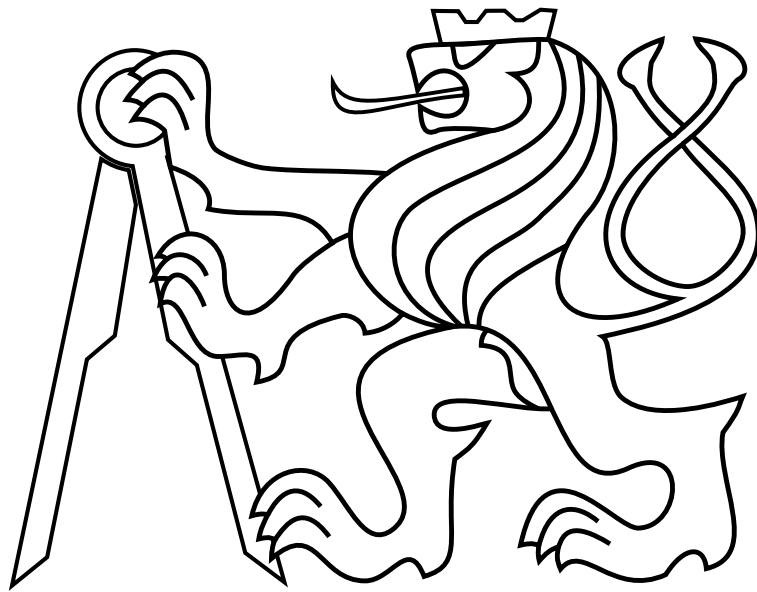


CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Master thesis



Jan Langr

Data Structures for Ray Tracing for Mobile Devices with Android OS

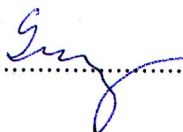
Department of computer graphics and interaction

Thesis supervisor: **doc. Ing. Vlastimil Havran, Ph.D.**

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne.....27.5-2016.....

..........

Declaration

I hereby declare that I have completed this thesis independently and that I have used only the sources (literature, software, etc.) listed in the enclosed bibliography.

In Prague on 27.5.2016


.....

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jan Langr**

Studijní program: Otevřená informatika

Obor: Počítačová grafika a interakce

Název tématu: **Datové struktury pro vrhání paprsku na mobilních zařízeních s OS Android**

Pokyny pro vypracování:


Nastudujte problematiku stavby prostorových datových struktur pro metodu vrhání paprsku. Zaměřte se na datové struktury a paralelní algoritmy pro jejich stavbu umožňující stavbu v reálném čase, tedy lepší než 32ms pro středně velké scény kolem 100,000 trojúhelníků. Po dohodě s vedoucím práce vybrané metody implementujte, optimalizujte rychlost výpočtu a spotřebu paměti. Realizované algoritmy otestujte na sadě alespoň deseti scén o různé velikosti a distribuci trojúhelníků. Realizujte testovací aplikaci zobrazující scény v reálném čase.

Seznam odborné literatury:

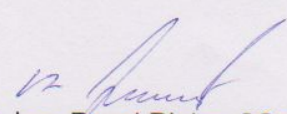
[1] J. Lee, W.-J. Lee, Y. Shin, S. Hwang, S. Ryu, J. Kim:
"Two-AABB traversal for mobile real-time ray tracing", SA '14 SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications
Article No. 14, doi:10.1145/2669062.2669088.

Vedoucí: doc.Ing. Vlastimil Havran, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016


prof. Ing. Jiří Žára, CSc.
vedoucí katedry




prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 25. 3. 2015

Acknowledgement

I would like to thank doc. Ing. Vlastimil Havran, Ph.D. for his endless patience, support during the work and helpful tips, without which this thesis could not have been made. I would also like to thank my family and girlfriend for their tolerance during my work on this thesis.

Abstrakt

Cílem této práce je nastudovat algoritmy umožňující rychlé vykreslování scén pomocí metody sledování paprsku a algoritmy pro rychlou stavbu akceleračních datových struktur, které by umožnily vykreslování dynamických scén v reálném čase na moderním mobilním zařízení s operačním systémem Android. Nejprve je proveden výzkum existujících Android aplikací, které vykreslují scény pomocí metody vrhání paprsku. Dále jsou zkoumány metody pro efektivní vykreslování pomocí vrhání paprsku a stavbu akceleračních datových struktur, které se pro toto vykreslování používají. V dalších kapitolách práce je předveden implementovaný vykreslovací systém, který je schopen vykreslovat scény v reálném čase, čehož je dosaženo využitím vícevláknového programování a nízkourovňových optimalizací kódu. Součástí tohoto systému je také algoritmus pro rychlou stavbu akcelerační datové struktury, u kterého je rovněž kladen důraz na optimalizaci, aby bylo možné datovou strukturu přestavovat v reálném čase. Výsledný software je otestován na sadě statických i dynamických scén na výkonném mobilním zařízení se systémem Android.

Abstract

The aim of this thesis is to examine ray tracing methods and methods of fast building of acceleration data structures for ray tracing which would allow for rendering of dynamic three-dimensional scenes in real time on a contemporary mobile device with the Android operating system. First, research of existing ray tracing applications for Android is made, along with algorithms focused on high performance rendering using ray tracing and fast construction of acceleration data structures used in ray tracing. An implementation of a rendering system which can render scenes in real time, which is achieved by using multithreading and low-level code optimizations, is presented in the next chapters. A part of this system is also an acceleration data structure builder. The implementation of this builder also uses low-level optimizations in order to be able to rebuild the needed data structures in real time. The resulting software is tested on both static and dynamic scenes on a high-end device with the Android operating system.

Contents

1	Introduction	1
1.1	Thesis structure	1
1.2	Ray tracing	3
1.2.1	Basics	3
1.2.2	Lighting model	5
1.2.3	Performance	7
1.3	Acceleration data structures for ray tracing	8
1.3.1	Construction schemes	8
1.3.2	Uniform grid	8
1.3.3	k-d tree	9
1.3.4	Octree	11
1.3.5	Bounding Volume Hierarchy	11
1.4	Dynamic scenes and interactive ray tracing	14
1.4.1	Dynamic scene composition	14
1.4.2	Object BVH	15
1.5	Android operating system	16
1.5.1	Android application development	16
1.5.2	Android devices	16
2	Research and other solutions	17
2.1	Android application development	17
2.1.1	Android device	17
2.1.2	Programming languages and tools	17
2.1.3	Parallel computation	19
2.2	Ray tracing on Android	20
2.2.1	Ray tracing dedicated hardware	21
2.3	Fast construction of acceleration data structures	21
2.4	Fast ray tracing	24

3	Design and implementation	25
3.1	Used technologies	25
3.2	Ray tracing library	25
3.2.1	Object BVH	26
3.2.2	Dynamic objects	26
3.2.3	Scene management	27
3.2.4	Library interface	28
3.2.5	Rendering	29
3.2.6	BVH traversal	32
3.3	Testing Java Android application	32
3.3.1	OBJ file parser	33
3.4	BVH construction	33
4	Performance testing and evaluation	39
4.1	Keyframe animation scenes	39
4.2	Static scenes	40
4.3	BVH construction - setup 1	41
4.4	BVH construction - setup 2	42
4.5	BVH construction - setup 3	43
4.6	Rendering performance and BVH quality	44
4.7	Discussion	45
5	Conclusion	46
5.1	Future development	46
6	Appendix	48
6.1	CD Contents	48

List of figures

1	Illustration of the process of rendering an image of a scene using ray tracing	3
2	Diffuse lighting in the Phong model	6
3	Specular lighting in the Phong model	7
4	Ray traversal of a uniform grid	9
5	Three phases of a k-d tree construction	10
6	Three phases of a top-down BVH construction	13
7	Tracing a ray through the object bounding volume hierarchy	27
8	Projection plane sampling using a SIMD ray and multiple threads	30
9	Keyframe animation scenes used for performance tests	40
10	Static scenes used for performance tests	41

List of tables

1	Krait 300 CPU specifications	17
2	Adreno 320 GPU specifications	17
3	BVH build performance - BVH built with a single thread	42
4	BVH build performance - BVH built with a single thread, NEON instructions	43
5	BVH build performance - BVH built with multiple threads, NEON instructions	44
6	Scene rendering performance for 1024x576 resolution	45
7	CD contents	48

List of algorithms

1	Naïve ray tracing algorithm	4
2	Naïve version of agglomerative clustering	22
3	Algorithm of the BVH builder based on binning	37
4	BVH node splitting based on binning	38

1 Introduction

In most of today's interactive computer or mobile applications where performance is key, rasterization is the method of choice for fast rendering of three-dimensional scenes. In rasterization, objects from the scene are projected onto a two-dimensional projection plane which represents the view area of a camera. In ray tracing, the process is inverse. Rays are shot from the viewpoint through the projection plane into the scene and objects intersected by these rays are then drawn onto the projection plane, pixel by pixel. Unlike rasterization, this method results in a natural occurrence of phenomena such as shadows and reflections, which have to be simulated in rasterization, often with rough approximations. This comes at a great cost, which is an orders of magnitude higher computational cost.

The main problem of the higher computational cost of ray tracing is the fact that for each ray, the scene needs to be searched for the closest object the ray intersects. Using a naïve approach, this is very expensive. A lot of research is therefore focused on acceleration data structures that store the scene objects in a way that allows for efficient traversal of the ray through the scene while skipping most of the uninteresting objects. When used with dynamic scenes, these structures often need to be rebuilt quickly in order to maintain interactivity of the application. Various algorithms have thus been proposed and are still sought that aim to construct the data structure with a trade-off between construction speed and the quality of the structure with respect to ray tracing performance.

With mobile devices such as smart phones or tablets gaining performance rapidly year by year, it is no longer necessary to restrain ray tracing use to high-end desktop or laptop computers. These devices today have 4 or 8 core processors, a few gigabytes of RAM and dedicated graphics processors. The newest high-end devices even support using their graphics processors for general-purpose computing (GPGPU).

The goal of this thesis is to examine the computing capabilities of a high-end mobile device with the Android operating system and implement a ray tracing system that would be able to render dynamic (moving) scenes at interactive frame rates. That inherently includes being able to construct acceleration data structures over the scene data at very high speeds.

1.1 Thesis structure

The first chapters of the thesis are an introduction to ray tracing, acceleration data structures used in ray tracing, the problematics of dynamic scenes and the Android system.

The next part focuses on the research of the device's hardware and software capabilities, existing ray tracing applications on the Android platform, and mainly on various data

structures suitable for ray tracing and algorithms for their fast construction. Last but not least, the chapter examines tricks and optimizations to make the process of ray tracing faster (apart from using acceleration structures).

The next chapter describes the rendering system and acceleration structure building algorithms implemented for the Android platform as part of this thesis.

In the last chapter, the implemented system is tested on a set of various scenes and configurations, and performance is measured.

1.2 Ray tracing

1.2.1 Basics

Ray tracing is a method of image synthesis (rendering) based on tracing the path of light from light sources in a 3D scene to the camera. A great advantage of this method is that phenomena such as reflection, refraction and shadows occur naturally during the rendering process, unlike in rasterization, where they have to be simulated and are never 100% accurate.

For each pixel of the projection plane, a primary ray is cast from the viewpoint (camera) through the projection plane positioned in the scene. If an intersection of this ray with an object in the scene is found, rays from all light sources are traced towards the intersection of the primary ray with the object. If any of these light sources are not occluded and rays from them arrive at the intersection point, the color of the pixel is calculated using the material properties of the object and properties of the light source. Additionally, if the material of the object is reflective or transparent, a reflected and/or refracted ray is cast from the intersection point in the direction of the reflection (refraction). The same process as for the primary ray is repeated for this ray. Any color this secondary ray 'collects' is then added to the final pixel color. An illustration of the process is depicted in figure 1 and algorithm 1.

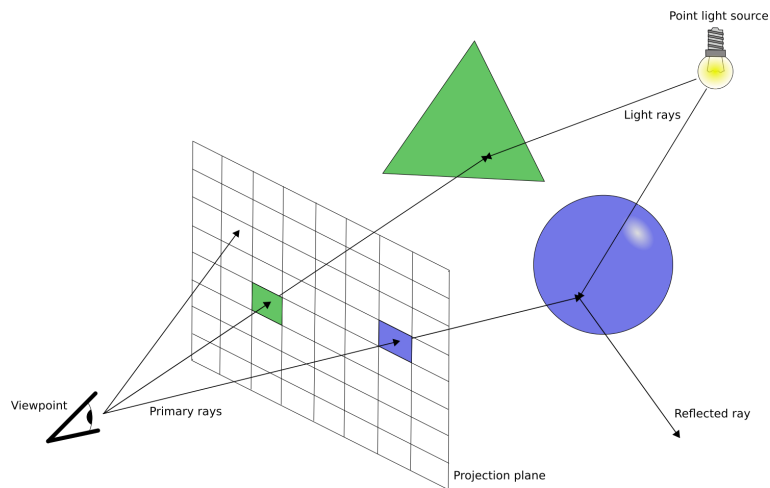


Figure 1: Illustration of the process of rendering an image of a scene using ray tracing

```
w ← projection plane width in pixels;
h ← projection plane height in pixels;
 $\vec{C}_p$  ← camera position;
for int i=0; i<w; i++ do
  for int j=0; j<h; j++ do
     $C_{pixel}^{\vec{}}$  ← (0, 0, 0);
     $\vec{V}$  ← ray from viewpoint through pixel [i,j];

    closestObjectEye ← null;
    min_t ← ∞;
     $\vec{I}$  ← null;
    foreach object O in scene.objects do
      t ← intersect  $\vec{V}$  with O;
      if  $t < min\_t$  then
        min_t ← t;
        closestObjectEye ← O;
         $\vec{I} \leftarrow t * \vec{V} + \vec{C}_p$ ;

    Compute ambient lighting;

    if closestObjectEye is not null then
      foreach point light P in scene.pointLights do
         $\vec{L}$  ← ray from P's position towards  $\vec{I}$ ;
        closestObjectLight ← null;
        min_tLight ← ∞;
        foreach object O in scene.objects do
          tLight ← intersect  $\vec{L}$  with O;
          if  $tLight < min\_tLight$  then
            min_tLight ← tLight;
            closestObjectLight ← O;

        if closestObjectEye = closestObjectLight then
           $C_{cur}^{\vec{}}$  ← Compute diffuse and specular lighting;
          Add  $C_{cur}^{\vec{}}$  to  $C_{pixel}^{\vec{}}$ ;
        else
           $C_{cur}^{\vec{}}$  ← Background color;
          Add  $C_{cur}^{\vec{}}$  to  $C_{pixel}^{\vec{}}$ ;
```

Algorithm 1: Naïve ray tracing algorithm

1.2.2 Lighting model

After obtaining the nearest object the primary ray intersects and a set of lights that illuminate the object at the point of intersection, a set of rules is needed for calculating the color that is returned back to the pixel. These rules are provided by a lighting model. One of the most commonly used lighting models is the Phong lighting model, which is also used in the rendering part of the implementation in this thesis. The Phong model calculates the pixel color based on three components:

- **Ambient component**, which simulates global illumination caused by light scattering and reflecting throughout the entire scene.
- **Diffuse component**, which takes into account the color the object emits when illuminated by a light source.
- **Specular component**, which simulates the direct reflection of a light source on the object's surface (so called specular highlight)

The ambient component depends on the color of the object's material (the color that it reflects) and the selected color of ambient lighting. This color is not calculated from the light sources present in the scene and is only a rough approximation that doesn't correctly represent light distribution in the scene and varying amounts of light coming at the object from each direction. The component is calculated using the following formula:

$$\vec{C}_a = k_a \vec{C}_A$$

where \vec{C} is a three-component vector describing a color in the RGB format with values ranging from 0 to 1. \vec{C}_A is the color of the ambient light in the scene and k_d represents the fraction of the ambient light that the object reflects.

The diffuse component depends on the color of the object's material and the angle between the surface normal at the point of intersection and the direction of the shadow ray. The diffuse component of the final color is calculated using the following formula:

$$\vec{C}_d = k_d \frac{\vec{N} \cdot \vec{L}}{|\vec{L}|} \vec{C}_L$$

where \vec{C}_d is the color of the object's material, \vec{C}_L is the color of the light emitted by a light source, \vec{N} is the normal vector on the object's surface at the point of intersection

with the view ray and \vec{L} is the shadow ray (in the direction towards the light source). As can be seen from the formula, the diffuse color seen by the camera or viewer is independent of its position relative to the object. The configuration can be seen in figure 2.

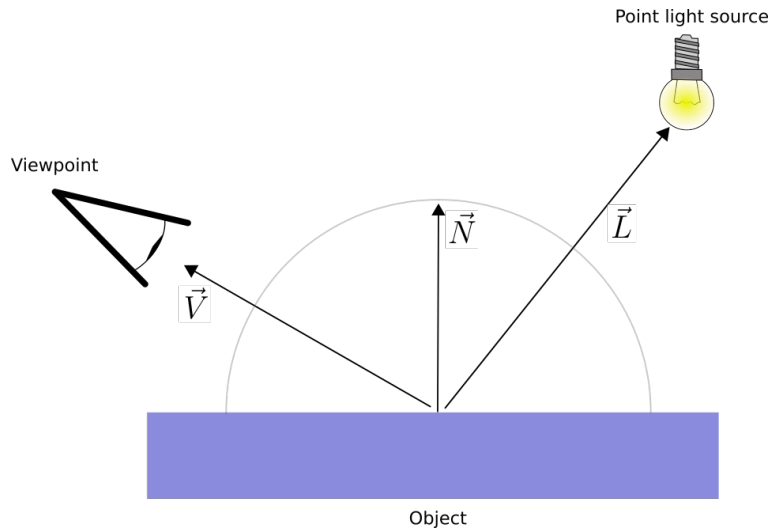


Figure 2: Diffuse lighting in the Phong model

The specular component simulates light rays completely reflected off the object's surface. It depends on the color of the light emitted by the light source and the angle between the view ray and the reflected light ray. The value of the specular component is calculated using this formula:

$$\vec{C}_s = k_s \left(\frac{\vec{V} \cdot \vec{R}}{|\vec{V}| |\vec{R}|} \right)^\alpha \vec{C}_L$$

$$\vec{R} = (2\vec{N} \cdot \vec{L})\vec{N} - \vec{L}$$

where \vec{R} is the reflected light ray, \vec{V} is the vector from the object intersection point to the viewpoint (camera), \vec{N} is the normal vector on the object's surface at the point of intersection, \vec{L} is the shadow ray, \vec{C}_L is the color of the light source, α is the shininess coefficient (defines the sharpness of the reflection) and k_s represents the fraction of the light that the object reflects. The configuration can be seen in figure 3.

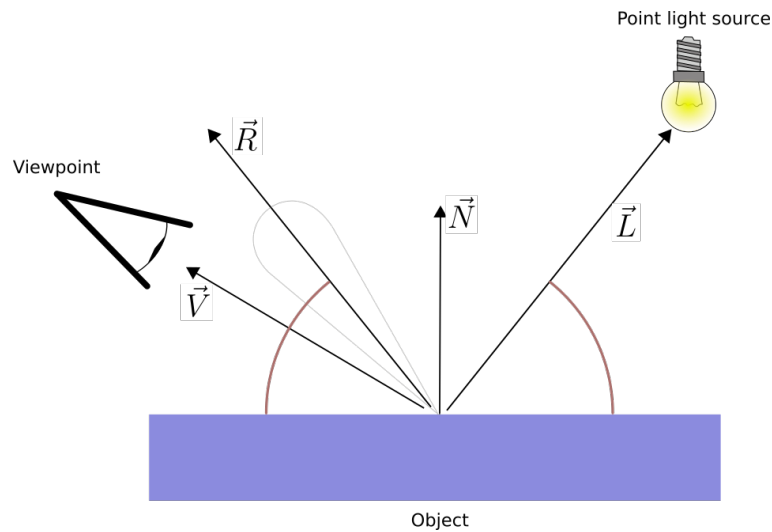


Figure 3: Specular lighting in the Phong model

The coefficients k_a , k_d and k_s need to be in the range $\langle 0, 1 \rangle$, where 0 means none of the light travels towards the viewpoint and 1 means that all the light travels towards the viewpoint.

It is important to note that further in this thesis, ray tracing of objects made only of triangles will be considered. So any larger models and scenes will always be made up of triangles. It is important to note this, because a ray tracer may also support rendering spheres, implicit surfaces (surfaces created using some mathematical function) etc.

1.2.3 Performance

When looking at algorithm 1, it is obvious that for every pixel of the image plane, all the objects in the scene need to be tested for intersection with the view ray in order to find the nearest one (if there is any). Additionally, for every secondary ray (reflection, refraction or shadow ray), all these tests must be performed again. The number of intersection tests that need to be performed increases quickly with larger amounts of objects in the scene, as opposed to the rasterization process, where every triangle is processed at most once and all the pixels it covers are colored in sequence. The amount of necessary intersection tests has a large impact on performance.

In order to achieve a reasonable rendering performance, it is necessary to dramatically decrease the number of ray-object intersection tests that need to be done for each ray. This can be achieved by storing the objects in an acceleration data structure. It is then possible to cull away large regions of space or large amounts of objects before actually starting to test intersections of rays with objects.

1.3 Acceleration data structures for ray tracing

As mentioned in the previous chapter, a significant boost in rendering speed can be obtained by storing the scene objects in an acceleration data structure. This structure is usually represented by a tree where the objects are distributed among its leaves.

In regular data structures, the space of the scene is divided into a regular structure of cells and objects are placed into the cells they overlap. When trying to find an intersection of a ray with an object, only the cells that the ray overlaps are checked. In hierarchical data structures, the scene geometry or space is recursively subdivided into parts. The ray is first checked for intersection with the root node of the hierarchy, which bounds the entire scene. If an intersection is confirmed, the child nodes of the root node are checked, and so on. Until eventually the ray gets to triangles stored in leaf nodes.

1.3.1 Construction schemes

Data structures for ray tracing can be constructed in various ways. Those that employ space partitioning divide the scene's space into regions (and these regions into subregions, in some cases) which always cover the entire space (or the space of their parent region). The objects stored in these structures can be referenced in more than one cell. These data structures can be further divided into *regular* and *hierarchical*.

A different scheme for creating an acceleration data structure for ray tracing is object partitioning. Data structures using this scheme recursively divide the objects in the scene, not space, until either a small amount of objects remains or some termination criterion is met. The most commonly used object partitioning data structure is the Bounding Volume Hierarchy (BVH).

1.3.2 Uniform grid

A uniform grid is a regular data structure and a simple form of space partitioning. The idea is to divide the space occupied by the objects into a regular grid of cuboids of the same size and distribute the objects into these cells. When a ray is traced, only the cells the ray intersects are checked and the objects inside them are tested for intersection with the ray.

The advantage of this method is the computational complexity of its build, the creation of the structure is linear with respect to the number of objects in the scene. A major disadvantage is that rendering performance when using uniform grids drops when objects

in the scene are not distributed uniformly. For some areas of the scene, the ray has to intersect a lot of empty cells, and in other areas, there may be too many objects in one cell. The traversal of a uniform grid with a ray can be seen in figure 4. Only the light gray cells are intersected and therefore only objects C and D are tested for intersection with the ray.

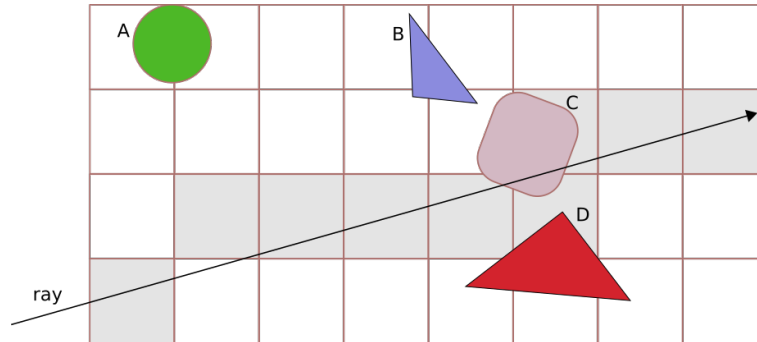


Figure 4: Ray traversal of a uniform grid

1.3.3 k-d tree

Unlike uniform grids, a k-d tree is a hierarchical space partitioning data structure. It is a form of binary space partitioning. At the root level of the hierarchy, all the scene objects are contained in an enclosing axis-aligned bounding box (AABB). The box is then divided into two parts by a dividing plane parallel to one of the three coordinate axes. The resulting cells are further recursively subdivided in this fashion until some criterion is met. A visualisation of three steps of building a k-d tree for 2D objects can be seen in figure 5. During traversal, the root node is checked for intersection first. When it succeeds, its children are tested for intersection with the ray and so on, until the ray eventually reaches the leaf nodes with triangles in them, or doesn't.

There are three important things that need to be repeatedly decided during the construction of a k-d tree:

- Along which coordinate axis to divide the current cell (i.e. which coordinate axis will the dividing plane be parallel to).
- The exact position of the dividing plane along that coordinate axis.
- Whether to further subdivide or stop and declare the current cell as a leaf node.

In order to decide correctly, some measure of the quality of a cell split is needed. Experiments have proven that a so called *Surface Area Heuristic (SAH)* gives correct values of

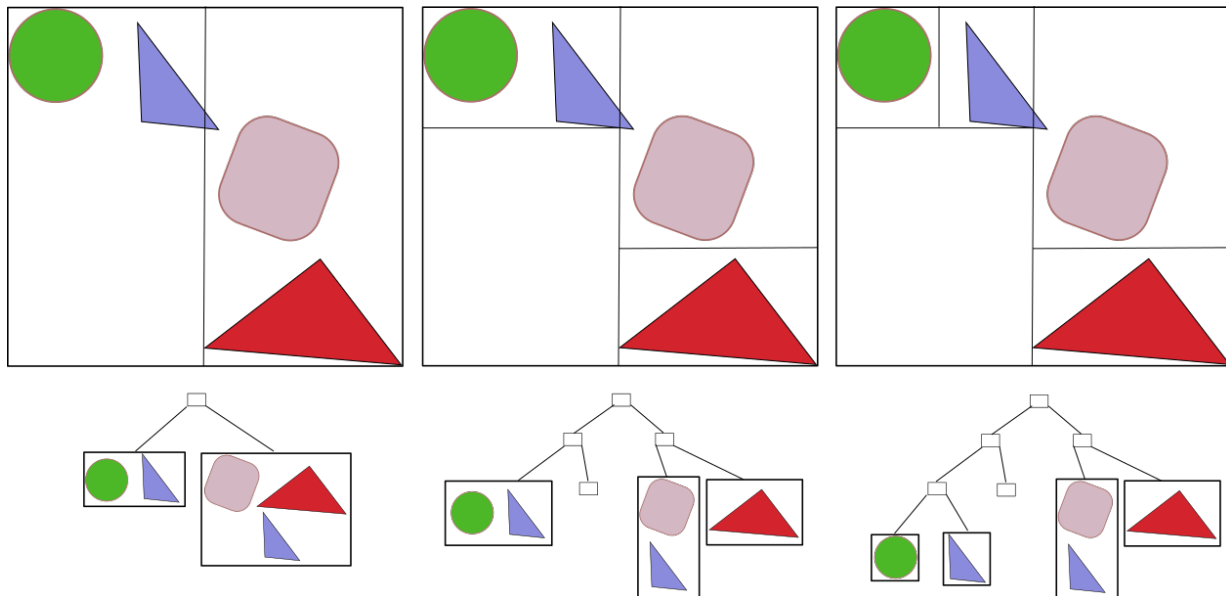


Figure 5: Three phases of a k-d tree construction

quality of tree cells when used for ray tracing. The SAH is based on geometric probability of a ray hitting an axis-aligned bounding box, given a uniform distribution of rays represented as infinite lines. It is calculated from the surface areas of the parent cell and the two new cells created by the split and the costs of the subtrees. The formula for calculating the cost of a k-d tree node using the Surface Area Heuristic is:

$$C(P) = \begin{cases} \frac{SA(V_L)}{SA(V_P)}(C_I + C(L)) + \frac{SA(V_R)}{SA(V_P)}(C_I + C(R)), & \text{if node is inner} \\ C_T N_T, & \text{if node is a leaf} \end{cases}$$

where $C(P)$ is the cost of the current node, $C(L)$ and $C(R)$ are the costs of the child nodes, $SA(V_P)$ is the surface area of the current node's cell, $SA(V_L)$ and $SA(V_R)$ are the surface areas of the cells of the child nodes, C_I is the cost of intersecting an axis-aligned bounding box with a ray, C_T is the cost of intersecting a triangle with a ray and N_T is the number of triangles in a leaf node. It is not important how large the values of C_I and C_T are, as long as their ratio is correct.

When building the tree from the root and subdividing cells (top-down construction), the aim is to minimize this cost when looking for the position of the dividing plane. Since the true cost of a node's children is not known when building the tree from the root node and subdividing nodes, an estimate must be made. The estimate has the following formula:

$$C(P) = \begin{cases} \frac{SA(V_L)}{SA(V_P)}(C_I + C_T N_L) + \frac{SA(V_R)}{SA(V_P)}(C_I + C_T N_R), & \text{if node is inner} \\ C_T N_T, & \text{if node is a leaf} \end{cases}$$

where the only difference is that the true cost of the child nodes is replaced by their cost as if they were leaf nodes.

When trying to find the minimal cost of a node split, there are $6N$ options for N triangles in the node, two for each triangle in 3 dimensions (The triangles must be culled to the volume of the current cell, as they can extend past its boundaries into other cells. Placing the dividing plane in other places than the edges of the triangles' bounding boxes is pointless, as the cost between these positions increases or decreases linearly, and therefore doesn't contain a local minimum. The selection of the axis in which to divide the node can be done in several ways:

- Selecting the axis in a round robin fashion (i.e. x, y, z, x, y, z).
- Selecting the axis along which the node's bounding box is widest.
- Performing the splitting plane search along all three axes and selecting the axis where the split creates the best node cost.

The splitting plane is obviously the most costly operation in the tree building process. There are a number of algorithms that aim to reduce the cost at the expense of accuracy, for example by selecting only a subset of possible dividing plane positions.

1.3.4 Octree

Octree is another type of hierarchical space partitioning data structure. As the name suggests, each inner node of the tree is divided into eight identical child nodes by splitting the node exactly in the middle in each coordinate axis. Nodes are further subdivided while using the similar criteria as in k-d trees, with the cost of the nodes also being computed with the SAH.

1.3.5 Bounding Volume Hierarchy

A bounding volume hierarchy is a hierarchical object partitioning data structure that encloses objects in bounding volumes. These bounding volumes can be of various shapes

and complexity, but the most common type used for ray tracing is the axis-aligned bounding box (AABB). Only AABBs will be considered when talking about bounding volume hierarchies further.

At the root level, all objects (triangles) are enclosed in a bounding box. The set of triangles is then divided into two groups. These groups of triangles are enclosed in their respective bounding boxes and the process continues recursively. The number of child nodes does not necessarily need to be two. A so called Quad-BVH can also be constructed, with a node having four child nodes instead of two. This scheme can be utilized with the use of vector instructions to compute the intersection of one ray with four bounding boxes at once. A bounding volume hierarchy can be built in several ways:

- In a top-down fashion, starting by enclosing the objects in a bounding box and recursively dividing the groups of objects until a termination criterion is met.
- In a bottom-up fashion, starting with every object contained in its own AABB and by combining these so called clusters in pairs of two into new clusters, until only a single cluster remains, which represents the root node of the hierarchy.
- By incremental insertion of objects, for example when the scene data are not available all at once.

Top-down construction

In a top-down construction of a BVH, the selection methods of the axis in which to divide the objects and where to position the imaginary dividing plane are similar to those in k-d trees (chapter 1.3.3), with the exception that there are only $N - 1$ possibilities in each axis for N objects. The surface area heuristic is used in bounding volume hierarchies as well. Three steps of a bounding volume hierarchy construction can be seen in figure 6.

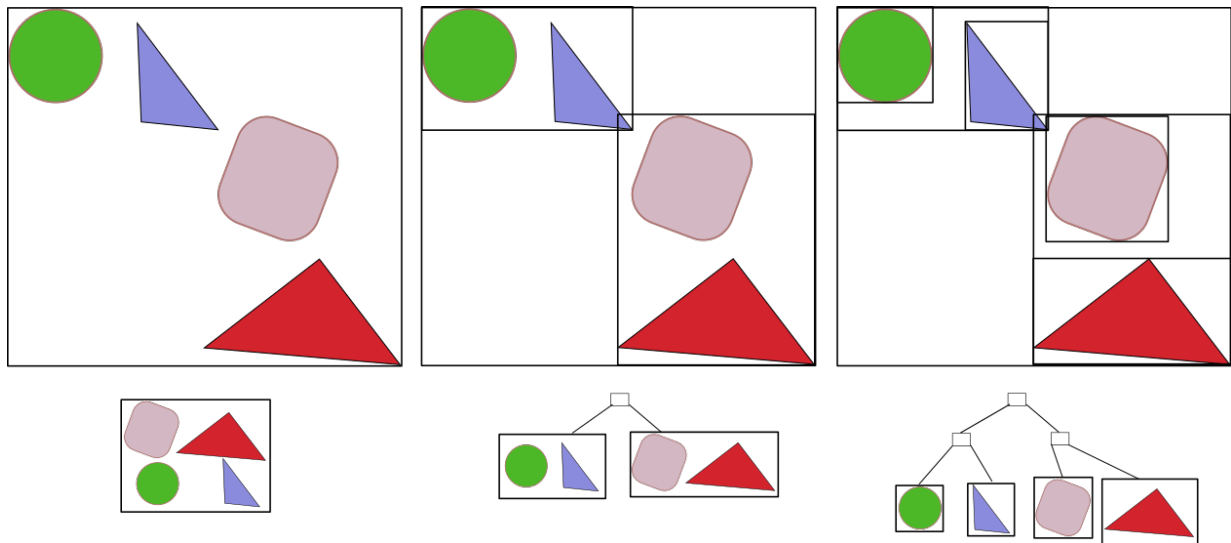


Figure 6: Three phases of a top-down BVH construction

Bottom-up construction

When constructing the BVH from the bottom, all objects are initially contained in their bounding boxes and create leaf nodes. Then, pairs of nodes are combined until only the root node remains. Nodes are combined based on a distance function which will be described later in chapter 2.3. The problem is that there are a lot of possibilities when choosing pairs of nodes to combine. Algorithms using this scheme try to limit the number of possible combinations by separating the nodes into groups and avoiding testing combinations of nodes that would result in new nodes with high costs.

Incremental construction

In an incremental construction of a bounding volume hierarchy, the hierarchy is constructed by inserting triangles one by one, for example when the entire data set is not available at the beginning of the build. Triangles are inserted one by one into the structure and the best place is found for each one.

1.4 Dynamic scenes and interactive ray tracing

Dynamic scenes are a type of 3D scenes in which the geometry or topology changes over time. The changes can be caused by moving or rotating objects, objects changing their shape, objects breaking into smaller objects etc. When acceleration data structures are used in ray tracing, this may pose a problem. When a scene is static, it is possible to build a data structure of its triangles once at the beginning and use this structure during ray tracing, possibly real time. The quality of the data structure can be prioritized at the expense of longer construction time.

For non-static scenes, however, the data structure constructed for the scene at one moment is no longer valid at a different moment, after the scene has changed. This may require that a part of the data structure or the whole data structure be rebuilt.

1.4.1 Dynamic scene composition

Dynamic scenes can be created in one or more of these ways:

- Using rigid objects and applying transformations to them. Rigid objects are represented by a set of triangles that doesn't change over time. Using transformations represented by matrices, they can be translated, resized and rotated within the coordinate system of the scene. An example of a rigid object could be for example a non-deformable car chassis.
- Using soft objects and applying transformations to them. Soft objects are represented by a set of triangles whose geometry might change over time. Additionally, the objects can be transformed the same way as rigid objects. An example could be a piece of deformable cloth controlled by a cloth simulation engine.
- Using keyframe animations, which means having a moving object or scene captured in several time moments, and for each of these moments, having the whole state of the object's geometry saved as a rigid model. By animating these models in quick succession, an illusion of a moving or deforming object can be achieved.

When considering how to manage an acceleration structure of the scene geometry for ray tracing, a number of options are available. One option is to take all triangles of all the objects (both static and dynamic) and build the data structure again each frame (or every time some change in the scene occurs). Another option is to separate static and dynamic geometry and only rebuild the dynamic geometry.

Keyframe animation objects require an acceleration data structure for each of their keyframes. They can be all built at the beginning without the need to rebuild them again,

at the expense of severely higher memory consumption. Or they can share the same memory space for the data structure and after each keyframe change, the data structure for the particular keyframe must be built again.

For soft objects which change their shape over time, there is no way to circumvent the need to rebuild (or at least refit) the data structure after the object's shape is modified.

For the cases when rebuilding the data structure of a set of triangles often and fast is unavoidable, a slightly different approach of building data structures is needed. Algorithms suited for this need often sacrifice data structure quality in favor of performance and use approximations and heuristics to get as close to a high-quality solution as possible.

1.4.2 **Object BVH**

For rigid objects, an interesting approach is possible. For the triangles of each object, an acceleration data structure is built only once in the object's model space. Then, for each movement, rotation or size change, a new transformation matrix is available. Using this transformation matrix, only the bounding box of the root node of the data structure is transformed into world coordinates, not all the geometry. This transformed bounding box is then wrapped in another bounding box aligned with the coordinate axes of the world coordinate system. When this is done for all the rigid moving objects, a data structure is built above the new bounding boxes. When a ray is traced through the scene, it first traverses this data structure of objects. When it hits an object's bounding box, it is transformed by an inverse of the object's transformation matrix. It is then traced in the coordinate space of the triangle mesh. The data structure of the object's triangles therefore never needs to be rebuilt and the object can still freely move within the scene. This approach is used in the implementation part of the thesis and is further described in chapter 3.2.1.

1.5 Android operating system

Android is an operating system used mainly in handheld mobile devices such as smartphones and tablets. The system is based on Linux. Originally created by Android, Inc., it is now owned and developed by Google. The first version of the system was released in 2007.

1.5.1 Android application development

Applications for Android devices are developed using the Android Software Development Kit. An application can either be run on a real device or in an Android device emulator. The main language Android applications are written in is Java, but using native code written in C or C++ is also possible with the use of Java Native Interface (JNI) and the Android Native Development Kit (NDK) supplied by Google.

1.5.2 Android devices

Android devices range from smart phones and tablets to smart watches and televisions. The focus of this thesis are devices which are often used as a desktop computer where mobility is key, therefore phones and tablets. These devices are today already powerful enough to run 3D games and application with advanced graphics. As Android gaming is a large market, it is the aim of this thesis to examine if these applications can be further enhanced by using ray tracing and still be run on contemporary devices.

2 Research and other solutions

2.1 Android application development

2.1.1 Android device

The device with the Android operating system used in the implementation and testing parts of this thesis is the Samsung Galaxy S4 (model i9505), a high-end smart phone from Samsung released in 2013. The device uses the Snapdragon 600 SoC (System on Chip) with the ARM CPU architecture (ARMv7 instruction set). The system uses 2 GB of 600 MHz dual-channel LPDDR3 random access memory. It also has a separate graphics processor - the Qualcomm Adreno 320, which shares memory with the CPU.

The CPU and GPU specifications of the Snapdragon 600 SoC are shown in the tables that follow (data taken from [1] and [2]).

CPU (Krait 300)		
Parameter	Value	Units
# of cores	4	-
Core frequency	1900	MHz
Level 0 instruction cache per core	4	KB
Level 0 data cache per core	4	KB
Level 1 instruction cache per core	16	KB
Level 1 data cache per core	16	KB
Level 2 instruction cache per core	2048	KB
Level 2 data cache per core	2048	KB

Table 1: Krait 300 CPU specifications

GPU (Adreno 320)		
Parameter	Value	Units
Core frequency	400	MHz
# of rendering pipelines	16	-

Table 2: Adreno 320 GPU specifications

2.1.2 Programming languages and tools

Java

The default language used when programming applications for Android devices is Java. The Android Java API contains everything that is needed to create an application including its structure, user interface, communication with various sensors the device has, etc. Because both ray tracing and building of data structures for ray tracing are performance-hungry tasks, a lot of effort must be put into optimizing the code. Since Java doesn't offer a lot of options of low-level optimizations and custom memory management, other means of programming applications for Android were sought.

Native code development

After making further investigations, it turned out that it is possible to write native code for Android in C or C++. Using the Android NDK (Native Development Kit), which is an official toolset for developing native code for Android, it is possible to write parts of applications (or even entire applications without the use of Java) in C or C++.

This means that parts of an application that are computation-expensive can be written in native code and the computed result can be used in the Java part of the application. It is, however, not recommended to use native code extensively for all tasks or write an entire application in native code, as it has to be compiled for each Android device architecture separately, whereas Java applications run across all devices in a Java Virtual Machine. Also, the complexity of the application would be greater, since the Java API contains lots of classes and tools that allow for simple application state management, rendering etc.

The Android NDK contains several C and C++ libraries, including for example helper libraries for debug logging, manipulating Bitmap objects from Java, and a port of the C++ standard library. The native code has to be separately compiled before packaging and launching the application from the IDE.

Java Native Interface

The bridge between the Java part of an application and the native code is provided by JNI (Java Native Interface), which is a standard interface used on other platforms as well. This interface provides means of communication and data transfer from Java code to C/C++ code and vice-versa. In order to perform native code, a method declaration must be written in the Java part of the application. This method must have the keyword *native*. The method is then mirrored in the native code and its body is implemented there. An example of a Java native method and its native code counterpart follows:

```
Java:  private native int doSomething(int number)
C++:   JNIEXPORT jint JNICALL Java_path_to_class_ClassName_doSomething
      (JNIEnv* env, jobject thiz, jint number)
      { return 0; }
```

The JNIEnv object contains all functions necessary for the Java-native communication, for example conversion from Java to native arrays. The thiz object is a reference to the object the method was called on. If the method is static, this reference is of type jclass and refers to the class this method belongs to. Other arguments mirror the arguments of the Java declaration of the function.

2.1.3 Parallel computation

Since both ray tracing and construction of hierarchical data structures of objects for ray tracing can benefit from parallelization of tasks, options of parallelization on the described Android device were studied thoroughly.

OpenCL

The Adreno 320 GPU on the Snapdragon 600 System on Chip is one of the first Adreno graphics processors to support OpenCL (in version 1.1). OpenCL (Open Computing Language) is a framework for parallel computation on various devices, including CPUs, GPUs, DSP (Digital Signal Processor) etc. The functions to be run in parallel on the devices are written in a C99-like programming languages, and their running and data management and transfer is done through the OpenCL API.

OpenCL is not officially supported by Google on Android devices, and on the Galaxy S4, is therefore only unofficially supported by Qualcomm. It took some time to set it up on the device and run it, as there wasn't much information and documentation during the time of working on this thesis. It was actually necessary to pull the OpenCL library *.so file from the device in order to compile the application.

A simple ray tracer was written in OpenCL and run on the Adreno 320 successfully, but after trying to implement a BVH traversal on the GPU, serious problems were run into. The performance drop was catastrophic, sometimes even freezing or restarting the device. The source of the problems may have been non-coalesced access to the global memory of the device (where each processing unit accessed different areas of the memory). After no solution was found for these problems, computation on the GPU was abandoned and left for future investigation.

RenderScript

RenderScript is an Android framework for parallel computation. It is similar in nature to OpenCL, with the exception that the running of parallel tasks and data management is done in the Java language. Also, the framework decides on its own which computation device (CPU or GPU) to use and this choice cannot be altered. This option was not tested in any way.

POSIX threads

It was found that the NDK supports POSIX threads through the use of the `pthread.h` header file. It is therefore possible to utilize the four processor cores of the Krait 300 CPU. This is also the means of parallelization that is used later in chapter 3 for both rendering and data structure building.

ARM NEON instruction set

It was found out that the device's CPU also supports the ARM NEON instruction set. It is a set of vector instructions of the ARM architecture similar to the SSE instructions for Windows systems. These are also used in chapter 3 and provide a large boost in performance of the rendering system.

2.2 Ray tracing on Android

When searching for articles discussing the use of ray tracing in Android applications, only a few applications with little to no documentation were found on the web and Google Play Store.

The first application for the Android platform is "Raytracer Demo" by Nic Dahlquist [3]. The application displays several spheres with reflections and an environment map. The user can interact with the spheres using the device's touch screen (move them in a plane parallel to the projection plane). There are also options to turn off the environment map and reflections. The source code is available for free. The application is implemented using a single Android Activity Java class. The rendering is implemented in C++ for multiple threads run on the CPU. There are just spheres present in the demo, no triangles, and a bounding volume hierarchy is built using the sphere data each frame. The communication between the user interface written in Java and the rendering code written in C++ is done using Java Native Interface. That includes passing the Bitmap object to the native side

of the application, which then accesses the raw pixel data associated with this Bitmap object and writes the correct colors to it. The rendering parameter changes done by the user are also transferred using the Java Native Interface. The application runs at about 6 frames per second on the device mentioned in chapter 2.1.1 with all the features on. The resolution of the rendered image cannot be selected and seems to be equal to the native resolution of the device's display, which in this case is 1920 x 1080. Framerates of 6-10 frames per second are reported by other users with various other high-end devices.

The second application is "Multi-core CPU Raytracing" by Wizapply Project [4]. The application lets the user select the resolution factor (4, 2, 1.5 and 1) which divides the resolution, resulting in smoother animation for lower resolutions. The user can also specify the number of threads the application is allowed to use (assuming a multi-core processing unit is present on the device). A number of spheres and a floor are then rendered, while each of the spheres follows a predefined trajectory. The framerate highly depends on the selected resolution factor and number of threads, but also each added/removed sphere. The source code for this project is available for purchase from the creators. The application is also available for Mac OS X and Windows.

A similar application by the same author is "Real-Time GPU Raytracing" [5]. It is very similar except it allegedly runs on the graphics processor of the device. There is no source code available and thus no way to confirm that.

2.2.1 Ray tracing dedicated hardware

Several articles can also be found on proposals of ray tracing oriented mobile hardware. One of them is [6], in which Lee et al. propose a hardware unit with two ray-AABB intersection testing units. They then compare the system with existing hardware-based approaches at real time ray tracing.

2.3 Fast construction of acceleration data structures

With computer hardware already fast enough for real time ray tracing, focus is being set on ray tracing dynamic scenes. That requires algorithms for fast rebuilding or updating of acceleration data structures containing the scenes' changing geometry. Multiple different types of data structures are suited for ray tracing and for particular scene types. K-d trees have shown the best performance in ray tracing so far, but their construction is not as simple as in bounding volume hierarchies because of the high number of potential dividing plane position at each level of the hierarchy during a top-down build. Bounding volume hierarchies can simply be refitted from the bottom up for minor changes in scene geometry.

Larger changes however result in degradation of the hierarchy’s quality and a full rebuild is necessary.

Because of the failure to successfully use the device’s GPU for parallel computation described in chapter 2.1.3, a decision was made to utilize the full potential of the CPU instead, possibly using multiple threads and vector instructions, where applicable. Further research in this and the next chapter is therefore focused on algorithms devised for a CPU.

In [7], two bounding volume hierarchy construction algorithms are proposed. They are both based on the process of agglomerative clustering. The idea of agglomerative clustering is to start with an array of elements (clusters) and make pairs of these elements into new clusters. A new cluster replaces the two old clusters it is made of and the process continues until only one cluster remains. A naïve version of the process is shown in algorithm 2. The problem of the naïve approach is its $O(N^3)$ computational complexity.

```

C ← singleton clusters made from triangles;
A ← null;
B ← null;
while size(C) > 1 do
    bestValue ← ∞;
    foreach cluster Ca in C do
        foreach cluster Cb in C do
            if Ca ≠ Cb and d(Ca, Cb) then
                bestValue ← d(Ca, Cb);
                A ← Ca;
                B ← Cb;
        D ← new cluster(A, B);
        C ← C - {A} - {B} + {D};
return C;

```

Algorithm 2: Naïve version of agglomerative clustering

where $d(C_i, C_j)$ is a symmetrical distance function which denotes how close to each other the clusters are. The function can have different meanings in different contexts. In this case (ray tracing), the authors follow the research in [8] by making this function equal to the surface area of the bounding box that contains the bounding boxes of the two clusters C_i and C_j .

The first algorithm aims to reduce the computational complexity of finding the best pair of clusters to combine by using a simple low quality k-d tree to store the active clusters

and a heap to store computed distance function values for cluster pairs.

When searching for the best pair of clusters to combine, the k-d tree is traversed. An inner node contains information to calculate the lower bound on the distance value function for the elements in its subtree. This allows for culling of subtrees whose traversal would not yield a better cluster combination candidate. The k-d tree itself is built using a simple top-down construction scheme where each node is split in its spatial median along the axis in which the node has the longest extent.

The heap stores the best match for each cluster along with the value of the distance function for the pair of clusters. They report an improvement of performance from the naïve quadratic to almost linear.

The second proposed algorithm does not use the heap. A random cluster A is selected at the beginning and a best match B is found for this cluster. If the best match of B is A, a new cluster is formed, otherwise there is a better match for B (C), and the same process continues with B and C.

Another algorithm for bounding volume hierarchy construction for ray tracing based on agglomerative clustering is proposed in [9] by Gu et al. The article addresses the agglomerative clustering algorithms proposed by Walter et al. in [7] described before. It is noted that the build times of the proposed algorithms are still higher than those of top-down bounding volume hierarchy builders based on binning.

Gu et al. therefore propose a different approach of reducing the number of possible cluster combinations during build. They first take all the scene triangles and enclose them in their bounding boxes, creating so called singleton clusters. For each of the clusters, a morton code of its bounding box center is calculated. This places each cluster at a position on a Z-order curve (or space-filling curve) that covers the whole scene space. The clusters are then sorted by the morton code assigned to them.

The algorithm then has two phases, a top-down division phase and a bottom-up clustering phase. During the division phase, the set of singleton clusters is recursively divided into two parts using the next digit of the morton code. In case of running out of digits, the singleton cluster subsets are further halved. Once a predefined number of clusters remains in one of the branches, the second phase begins. The clusters are clustered until only a predefined number of clusters remains. The resulting clusters are merged with resulting clusters from the sibling branch, and the process continues up, until only one cluster remains. The number of clusters that is created at each level is controlled by a so called *cluster count reduction function*.

The results presented in [9] show similar or better performance and tree quality when compared to a binning top-down builder.

A fast top-down building algorithm based on binning is presented in [10]. The idea of splitting an inner node is to first obtain the bounding box enclosing the centers of the bounding boxes of triangles contained in the node. This bounding box is then split into several bins along the axis of its longest extent. Triangles are then distributed based on the coordinates of their bounding box centers into these bins. For each of these bins, a bounding box is kept containing all triangles in that bin. The bins are then iterated over and the surface area heuristic is evaluated for planes between the bins, counting the bins on the left and right from the plane into the left, resp. right subtree.

The next step after locating the best split is to reorder triangles into two subarrays for the subtrees. This is done by a linear sweep over the triangles in the current node and organizing them either at the beginning or end of the new ordered subarray.

The subdivision is terminated when either a low number triangles in a subtree has been reached, when the bounding box of centers is too small or when the estimated cost of a subdivision is larger than the cost of a leaf.

Parallelization of this algorithm is also proposed, with two phases described: a horizontal and vertical phase. During the horizontal phase, the tree subdivision process runs in one thread and the binning process for finding the best node split is parallelized. In the vertical phase, enough subtrees for further subdivision have been created to allow for parallelization of subtree builds.

This algorithm is implemented as part of this thesis and the implementation is further described in chapter 3.4.

2.4 Fast ray tracing

Rendering dynamic scenes would not be possible without optimizing the rendering algorithm itself. Research of performance tweaking techniques was therefore made. In [11], tracing large packets of rays is discussed and implemented for primary and shadow rays. During the implementation part of this thesis, an effort was made to trace packets of rays, but led to no performance gains, strangely. In the article, performance is compared to tracing 2x2 SIMD rays using vector instructions, and is shown to be better. Using vector floating point instructions and modifying the rendering algorithm to process four rays at once (2x2 SIMD ray) can also improve performance. This approach is used in the implemented ray tracer and actually brought a huge rendering performance improvement over tracing single rays.

3 Design and implementation

In this chapter, the application and algorithms implemented as part of this thesis are described. In the first part, the created rendering system is described, including tested, used and unused optimizations researched in chapter 2. In the second part, the selected BVH building algorithm is described, including implementation and optimization details.

3.1 Used technologies

As mentioned in chapter 2, efforts to utilize the Adreno 320 GPU of the available Android device were abandoned after encountering a problem with performance probably caused by random global memory accesses. Focus was thereafter set on the CPU and maximal utilization of its resources. Using the Native Development Kit along with utilizing POSIX threads and ARM NEON vector instructions mentioned in chapter 2.1.3 was a clear choice.

3.2 Ray tracing library

The core part of the project, a rendering system, is designed as an Android library module that can be integrated into any Android Java application and used for rendering. It takes a reference to a Bitmap object (from the Android API) as input, accesses its raw pixel data as an array of unsigned integer values, and renders into it whatever scene the user has set up. The library also offers a number of methods for the loading of geometry, lighting, camera parameters and object data.

The library contains one Java class, `RayTracerLibrary`. This class serves as an interface to the native part of the library and provides the above mentioned access to the rendering resources. It does not need to be instantiated, access is static.

The rest of the library is native C++ code. The `raytracer.cpp` file contains the necessary JNI counterparts of the library's native functions. These serve for the communication of the Java part of the library with the native code. Scene and camera data are all encapsulated in an instance of the `RayTracerContext` class. This gives the possibility to have several different independent rendering environments.

3.2.1 Object BVH

Before explaining the scene management functions, it is necessary to mention the way scenes are managed in the library. It is important to know the difference between these two things:

- A **mesh**, represented by the `RigidMeshTemplate` class, is a set of geometry and topology information, i.e. triangle data including vertex coordinates. It manages its own hierarchical data structure.
- An **object**, represented by the `RigidObject` class, which can be thought of as an instance of a mesh. It references a `RigidMeshTemplate` object and a `Material` object, which gives information about the object's material properties. It further contains information about the rigid transformation currently applied to the object in the form of a 4x4 matrix.

Using this mechanism, it is not necessary to have multiple copies of the same mesh in memory for cases when there is to be more than one of the objects that the mesh represents in the scene (for example multiple cars of the same type on a road). These mesh copies would also have to be transformed in space after each movement of the object represented by the mesh. Instead, there are only multiple objects referencing the same mesh and each has its own transformation matrix.

For each of these instances of the `RigidObject` class, the bounding box of the mesh they reference is transformed from its model space to the world space using the object's rigid transformation and contained in a bounding box that is aligned with the world's coordinate axes.

After all the objects' bounding boxes in the world space are constructed, a bounding volume hierarchy is constructed above them. When tracing rays through the scene, the rays are first tested against this object bounding volume hierarchy. When an intersection is found, the ray is transformed using the *inverse* of the particular object's transformation and is traced in the coordinate space of the mesh that the object references. The acceleration structure for the mesh can therefore only be built once for each mesh and doesn't require any rebuilding. Figure 7 shows the described process.

3.2.2 Dynamic objects

Objects can reference more than one mesh. This allows for animated objects with so called keyframes. Each keyframe represents a model or scene in some state, and when these keyframes are rendered in quick succession one after each other, they give the illusion of a

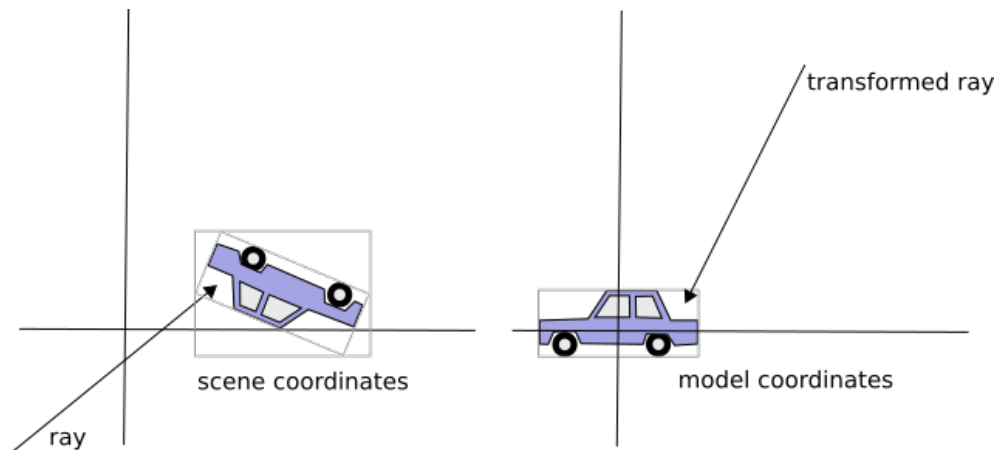


Figure 7: Tracing a ray through the object bounding volume hierarchy

moving or animated model.

In the current version of the library, each mesh has its own acceleration data structure, so it is not possible to save memory by having only one data structure for an animated object and rebuild the data structure for every keyframe during animation. This functionality could, however, be easily added. For the purposes of performance testing, it is possible to rebuild the data structure of a mesh before each rendering takes place.

3.2.3 Scene management

Apart from meshes and objects, the `RayTracerContext` class also manages materials and point lights. Materials are assigned to objects, not meshes. It is therefore possible to instantiate a mesh using different materials. Point lights are the only type of light supported at the moment, and have the *position* and *light color* parameters.

For dynamic scenes, the `RayTracerContext` also has a transform queue for objects. When an object transformation is requested from the outside, it is put on this queue and the queue is processed before the next rendering occurs. This way, transforming all objects before each render is avoided when their transformation matrix has not changed from the last rendering pass.

The class also manages an instance of the `Camera` class. This class contains necessary information the ray tracing function needs to render the scene. That includes the position of the camera in the world space, the direction the camera is pointed in, the horizontal field of view (FOV) of the camera, the projection plane resolution, and some other parameters

further described in section 3.2.5.

3.2.4 Library interface

The `RayTracerLibrary` Java class provides a number of methods to communicate with the above described `RayTracerContext` C++ class. The methods for scene management are described below:

- **changeBitmap** accepts a Java Android Bitmap object as a parameter and changes the Bitmap object the ray tracer will render to. In case this function is never called, the library creates its own Bitmap with the resolution of 1024 x 512 pixels during the initialization of the Java library class.
- **addMaterial** adds a new material to the library. The properties of the material are given as parameters to the function. They include the color of the material, diffuse and specular constants, shininess coefficient, transparency and index of refraction (in case the material is transparent)
- **addPointLight** accepts as parameters the position and color of the light and adds this light to the library's array of point lights.
- **createMeshTemplate** takes the number of triangles this mesh will have as a parameter and allocates the needed memory for it.
- **addTriangle** takes as parameters the ID of the mesh the triangle should be added to, three vertex coordinates and three normal vectors. In case a mesh with the given ID is not present in the library yet, the triangle is not added.
- **createObject** accepts two parameters: number of meshes the object will reference (more than one for animated meshes) and material ID. In case the material with the given ID is not present in the library yet, the object is not created.
- **addMeshToObject** takes an ID of an object and an ID of a mesh as parameters and adds the mesh to the object, if both the mesh and the object are present in the library already.
- **selectObjectMesh** takes an object ID and a mesh index as parameters. This function is for animated objects that reference multiple meshes and is used to select which of the several meshes the object will currently use.

Next, there is function `transformObject` that accepts an object ID and a transformation matrix as parameters. If the object exists, its transformation is put on the above mentioned transformation queue in the `RayTracerContext` object along with the new transformation

matrix and is then transformed before the next rendering of the scene. The interface also contains functions to move and rotate the camera.

Finally, the class contains a `rayTraceScene` method that renders the scene with the current camera parameters and object and light transformations into the supplied Bitmap object.

3.2.5 Rendering

The core function of the library is to render the scene configuration using ray tracing and the constructed acceleration data structures. This is the part where POSIX threads and NEON vertex instructions are utilized. The `arm_neon.h` header provides several new data types, of which these are further used:

- `float32x4_t`, a vector of four floating point numbers conforming to the IEEE 754 standard
- `uint32x4_t`, a vector of four unsigned integer numbers
- `int32x4_t`, a vector of four signed integer numbers

These vector data types store the four primitive data type variables in consecutive memory. This fact is exploited and an auxiliary C++ union is defined:

```
typedef union {
    float32x4_t flt4;
    float fltx4[4];
    uint32x4_t uint4;
    int32x4_t int4;
    int32_t intx4[4];
    uint32_t uintx4[4];
} float_uint32x4_t;
```

This union allows to access the floating point data as integers and perform bitwise operations on it (bit clearing etc.).

There are several ways to use vector instructions in ray tracing. A single ray can be traced and tested for intersections against four bounding boxes or triangles at the same time. Or, four rays can be traced at the same time intersecting only one bounding box or triangle at a time. The latter is what was chosen for the rendering in this library. For that purpose, two new structures were created:

3.2 Ray tracing library

- A `SIMD_Vector` structure, which, instead of one three-dimensional vector, contains four. There are a total of three `float32x4_t` variables, one for each dimension. Each of them holds the coordinates of four different vectors in that particular dimension.
- A `SIMD_Ray` structure that holds three `SIMD_Vector` structures inside: a ray origin, direction and inverse direction (a vector with components calculated as inverse components of the direction vector). Of course, all three structures hold data for four independent rays.

where SIMD stands for 'Single Instruction, Multiple Data'. Instead of tracing single rays from the camera, these SIMD rays are traced, which for coherent rays means tracing four rays is a little more or equally expensive as tracing a single ray. The same is applied to shadow rays (rays traced from light sources towards objects intersected by view rays).

Since the rendering can be run in multiple threads, the image is divided vertically into narrow strips. The threads' work on these strips is interlaced to allow for better work distribution among the threads, otherwise their load might become unbalanced.

In the first step of the rendering process, the projection plane is set up in the world space and the first ray packet is computed. During the rendering process, the packet is then moved across the projection plane by adding `pixel_step_x`, `pixel_skip_y` and `row_step_back` vectors to it. The whole setup is shown in figure 8.

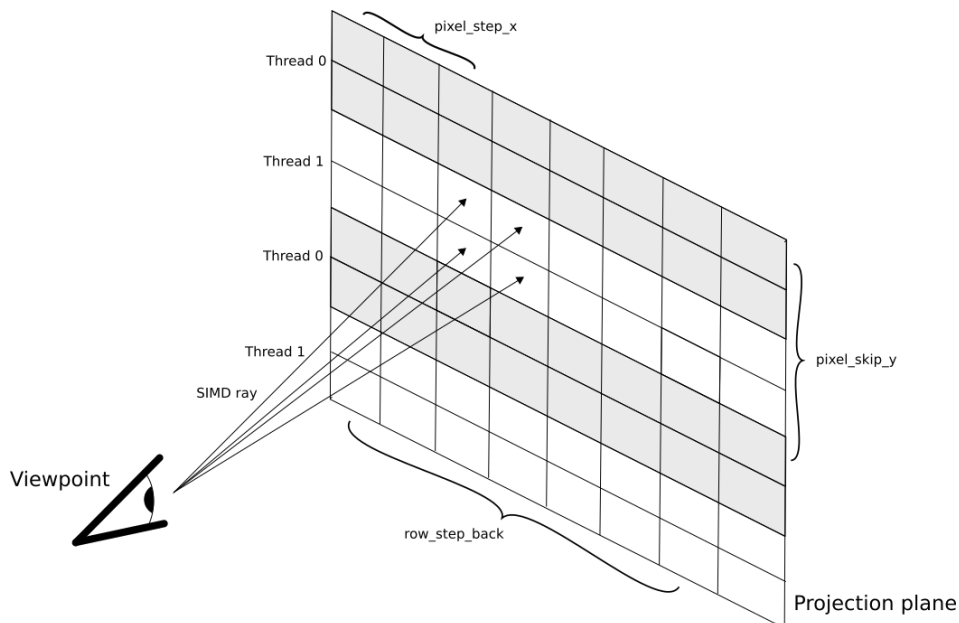


Figure 8: Projection plane sampling using a SIMD ray and multiple threads

For each position of the SIMD view ray, the color of the four pixels is set to $(0, 0, 0)$ (black). The object bounding volume hierarchy (described in chapter 3.2.1) is then traversed with the SIMD ray. If an object is hit by at least one of the four rays, the SIMD ray is transformed using the inverse of the object's transformation matrix and is further traced through the bounding volume hierarchy of the object's mesh. For each of the four rays, ID of the nearest intersected triangle and the object it belongs to is stored, as well as the distance to the respective object, if there is any.

Then, if at least one of the rays is valid (hit a triangle), the intersected triangles are loaded from memory, along with normal vectors at the intersection points, which are calculated using linear interpolation. For the invalid rays (did not hit any triangles), dummy triangles are loaded, as the computation still performs on all four rays simultaneously and triangle data need to be present. Materials of the triangles are loaded as well from the intersected object data.

After the data is loaded, all point lights are iterated over. For each point light, a SIMD ray is constructed that contains the rays from the point light to the four intersection points. The object bounding volume hierarchy is then traversed with this SIMD light ray. If the light illuminates at least one of the objects the original view SIMD ray intersected, the diffuse and specular color components are calculated (with invalid rays using dummy values again).

After each point light is processed, the computed colors of the four pixels are cleared using the bit mask of the valid rays (the rays that intersected a triangle). The cleared pixels' color is set to the color of the background.

If secondary rays are enabled (reflections and/or refractions), then, based on the object's material, a reflected or refracted ray is fired, and the whole process is repeated, adding color to the four pixels. Secondary rays are not fired by calling a recursive function, but are put on a stack instead. Each stack item contains information about the ray, recursion depth and the coefficient by which the color collected by this ray should be multiplied when adding it to the final pixel color. For example, if an object reflects 50 % of light and a ray is reflected off this object, the resulting color is multiplied by 0.5. This way, unnecessary function calls are avoided. When the maximum allowed recursion depth is reached, no more secondary rays are fired. When the stack is empty, the algorithm proceeds to the final step, coloring the pixel in the bitmap.

Finally, the Bitmap object's pixel array is accessed and the corresponding pixels are colored. The SIMD view ray is then translated to the next position on the projection plane and the process is repeated.

3.2.6 BVH traversal

As SIMD rays are used for the acceleration of rendering, the algorithm for the traversal of rays through the bounding volume hierarchy needs to be modified accordingly.

For both primary and shadow rays, the traversal is implemented using a stack. When at least one ray of the four rays in the SIMD packet intersects a bounding box, one of its child nodes is put on this stack (only if at least one of the rays intersects its bounding box) and one is selected as the current node (again, only if at least one of the rays intersects its bounding box). If none of the two child nodes are intersected by any of the four rays, the traversal stack is popped. If the ray enters a leaf node, triangles are intersected and intersections noted. When the traversal stack is empty, the traversal is over. Traversal of a tree branch can be ended prematurely when none of the rays intersects a node's bounding box or when the intersections of the rays with the bounding box are further away than the nearest found triangle intersections.

Unlike for primary rays, shadow ray traversal can end prematurely in a different way. Throughout the traversal, a value is stored for each of the four rays indicating whether they are occluded and do not light up the given triangles intersected by the primary rays. When all four rays have this value false, the traversal can be ended prematurely.

3.3 Testing Java Android application

To test the implemented ray tracing library, a simple Android application was written in Java. The application consists of the `MainActivity` class that extends the `Activity` class, which is a basis for an application's 'window' on Android. Into this window, a `View` object is embedded. A `View` allows the manipulation of the `Bitmap` object that is drawn to the screen. The `View` gets refreshed in regular intervals fast enough, so its `onDraw` method is used to call the library's rendering method. That results in a fluent experience, provided the library is fast enough with the rendering and data structure rebuilding.

There are also 10 buttons added, six for the movement of the camera and 4 for its rotation. Animated objects (objects with more than one mesh assigned) are encapsulated by the `AnimatedObject` Java class, which changes the selected mesh of the object in intervals of selected length. This allows for the animation of the objects.

The `View` class also contains methods containing scene definition and updates (hard-coded for the purposes of testing in chapter 4).

3.3.1 OBJ file parser

A necessary part of the testing application is a parser of meshes in the OBJ format. The files are packaged with the application and then access using a resource locator. A simple parser was implemented for the purposes of this thesis. It accepts a file stream and parses the files line by line. At this point, vertex coordinates, normal coordinates, triangles and quads are supported. The parser directly calls the library's functions for adding meshes and triangles.

3.4 BVH construction

The BVH building algorithm based on the surface area heuristic which uses binning during the node splitting phase described in [10] is implemented within the ray tracing system.

Since the upper bound of the number of nodes of a bounding volume hierarchy is known ($2n - 1$ for n triangles), it is possible to allocate the space beforehand and avoid dynamic memory allocation throughout the building process. An array of node objects is therefore allocated before the build to serve as an object pool. A single BVH node has the following structure:

```
struct BVH_node {
    AABB box;
    union {
        int firstPrim;
        int leftChild;
    };
    union {
        // bits 0 - 30: count/index, bit 31: 0 - inner node, 1 - leaf
        int numPrim;
        int rightChild;
    };
};
```

The `leftChild` and `rightChild` variables are used for inner nodes and are indices pointing into the node array to the left, resp. right child node of a node, `firstPrim` and `numPrim` are used for leaf nodes, where *prim* is short for a geometric primitive (triangle in this case). The `firstPrim` variable denotes the first triangle from the triangle array that the leaf node addresses, `numPrim` denotes the number of triangles in that leaf node. To differentiate inner

nodes from leaf nodes without additional memory consumption, the most significant bit of `numPrim/rightChild` is 0 for inner nodes and 1 for leaves. Before operating with a leaf, the bit has to be cleared using the `0x80000000` bit mask. The presence of the `union` constructs is not necessary, it only serves for better orientation in construction and traversal code. `AABB` represents a bounding box by six floating point values, and therefore takes 24 bytes of memory. The whole BVH node consumes 32 bytes of memory.

As mentioned in the previous paragraph, triangles are stored and supplied to the building algorithm in an array. However, they are not accessed directly, but via an auxiliary array of triangle indices (triangle reference array). This allows for minimal data movement when for example sorting or otherwise manipulating the triangle array.

The construction algorithm implemented is a top down builder based on binning described in chapter 2.3. It is a top-down builder which aims to reduce the number of possible options of dividing an inner node of a bounding volume hierarchy by projecting triangles into bins and only testing several split options using these bins.

The algorithm terminates dividing nodes when the number of triangles drops below a certain threshold. The process is shown in pseudocode in algorithm 3.

The bitwise operations needed to distinguish inner and leaf nodes are left out of algorithm 3 for the sake of readability. The algorithm is not recursive and uses a stack instead. As can be seen, data movement is minimized by directly continuing the build with one of the child nodes instead of storing them both on the stack and then popping the stack right after.

All the expensive computations happen in the `getDividingTri` function, which returns the index of the first triangle that should go into the right child from a subarray of triangle indices. The selection is done using the surface area heuristic and binning to estimate the best way to divide the current node. The function accepts a part of the triangle reference array as parameter (index of the first triangle and number of triangles), along with the area of the current node's bounding box, and references to both the left and right child's bounding box, which are filled in this function during the process of finding the best split.

The idea is to divide the bounding box of triangle centers of the current node into several bins of equal size. The division happens along the widest axis of the bounding box enclosing the *centers* of the triangles' bounding boxes, not their vertices. Then, triangles are distributed into these bins based on the coordinates of their bounding box centers. For each bin, two bounding boxes are kept, `leftBox` and `rightBox`. Initially, they are both sized to enclose the triangles in the particular bin. After all the triangles are distributed, the array of bins is iterated over in both directions. When going in one direction, the `leftBox`

of each bin is resized to accommodate all `leftBoxes` of the bins before it. In the other direction, the same is done with the `rightBox` of each bin. This way, bounding boxes of each potential left and right child are obtained for several splitting plane positions. During the iteration, the triangle counts `leftCount` and `rightCount` are updated along with the boxes.

After evaluating the bounding boxes of bins, the array of bins is iterated over once more and the best split is selected based on the surface area heuristic, which can now be calculated because both the bounding box areas and numbers of triangles in the left and right children for each particular configuration are known.

The last step is to sort the triangles in the subarray based on which child node they belong to, left or right. This is done by one iteration over the triangle reference array. A copy of the subarray of triangle indices is made and triangle indices are stored either at the beginning or at the end of the original subarray, based on which child they belong to. During this process, the index of the first triangle that belongs to the right child is found and returned to the main building function. The resulting left and right triangle index subarrays then undergo the subdivision process again.

The `getDividingTri` function is implemented both with and without the use of NEON vector instructions. Performance testing in chapter 4 shows that using the vector instructions leads to large performance gains in the build process. Pseudocode of the binning process can be seen in algorithm 4.

The specific parts where NEON instructions are used is the calculation of the bounding box enclosing the triangle centers and the triangle distribution among bins. For the triangle center bounding box calculation, triangle centers are processed in packs of four and contained in four different bounding boxes. When all centers are processed, the four bounding boxes are merged into one. In the triangle distribution phase, four indices of bins in which triangles belong are calculated at once. The triangles are then distributed using standard non-vector instructions.

This algorithm can also be parallelized using multiple threads. Since the tested Android device has a 4-core CPU, this fact is also utilized. From the proposed approach described in [10], only the parallelization of the subtrees was implemented (called vertical phase). The parallelization in the upper part of the BVH when subtrees are being generated was also tested, but only yielded performance drops, probably due to the high cost of starting and joining threads. Using a thread pool might be the solution in this case, but was not further tested.

In the first phase of the multi-threaded version of the algorithm, a number of nodes to further subdivide is created. These nodes are then supplied to several threads as jobs. The

generation of subtrees created in the first phase by one thread is controlled by a constant, n_S , which indicates the threshold number of triangles in a node. If the number of triangles in a node drops below this value, the node (future subtree) is not further subdivided, but is put on a stack instead. When all nodes are processed in the first phase using this rule, a list of nodes for further splitting is ready for the second phase. Multiple threads are started, each taking nodes from the subtree array, until the subtree array is empty.

Access to the subtree array is implemented using an atomic integer variable. Each thread is able to atomically fetch and increment its value without interfering with the work of other threads. This way, threads which process smaller subtrees can fetch new subtrees to process dynamically and no manual distribution of subtrees among threads is necessary.

It is also important to note that during the implementation, ShinyProfiler ([12]) was used to find the most computationally expensive parts of code and remove bottlenecks.


```

foreach triangle T in bvh.triangleArray do
  └ bvh.rootNode.AABB.contain(T);

if size(bvh.triangleArray) < numTriInLeaf then
  └ bvh.rootNode.firstPrim ← 0;
  └ bvh.rootNode.numPrim ← size(bvh.triangleArray);
else
  bvh.triRefArray ← new int[size(bvh.triangleArray)];
  for int i=0; i < size(bvh.triangleArray); i++ do
    └ bvh.triRefArray[i] ← i;
  nodeList ← new BVH_node[2 * size(bvh.triangleArray)];
  curNode ← nodeList[0];
  toSubdivideStack ← new int[64]; numToSubdivide ← 0;
  while true do
    leftChild ← nodeList[numBvhNodes++];
    rightChild ← nodeList[numBvhNodes++];
    firstTri ← curNode.firstTri;
    numTri ← curNode.numTri;
    dividingTri ←
      getDividingTri(firstTri, numTri, area(curNode.box), leftChild.box, rightChild.box);

    leftChild.firstTri ← firstTri;
    leftChild.numTri ← dividingTri;
    rightChild.firstTri ← firstTri + dividingTri;
    rightChild.numTri ← (numTri - dividingTri);

    curNode.leftChild ← numBvhNodes - 2;
    curNode.rightChild ← numBvhNodes - 1;
    if (leftChild.numTri) > numTriInLeaf then
      └ if (rightChild.numTri) > numTriInLeaf then
          └ toSubdivideStack[numToSubDivide++] ← curNode.rightChild;
          └ curNode ← leftChild;
      else if (rightChild.numTri) > numTriInLeaf then
          └ curNode ← rightChild;
      else if numToSubdivide > 0 then
          └ curNode ← nodeList[toSubdivideStack[--numToSubdivide]];
      else
          └ break;

```

Algorithm 3: Algorithm of the BVH builder based on binning

```
centersBox ← new AABB;
for int  $i = \text{firstTri}; i < \text{firstTri} + \text{numTri}; i++$  do
    | centersBox.contain(triArray[triRefArray[i]].center);
axis ← centersBox.widestAxis;
for int  $i = \text{firstTri}; i < \text{firstTri} + \text{numTri}; i++$  do
    | binIndex ← numBins * (triArray[triRefArray[i]].center[axis] -
    |   centersBox.min[axis]) / (centersBox.max[axis] - centersBox.min[axis]);
    | triArray[triRefArray[i]].binIndex ← binIndex;
    | binList[binIndex].leftBox.contain(triArray[triRefArray[i]));
    | binList[binIndex].rightBox.contain(triArray[triRefArray[i]));
    | binList[binIndex].numTri++;

for int  $i = 1; i < \text{numBins}; i++$  do
    | binList[i].leftBox.contain(binList[i-1].leftBox);
    | binList[i].leftTriNum ← binList[i].numTri + binList[i-1].leftTriNum;
for int  $i = \text{numBins} - 2; i \geq 0; i--$  do
    | binList[i].rightBox.contain(binList[i+1].rightBox);
    | binList[i].rightTriNum ← binList[i].numTri + binList[i+1].rightTriNum;

bestSAHvalue ← ∞;
divBinIndex ← 0;

for int  $i = 0; i < \text{numBins} - 1; i++$  do
    | SAHvalue ← (binList[i].leftBox.area * binList[i].leftTriNum +
    |   binList[i+1].rightBox.area * binList[i+1].rightTriNum);
    | if SAHvalue < bestSAHvalue then
    |   | bestSAHvalue ← SAHvalue;
    |   | divBinIndex ← i;

triRefArrayCopy ← copy of triRefArray;
leftIndex ← firstTri;
rightIndex ← firstTri + numTri - 1;
for int  $i = 0; i < \text{numTri}; i++$  do
    | if triArray[triRefArrayCopy[i]].binIndex > divBinIndex then
    |   | triRefArray[rightIndex--] ← triRefArrayCopy[i];
    | else
    |   | triRefArray[leftIndex++] ← triRefArrayCopy[i];
return (leftIndex - firstTri);
```

Algorithm 4: BVH node splitting based on binning

4 Performance testing and evaluation

In this chapter, performance of both the implemented BVH building algorithm and the rendering system is tested and usability for real-time applications is evaluated. Tests are performed on 15 scenes using meshes with varying triangle count and distribution. All tests are done under these conditions and settings:

- Rendering is done using the multithreaded algorithm with the use of NEON instructions, as described in section 3.2.5, using 4 threads.
- Primary and shadow SIMD rays are fired. Diffuse and specular color components are calculated for each pixel.
- One point light is added to each scene.
- Rendering performance is measured as the duration of one frame rendering in milliseconds.
- Quality of the constructed BVH is evaluated using the surface area heuristic. Also, the average number of intersections of a ray with a bounding box and with a triangle are measured for each BVH. It is important to note that these values are measured for SIMD rays, which are packets of four single rays.
- Resolution of the rendered image is set to 1024x576, which follows from the 16:9 aspect ratio of the device's screen.
- Rendering performance is measured for the camera position, angle and field of view which can be seen in figures 9 and 10.

As mentioned in chapter 3.4, the algorithm for node splitting was implemented both with and without the use of NEON instructions. Additionally, the whole build process was implemented with the utilization of both one and several threads. Three build configurations are therefore tested and evaluated in the following sections.

4.1 Keyframe animation scenes

In the first set of scenes, each scene contains one rigid object which references 10 meshes. Each of these meshes contains the same geometry, but the topology is different. The rigid object loops through the meshes creating an animation. In these scenes, the measured values are averaged across all 10 meshes. Rendering performance that is measured does not include rebuilding of the BVH for each mesh before rendering each frame. This can easily be calculated by summing the frame time in milliseconds with the BVH build time

in milliseconds. The selected keyframe animation scenes can be seen in figure 9 including their triangle counts.

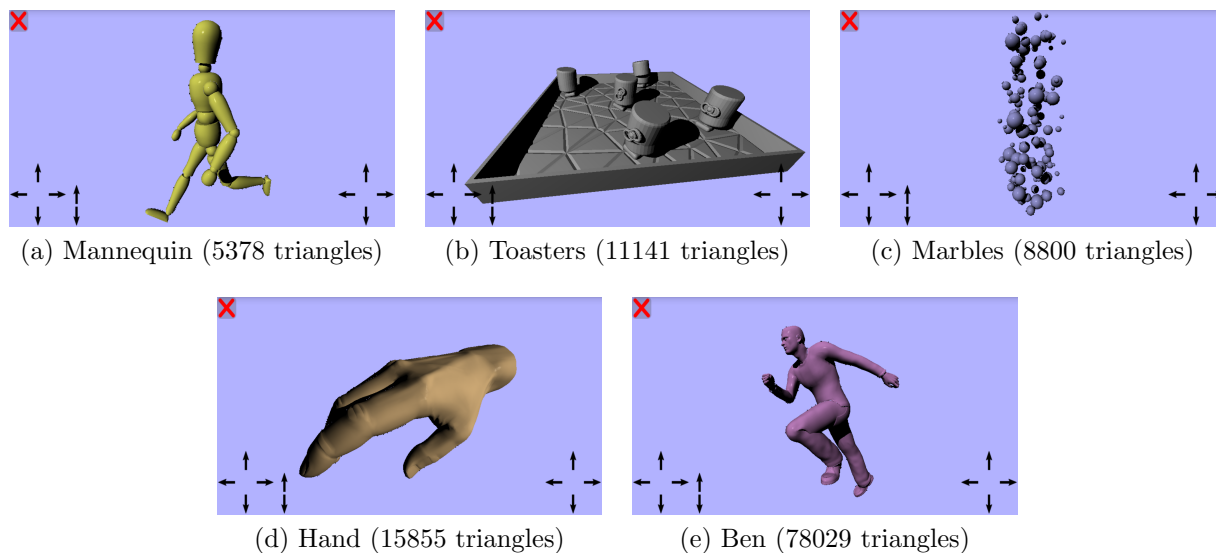


Figure 9: Keyframe animation scenes used for performance tests

4.2 **Static scenes**

In the second set of scenes, each scene contains one rigid object which references one triangle mesh. As with the dynamic scenes, rendering performance is again measured without the BVH being rebuilt before each frame. The selected scenes including their triangle counts can be seen in figure 10.

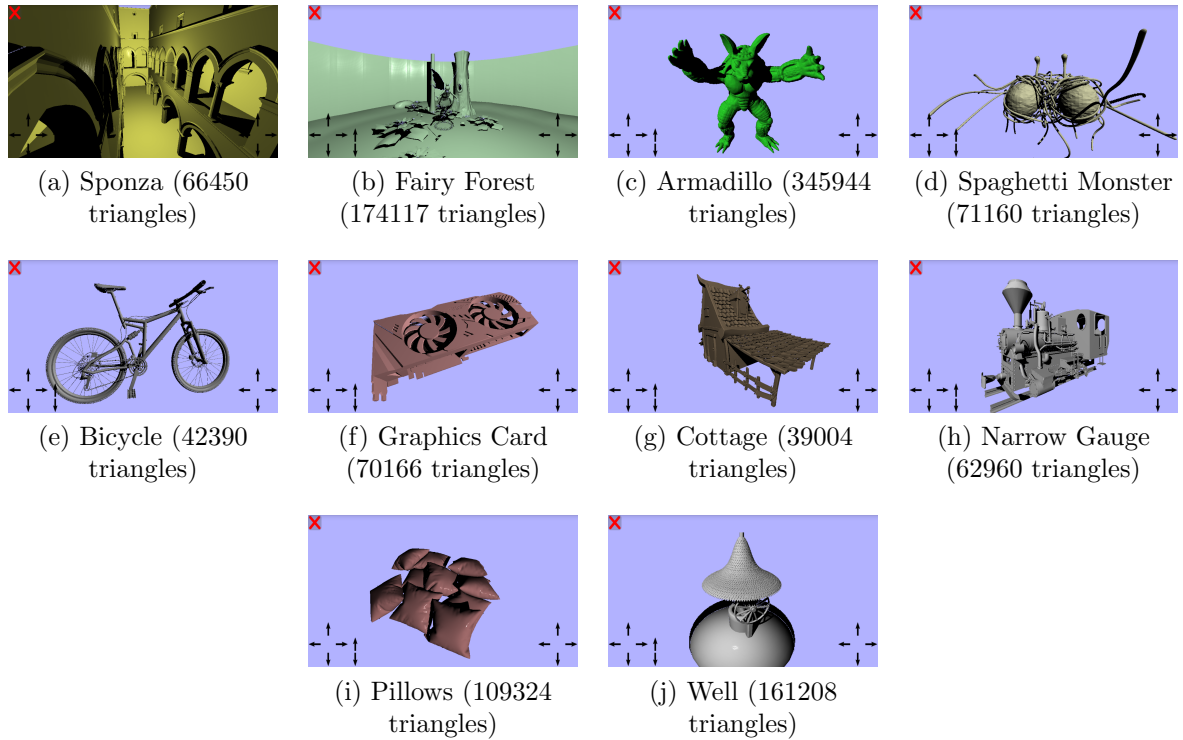


Figure 10: Static scenes used for performance tests

4.3 BVH construction - setup 1

In the first BVH algorithm testing setup, the single-threaded version is used. For the node split search, the algorithm without SIMD NEON instructions is used. The builder is configured as follows:

- The cost of a ray-bounding box intersection (C_t) and ray-triangle intersection (C_i) are set to 1.0 and 2.0, respectively.
- The threshold value for the number of triangles in a leaf node is set to 10.
- The number of bins for the node splitting phase is set to 16.

The results of BVH construction times can be seen in table 3. For the keyframe animation scenes, the build time is averaged across 10 triangle meshes.

Scene	triangles	Build time [ms]
Mannequin	5378	11.17
Toasters	11141	23.04
Marbles	8800	14.95
Hand	15855	32.61
Ben	78029	216.95
Sponza	66450	175.32
Fairy Forest	174117	575.65
Armadillo	345944	2326.57
Spaghetti Monster	71160	200.74
Bicycle	42390	118.71
Graphics Card	70166	237.88
Cottage	39004	107.33
Narrow Gauge	62960	335.85
Pillows	109324	486.76
Well	161208	493.74

Table 3: BVH build performance - BVH built with a single thread

4.4 BVH construction - setup 2

In the second BVH algorithm setup, the single-threaded version is used as well. For the node split search, however, the algorithm which utilizes SIMD NEON instructions is selected. The builder is configured the same way as in setup 1:

- The cost of a ray-bounding box intersection (C_t) and ray-triangle intersection (C_i) are set to 1.0 and 2.0, respectively.
- The threshold value for the number of triangles in a leaf node is set to 10.
- The number of bins for the node splitting phase is set to 16.

The results of BVH construction times can be seen in table 4. For the keyframe animation scenes, the build time is averaged across 10 triangle meshes. In the fourth column, the relative improvement in build times over setup 1 is shown.

Scene	triangles	Build time [ms]	Improvement (vs. setup 1) [%]
Mannequin	5378	6.48	41.99
Toasters	11141	16.46	28.56
Marbles	8800	11.26	24.68
Hand	15855	28.53	12.51
Ben	78029	155.33	28.40
Sponza	66450	129.15	26.33
Fairy Forest	174117	409.09	28.93
Armadillo	345944	1122.8	51.74
Spaghetti Monster	71160	137.82	31.34
Bicycle	42390	71.84	39.48
Graphics Card	70166	138.95	41.59
Cottage	39004	69.97	34.81
Narrow Gauge	62960	121.43	63.84
Pillows	109324	209.81	56.90
Well	161208	363.31	26.42

Table 4: BVH build performance - BVH built with a single thread, NEON instructions

4.5 BVH construction - setup 3

The third setup uses the multi-threaded two-phase algorithm. For the node split search, the algorithm which utilizes NEON instructions is used. The builder is configured as follows:

- The cost of a ray-bounding box intersection (C_t) and ray-triangle intersection (C_i) are set to 1.0 and 2.0, respectively.
- The threshold value for the number of triangles in a leaf node is set to 10.
- The number of bins for the node splitting phase is set to 16.
- The constant which determines the threshold value for the number of primitives in a subtree which is processed by a thread (described in chapter 3.4) is set to 8. That means that subtrees for the second phase of the build contain anywhere between 1/8 and 1/16 of the total number of triangles in the mesh.

The results of BVH construction times can be seen in table 5. For the keyframe animation scenes, the build time is averaged across 10 triangle meshes. In the fourth column, the relative improvement in build times over setup 2 is shown, in the fifth column, the build times are compared to build times in the first setup.

4.6 Rendering performance and BVH quality

Scene	triangles	Build time [ms]	Improvement (vs. setup 2) [%]	Improvement (vs. setup 1) [%]
Mannequin	5378	6.02	7.10	46.11
Toasters	11141	18.48	-12.27	19.79
Marbles	8800	9.95	11.63	33.44
Hand	15855	21.5	24.64	34.07
Ben	78029	128.52	17.26	40.76
Sponza	66450	113.74	11.93	35.12
Fairy Forest	174117	309.57	24.33	46.22
Armadillo	345944	639.43	43.05	72.52
Spaghetti Monster	71160	105.74	23.28	47.32
Bicycle	42390	60.49	15.80	49.04
Graphics Card	70166	142.67	-2.68	40.02
Cottage	39004	56.15	19.75	47.68
Narrow Gauge	62960	105.25	13.32	68.66
Pillows	109324	137.85	34.30	71.68
Well	161208	276.89	23.79	43.92

Table 5: BVH build performance - BVH built with multiple threads, NEON instructions

The build times for the *Toasters* and *Graphics Card* scenes are actually worse than in setup 2. Experiments have shown that setting the number of primitives in a subtree which is processed by a thread to 4 instead of 8 (leaving more work for the second, parallel phase of the build) for these scenes improved the result, which was then better than in setup 2. For other scenes, this change resulted in worse build times.

4.6 Rendering performance and BVH quality

Rendering performance is measured for all listed scenes using the camera setup visible in figures 9 and 10 and resolution 1024x576. Using different viewpoints and resolutions influences the rendering performance significantly. The cost of the built data structures is evaluated using the surface area heuristic. The average number of intersections of a SIMD ray with a bounding box and a triangle during one traversal of the BVH is also measured. The results can be seen in table 6.

4.7 Discussion

Scene	triangles	BVH cost	Avg. ray-box tests	Avg. ray-tri tests	Render time [ms]
Mannequin	5378	34.99	8.83	4.79	76.78
Toasters	11141	76.28	28.69	14.03	221.11
Marbles	8800	61.98	20.67	12.11	82.98
Hand	15855	48.39	21.46	11.09	165.33
Ben	78029	46.39	18.14	8.8	121.58
Sponza	66450	136.74	58.08	22.15	677.28
Fairy Forest	174117	53.06	30.28	10.05	355.66
Armadillo	345944	48.56	20.13	6.48	191.22
Spaghetti M.	71160	42.72	21.08	7.74	222.08
Bicycle	42390	51.62	16.96	9.54	207.67
Graphics C.	70166	225.17	49.77	24.44	229.73
Cottage	39004	81.23	28.35	13.64	181.35
Narrow G.	62960	49.47	23.85	9.49	238.09
Pillows	109324	103.41	34.65	17.38	243.97
Well	161208	70.2	22.56	9.95	255.6

Table 6: Scene rendering performance for 1024x576 resolution

4.7 Discussion

The results show that for an interactive application that aims to maintain 30 frames per second, the implemented BVH builder is able to do a per-frame-rebuild of a BVH for a triangle mesh of roughly 20000 - 30000 triangles, although the build time highly depends on the triangle distribution within the mesh and the selected algorithm constants, and can vary significantly from mesh to mesh.

Results of the testing with setup 2 show that using the NEON instruction set in the node splitting phase has a huge impact on build performance. Evaluation of the tests with setup 3 shows that splitting the whole build into two parts and building subtrees in parallel using multiple threads yields lower or higher performance boosts, depending on the selected mesh and how work is divided between the two build phases. For the *Toasters* scene, the improvement over the basic, single-threaded, non-SIMD algorithm is only 19%, while for the *Armadillo* model, the improvement is over 72%.

5 Conclusion

The research done in this thesis has shown that interactive applications using ray tracing are already very possible and efforts have already shifted from efficient ray tracing to efficient acceleration data structure building, allowing to move from interactive rendering of static scenes to interactive rendering of dynamic ones, which is the ultimate goal.

The results of the evaluation in chapter 4 show that a contemporary high-end mobile device with the Android operating system is certainly capable of rendering 3D scenes using ray tracing, and is even able to do it in real time, depending on the quality of the implementation, scene complexity and the hardware specifications of the device.

The implemented ray tracing system and BVH builder are able to render a dynamic scene at an interactive framerate. Since the screen of a mobile device is small, the resolution can be lowered without a significant impact on user experience, allowing for even higher framerates than those measured in chapter 4. The implemented BVH builder is able to rebuild the data structure of one or several meshes before the rendering of each frame takes place, depending on the complexity and number of the meshes.

5.1 Future development

An advantage for partly dynamic and partly static scenes, or scenes which use rigid body motion and do not require fast rebuilding of acceleration data structures, using k-d trees for certain models could speed up the rendering phase significantly, since, k-d trees are currently the most efficient acceleration data structures for ray tracing. Combining meshes with bounding volume hierarchies and meshes with k-d trees would be possible using the object bounding volume hierarchy described in chapter 3.2.1.

Another possible path of examination and development is using the dedicated graphics processor on the device, which requires differently designed algorithms for both rendering and data structure construction.

Bibliography

- [1] SpecOut. Snapdragon 600 system on chip technical specifications. <http://system-on-a-chip.specout.com/1/296/Qualcomm-Snapdragon-600-APQ8064T>, 2015.
- [2] Klaus Hinum. Qualcomm adreno 320 gpu specifications. <http://www.notebookcheck.net/Qualcomm-Adreno-320.110713.0.html>, 2014.
- [3] Nic Dahlquist. Ray tracer demo. <https://github.com/ndahlquist/raytracer>, 2015.
- [4] Wizapply Project. Multi-core cpu raytracing. <http://wizapply.com/mp2mark/>, 2012.
- [5] Wizapply Project. Real-time gpu raytracing. <https://play.google.com/store/apps/details?id=com.raytracing.wizapply&hl=en>, 2011.
- [6] Jaedon Lee, Won-Jong Lee, Youngsam Shin, Seokjoong Hwang, Soojung Ryu, and Jeongwook Kim. Two-aabb traversal for mobile real-time ray tracing. In *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*, page 14. ACM, 2014.
- [7] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. Fast agglomerative clustering for rendering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 81–86. IEEE, 2008.
- [8] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *Computer Graphics and Applications, IEEE*, 7(5):14–20, 1987.
- [9] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference, HPG '13*, pages 81–88, New York, NY, USA, 2013. ACM.
- [10] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 33–40. IEEE, 2007.
- [11] Ryan Overbeck, Ravi Ramamoorthi, and William R Mark. Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 41–48. IEEE, 2008.
- [12] Aidin Abedi. Shiny profiler. <https://code.google.com/p/shinyprofiler/>, 2011.

6 Appendix

6.1 CD Contents

Directory name	Description
NativeRayTracer	The implemented application (three eclipse IDE projects) without scene data
Documentation	Source code documentation for the NativeRayTracer-Library
Screenshots	Screenshots of the tested scenes
dp_langrjan.pdf	The master thesis text

Table 7: CD contents