Graphics and Computational Programming Assignment 1

Graham Rigler  
i7465070

# Part 1

## Brute-Force Ray-Triangle Intersection

### Theory

The brute-force approach for testing a ray-triangle intersection is to test each ray against all other triangles or vice-versa, so called due to the fact that there is no other preliminary testing with computationally cheaper methods to see whether there is any reasonable chance of collision before the more-expensive ray-triangle intersection test is done.

The actual ray-triangle intersection test is done through first testing whether the ray will collide with the supporting plane of the triangle, then gathering the intersection point with that plane and testing whether it is within the bound of the triangle. This is outlined in more detail through a series of equations below.

(1)

A point on a ray is defined using the as using the parametric equation (1), where defines the ray’s origin, the direction and a parameter which is a scalar value defining how far along the point lies.

(2)

(3)

The supporting plane is defined by (2), but can be expressed as a product of its normal by (3). If any point on this ray, as defined by (1) is to exist on the plane then it must satisfy this equation. Therefore if is substituted by, then the equation can be rearranged into the form below.

(4)

Logically, this means that if or then the ray does not intersect with that plane due to either being parallel in the former case or the plane existing in the inverse direction of the ray, as shown in Figure 1.

Once the intersection with the plane has been validated the intersection point can be found by using (1) to give the intersection point. A test must then be done for whether this point is within the bounds of the triangle, defined by three vertices and. Using the defining vertex points of the triangle, the three edges of the triangle can be defined as (5).

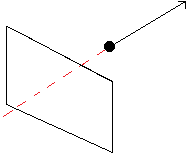


Figure 1 - Simple diagram of a plane (shown with a rectangular sub-section) existing behind a ray with an arrowed direction. As shown, the section of the ray that would exist for a line (shown in red) would collide with plane, but does not for the ray.

(5)

By calculating the triple product of , where and are the vertex points that make up each edge of the triangle, a scalar value is given that will be positive for if the contact point is within the triangle bounds of that edge. Therefore, if this is tested positively for each bounding edge of the triangle then the point is within the triangle. This is shown in (6).

(6)

### Implementation

The implementation of

At a higher level than the actual test for a ray-triangle intersection, ordering of operations can be done to reduce the number of overall operations. The ordering of these calculations have been done in a specific way as shown in the pseudocode below.

|  |
| --- |
| Brute-Force Pseudocode |
| for each triangle  n := CalculateNormal()    for n from 0 to 255    for m from 0 to 255  #Defining origin and direction of ray  p := (0,m,n)  d := (1,0,0)    #Testing collision  if collision(p,d,triangle)  Draw(m,n)  end if    end for    end for    end for |

In this pseudocode the triangles are iterating over the rays, this is so that the normal for each triangle is being calculated only one as this is a computationally expensive operation and should be done a minimal amount of times. It should also be noted that in the implementation of this algorithm (as well as any others mentioned in this report) rays are not instantiated prior to the ray-triangle test, instead the origin and direction are used as arguments to the collision test function. This has been done as it is wholly unnecessary to instantiate them as memory as it only increases the memory footprint of the program.

## Bounding Volumes

### General Theory

A bounding volume (BV) is a representation of an object in world-space using some primitive shape, such as a cube or a sphere. This allows for more complicated geometry to be represented in a simplistic, if not completely accurate, form. A popular implementation of BVs is the Axis-Aligned Bounding Box (AABB), this encompasses an object in a box that is relative to the X, Y and Z axis as opposed to being oriented to the object (although this implementation exists as the Oriented Bounding Box). The purpose of this in the context of this assignment is that it allows for each ray to be tested against the bounding volume before being tested against triangle, this is a computationally cheaper test and has been explained further below.

As can be seen in Figure 3, the arbitrary object in this example is taking a relatively small portion of the screen. This means if each ray, which in the context of this example is equivalent to a pixel on the screen, was to test against each triangle on the object then there would be a lot of expensive calculation being done needlessly as there it is clear there is no way in which those rays will be colliding with the object. Therefore, a check must be done to ensure that the ray’s origin is within the BV in the relevant axis, for this instance the only relevant axis are Y and Z as the ray’s direction is always constant in the X axis as 1.

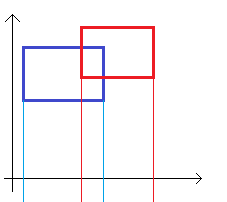


Figure 2 - Diagram showing the projection of the minimum and maximum values of two AABBs onto an arbitrary axis for collision testing.

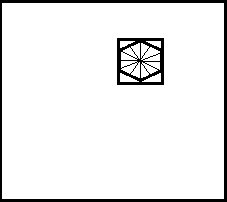


Figure 3 - Simple diagram of an arbitrary triangulated object, surrounded with a tight-fitting AABB, displayed in a on a screen.

In order to do this, the minimum and maximum values of the BV must be projected onto the axis, as is shown in Figure 2 for an arbitrary axis. In this figure the minimum and maximum values have been displayed against the axis using lines. When AABBs are used for collision detection between objects, each object has a BV in 3D and is said to pass the collision test if the minimum value of one is less than the maximum value of another (or vice-versa) on all axis, however in this context the ray does not need to have a bounding volume per-say as it does not exist as a physical object. Instead the Y and Z bounding values for the object’s AABB are projected and used to test against the origin of the ray. The equations for this simple test are shown below.

(7)

Where is the origin of the ray, and and define the minimum and maximum bounds of the AABB of an object.

(8)

As can be seen in (7), the ray’s origin is defined in the Y and Z axis in this example and as it’s direction is constant in the X axis, the only axis of the object that need to be tested of the object are the Y and Z axis as is shown in (8).

If this test has not passed then this ray will not test against any triangles at all whereas the ray would have tested against every single triangle using brute-force, wasting processing time. If the test has passed, then it can be seen that there is a chance for this ray to collide with a triangle within the object (although it is important to note that this is not guaranteed, due to the inaccuracy in AABB and it not exactly representing the object) therefore a brute-force test will continue for this ray.

When an object encapsulated by a BV is rotated in any way, then the bounds of the BV must be recalculated as they will likely no longer be fitting the object and it is possible to exist outside of those bounds. An example of this has been shown in Figure 4. There are multiple methods of solving this problem; simply iterating through all vertices of the object again and constructing a new AABB with the rotated set of vertex points, a ‘Hill-climbing’ that requires a data structure in the minimum and maximum vectors of the AABB store information on their neighbours, and even rotating the minimum and maximum bounds of the AABB (effectively turning it into an oriented bounding box) and then finding the new minimum and maximum values from these two points. (Ericson 2004, pp.82-87)

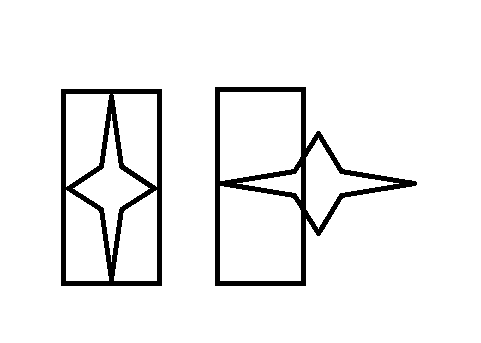


Figure 4 - Diagram showing an arbitrary object with an AABB on the left and then the same object after a rotation with an un-updated AABB, showing the effects of rotation on the accuracy of a BV.

### Implementation

The implementation of this algorithm can be seen in pseudocode form below. For ease of reading, the differences between this pseudocode and that of the brute-force implementation have been marked in blue.

|  |
| --- |
| AABB BV Collision |
| bv := CalculateAABB()  for each triangle  n := CalculateNormal()    for n from 0 to 255  if n outside bv  continue to next n iteration  else  for m from 0 to 255  #Defining origin and direction of ray  p := (0,m,n)  d := (1,0,0)  if m outside bv  continue to next m iteration  else  #Testing collision  if collision(p,d,triangle)  Draw(m,n)  end if    end for    end for    end for |

This implementation makes use of the fact that the rays are iterating at two levels, in that if n value is not within the BV of the object then there is no use iterating through m and carrying out more checks. While this does increase the amount of checks that must be carried out for a successful ray-triangle test to be performed, it eliminates other unnecessary operations from being carried out.

The actual calculation of the BV has been omitted from the timing of this algorithm as in most applications there will already be an implementation of a BV in place, namely within physics systems. However this in itself should not taint the data in any way as it would be a pre-computation that would be carried out on the object when it is loaded into the application, as opposed to being done each time it is being checked against a ray.

### Per-Triangle

#### Differences

While an AABB can be calculated to encompass the entire object to initially test for a possible collision, then carrying on with a brute-force pass if this passes, it is also possible to create an AABB for each triangle. The intention of this is that, as the triangles are being iterated through at the highest loop in the algorithm, the initial BV check will be representative of the likelihood that a ray-triangle collision will occur, potentially reducing the number of failed and therefore unnecessary checks are being made.

#### Memory Overhead vs. Performance

There are two methods for implementing this algorithm, one involves calculating and storing an AABB for each triangle when an object is loading in and the other involves calculating it for each triangle as it is tested. For objects made up of a small number of triangles, the increased memory footprint involved in storing 6 more floating point values for each face does not have a huge impact however it is clear that as the amount of faces increases then this becomes more of an issue. Therefore for the implementation that has been done for this assignment the BV has been calculated per-triangle at the same stage as the normal. This is a comparison of 9 floating point values (the components of the three vertex points that make up the triangle), to find the minimum and maximum values in each axis.

### Bounding Volume Hierarchy

As was previously discussed, one of the main issues with this method of ray-object intersection testing is that once the ray is determined to pass the initial AABB test then the brute-force algorithm is essentially used where that ray is then tested against every triangle (or triangle against every valid ray as the case may be). A potential solution to this problem is to create a ‘Bounding Volume Hierarchy’ (BVH), in which a tree-like structure of BVs are used to partition the object into discrete sub-sets of triangles in a hierarchy. This solution also has parallel implementations that have been developed in recent years to a variety of extents and have been used extensively in ray-tracing applications. (Lauterbach et al. 2009)

Due to time constraints of this project, an implementation of a BVH was not attempted for this project. However, it is likely that the speedup gained from this would allow for potentially real-time ray-tracing to be achieved, albeit to a low-quality compared to a standard rendering pipeline, from this solution. (Gunther et al. 2007) Existing parallel implementations of BVHs, such as the ‘Two-AABB’ system produced by Lee et al. (2014) and the more-modern ‘MBVH’ structure from Viitanen et al. (2016) that is based off of it, have actually proven to work on lower-performance systems such as mobile devices in real-time through the use of parallel processing architectures and BVH data structures.

## Algorithm Analysis

### Brute-Force

In order to evaluate the brute-force algorithm accurately, a high number of performance benchmarks have been taken on a single machine using a variety of models containing different numbers of triangles. This ranges from a simple, publicly available, model of a Gourd that uses 648 triangles to a triangulated 3D scan of man that uses 79,996 triangles. 256 full brute-force tests were run for each model, apart from the highest-poly mesh due to the length of time each test was taking. Each test was timed using ‘steady\_clock’ from C++’s standard library, this implementation of a timer contains some steadying to account for process scheduling done by the OS, that would be otherwise unaccounted for if ‘high\_resolution\_timer’ was used, as the only time that should be measured when comparing them is the time that is actually spent computing the process due to potential differences in process scheduling between tests. Despite this there was still a relatively large variance in the time taken between tests which increased as the time taken for the algorithm finished, ranging from a standard error of 76.2ms for the Gourd and 1416.0ms for the 3D scan. Graphs of all of this data is available in appendices 1.a-1.f; the trend-lines have been plotted with the gradients shown, the gradients of which are a good indicator of the variance between tests.

### Bounding Volume

The bounding volume implementations vastly outperformed the brute-force implementation, however it is worth noting that the performance of this implementation of a BV test depends greatly on the size of the object on the screen. As can be seen in Figure 5 the ‘Magnolia’ test-render took up more space on the screen compared to the ‘Typhoon’ test-render and on average took longer to render despite having a lower number of faces and therefore less triangles to test against. This is due to the increased size means that more rays will collide with the BV in the initial test and will therefore test against more triangles.

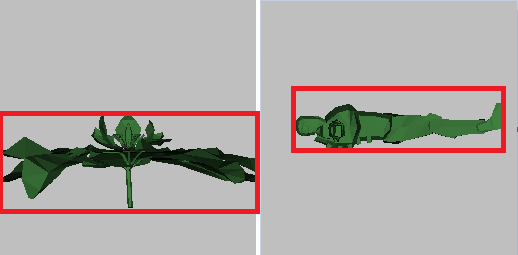


Figure – Two renders produced from the BV implementation, Magnolia is on the left and Typhoon is on the right. Red boxes have been added to give a visual indication of what the BV for both models would be, as can be seen the Magnolia’s is larger than the Typhoon’s and therefore takes longer to render, despite having a lower number of triangles.

#### Per-Object AABB

As can be seen in appendices 2-b, not only did the per-object AABB implementation perform much faster than the brute-force implementation the results produced seem to be more consistent for objects with a low number of faces. This can be seen in comparing the gradients of the trendlines produced for both appendices 1 and appendices 2-b; if the results were perfectly produced then there should be a gradient of 0 (i.e. the same result is produced for each iteration), however this is not feasible due to lack of control over process scheduling over time and such. Despite this the gradient does not grow over time as quickly as the brute-force implementation.

#### Per-Triangle AABB

As can be seen in Appendix 2 - a, the differences between this algorithm and its per-object counterpart for objects with a lower number of triangles is minimal. However there is a notable divergence for the tests done with a relatively high number of triangles that, due to the linear nature of this algorithm, will presumably continue to diverge. This shows that while the performance increase from using this method is almost within the range of being related to an error, there is an increasing notable performance increase in using this method for higher ranges.

# Part 2

## Sectioned Parallelism

This implementations that have been made do not rely on any data from each other, in that the result of one ray-triangle intersection test does not rely on the other, meaning that the entirety of any algorithm implementation mentioned in this report could be computed entirely in parallel. However, the method of doing that is variable, this is explained in in more Figure 5 below.

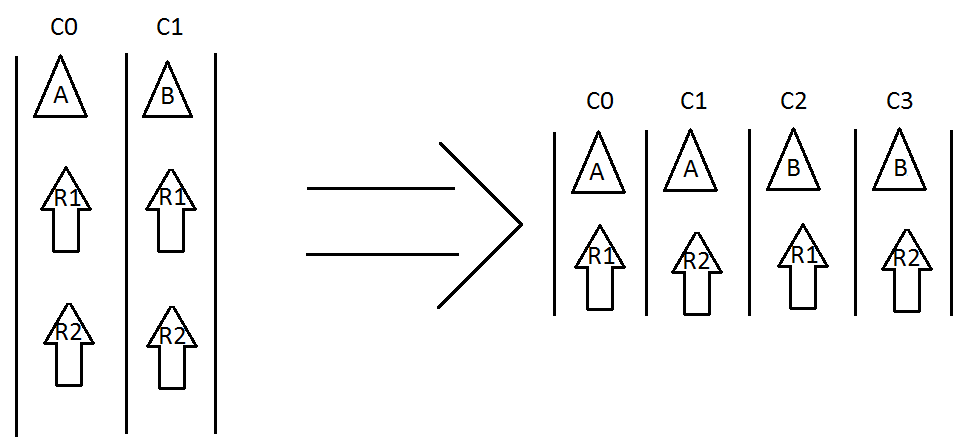


Figure - A rudimentary parallel system on the left, showing each triangle being tested against all rays in parallel. On the right is a parallel system made to make use of more processing core in which each ray-triangle test is being done in parallel. This is showing that if there are more physical cores available then spreading as many atomic calculations across cores (assuming now communication between non-shared memory is required) will reduce the time taken to process the problem overall.

It is important to note that the amount of calculations that can be done in parallel is always restricted by the number of cores available, as well as the amount of data that there is to compute. For example, if there are only two ray-triangle tests to be computed then attempting to use 4 processing cores would result in the same speedup as using 2 while reducing the efficiency. This has concept has been extrapolated to a degree with Figure 6.

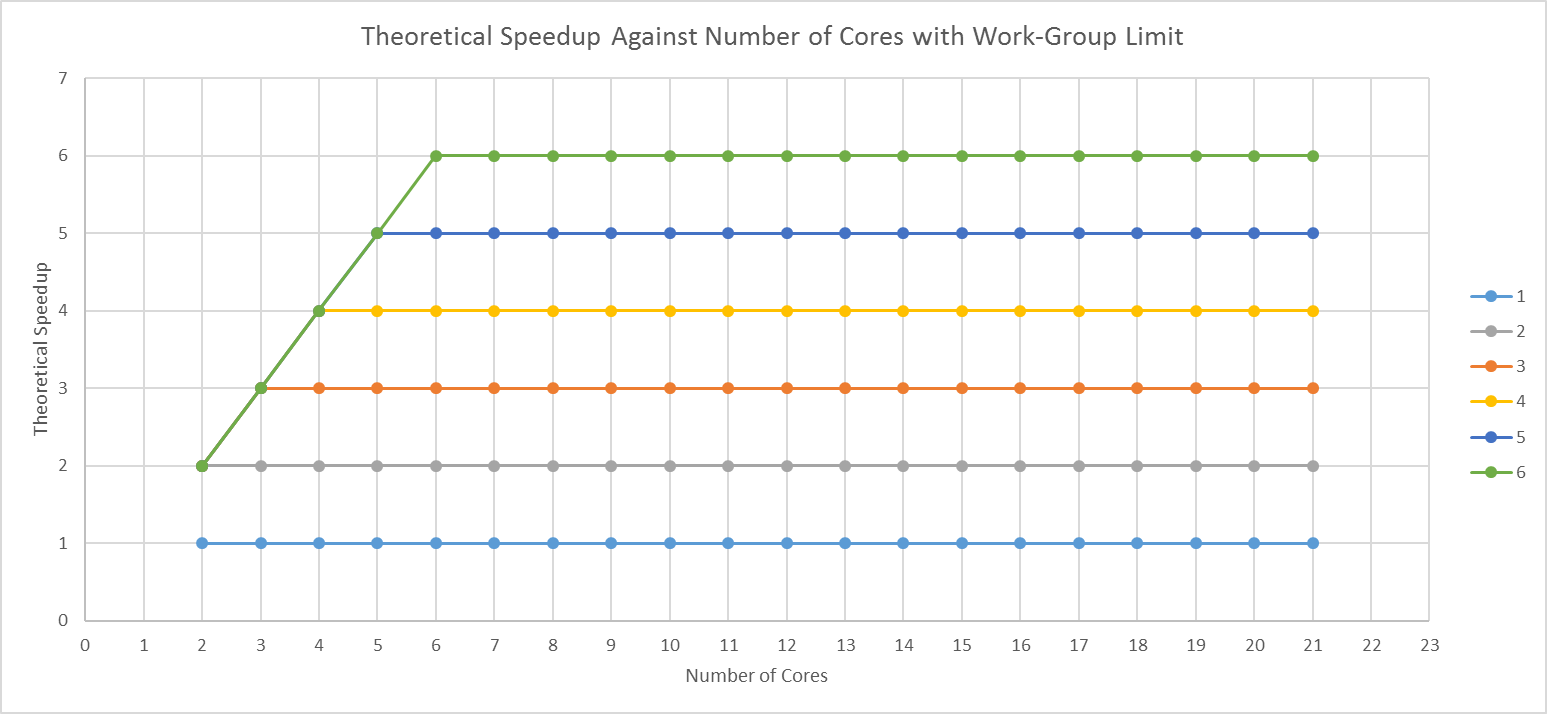
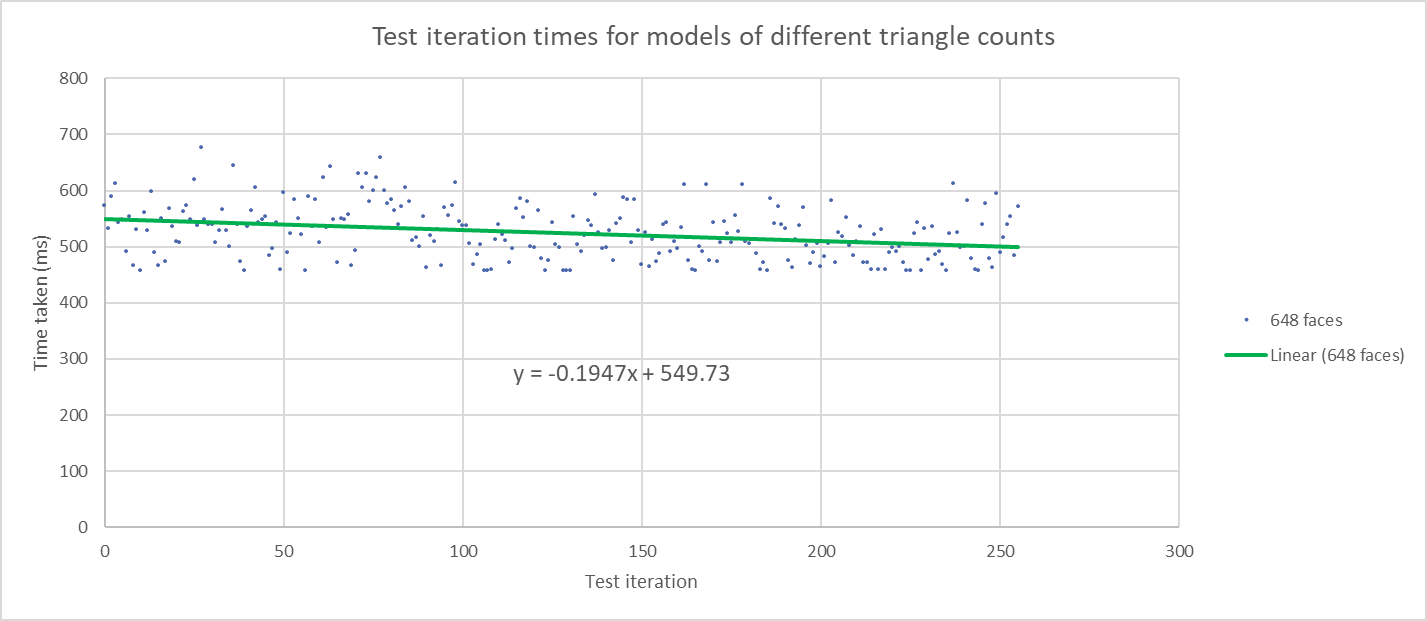


Figure 7 - Theoretical Speedup against number of cores, where theoretical speedup is being limited by the work-size of each data series. As can be seen the theoretical speedup can only increase if there is enough work to be done as there are cores. In the context of this assignment this is equivalent to either the amount of triangles to calculate intersections for or, in the case of a highly parallelised implementation, the number of ray-triangle intersection tests to carry out.

# Appendices

## Brute-Force Algorithm Data



Appendix 1 - – Gourd model (640 faces)

Appendix 1 - – Typhoon model (1283 faces)

Appendix 1 - – Magnolia model (1283 faces)

Appendix 1 - – Teapot model (6320 faces)

Appendix 1 - – Reduced polygon version of a 3D scanned model of a man (79,996 faces). The number of tests taken for this model are lower (32) due to the time taken to complete each data-point.

Appendix 1 - f – Average values of graphs a-e (with y-intercept set to 0), showing what appears to be a linear increase in computation tale with the size of input. This equates to O(n) in big-O notation, in which the number of faces is n. Ideally more models could have been tested in the range of 10,000-75,000 to reduce the gap between that data-points gathered.

## Bounding-Volume Algorithm

### Compared

Appendix 2 - a – Averaged render-time for both the object-based bounding volume and the triangle-based bounding volume implementations. This shows a clear divergence as the number of faces increases, clearly displaying the improvement the triangle-based implementation makes over the other.

### Object-Based Bounding-Volume

Appendix 2 - b - i

Appendix 2 - b - ii

Appendix 2 - b - iii

Appendix 2 - b - iv

Appendix 2 - b - v

Appendix 2 - b - vi

### Triangle-Based Bounding-Volume

Appendix 2 - c - i

Appendix 2 - c - ii

Appendix 2 - c - iii

Appendix 2 - c - iv

Appendix 2 - c - v

Appendix 2 - c - vi

# References

Ericson, C., 2004. *Real-Time Collision Detection*. CRC Press, Inc.

Gunther, J., Popov, S., Seidel, H. P. and Slusallek, P., 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal, *2007 IEEE Symposium on Interactive Ray Tracing* (pp. 113-118).

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D. and Manocha, D., 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum*,28 (2), 375-384.

Lee, J., Lee, W.-J., Shin, Y., Hwang, S., Ryu, S. and Kim, J., 2014. Two-AABB traversal for mobile real-time ray tracing. *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*, Shenzhen, China. 2669088: ACM. 1-5.

Viitanen, T., Koskela, M., J, P., #228, #228, skel, #228, inen and Takala, J., 2016. Multi bounding volume hierarchies for ray tracing pipelines. *SIGGRAPH ASIA 2016 Technical Briefs*, Macau. 3005384: ACM. 1-4.