# Graphics and Computational Programming

Introduction

For this assignment, the implementation of 2 ray intersections is required one of which is brute-force ray intersection and the other has been chosen to be Möller-Trumbore ray intersection. Both intersection functions will be tested to see if they work correctly, and how effectively they work against varying amounts of triangles.

To make this balanced for both intersection algorithms, there are constraints set, these are:

1. The rays are to be defined by the parametric equation: R = P + d . t, Where d = (1, 0, 0), P = (0, m, n), 0 >= m < 256 and 0 >= n < 256.
2. There will be 256 by 256 rays due to m and n creating a total of 65536 rays
3. This sets the bounds of the rays to 0 <= x < 256, 0 <= y < 256, 0 <= z < 256 and all objects used must be within these bounds
4. Finally, each object used is defined by a set of triangular facets. The number of triangles and shape of the object are changeable.

The final part of the assignment involves research and calculation of parallel performance indicators of Brute-Force Ray Intersection.

Part 1.1 – Implementation of Brute-Force Ray Intersection

To implement brute force ray intersection, rays are cast between 0 and 256 for each x, y and z coordinate, calculating the intersection points between the ray cast and an object loaded into the program.

To calculate the intersection point, other calculations must first be made using predefined ray origin, ray direction and object vertices, these calculations are as follows:

Calculating normal

The normal of 3 vertices making up a triangle is firstly calculated.

//Function used to find the normal between 3 vectors

Vector FindNormalVector(Vector& \_a, Vector& \_b, Vector& \_c)

//24 operations

{

//subtraction functions used for the calculation of vectors \_p1 and \_p2

Vector \_p1;

\_p1.Subtraction(\_b, \_a);

Vector \_p2;

\_p2.Subtraction(\_c, \_a);

//cross product function used to calculate the normal vector before normalization

Vector \_n;

\_n.CrossProduct(\_p1, \_p2);

//normalize funciton used to normalize the normal vector

\_n.Normalize();

//returns the normalized vector

return \_n;

}

Calculating Constant

This normal is used to then produce a constant variable, K. If K is equal to zero, this then means that the ray is parallel to the triangle and therefore is no intersection.

//function used to calculate the constant for the plane

float FindConstant(Vector& a, Vector& b)

//5 operations

{

//dot product function used to calculate the constant value for the plane

float \_k = a.DotProduct(a, b);

//returns the constant value

return \_k;

}

Calculating Intersection Parameter

The intersection parameter, t, uses the constant, K, along with the normal, ray origin and ray direction to create a value representing the distance between the ray origin and the intersection point. If t is a negative value, this shows that the ray is behind the triangle and therefore there is no intersection point.

//Function used to calculate the intersection parameter between the ray and triangle

float FindIntersectionParameter(float k, Vector& n, Vector& p, Vector& d)

{

//Variable decleration

float \_t;

//checks if the ray and triangle are parallel

if ((n.DotProduct(n, d) != 0))

{

//uses the dot product function along with an algorithm to calculate the parameter value at the intersection point

\_t = (k - n.DotProduct(n, p)) / (n.DotProduct(n, d));

}

else

{

\_t = 0;

}

//returns the intersection parameter

return \_t;

}

Calculating Intersection Point

With the intersection parameter, the intersection point can now be calculated. Using the equation R = P + D . t the ray can be calculated by multiplying the (D) direction variable by (t) the intersection parameter and then adding (P) the ray’s origin.

//Function used to calculate the point of intersection between the ray and the triangle

Vector FindIntersectionPoint(Vector& \_p, float \_t, Vector& \_d)

{

//calculates the directio vector after being multiplied by the intersection parameter

\_d2.SetPoint(\_d.x() \* \_t, \_d.y() \* \_t, \_d.z() \* \_t);

//uses the addition function to calculate the intersection point between the ray and the triangle

\_q.Addition(\_p, \_d2);

//returns the point of intersection

return \_q;

}

Edge Checks for Ray Point Intersection

The final check is between the ray and the edges of the triangle. Using each of the triangles vertex position and the intersection point of the ray, the edge of each triangle can be found. If the intersection point is to the left of each edge, then the ray is inside of the triangle.

//calculates a triangle edge through subtraction of vertice a and b of the triangle

edge.Subtraction(tri.b(), tri.a());

//calculates the variable vp through subtraction of ray intersection

//and triangle vertice a

vp.Subtraction(ray.q(), tri.a());

//sets the perp variable equal to the cross product of the edge and vp variable

perp.CrossProduct(edge, vp);

//checks if the intersection point is to the right of triangle

if (perp.DotProduct(tri.n(), perp) < 0)

{

return false;

}

//repeats the above step for each edge making sure the ray intersection point is

//to the left of each edge

edge.Subtraction(tri.c(), tri.b());

vp.Subtraction(ray.q(), tri.b());

perp.CrossProduct(edge, vp);

if (perp.DotProduct(tri.n(), perp) < 0)

{

return false;

}

edge.Subtraction(tri.a(), tri.c());

vp.Subtraction(ray.q(), tri.c());

perp.CrossProduct(edge, vp);

if (perp.DotProduct(tri.n(), perp) < 0)

{

return false;

}

Part 1.2 – Testing of the Brute-Force Ray Intersection

To evaluate the effectiveness of brute force ray intersection, multiple tests were done.

Time Tests

A timer is set to record the time taken between starting the brute force ray intersection and finishing the function, displaying the results to the console in seconds. This allows the comparison of this function against other functions and can be used to test how effective brute force ray intersection is.

Within release mode of Visual Studio 2015, an object of 6768 vertices and 2256 triangles can be rendered fully through brute force ray intersection in an average time of 2.7924 seconds through 10 tests as shown below as shown in Fig 1.

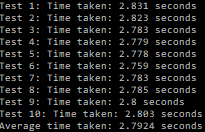


Figure 1: time take over 10 iterations of the brute force ray intersection function, displaying an average time at the end

Visual Tests

A visual representation of the object is also rendered as the program performs the brute force ray intersection function. Each intersection point on the object is drawn as a pixel on the screen as a ray hits it, gradually storing each pixel as the program loops through each ray. This is used as a visual representation of the object in which the brute force ray intersection function works as shown in Fig 2.

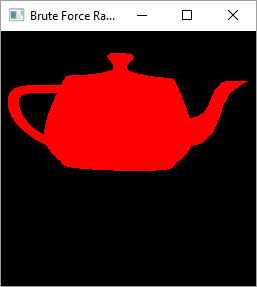


Figure 2: The rendered object displayed upon an iteration of the brute force ray intersection function

Triangular Incrementation Tests

The final testing was seeing how well Brute force ray intersection worked against varying levels of triangles. Starting with a cube of 500 triangles and increasing in increments of 500 triangles until 5000 triangles. Each object was tested for 10 iterations of the intersection function and an average time was produced. These are shown in the graph below Fig 3:

Figure 3: A graph showing the positive correlation between time taken and number of triangles present within an object with Brute-Force ray intersections.

Here it is shown that the increase in triangles makes a clear impact on the time take for the intersection function to do a single iteration with a positive correlation between time taken and the number of triangles within the object. The values range from 1.9387 seconds at 500 triangles to 18.531 seconds at 5000 triangles.

Changes made for Efficiency

Another change was allowing functions to return early, if a ray was found to have not hit a triangle. This was done by making Boolean functions and allowing the function to return false anytime an intersection test was done and the ray was shown to have not hit the triangle.

Another change included casting only rays that were within the bounds of the object by storing the highest and lowest values of the objects y and z coordinates and restricting the rays coordinates to be within those coordinate values.

By creating these changes, the speed of the program increased significantly from originally roughly 20 seconds on brute force ray intersection down to roughly 3 seconds.

Part 1.3 – Research into an Alternate Ray Intersection Algorithm

Through research into other ray intersection algorithms, the Möller–Trumbore intersection algorithm provides a robust and fast algorithm that calculates the intersection between a ray and a triangle. Other ray intersection functions were provided however Möller–Trumbore allowed for simple implementation, whilst maintaining speed and reducing memory storage costs. Another advantage of using this algorithm was the number of helpful articles and citations available through the internet. Many searches provided Möller–Trumbore as the top result allowing easy access to a wide variety of solutions and algorithms.

Part 1.4 – Implementation of Möller–Trumbore Ray Intersection

To implement Möller-Trumbore Ray Intersection, multiple sub calculations must first be completed.

Edge Calculations

Firstly 2 edges of the triangle must be calculated using the triangle vertices (A, B and C).

//calculates edge 1 of the triangle

edge1.Subtraction(tri.b(), tri.a());

//calculates edge 2 of the triangle

edge2.Subtraction(tri.c(), tri.a());

pvec Calculation

pvec vector must then be calculated. pvec is used to calculate both the determinant and u float variable in later calculations.

//calculates the pvec used for calculating the determinant and float 'u'

pvec.CrossProduct(ray.d(), edge2);

determinant Calculation

The determinant float variable is calculated. The value of the determinant is checked to see if it is less than or equal to zero, if so, then the ray misses the triangle.

//used to check for back-face culling and other calculations

float determinant = edge1.DotProduct(edge1, pvec);

//back-face culling

if (determinant <= 0)

{

return false;

}

inverse determinant Calculation

The inverse determinant is calculated. This float variable is used to calculate the float variables u, v and t.

//variable used for calculating float variables 'u', 'v' and 't'

float invDeterminant = 1 / determinant;

tvec Calculation

tvec vector stores a vector which is used in the calculation of the u variable

//stores vector used for calculation of 'u'

tvec.Subtraction(ray.p(), tri.a());

u float calculation

U float variable is then calculated. This value is then checked to see whether it is between 0 and 1, if not, the ray point is outside of the triangle.

//variable used to check if the ray point is outside the triangle or not

float u = invDeterminant \* (tvec.DotProduct(tvec, pvec));

//check for whether the ray point is outside the triangle

if (u < 0.0 || u > 1.0)

{

return false;

}

qvec Calculation

qvec vector is calculated, this result is then used within the calculation of v.

//variable used for the calculation of 'v'

qvec.CrossProduct(tvec, edge1);

v float Calculation

V float variable is calculated. This value is then checked to see if it is greater than 0 and if the addition of u and v is less than 1, if not then the ray point is outside of the triangle.

//variable used to check if the ray point is outside the triangle or not

float v = invDeterminant \* ray.p().DotProduct(ray.d(), qvec);

//check for whether the ray point is outside the triangle

if (v < 0.0 || u + v > 1.0)

{

return false;

}

intersect parameter Calculation

Once all checks have been done the intersection parameter, t, can safely be calculated.

//intersection parameter used to calculate the intersection point

float t = invDeterminant \* edge2.DotProduct(edge2, qvec);

intersection point Calculation

Finally, the intersection point can be calculated using the equation Q = P + D \* t where Q is the intersection vector, P is the ray origin, D is the ray direction and t is the intersection parameter.

//calculation to work out the vector from multiplying the ray direction vector

//by the intersection parameter

d2.SetPoint(d2.x() \* t, d2.y() \* t, d2.z() \* t);

//calculates the intersection point

q.Addition(ray.p(), d2);

Part 1.5 – Testing of Möller-Trumbore Ray Intersection

To evaluate the effectiveness of Möller-Trumbore ray intersection, multiple tests were done.

Time Tests

A timer is used to test the speed and efficiency of the Möller–Trumbore ray intersection by storing the time taken through 10 calls of the intersection function and producing an average value of the results. As shown below, within the release mode of the program, through 10 calls of the intersection function, the average time taken was 1.7762 seconds as shown in Fig 4.

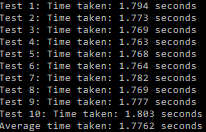


Figure 4: time take over 10 iterations of the Möller–Trumbore ray intersection function, displaying an average time at the end

Visual Tests

Another test conducted was the visual representation of the object. This shows that the intersections have correctly been performed and produced the object as defined by its vertices. Each intersection point is used to draw a single pixel, forming each triangle and eventually rendering the whole object once all intersections have been completed as shown in Fig 5.



Figure 5: The rendered object displayed upon an iteration of the Möller–Trumbore ray intersection function

Triangular Incrementation Tests

The final testing was seeing how well Möller–Trumbore ray intersection worked against varying levels of triangles. Starting with a cube of 500 triangles and increasing in increments of 500 triangles until 5000 triangles. Each object was tested for 10 iterations of the intersection function and an average time was produced. These are shown in the graph below Fig 6:

Figure 6: A graph showing the positive correlation between time taken and number of triangles present within an object with Möller-Trumbore ray intersections.

Here it is shown that the increase in triangles makes a clear impact on the time take for the intersection function to do a single iteration with a positive correlation between time taken and the number of triangles within the object. The values range from 0.9124 seconds at 500 triangles to 8.6514 seconds at 5000 triangles.

Triangular Incrementation Tests

One way to improve the ray intersection would be to make sure only rays within the bounds of the object were cast and that functions returned early when a ray was found to have not intersected with a triangle.

Another change was allowing functions to return early, if a ray was found to have not hit a triangle. This was done by making Boolean functions and allowing the function to return false anytime an intersection test was done and the ray was shown to have not hit the triangle.

Through these changes the execution time of the intersection function was reduced from an average of 13 seconds to an average of 1.7762 seconds.

Part 1.6 – Comparison of Brute-Force Ray Intersection and Möller-Trumbore Ray Intersection

Through testing of both ray intersection algorithms, the Möller–Trumbore ray intersection is faster than the Brute force ray intersection. The Möller–Trumbore algorithm produced an average time of 3.3238 seconds over 10 function iterations, whereas the Brute force algorithm produced an average time of 4.9319 seconds over 10 function iterations.

Another difference between the two intersection functions was the amount of intersections stored once the calculations had been completed. Möller-Trumbore ray intersection had almost half the amount of calculations as Brute force ray intersection. Möller-Trumbore had a total of 65298 intersections for 1500 triangles whereas Brute force had 130534 intersections for the same number of triangles. The reason for this is due to the back-face culling implemented into Möller-Trumbore ray intersection. This is done through the checks made on the determinant (less than or equal to 0), allowing the intersection function to avoid any triangles that aren’t visible which roughly equals to half of the triangles within a symmetrical object and within this case, a cube.

However, with the calculations of the intersections, both methods produced the same results where the intersections matched and rendered the same object in the same way as shown within Fig 2 and Fig 5.

Below is a graph showing the difference in time with both intersection functions as the number of triangles is increased. The Möller–Trumbore intersection algorithm is more efficient based upon its correlation in speed and increase in triangles. At 500 triangles, the difference is 0.8663 seconds reaching a difference of 9.8796 seconds at 5000 triangles as shown in Fig 6.

Figure 6: A graph showing the positive correlation between time taken and number of triangles present within an object with both Möller-Trumbore and Brute force intersection algorithms.

Part 2.1 – Research into Amdahl’s Law and Gustafson’s Law for parallel performance of Brute-force Ray Intersection

Amdahl’s law is an algorithm, which shows the speedup in latency based upon the increase in the system hardware if the workload is constant. What this means is that the algorithm can show the increase in speed of a program when the number of processors within the computer system are increased. As the ray intersection function workload is fixed, this means that this law is better for this program rather than Gustafson-Barsis’s law which is based more upon the increase of workload being scalable to the number of processors.

The simplified algorithm for the speedup is

S = T(1) / T(n)

where T(1) is the amount of time required to run the program and T(n) is the amount of time required to run the parallel program.

A more complex algorithm for speedup is:

S = T(1) / ( T(1) ( B + 1 / n ( 1 - B ) ) )

where T(1) is the amount of time required to run the program, T(n) is the amount of time required to run the parallel program and B is the amount of the program in which is serial.

With brute force ray intersection, all the loops can be parallelised, therefore meaning that B would be 0 as none of the code is serial and all the intersection function is within these loops.

Part 2.2 – Calculation of parallel performance indicators for Brute-Force Ray Intersection

Amdahl’s Law Research

For the analysis of the parallel performance of the brute-force ray intersection, a cube made up of 1500 triangles will be used for testing and calculating results along with a quad core processor.

The speed with 4.9361 would calculate S = 4.9361 / ( 4.9361 ( 0 + 1 / 8 ( 1- 0 ) ) ) so S = 8.

Therefore, following this calculation, it is possible to simplify to S = 1 / ( (1 – alpha ) + ( alpha / n ) ) producing the same result of 8.

S = 1 / ( ( 1 – 1 ) + ( 1 / 8 ) ) = 8.

Following this with different amounts of processors, a linear graph is produced with direct correlations between speedup and the number of processors and also efficiency (speedup / number of processors) and the number of processors. These are displayed in the graphs below in Fig 7 and 8

Figure 7 (left): A graph showing the correlation between speedup and No. of processors with brute force ray intersection using parallelisation.

Figure 8 (right): A graph showing the correlation between efficiency and No. of processors with brute force ray intersection using parallelisation.

Operation Calculations

Throughout the brute force intersection algorithm, there is a total of 112 operations within the program assuming 1 triangle and 1 ray, however with 1500 triangles and 65,536 rays (256 x 256) there is 8,159,275,500 operations. As the intersection function is completed within 5.5538 seconds with 1 core, this means 1,469,133,836 operations are done per second.

With 2 cores, the operations are reduced from 8,159,275,500 operations to 4,079,637,750 operations with parallelisation.

With 3 cores, the operations are reduced from 8,159,275,500 operations to 2,719,758,500 operations with parallelisation.

With 4 cores, the operations are reduced from 8,159,275,500 operations to 2,039,818,875 operations with parallelisation.

With 5 cores, the operations are reduced from 8,159,275,500 operations to 1,631,855,100 operations with parallelisation.

With 6 cores, the operations are reduced from 8,159,275,500 operations to 1,359,879,250 operations with parallelisation.

With 7 cores, the operations are reduced from 8,159,275,500 operations to 1,165,610,786 operations with parallelisation.

With 8 cores, the operations are reduced from 8,159,275,500 operations to 1,019,909,437.5 operations with parallelisation.

Speedup and Efficiency Calculations

Through changing the amount of threads used when running the brute force ray intersection algorithm within visual studio, different times were produced and documented as shown in Fig 9.

|  |  |  |  |
| --- | --- | --- | --- |
| number of threads | Times | Speedup | Efficiency |
| 1 | 5.5538 | 1 | 1 |
| 2 | 5.2338 | 1.061141045 | 0.530570522 |
| 3 | 5.1634 | 1.075609095 | 0.358536365 |
| 4 | 5.1022 | 1.088510838 | 0.27212771 |
| 5 | 5.0808 | 1.093095575 | 0.218619115 |
| 6 | 5.0668 | 1.096115892 | 0.182685982 |
| 7 | 5.0529 | 1.099131192 | 0.157018742 |
| 8 | 5.0432 | 1.101245241 | 0.137655655 |

Figure 9: A table containing data collected from changing the number of threads used for brute force ray intersection algorithm.

These times were then used to calculate the speedup which was calculated by dividing each time by the time of 1 thread. A graph was then produced showing the results. This is shown in Fig 10.

Figure 10: A graph showing the correlation between speedup and No. of processors with brute force ray intersection using parallelisation.

Finally, the efficiency was calculated by dividing each speedup by the amount of processors that were used for it, for example the speedup of 1.093095575 used 5 cores so produced an efficiency of 0.218619115. A graph was then produced showing the results. This is shown I Fig 11.

Figure 11: A graph showing the correlation between efficiency and No. of processors with brute force ray intersection using parallelisation.

Part 2.3 – Analysis and Evaluation of the parallel performance of Brute-Force Ray Intersection

In figure 10 in part 2.2, it is clear to see that the more processors added to the program allows for an increase in the speedup of the program, however the more the processors added, the lower the increase in speedup. What this shows is that beyond a certain number of processors, the speedup becomes negligible, meaning that adding more processors doesn’t make enough of a difference to be effective. This is shown after 6 cores within figure 10, as the graph begins plateau.

A similar result is shown within Figure 11 in part 2.2. This graph shows the correlation between efficiency of the program and the number of processors within the system running the program. As the number of processors increases, the efficiency of the program also increases, however, the more processors, the less effect on the efficiency of the program to a point where the number of processors becomes negligible as there is not enough of a difference for it to be effective. This is shown after 6 cores as the graph begins to plateau.

With Amdahl’s law, the equation will always produce a linear value as the value for alpha is 1. What this means is that all the code is parallelisable for the brute force ray intersection function within the program so the results are not of use to this program.

References:

ScratchaPixel, 2016. Ray Tracing: Rendering a Triangle [online]. Available from: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection> [Accessed 15 November 2017].

Möller , T., Trumbore, B., 1997. Fast, Minimum Storage RayTriangle Intersection. [online]. Available from: <https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf> [Accessed 17 November 2017].

Wikipedia, 2017. Möller -Trumbore intersection algorithm. [online]. Available from: <https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm> [Accessed 17 November 2017].

Aaron Michalove, n.d. Amdahl’s Law [online]. Available from: <https://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html> [Accessed 20 November 2017].

Matt Bach, 2015. Estimating CPU Performance using Amdahl’s Law. [online]. Available from: <https://www.pugetsystems.com/labs/articles/Estimating-CPU-Performance-using-Amdahls-Law-619/> [Accessed 22 November 2017].