

## **S0012E - Profiling and Optimization**

Tungjai T. Mady

Computer Game Programming, Lulea University of Technology

S0003E: Game Development Project

18 November, 2024

<b>Analysis Tools.....</b>	<b>3</b>
Chrono Time.....	3
Microsoft Visual Studio Performance Profiling.....	3
AMD uProf.....	3
<b>Optimization.....</b>	<b>4</b>
Original Code.....	4
Raytracer::Raycast.....	5
Optional Class.....	6
Bounding Volume Hierarchy.....	7
Multi-Threading.....	7
<b>Memory Leaks.....</b>	<b>8</b>
<b>Analysis Tools.....</b>	<b>8</b>

## Analysis Tools

### Chrono Time

The first tool that will be used is built in library chrono, this work by getting the time before the raytrace, and then one more after the raytrace. This will give us the time taken for each frame to be created.

```
auto start = std::chrono::high_resolution_clock::now;
rt.Raytrace();
auto end = std::chrono::high_resolution_clock::now();
```

### Microsoft Visual Studio Performance Profiling

This is the build-in profiling tool for Microsoft Visual Studio. The main use of this software is to see where the code spends the majority of the time on creating the frame.

### AMD uProf

This is a profiling tool for AMD chips. This will be mainly used to check and report cache hit/miss rate and thread utilization. The two main things that we will look at are the IC\_MISS\_RATIO and CPI. The result will be a screenshot from the uProf or a text taken from the terminal using a command line argument for uProf. Such as:

“amduprofPcm -m 11 -C .\trayracer.exe”

IC\_MISS\_RATIO is the indicator of how often the cpu fails to fetch instructions from the cache. A lower hit means a better performance.

The Cycle Per Instruction (CPI) indicates how effective the cpu thread is being utilized. The lower the CPI the more effective it is, since it means that the code is optimized, and takes less cycles to complete.

Instructions Per Cycle (IPC) measures the number of instructions a CPU can execute per clock cycle. A higher IPC generally indicates better CPU efficiency, as it means more instructions are completed in each cycle.

# Optimization

## Original Code

Figure 1.1

```
Width: 300
Height: 300
Ray Per Pixel: 1
Sphere Amount: 36
Duration: 66.0381 sec
Total Number of Rays: 90000
Total Rays: 0.00136285MRays/s
```

*Output from code using chrono library (Using Debug Mode)*

Figure 1.2

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
trayracer (PID: 24036)	22230 (100,00 %)	0 (0,00 %)
[System Code] ntdll.dll!0x00007ffae4eeaf08	22219 (99,95 %)	464 (2,09 %)
mainCRTStartup	21754 (97,86 %)	0 (0,00 %)
__scrt_common_main	21754 (97,86 %)	0 (0,00 %)
__scrt_common_main_seh	21754 (97,86 %)	0 (0,00 %)
invoke_main	21754 (97,86 %)	0 (0,00 %)
main	21754 (97,86 %)	0 (0,00 %)
Raytracer::Raytrace	21372 (96,14 %)	1 (0,00 %)
Raytracer::TracePath	21356 (96,07 %)	0 (0,00 %)
Raytracer::Raycast	13900 (62,53 %)	1 (0,00 %)
Raytracer::TracePath	7425 (33,40 %)	0 (0,00 %)

*Visual Studio Profiling on Original Code*

The profiling shows that most of the time the resource is spent on Raytracer::Raycast.

## uProf Command Line Argument

### IPC CORE METRICS

Metric,Core-0  
 Utilization (%),1.03  
 Eff Freq,42.97  
 IPC (Sys + User),0.55  
 CPI (Sys + User),1.81  
 Branch Misprediction Ratio,0.05

### L1 CACHE CORE METRICS

Metric,Core-0  
 IC (32B) Fetch Miss Ratio,0.33  
 Op Cache (64B) Fetch Miss Ratio,0.60  
 IC Access (pti),255.96  
 IC Miss (pti),83.58  
 DC Access (pti),587.72

We can see that the hit rate (  $100 - (83.58 / 255.96) * 100$  ) = 67.346%

### L2 CACHE CORE METRICS

Metric,Core-0  
 L2 Access (pti),154.68  
 L2 Access from IC Miss (pti),98.36  
 L2 Access from DC Miss (pti),36.66  
 L2 Access from HWPF (pti),24.99  
 L2 Miss (pti),80.99  
 L2 Miss from IC Miss (pti),50.87  
 L2 Miss from DC Miss (pti),15.90  
 L2 Miss from HWPF (pti),14.22  
 L2 Hit (pti),69.10  
 L2 Hit from IC Miss (pti),40.21  
 L2 Hit from DC Miss (pti),18.30  
 L2 Hit from HWPF (pti),10.77

## Raytracer::Raycast

Remove the majority of the code. Such as, the sorting unnecessary sorting algorithm, the iteration that does nothing. Remove the while loop, then combine them into one for loop. The code ended up looking like this:

**Figure 2.1**

```
for (Object* object : world){
    auto opt = object->Intersect(ray, closestHit.t);
    if (opt.HasValue()){
        hit = opt.Get();
        assert(hit.t < closestHit.t);
    }
}
```

```

closestHit = hit;
closestHit.object = object;
isHit = true;
numHits++; }

```

Figure 2.2

```

Width: 300
Height: 300
Ray Per Pixel: 1
Sphere Amount: 36
Duration: 2.43482 sec
Total Number of Rays: 90000
Total Rays: 0.0369637MRays/s

```

Output from code using chrono library (Using Debug Mode)

Figure 2.3

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
trayracer (PID: 4884)	14466 (100,00 %)	0 (0,00 %)
[System Code] ntdll.dll!0x00007ffb5874af08	14455 (99,92 %)	728 (5,03 %)
mainCRTStartup	13727 (94,89 %)	0 (0,00 %)
__scrt_common_main	13727 (94,89 %)	0 (0,00 %)
__scrt_common_main_seh	13727 (94,89 %)	0 (0,00 %)
invoke_main	13727 (94,89 %)	0 (0,00 %)
main	13727 (94,89 %)	7 (0,05 %)
Raytracer::Raytrace	13089 (90,48 %)	17 (0,12 %)
Raytracer::TracePath	11825 (81,74 %)	9 (0,06 %)
Raytracer::Raycast	5267 (36,41 %)	64 (0,44 %)
Raytracer::TracePath	4381 (30,28 %)	0 (0,00 %)

Raytracer::Raycast went down to 36.41% from 62.53%

**uProf Command Line Argument****IPC CORE METRICS**

Metric,Core-0  
 Utilization (%),1.88  
 Eff Freq,80.22  
 IPC (Sys + User),0.51  
 CPI (Sys + User),1.97  
 Branch Misprediction Ratio,0.05

**L1 CACHE CORE METRICS**

IC (32B) Fetch Miss Ratio,0.34  
 Op Cache (64B) Fetch Miss Ratio,0.66  
 IC Access (pti),290.15  
 IC Miss (pti),99.80  
 DC Access (pti),604.29

We can see that the hit rate (  $100 - (99.80 / 290.15) * 100$  ) = 65.604 %

**L2 CACHE CORE METRICS**

Metric,Core-0  
 L2 Access (pti),146.99  
 L2 Access from IC Miss (pti),108.27  
 L2 Access from DC Miss (pti),36.45  
 L2 Access from HWPF (pti),23.74  
 L2 Miss (pti),57.37  
 L2 Miss from IC Miss (pti),30.21  
 L2 Miss from DC Miss (pti),13.68  
 L2 Miss from HWPF (pti),13.48  
 L2 Hit (pti),64.85  
 L2 Hit from IC Miss (pti),36.71  
 L2 Hit from DC Miss (pti),17.94  
 L2 Hit from HWPF (pti),10.27

Overall the biggest improvement is the time, the time taken to create one frame went from 60 secs all the way down to 2.4 secs that is 25 times faster than the original. For IPC, L1, and L2 the difference is negligible.

## Optional Class

Completely remove optional class, the HasValue function of that class is used to check if the object is a null pointer or not, therefore we could easily add a similar function in the HitResult class. The Get function is used to return the HitResult, but since we're the HitResult calling it from the HitResult class there's no need to return itself.

**Figure 3.1**

```
bool HasValue() {  
    if (object != nullptr)  
        return true;  
    else  
        return false;  
}
```

*Simplified version of the HasValue() inside the HitResult class.*

Since this is just refactoring the code there's no performance even though Figure 3.2 shows that the frame takes 0.2sec slower this is negligible.

**Figure 3.2**

```
Width: 300  
Height: 300  
Ray Per Pixel: 1  
Sphere Amount: 36  
Duration: 2.6301 sec  
Total Number of Rays: 90000  
Total Rays: 0.0342192MRays/s
```

*Output after removing the Optional class*



Figure 3.3

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
traytracer (PID: 28808)	5003 (100,00 %)	0 (0,00 %)
[System Code] ntdll.dll!0x00007ffc70e6af08	4974 (99,42 %)	457 (9,13 %)
mainCRTStartup	4517 (90,29 %)	0 (0,00 %)
__scrt_common_main	4517 (90,29 %)	0 (0,00 %)
__scrt_common_main_seh	4517 (90,29 %)	0 (0,00 %)
invoke_main	4513 (90,21 %)	0 (0,00 %)
main	4513 (90,21 %)	4 (0,08 %)
Raytracer::Raytrace	3966 (79,27 %)	2 (0,04 %)
Raytracer::TracePath	3606 (72,08 %)	3 (0,06 %)
Raytracer::Raycast	1619 (32,36 %)	17 (0,34 %)
Raytracer::TracePath	1393 (27,84 %)	3 (0,06 %)

*Hot path from Visual Studio after Removing Optional Class*

### uProf Command Line Argument

#### IPC CORE METRICS

Metric,Core-0

Utilization (%),1.03

Eff Freq,42.97

IPC (Sys + User),0.55

CPI (Sys + User),1.81

Branch Misprediction Ratio,0.05

#### L1 CACHE CORE METRICS

IC (32B) Fetch Miss Ratio,0.35

Op Cache (64B) Fetch Miss Ratio,0.64

IC Access (pti),284.21

IC Miss (pti),98.29

DC Access (pti),596.49

We can see that the hit rate (  $100 - (98.29 / 284.21) * 100$  ) = 65.416 %

#### L2 CACHE CORE METRICS

Metric,Core-0

L2 Access (pti),160.95

L2 Access from IC Miss (pti),166.81

L2 Access from DC Miss (pti),37.85

L2 Access from HWPF (pti),24.26

L2 Miss (pti),79.57

L2 Miss from IC Miss (pti),48.80

L2 Miss from DC Miss (pti),17.26

L2 Miss from HWPF (pti),13.51

L2 Hit (pti),74.36

L2 Hit from IC Miss (pti),43.44

L2 Hit from DC Miss (pti),20.09

L2 Hit from HWPF (pti),10.75

As you can see, removing the optional class does not do much for the IPC, or memory. However since it does not have any use, it is better to get rid of it.

## Vec3

Removed 2 bool functions that check if it is zero, and is normalized. The two functions are called on the constructor which stalls the program down a lot. As well as small changes by returning the vec3 without making it a new variable and returning it. The transform function in mat4.h is rework, and return the necessary variable instead of making a variable, add then one by one.

**Figure 4.1**

```
Width: 300
Height: 300
Ray Per Pixel: 1
Sphere Amount: 36
Duration: 1.02845 sec
Total Number of Rays: 90000
Total Rays: 0.0875103MRays/s
```

*Frame now takes 1.03 secs to create from 2.6 secs.*

### uProf Command Line Argument

#### IPC CORE METRICS

Metric,Core-0  
Utilization (%),1.66  
Eff Freq,71.47  
IPC (Sys + User),0.49  
CPI (Sys + User),2.04  
Branch Misprediction Ratio,0.04

#### L1 CACHE CORE METRICS

IC (32B) Fetch Miss Ratio,0.35  
Op Cache (64B) Fetch Miss Ratio,0.67  
IC Access (pti),353.51  
IC Miss (pti),123.75  
DC Access (pti),634.34

We can see that L1 hit rate is 64.994% by calculation:

$$(100 - (123.75 / 353.51) * 100) = 64.994\%$$

#### L2 CACHE CORE METRICS

Metric,Core-0  
L2 Access (pti),163.23  
L2 Access from IC Miss (pti),198.90  
L2 Access from DC Miss (pti),37.81  
L2 Access from HWPF (pti),20.37  
L2 Miss (pti),68.07  
L2 Miss from IC Miss (pti),39.40  
L2 Miss from DC Miss (pti),17.91  
L2 Miss from HWPF (pti),10.75  
L2 Hit (pti),86.32  
L2 Hit from IC Miss (pti),57.88  
L2 Hit from DC Miss (pti),19.20

L2 Hit from HWPf (pti),9.6

**Figure 3.3**

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
↳ traytracer (PID: 26544)	3352 (100,00 %)	0 (0,00 %)
↳ [System Code] ntdll.dll!0x00007ffc70e6af08	3330 (99,34 %)	459 (13,69 %)
↳ mainCRTStartup	2871 (85,65 %)	0 (0,00 %)
↳ __srt_common_main	2871 (85,65 %)	0 (0,00 %)
↳ __srt_common_main_seh	2871 (85,65 %)	0 (0,00 %)
↳ invoke_main	2868 (85,56 %)	0 (0,00 %)
↳ main	2868 (85,56 %)	0 (0,00 %)
↳ Raytracer::Raytrace	2339 (69,78 %)	2 (0,06 %)
↳ Raytracer::TracePath	2030 (60,56 %)	5 (0,15 %)
↳ Raytracer::TracePath	777 (23,18 %)	2 (0,06 %)
↳ Raytracer::Raycast	643 (19,18 %)	44 (1,31 %)
↳ std::vector<Object *,std::allocator<Object *> >::vector<Object *,std::allocator<Object *> >	462 (13,78 %)	1 (0,03 %)

*Hot path from Visual Studio after Vec3 Optimization*

The figure shows the cpu spendless time in the RayTrace function from 79.27% to 69.78 by improving the vec3 class.

Overall having two unnecessary functions really slows the code down. The reason it's unnecessary is there's no point in testing every vector to see if they're normalized or zero vectors. The function to test out those could be called with constant time.

## Multi-Threading

Multithreading is a programming technique that allows the program to run simultaneously in parallel. In this case the architecture of this multithreading technique is influenced by this slack overflow [ThreadPoolClass](#).

Raytracer Class now has AssignJob() which will divide the screen into chunks based on our desired number of chunks, which is equivalent to the number of jobs as well. It will loop through the number of chunks and add it to the QueueJob(vec 2), the queue will first lock the chunkInfo (which is a queue of vec2) by using the mutex lock, the reason is safety since it is a shared resource, and it might override each other, lastly after queuing it will notify one of the thread, the reason will be explained in loop function. Then it will wait until all the jobs are completed before returning RayNum.

The threadpool has a function SpawnThread(), it will get the current hardware concurrency, and spawn the threads base of that number.

The RaytraceChunk function takes in vec2 from ChunkInfo, so it will just loop through the chunk it is given. The core logic of the raytrace hasn't been changed, only refactoring to support vec2 as a parameter. However once the RayTraceChunk function is finished we increase JobComplete by 1, using atomic int, and the method fetch\_add(), this will read and write the variable at the same time, and prevent any override.

The ThreadLoop function, it will be locked until it has been notified from the queue function. It will return if the chunkInfo is empty or bShouldTerminate is true. bShouldTerminate is a safety condition for and setted to be false in the Stop() function.

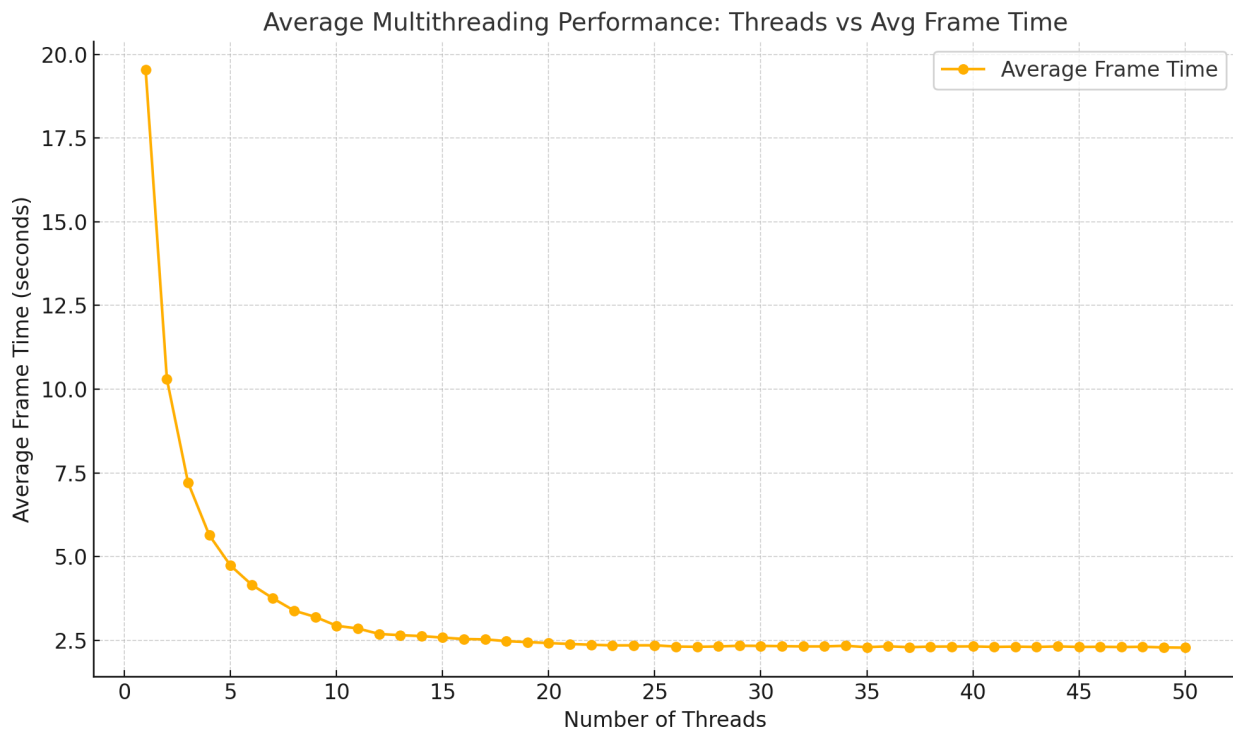
The Stop function sets the bShouldTerminate to false, and kills all the threads.

**Figure 5.1**

```
Width: 1000
Height: 1000
Ray Per Pixel: 2
Sphere Amount: 32
Duration: 21.4399 sec
Total Number of Rays: 2e+06
Total Rays: 0.093284 MRays/s
```

*Before Multithreading Optimization*

The test below (**Figure 5.2**) shows the number of cores used for multithreading and the time taken, the test is being done 5 times, and the average is taken. The parameter for multithreading is the same as in (**Figure 5.1**). This shows that it's being rendered with a single core, and multithreading is significantly better than not multithreading. One of the reasons that it gets worse performance after 6 cores, could be because the number of time spent on creating the threads is not worth it. **Figure 5.2**



*Shows the number of core vs time taken for a single frame, from 1 cores to 50 cores*

Overall there's a trend going down from 1 cores to 50 cores. However there's a very steep drop down from 1 thread to 10 threads, it went from 19 secs to just hovering around 3 secs. Then a small decrease from 10 to 24 (which is the hardware concurrency of the tested pc's hardware) from 3 secs to just around 2.4 secs. Beyond that the trend is still doing down but with minor decrease is a minor difference.

## uProf Command Line Argument

### IPC CORE METRICS

Utilization (%),82.78  
 Eff Freq,3593.72  
 IPC (Sys + User),0.82  
 CPI (Sys + User),1.22  
 Branch Misprediction Ratio,0.01

### L1 CACHE CORE METRICS

Metric,Core-0  
 IC (32B) Fetch Miss Ratio,0.13  
 Op Cache (64B) Fetch Miss Ratio,0.03  
 IC Access (pti),5.54  
 IC Miss (pti),0.70

DC Access (pti),652.69

L1 Hit Rate

$(100 - (0.7 / 5.54) * 100) = 87.365 \%$










The Hit rate went up from 64.994 %

### L2 CACHE CORE METRICS

Metric,Core-0  
 L2 Access (pti),4.35  
 L2 Miss (pti),1.32  
 L2 Hit (pti),2.65

*Uprof AMD Information Showing hit and miss*

**Figure 5.3**

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
 trayracer (PID: 41288)	158173 (100,00 %)	0 (0,00 %)
 [System Code] ntdll.dll!0x00007fff5b08af38	158133 (99,97 %)	478 (0,30 %)
 std::thread::_Invoke<std::tuple<void (__cdecl Raytracer::*)(void),Raytracer *>,0,1>	153868 (97,28 %)	0 (0,00 %)
 std::invoke<void (__cdecl Raytracer::*)(void),Raytracer *>	153868 (97,28 %)	0 (0,00 %)
 Raytracer::ThreadLoop	153868 (97,28 %)	0 (0,00 %)
 Raytracer::RayTraceChunk	153867 (97,28 %)	1176 (0,74 %)
 Raytracer::TracePath	149687 (94,63 %)	466 (0,29 %)
 Raytracer::Raycast	82768 (52,33 %)	5532 (3,50 %)
 Raytracer::TracePath	58675 (37,10 %)	213 (0,13 %)

*The hot path for after the multithreading*

Here we can see that the path itself is different than before. The multithreading works by adding the task in a queue and then the loop thread will keep on looping until all the thread is finished, that's why. At the same time the RayTrace function has been broken down to many small functions, and one of them is GetColor().

## Bounding Volume Hierarchy

The bounding box is useful since instead of testing the ray against all the spheres, we can break down the box and only check spheres within that box.

Class bounding box, the bounding box class contain:

Vec3 Max, Min, Center which is the max vector of that box, same as the min, and the center is the center of the cube.

There's GrowToInclude(sphere), this function will take the sphere in, and check if the sphere is within the box or not, if not it will increase the size of the box to include that sphere.

BoxIntersection(ray, maxDist) this will check if the ray hit the bounding box, and if the distance from the ray origin and maxDist. The method used here is called slab method, known to be an efficient method.

Other than there's Size() which returns the size of the bounding box in vec3.  
SurfaceArea() which returns a float of the volume of the cube.

Overall the bounding box class is a very straightforward class without any major complexity to code. But at the same time it is also the core of using bvh.

```
class BoundingBox
{
public:
    vec3 Min = vec3(0,0,0);
    vec3 Max = vec3(0,0,0);
    vec3 Center = (Min + Max) / 2;

    void GrowToInclude(Sphere* sphere)
    bool BoxIntersection(Ray ray, float maxDist)
    float DistanceTo(const vec3& point) const
    vec3 Size()
    int GetLargestAxis()
    float SurfaceArea()
}
```

The Node class keeps track of all the bounding boxes, number of spheres, and a pointer to its children. This class is significantly more complex than the bounding box, and contains a variety of methods to make sure the function runs smoothly.

Since it keeps track of the number of spheres, it has a function called AddSphere(sphere) which will take in a sphere and call the function GrowToInclude(sphere) so that the bounding box automatically includes that one as well.

Build(vec<spheres>) To make things even easier when building a box, it will take in a vector of sphere, automatically built.

FindBestSplit function will take the size of the bounding box, and then find the length in all of the 3 axes then scale it down by a constant number then we will iterate through it and check which split is the best. Inorder for this to work we have a function called EvaluateSplit, this function uses a method called surface area heuristic, this will estimate the cost which the ray will be traversing.

```
float surfaceAreaA = ASpheres.bounds.SurfaceArea();
float surfaceAreaB = BSpheres.bounds.SurfaceArea();
float surfaceAreaParent = RootAxis->bounds.SurfaceArea();
float costA = (surfaceAreaA / surfaceAreaParent) * (ASpheres.spheres.size());
float costB = (surfaceAreaB / surfaceAreaParent) * (BSpheres.spheres.size());
float overlapPenalty = OverlappedCount * 1.5;
```

It will take the cost of both a and b, and also a penalty of overlapping, since overlapping will cost more work for iterating later on.

```
Node(BoundingBox box, const std::vector<Sphere*> sp)
BoundingBox bounds;
std::vector<Sphere*> spheres;
Node* ChildA = nullptr;
Node* ChildB = nullptr;
void AddSphere(Sphere* sphere)
void Build(const std::vector<Sphere*> spheres)
bool IsLeaf()
int GetTreeSize(Node* root)
void FindNoOverlapSplit(Node* Root, int& SplitAxis, float& SplitPos)
int EvaluateNoOverlapSplit(Node* RootAxis, int Axis, float Pos)
void FindBestSplit(Node* Root, int &SplitAxis, float &SplitPos)
void NormalSplit(Node* Root, int &SplitAxis, float &SplitPos)
float EvaluateSplit(Node* RootAxis, int Axis, float Pos)
void SplitNode(Node* parent, int depth)
bool bInA(Sphere* sp, int axis, float Spos)
bool bInB(Sphere* sp, int axis, float Spos)
bool bInAB(Sphere* sp, int axis, float Spos)
```



In order for this to work we need to cooperate with the raytracer class. The Raytracer has a member called MainNode, first we build the node to get all the spheres, build the bounding box with all that sphere and then split it down.

In the Raycast function instead of iterating through all the objects, we use the Node to get the bounding box that intersects with the ray. The problem here is that there could be many Nodes that intersect with the bounding box, and is a leaf node therefore we have to check against all of them.

We did this by stack method, first we push the MainNode to the stack and then keep on looping, if the bounding box does not intersect the ray then it will be ignored. If the bounding box intersects the ray but it is not a leaf node, then it will keep iterating until it's a leaf node and check only when it's the leaf node.

```
Node* curr;
while (!StackNode.empty()) {
    curr = StackNode.top();
    StackNode.pop();










    if (!curr->bounds.BoxIntersection(ray, closestHit.t))
        continue;

    if (curr->IsLeaf())
        this->HitTest(curr, closestHit, ray);
    else {
        StackNode.push(curr->ChildA);
        StackNode.push(curr->ChildB);
    }
}
```

The HitTest() function will iterate through all the spheres within that node, and check if it is closer to the previous hit, if it's true then it will set the closest to that sphere.

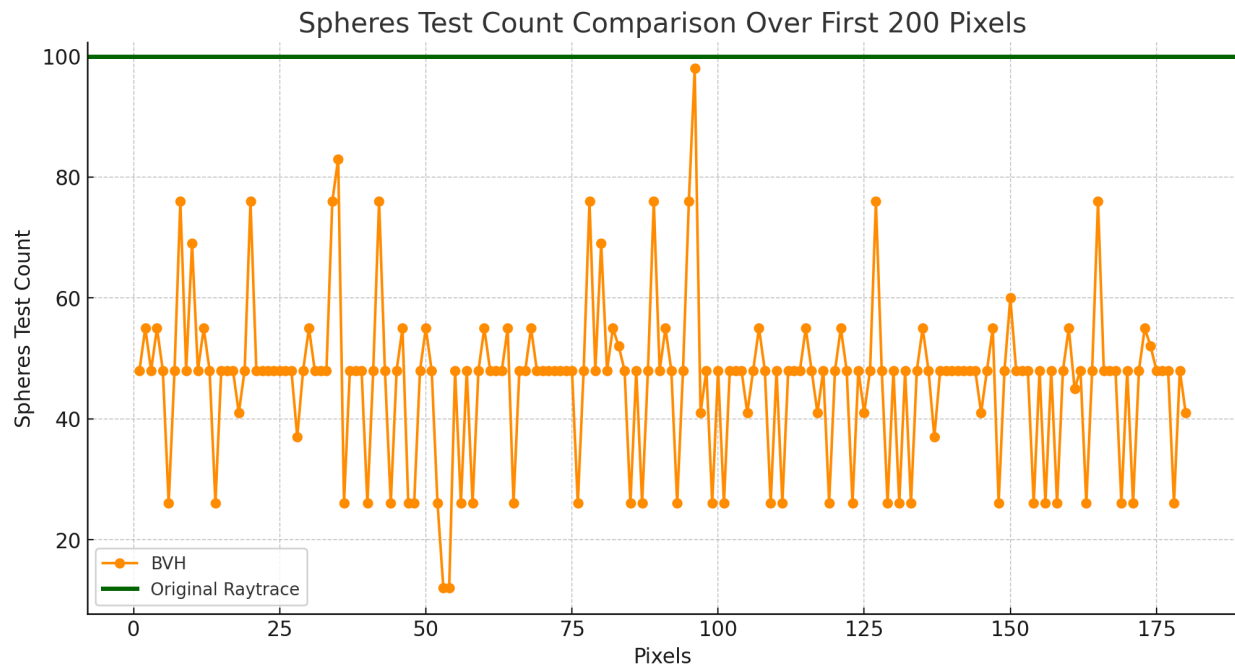
```
Raytracer::HitTest(Node*& node, HitResult& closestHit, Ray ray) {
    HitResult hit;
    for (Sphere* sphere : node->spheres)
    {
        this->test++;
        hit = sphere->Intersect(ray, closestHit.t);
        if (hit.HasValue())
        {
            if (hit.t < closestHit.t) {
                closestHit = hit;
                closestHit.object = sphere;
            }
        }
    }
}
```

Figure 6.1

Hot Path		
Function Name	Total CPU [unit, %]	Self CPU [unit, %]
 traytracer (PID: 19076)	532743 (100,00 %)	0 (0,00 %)
 [System Code] ntdll.dll!0x00007fff5b08af38	532713 (99,99 %)	516 (0,10 %)
 std::thread::_Invoke<std::tuple<void (__cdecl Raytracer::*)(void),Raytracer *>,0,1>	512761 (96,25 %)	0 (0,00 %)
 std::invoke<void (__cdecl Raytracer::*)(void),Raytracer *>	512761 (96,25 %)	0 (0,00 %)
 Raytracer::ThreadLoop	512761 (96,25 %)	0 (0,00 %)
 Raytracer::RayTraceChunk	512760 (96,25 %)	241 (0,05 %)
 Raytracer::TracePath	511702 (96,05 %)	277 (0,05 %)
 Raytracer::BVHRaycast	279552 (52,47 %)	656 (0,12 %)
 Raytracer::TracePath	229909 (43,16 %)	107 (0,02 %)

*This shows that path is very similar to the original raycast*

Figure 6.2

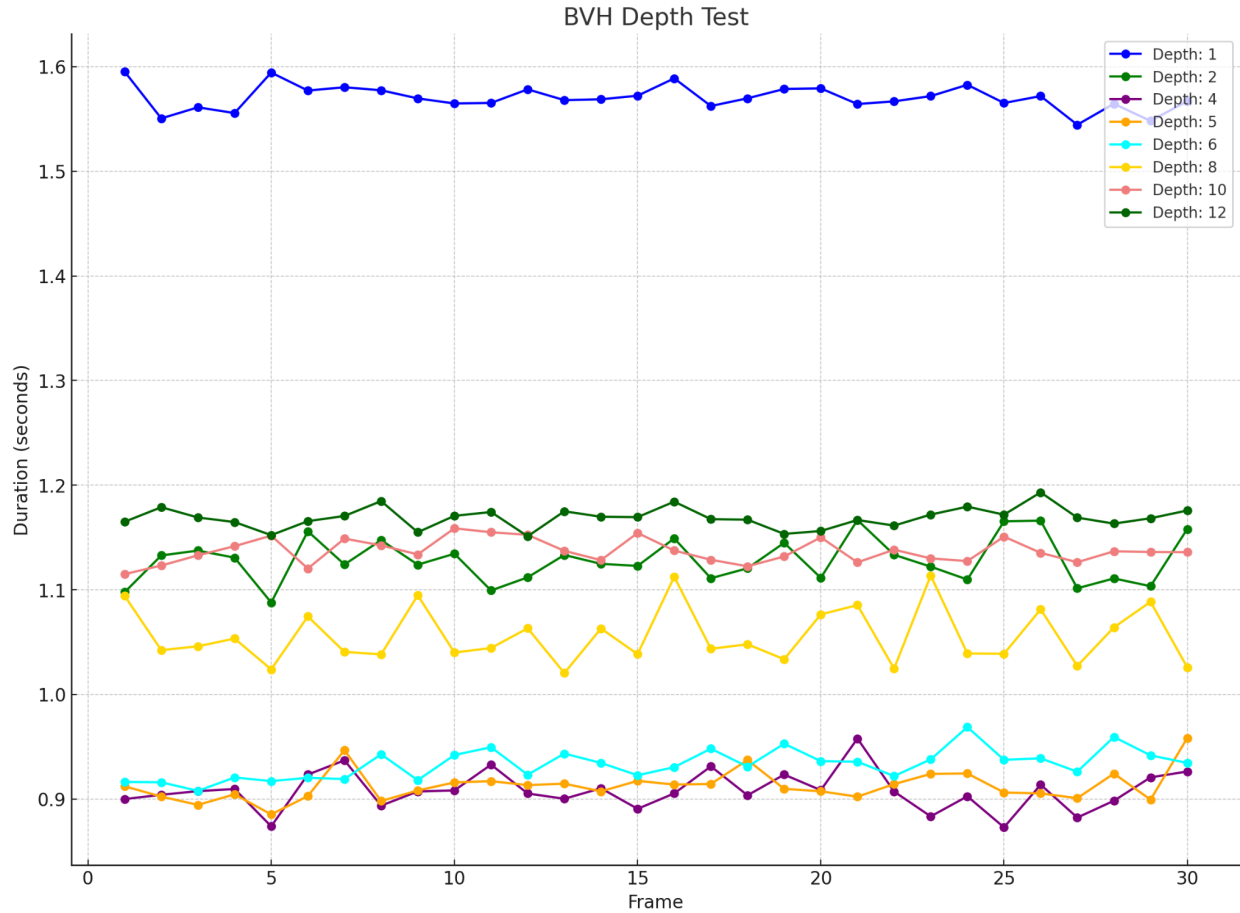


*Comparison between Original Ray Trace and the one that is using BVH*

As the data in **Figure 6.2** shows the number of spheres being tested dropped to around 46 spheres, which is less than half the original Ray trace function without BVH.

The depth of BVH is also being tested, after 12 runs the optimal depth is from 4 -6. One of the reasons this could be the case is that traversing through it using the stack method could be the bottleneck here.

**Figure 6.3**



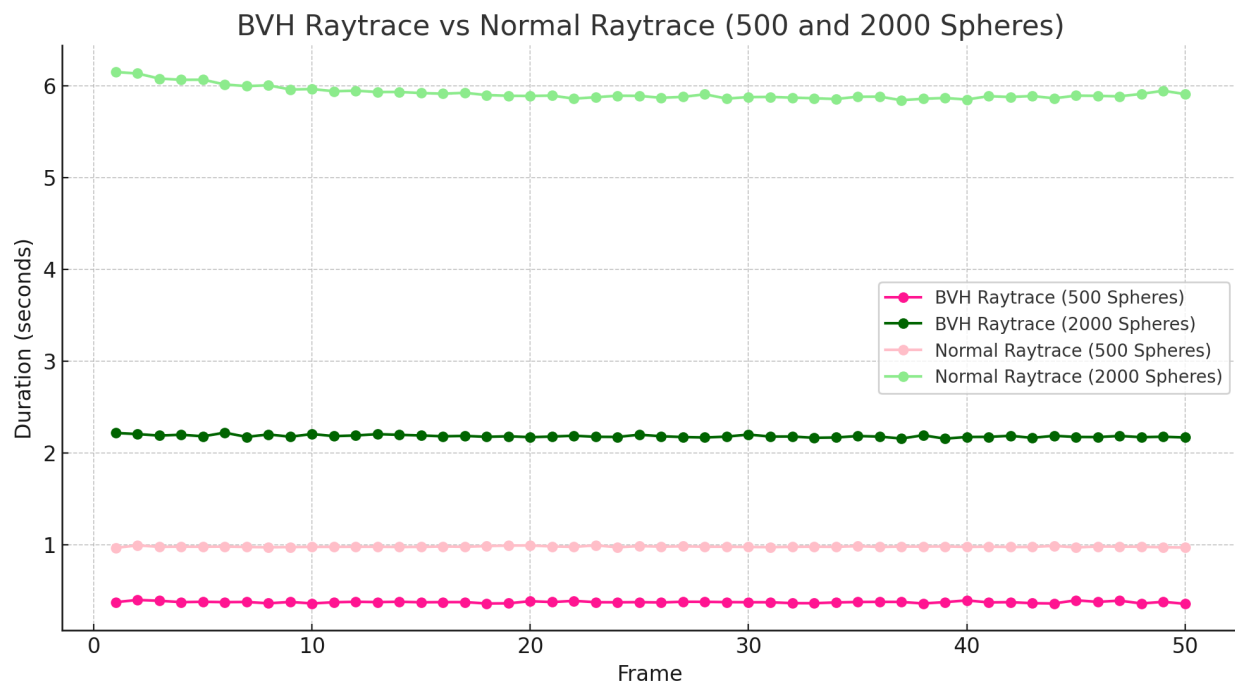
*Test for different BVH Depth*

The test below shows the comparison between BVH Raytrace and Original Raytrace. The best will be done with:

Width	500
Height	500
RPP	2
Max Bounce	5

Since BVH scales very effectively compared to the normal raytrace. The test will be done with 500 spheres, and 2,000 spheres.

**Figure 6.4**



#### *Comparison of BVH and Normal Raytrace*

To conclude both increase respectively to number of spheres, however BVH have a way better scaling than Normal Raytrace, this makes a lot of sense