

Open in app ↗

Sign up

Sign in

Medium

 Search

# Ray Tracing From Scratch: Grid & BVH Comparison



Muhammed Can Erbudak · Follow

8 min read · Jan 29, 2023



Listen

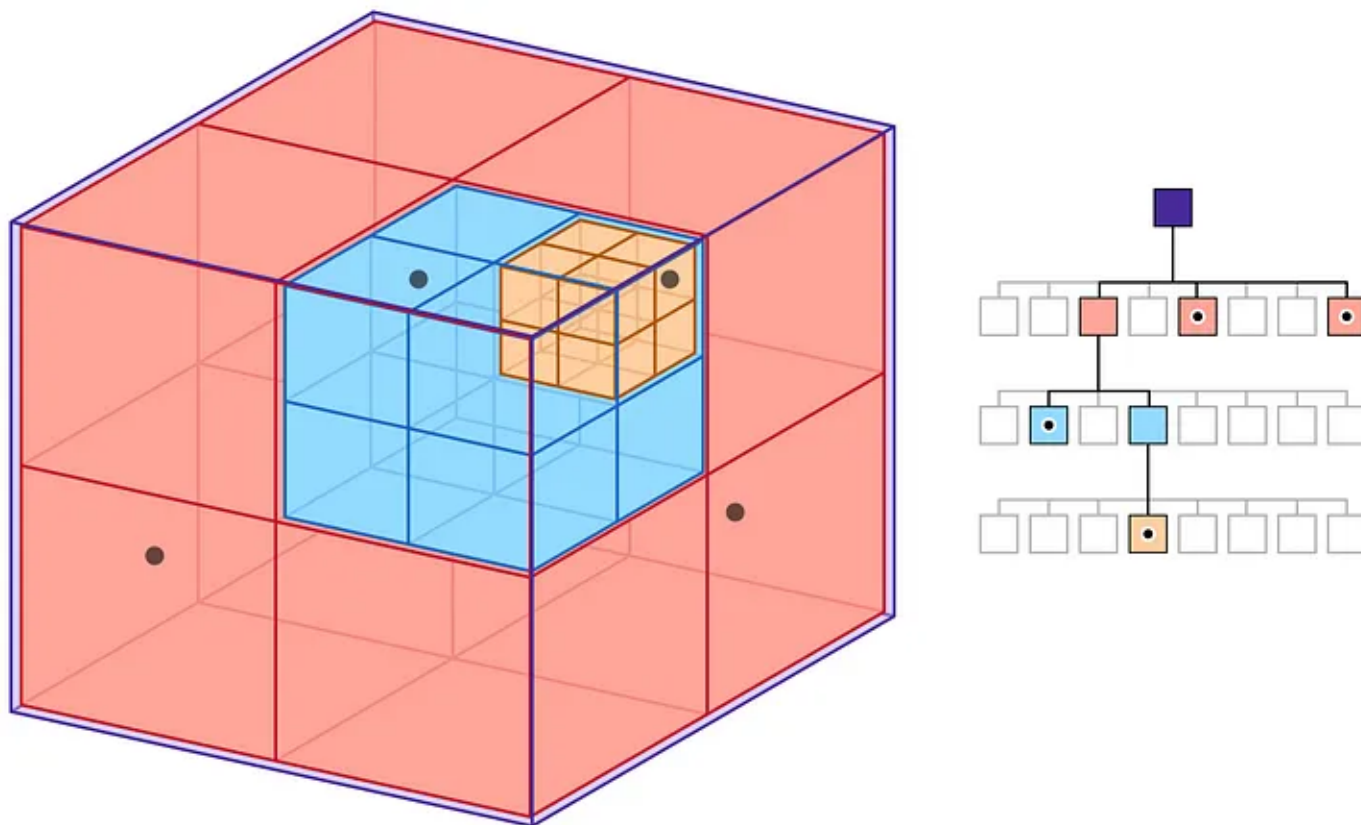


Share

In this blog post, in addition to the previously implemented **bounding volume hierarchy (BVH)** structure, the **regular grid** structure will be implemented and compared with the BVH structure.

Since ray tracing requires a lot of computing power, special data structures called as acceleration structures are used to optimize the process. There are two approaches used for acceleration structures. The first one is dividing the scene hierarchically. For example, bounding volume hierarchy divides the scene by combining bounding volumes of the objects. The details of that structure is explained in the [Optimizations & Transformations](#) blog post.

The other alternative approach is partitioning the space. In that approach, the space is divided into regions and objects are assigned to the region they are inside. Some of the examples are regular grids, octrees, and kd-trees. Regular grids uses more memory than other space partitioning structures. However, it is easier to trace rays using regular grids.

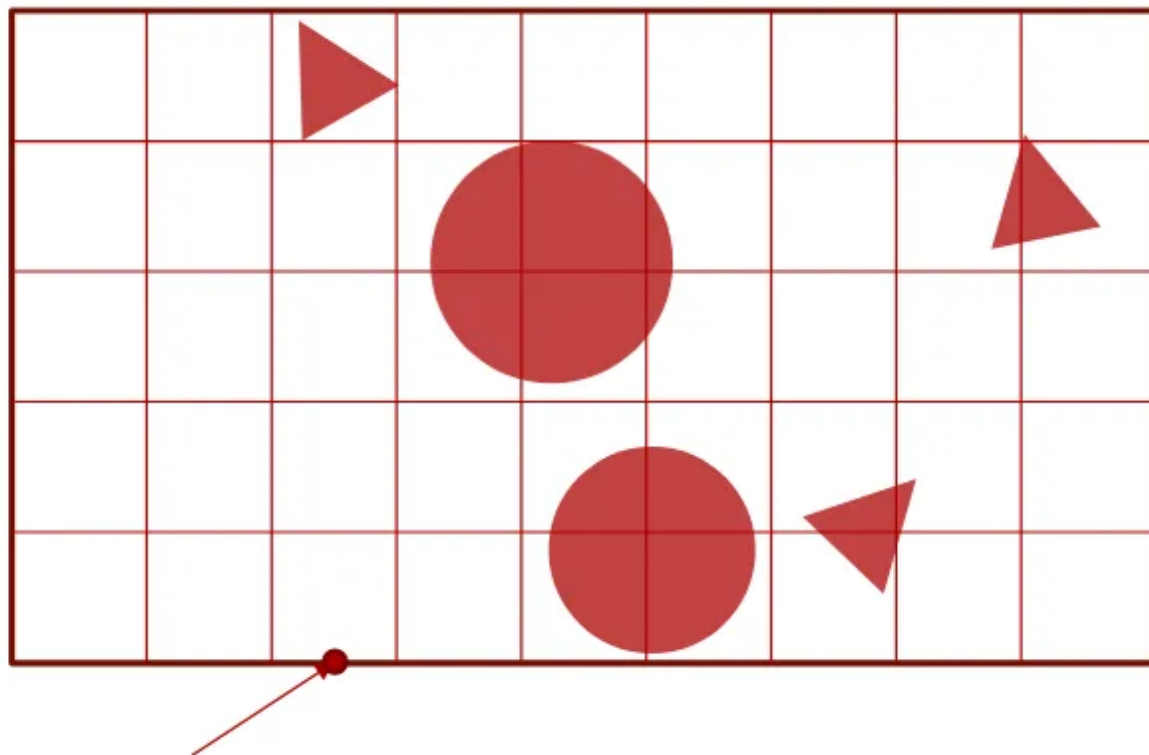


Octree figure. Retrieved from [2].

## Regular Grids

Regular grid structure divides the scene into equal sized voxels. Each voxel stores the primitives within itself. It is possible that one primitive can be stored by different voxels. Also, one voxel can store more than one primitives.

Similar to BVH, implementation of the regular grid structure can be divided into two steps: construction and tracing. The implementation of the construction part for grids is easier than the BVH construction. However, implementing tracing part requires more efforts than the BVH tracing.



Regular grid structure. Retrieved from [1].

Initially, the scene bounding box is generated as boundaries of the grid. This can be done merging the all primitive bounding boxes as if in the BVH.

Then, the grid is divided into voxels in each axis. The voxel count per axis can be decided as  $\sqrt[3]{N}$  where  $N$  is the number of primitives.

After that, the objects are assigned to the voxels. This can be done using the primitive bounding boxes as index boundaries and iterating through that indices.

In order to do that, it is necessary to find the axis ids for a given coordinate. This can be done using the following formula:

$$\vec{\Delta} = \frac{\vec{topRight} - \vec{bottomLeft}}{\# \text{ of voxels per axis}}$$

$$Idx = \frac{\vec{p} - \vec{bottomLeft}}{\vec{\Delta}}$$

Finding grid indices for a given point formula

```
Vec3i Grid::getAxisIds(const Vec3f& position) const
{
    Vec3f delta = (sceneBox.topRight - sceneBox.bottomLeft) / countPerDivision;
    Vec3f effectivePos = position - sceneBox.bottomLeft;

    int xIdx = effectivePos.x / delta.x;
    int yIdx = effectivePos.y / delta.y;
    int zIdx = effectivePos.z / delta.z;

    //degenerate cases
    if(xIdx >= countPerDivision) xIdx = countPerDivision - 1;
    if(yIdx >= countPerDivision) yIdx = countPerDivision - 1;
    if(zIdx >= countPerDivision) zIdx = countPerDivision - 1;

    return Vec3i(xIdx, yIdx, zIdx);
}
```

Finding grid indices for a given point code snippet

Note that there is a degenerate case if at least one of the axes of the point is on the scene bounding box boundary. In this case, the index will be one more the real index value. This generate case will occur since the scene bounding box is generated by merging the primitive bounding boxes and one of the primitives will have boundary point. It can be solved as given in the code.

Then, the objects can be added using the following code snippet:

```
void Grid::addPrimitive(const BoundingBox& boundingBox, const IntersectionData& intersectionData)
{
    Vec3i startIndices = getAxisIds(boundingBox.bottomLeft);
    Vec3i endIndices   = getAxisIds(boundingBox.topRight);

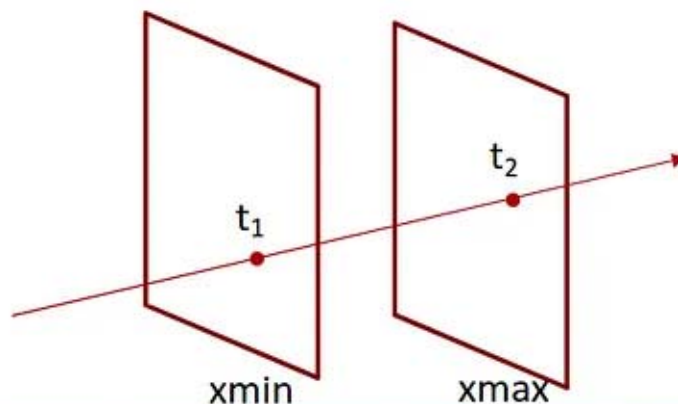
    for(int xIdx = startIndices.x; xIdx <= endIndices.x; xIdx++)
    {
        for(int yIdx = startIndices.y; yIdx <= endIndices.y; yIdx++)
        {
            for(int zIdx = startIndices.z; zIdx <= endIndices.z; zIdx++)
            {
                voxels[xIdx][yIdx][zIdx].possibleIntersections.push_back(intersectionData);
            }
        }
    }
}
```

Adding primitives into voxels

After the construction, trace part is implemented. In order to trace the grid, the initial voxel where the ray enters into the grid must be known. This voxel can be found by using the following operations to find the ray and bounding box intersection, and then the index of the voxel is calculated using the intersection coordinates:

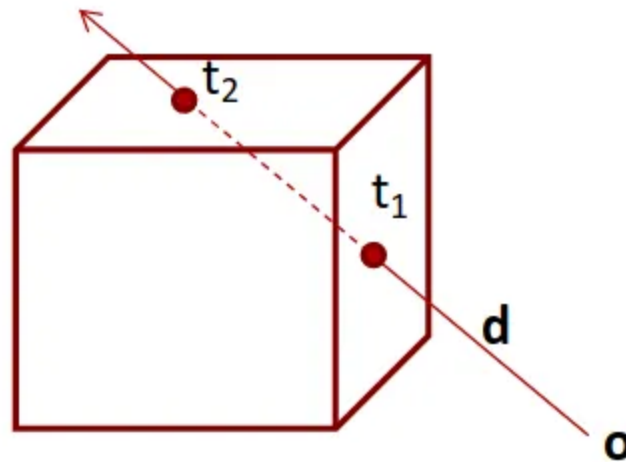
$$t_1 = (x_{min} - o_x) / d_x$$

$$t_2 = (x_{max} - o_x) / d_x$$



Swap  $t_1$  with  $t_2$  if  $t_2$  is smaller!

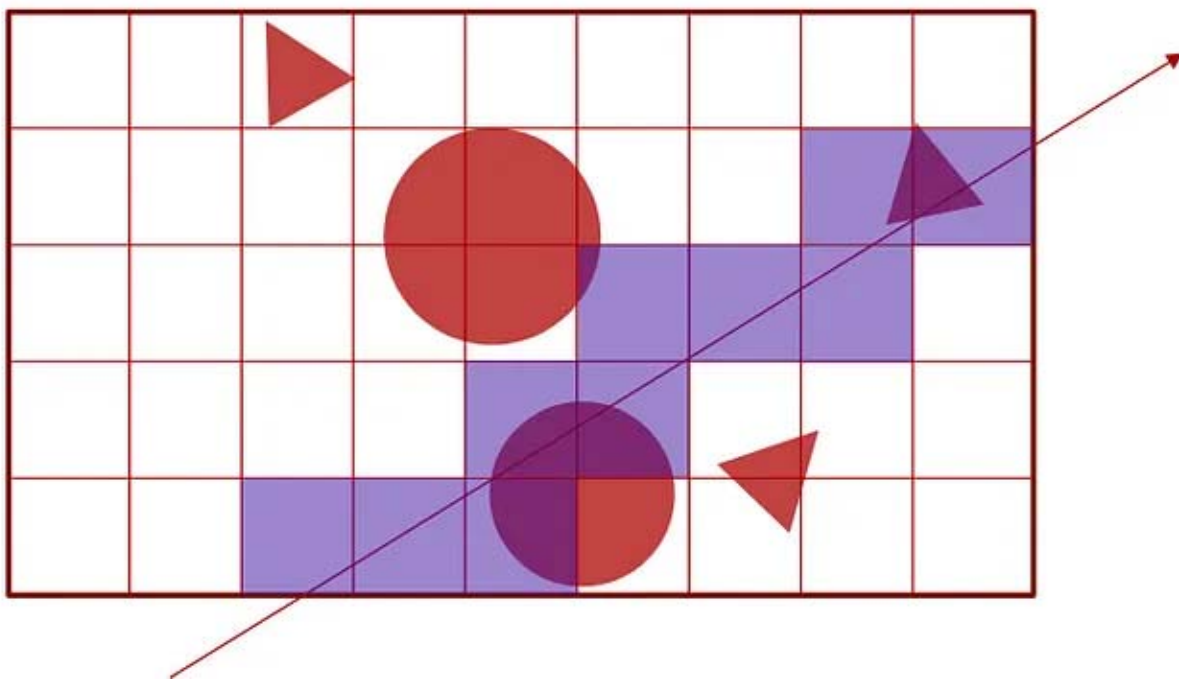
Finding t values. Retrieved from [1].



Ray- bounding box intersection. Retrieved from [1].

Note that it is possible a ray start within the grid, especially for shadow and reflection rays. In this case, ray origin coordinates are used for index calculations.

Then, the grid is traced iteratively. If there are primitives in the current voxel, their intersections with the ray is checked. If there is an intersection, the trace is stopped and that primitive is used for calculations. Also, the intersection with smallest  $t$  value is used for multiple intersections within one voxel case. Otherwise, the trace is continued until the end of the grid boundaries.





Grid traversal. Retrieved from [1].

One difficult point in tracing is to find the next voxel ray passes through knowing the current voxel and the ray. One solution is the algorithm given in [2], which is similar to midpoint rasterization algorithm:

```

if (tMaxX < tMaxY) {
    if (tMaxX < tMaxZ) {
        X = X + stepX;
        if (X == justOutX)
            return(NIL); /* outside grid */
        tMaxX = tMaxX + tDeltaX;
    } else {
        Z = Z + stepZ;
        if (Z == justOutZ)
            return(NIL);
        tMaxZ = tMaxZ + tDeltaZ;
    }
} else {
    if (tMaxY < tMaxZ) {
        Y = Y + stepY;
        if (Y == justOutY)
            return(NIL);
        tMaxY = tMaxY + tDeltaY;
    } else {
        Z = Z + stepZ;
        if (Z == justOutZ)
            return(NIL);
        tMaxZ = tMaxZ + tDeltaZ;
    }
}

```

Finding next voxel algorithm. Retrieved from [2].

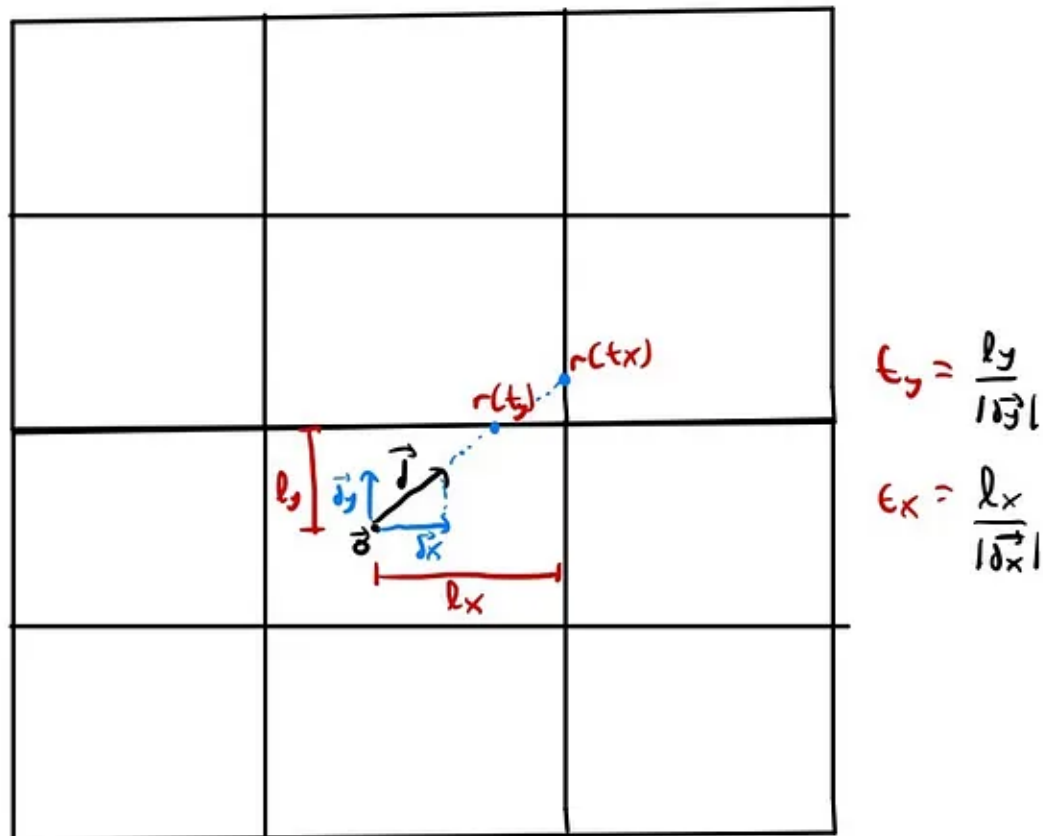
In that algorithm, the grid axis which the ray intersects first is found. In order to do that, the  $t$  values of the ray for each next voxel is found and the next voxel is the one with minimum  $t$  value. Note that the next voxel id can be one after or one before in one grid axis (excluding 45 degree angle ray, which is still works in that algorithm). Thus, the next index is found by adding step values, which is 1 if the ray direction in that axis is positive and -1 if it is negative. Then, the  $t$  value in the current axis is updated since

the next voxel will be used as the current one. Delta values for t, how much ray direction vector is required to pass one voxel, is calculated as follows:

```
Vec3f delta = (sceneBox.topRight - sceneBox.bottomLeft) / countPerDivision;
Vec3f deltaT;
if(ray.direction.x != 0) deltaT.x = abs(delta.x / ray.direction.x);
if(ray.direction.y != 0) deltaT.y = abs(delta.y / ray.direction.y);
if(ray.direction.z != 0) deltaT.z = abs(delta.z / ray.direction.z);
```

tDelta calculations code snippet

There is one part left for the tracing: How the t values (tMax in the algorithm) from the current voxel to the adjacent ones are calculated?



Finding t values

At first, the current index values are found in floating point:



```
//find adjacent axis
Vec3f delta = (sceneBox.topRight - sceneBox.bottomLeft) / countPerDivision;
Vec3f effectivePos = entryPoint - sceneBox.bottomLeft;

float xIdx = effectivePos.x / delta.x;
float yIdx = effectivePos.y / delta.y;
float zIdx = effectivePos.z / delta.z;
```

Finding current indices

Then, the next index for each grid axis where the first intersection in that axis will happen is found. If the ray direction in that axis is positive, it will be the first axis of the next voxel. If it is negative, the axis will be the next axis of the current voxel:

```
int nextIdx;

//going right
if(ray.direction.x > 0)
    nextIdx = int(xIdx) + 1;

//going left
else
    nextIdx = int(xIdx);
```

Finding next intersection axis id

Then, the difference (lx in the “Finding t values” figure) between the entry point (or ray origin) and the next axis will be intersected is calculated as signed:

```
diffX = sceneBox.bottomLeft.x + delta.x * nextIdx - entryPoint.x;
```

Entry point — grid axis intersection calculation

Finally, the  $t$  values are found by dividing the length values into the ray direction values in that axis. Note that it is possible that a ray direction in one axis is zero (e.g. horizontal ray). Thus, the zero division must be checked. For that case,  $t$  value will be set as infinity since it will not be used and it should not be less than the other  $t$  values.

```
Vec3f tMax(MAXFLOAT, MAXFLOAT, MAXFLOAT);
if(ray.direction.x != 0) tMax.x = diffX / ray.direction.x;
if(ray.direction.y != 0) tMax.y = diffY / ray.direction.y;
if(ray.direction.z != 0) tMax.z = diffZ / ray.direction.z;
```

Finding t values

## Performance Comparison

The grid and BVH structures are compared in terms of performance. In order to automatize that process, the following python script is prepared and used:

```
import subprocess
from scipy import stats
from math import sqrt
from numpy import mean

def runRT(rtPath, inputPath, accStrType):
    cmd = "time " + rtPath + inputPath + accStrType
    p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True, stderr=subprocess.PIPE)
    return str(p.stderr.readline())

def parseCmdTimeOutput(timeOutput):
    #get elapsed time
    elapsed = timeOutput.split(" ")[2]
    elapsed = elapsed[:-7]

    #get time info
    minutes, sec = elapsed.split(":")
    sec, ms = sec.split(".")
    return (int(minutes), int(sec), int(ms) * 10)

def inSec(timeTuple):
    return timeTuple[0] * 60 + timeTuple[1] + timeTuple[2] / 1000

def getCI(sampleList):
    sampleMean = mean(sampleList)
    tScore = stats.t.ppf(1 - 0.05, len(sampleList) - 1)
    CI = tScore * stats.tstd(sampleList) / sqrt(len(sampleList))
    return (sampleMean, CI)

def printRes(rtPath, inputPath, accStrType, sampleCount):
    outList = []
    for i in range(sampleCount):
        outList.append(inSec(parseCmdTimeOutput(runRT(rtPath, inputPath, accStrType))))
```

```

CI = getCI(outList)
print(outList)
print("Min: ", min(outList))
print("Max: ", max(outList))
print("CI: ", "{:.3f}".format(CI[0]) + "+" + "{:.3f}".format(CI[1]))

if(__name__ == "__main__"):
    rtPath = "../advanced-raytracing/raytracer "
    inputPathList = [
        "../hw1/inputs/simple.xml ",
        "../hw1/inputs/two_spheres.xml ",
        "../hw1/inputs/bunny.xml ",
        "../hw1/inputs/monkey.xml ",
        "../hw1/inputs/chinese_dragon.xml ",
        "../hw1/inputs/scienceTree_glass.xml ",
        "../hw2/inputs/dragon_metal.xml "]

    sampleCount = 10

    for inputPath in inputPathList:
        print("Input: ", inputPath)

        #Print BVH time data
        print("BVH")
        printRes(rtPath, inputPath, "bvh", sampleCount)

        #Print Grid time data
        print("Grid")
        printRes(rtPath, inputPath, "grid", sampleCount)

        print("")

```

At first the **time** cli tool is used by running the executable and using time command through Python via **runRT** function. **subprocess** module is used for that purpose.

After that, the output of the **time** command is parsed using **parseCmdTimeOutput** function. Note that milliseconds information is stored in precision two. Thus, it is multiplied with 10. Also, the parsed data is transformed to seconds using **inSec** function.

Note that running and testing inputs once do not provide statistically plausible results. Thus, they ran 10 times. Moreover, their statistical **confidence interval (CI)** is

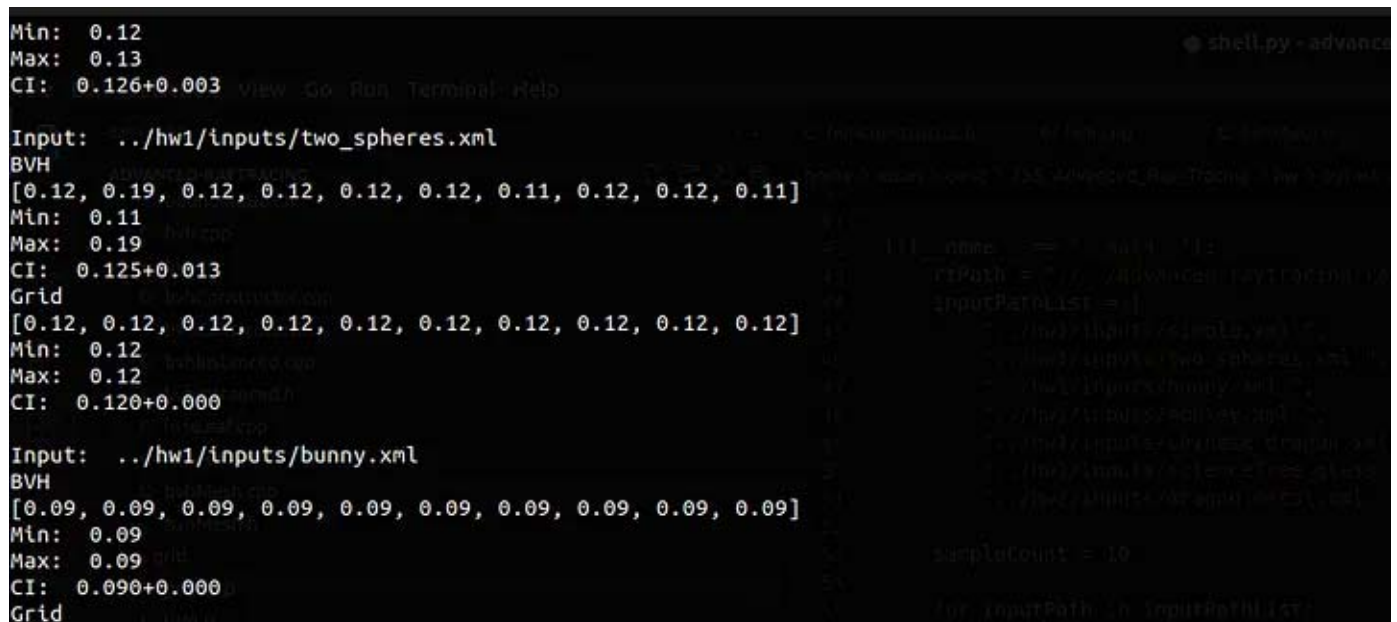
calculated as follows, where  $\bar{x}$  is the mean,  $s$  is the standard deviation,  $z$  is the  $z$  score and  $n$  is the sample size which is 10:

$$\bar{x} \pm z \frac{s}{\sqrt{n}}$$

Confidence Interval (CI) calculation. Retrieved from [5].

Since the sample size is less than 30, t-score is used instead of z-score. Also, degree of freedom is  $10 - 1 = 9$  and the 95% confidence level is used. For that purpose, `getCI` function is used.

Finally, the outputs are printed using `printRes` function. One of the example outputs is:



```
Min: 0.12
Max: 0.13
CI: 0.126+0.003

Input: ../hw1/inputs/two_spheres.xml
BVH
[0.12, 0.19, 0.12, 0.12, 0.12, 0.12, 0.11, 0.12, 0.12, 0.11]
Min: 0.11
Max: 0.19
CI: 0.125+0.013
Grid
[0.12, 0.12, 0.12, 0.12, 0.12, 0.12, 0.12, 0.12, 0.12, 0.12]
Min: 0.12
Max: 0.12
CI: 0.120+0.000

Input: ../hw1/inputs/bunny.xml
BVH
[0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09]
Min: 0.09
Max: 0.09
CI: 0.090+0.000
Grid
```

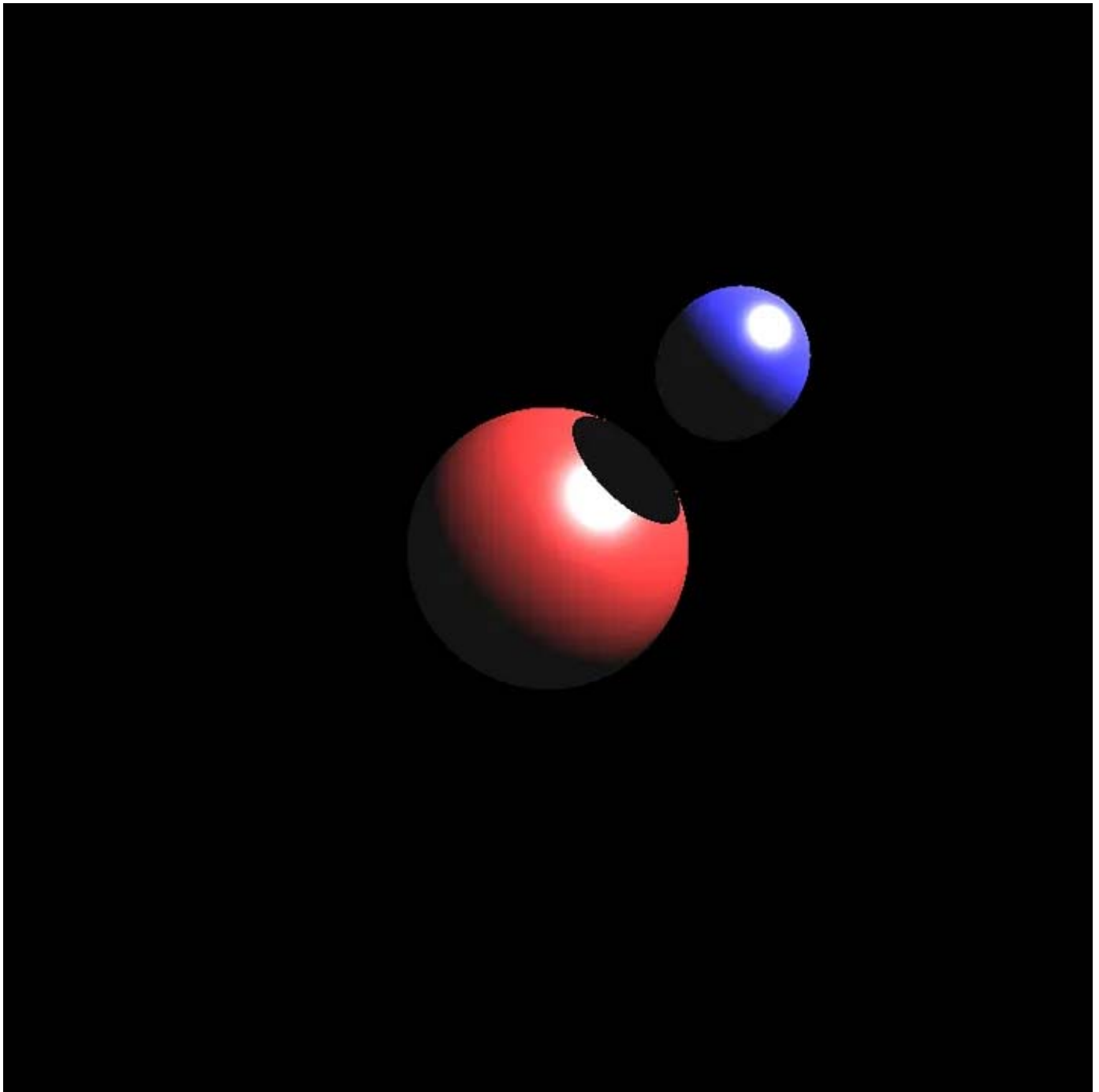
Example output for automation

The results are given as table:

Input Name	Rendering Time					
	BVH			Grid		
	min	max	CI	min	max	CI
simple.xml	0.12 s	0.13 s	$0.123 \pm 0.003$ s	0.12 s	0.13 s	$0.126 \pm 0.003$ s
two_spheres.xml	0.11 s	0.19 s	$0.125 \pm 0.013$ s	0.12 s	0.12 s	$0.120 \pm 0.000$ s
bunny.xml	0.09 s	0.09 s	$0.090 \pm 0.000$ s	0.11 s	0.11 s	$0.110 \pm 0.000$ s
monkey.xml	0.43 s	0.52 s	$0.468 \pm 0.016$ s	0.97 s	1.02 s	$1.001 \pm 0.010$ s
chinese_dragon.xml	3.33 s	3.45 s	$3.415 \pm 0.020$ s	1.51 s	1.61 s	$1.546 \pm 0.018$ s
scienceTree_glass.xml	0.71 s	0.81 s	$0.746 \pm 0.023$ s	15.81 s	19.4 s	$17.340 \pm 0.852$ s
dragon_metal.xml	11.37 s	11.89 s	$11.588 \pm 0.084$ s	161.8 s	171.85 s	$166.576 \pm 1.993$ s

Results table

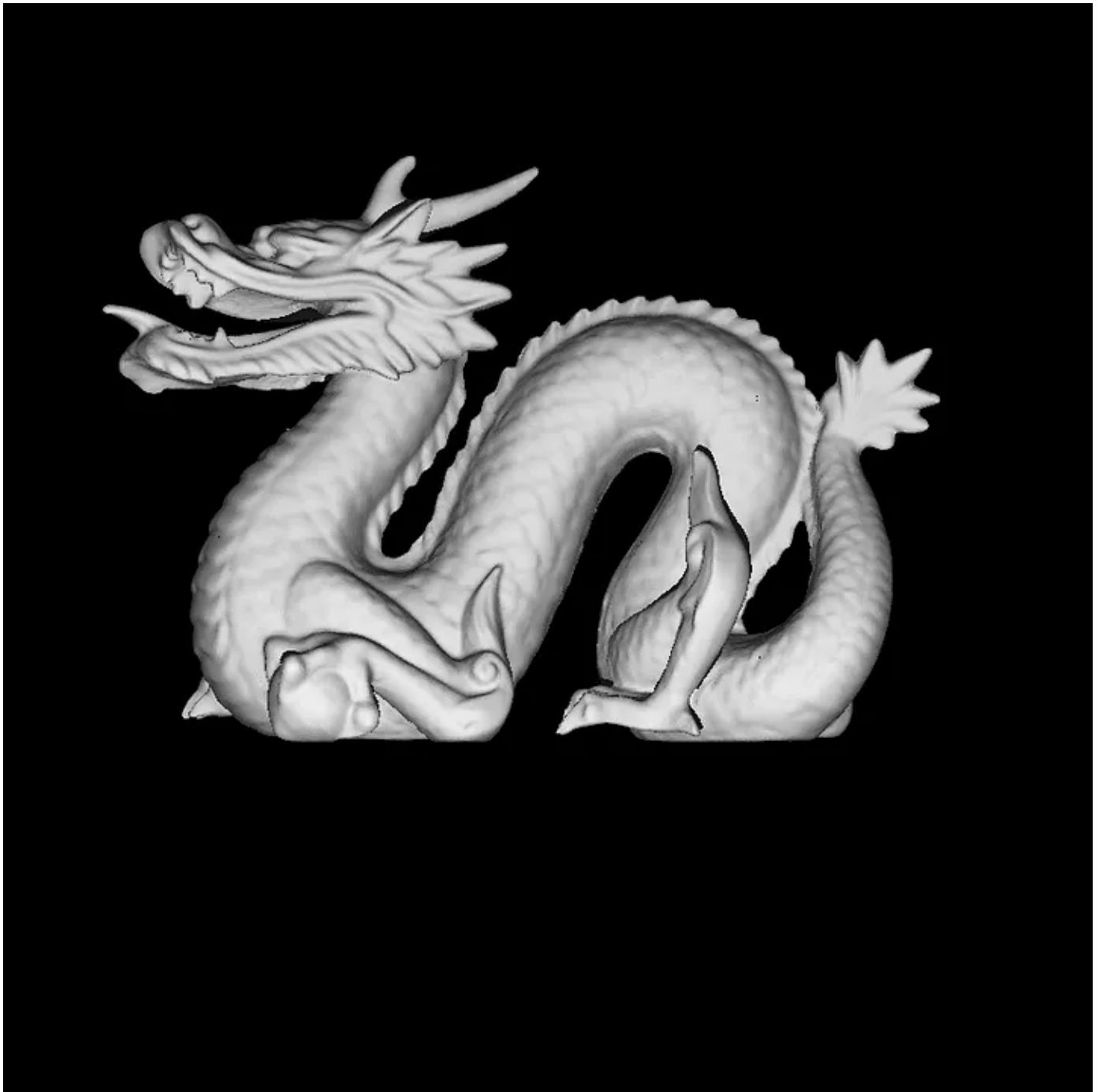
As can be seen in the results, simple scenes are almost the same in terms of performance for both structures.



Two spheres

On the other hand, the Grid structure outperforms the BVH if the scene has one mesh and the voxels do not have too many primitives. One of the example is that Chinese dragon.





Chinese dragon

However, the Grid structure performs poorly if the scene have multiple objects (e.g. Dragon metal) and they are distant since the scene box and voxels get bigger in that case. It results in more primitive in each voxel. It might even get worse as linear time  $O(n)$  as compared to logarithmic time  $O(\log n)$  of the BVH structure. For example, while BVH can render Marching Dragons scene in 5 seconds, the Grid could not finish rendering even after 50 minutes.



Dragon metal



Marching dragons

As a conclusion, the regular grid structure might outperform BVH for single mesh scenes. However, for multiple mesh scenes, BVH is still the better option.

That brings me to the end of the blog post and end of my ray tracing journey for CENG 795: Advanced Ray Tracing course. I would like to thank our instructor Prof. Dr. Ahmet Oğuz Akyüz for such an interesting and great learning experience.

## Sources

1. Akyüz, A. O. (2022). CENG 795 Special Topics: Advanced Ray Tracing. Middle East Technical University.
2. [Amanatides, J., & Woo, A. \(1987\). A Fast Voxel Traversal Algorithm for Ray Tracing. University of Toronto.](#)
3. <https://developer.apple.com/documentation/gameplaykit/gkoc-tree>
4. <https://www.the-analytics.club/python-shell-commands>
5. <https://www.statisticshowto.com/probability-and-statistics/confidence-interval/>

Ray Tracing

Grid

Bvh

Comparison

Optimization



Follow

## Written by Muhammed Can Erbudak

11 Followers · 3 Following

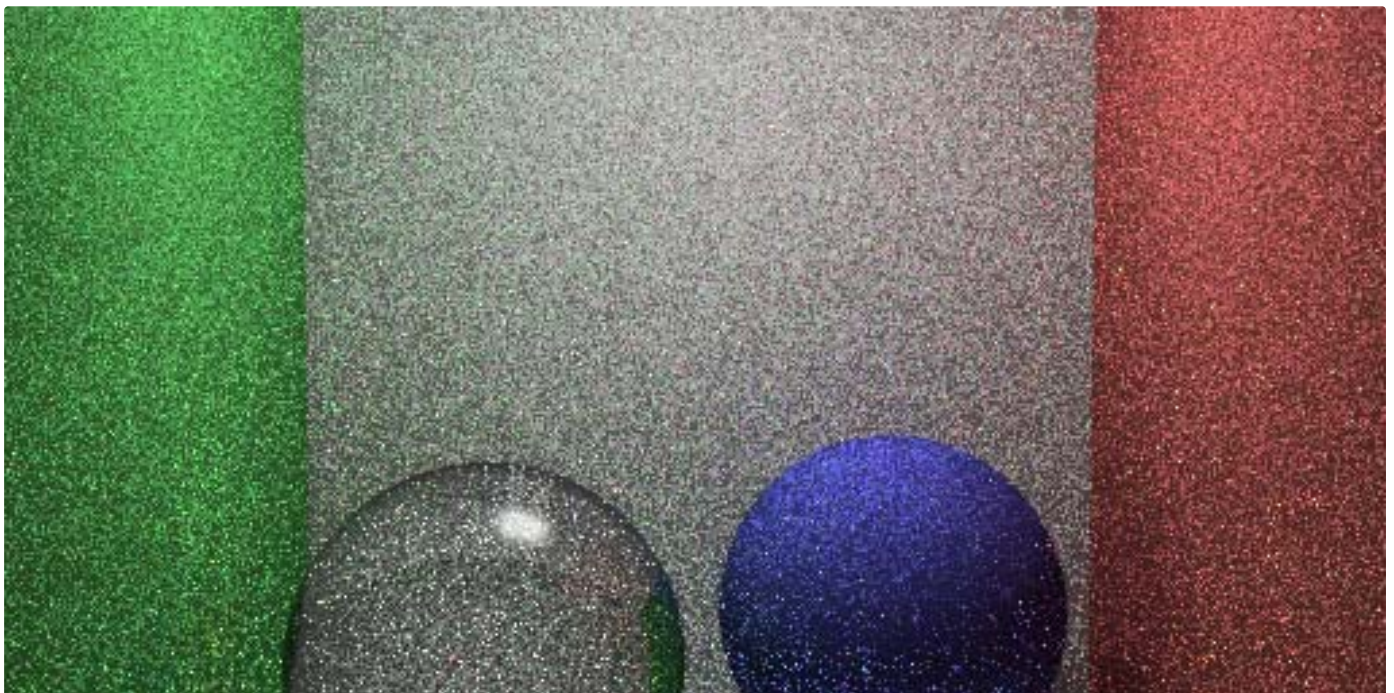


No responses yet

What are your thoughts?

Respond

## More from Muhammed Can Erbudak



 Muhammed Can Erbudak

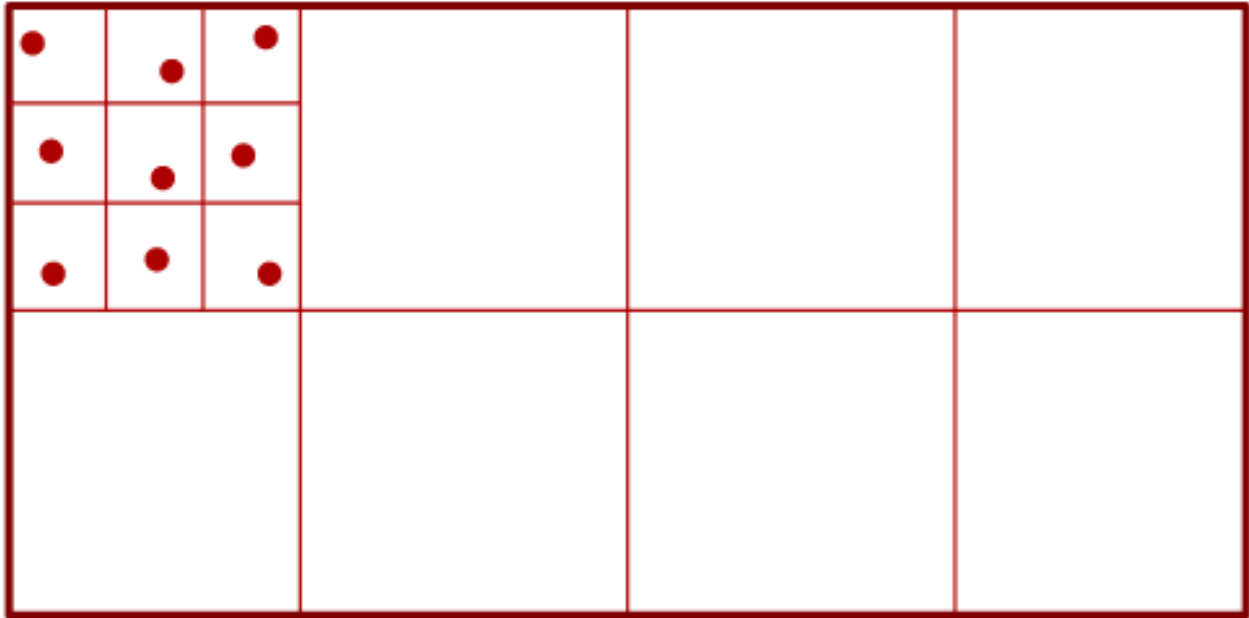
### Ray Tracing From Scratch: BRDFs, Object Lights & Path Tracing


The current ray tracer does not support indirect lighting from non-emissive objects. In order to provide that, the method called path...

Jan 26, 2023  4





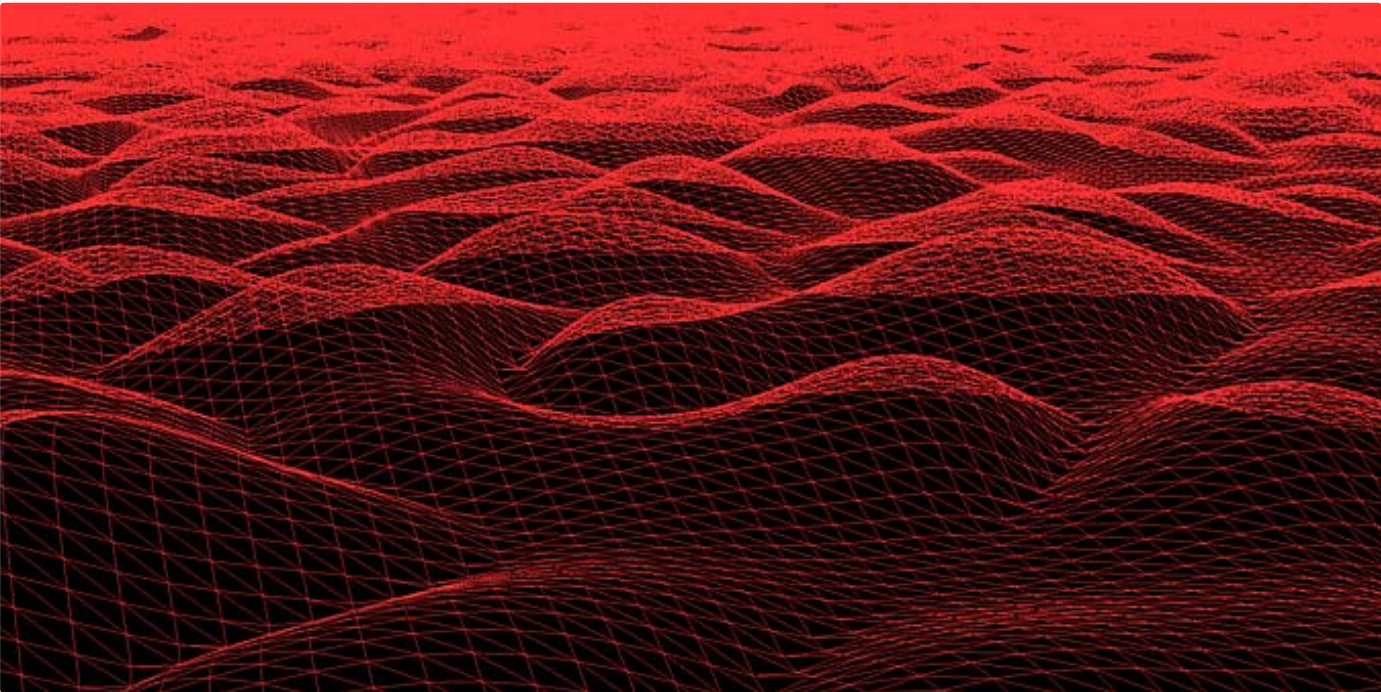



 Muhammed Can Erbudak

## Ray Tracing From Scratch: Multisampling & Distribution Ray Tracing

The previous versions of the ray tracer have good-looking and realistic results. However, they do not have some details such as soft...

Dec 1, 2022



 Muhammed Can Erbudak

## Terrain Rendering via Perlin Noise and OpenGL Geometry Shaders

Perlin Noise one of the techniques used for procedural generation. In this blog post, how a procedural terrain can be generated using...

Jun 16, 2022 🖱 1



 Muhammed Can Erbudak

## Ray Tracing From Scratch: Texture, Normal & Bump Mapping

The current version of the ray tracer is capable of rendering realistic scenes. However, one important feature is missing: Textures...

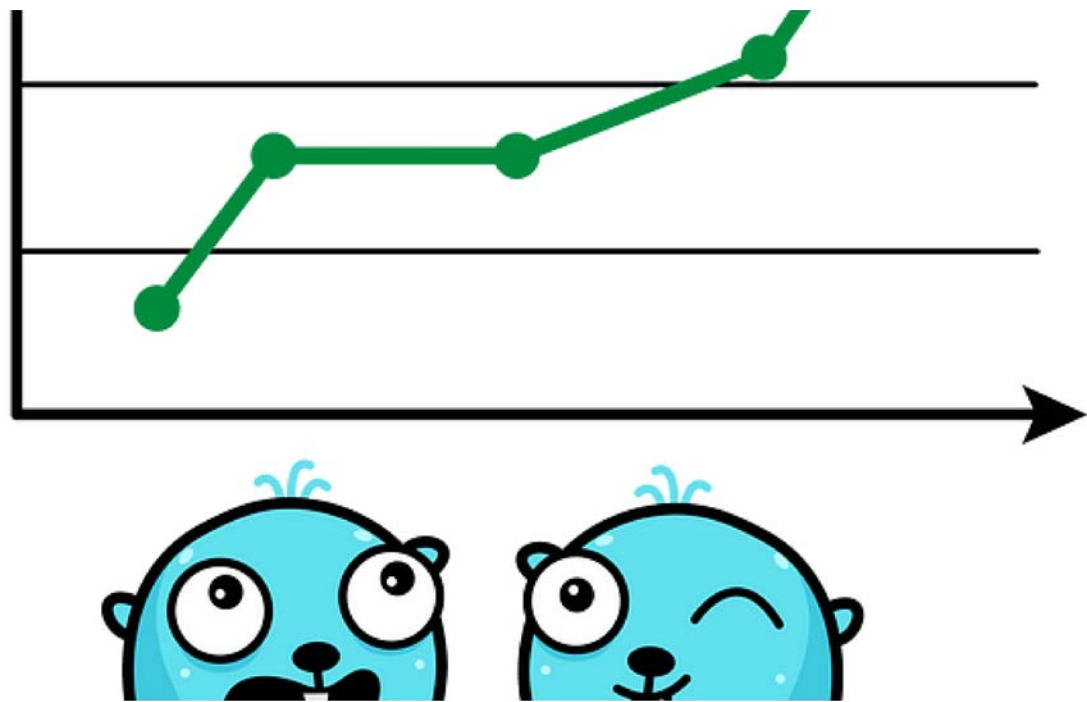
Dec 20, 2022




See all from Muhammed Can Erbudak



Recommended from Medium



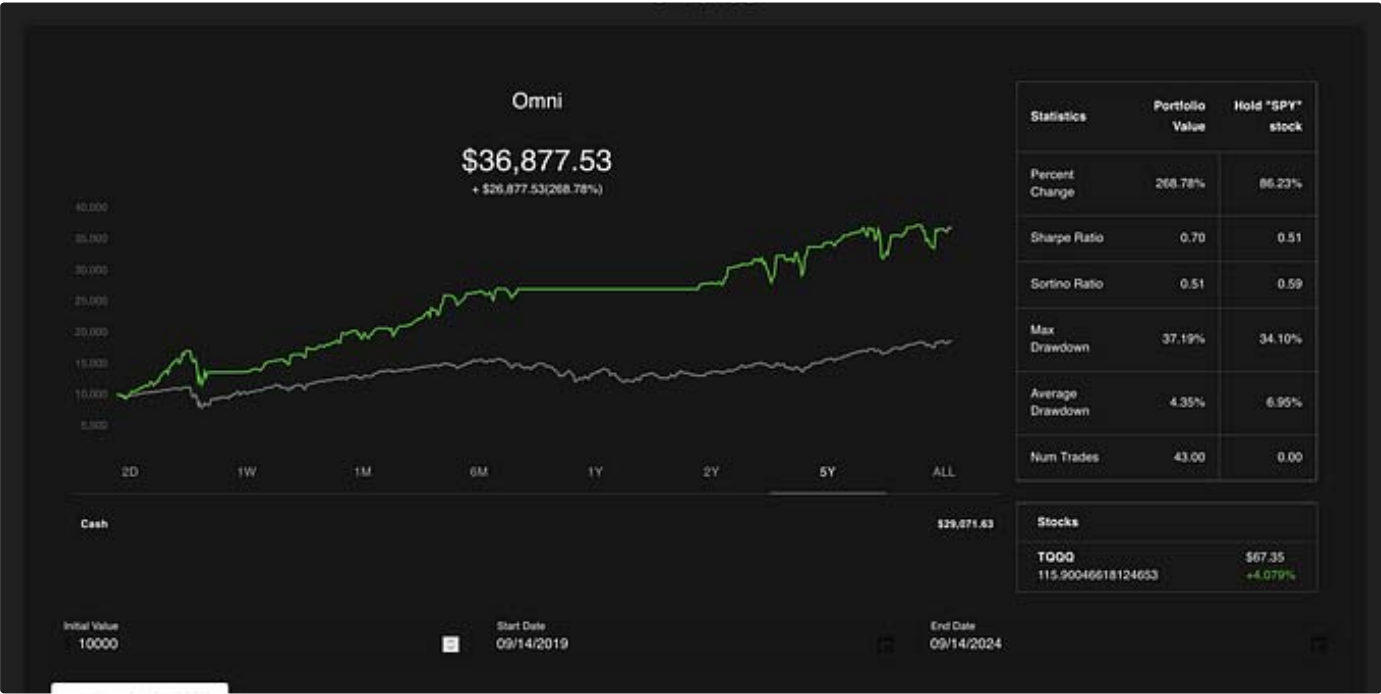
 Salvatore Gonda

**Go & BPMN—A programmatic approach**

A short foreword

Oct 4, 2024  18





In DataDrivenInvestor by Austin Starks

# I used OpenAI's o1 model to develop a trading strategy. It is DESTROYING the market

It literally took one try. I was shocked.

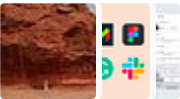
★ Sep 15, 2024    🖐️ 8.1K    💬 197



## Lists



**Icon Design**  
38 stories · 471 saves

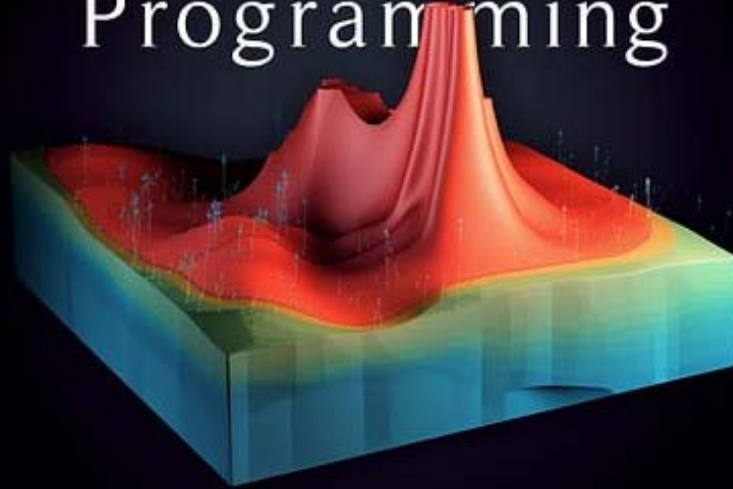




**Interesting Design Topics**  
258 stories · 927 saves



**Natural Language Processing**  
1881 stories · 1515 saves

# Nonlinear Programming



 In Towards Data Science by Maxime Labonne 

## The Art of Spending: Optimizing Your Marketing Budget with Nonlinear Optimization

Introduction to CVXPY to maximize marketing ROI

 May 23, 2023  302  2



## Always Free

### 24 GB RAM + 4 CPU + 200 GB



@harendra2



@harendra21



@harendra21



Harendra

## How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

★ Oct 26, 2024 🖱 8.4K 💬 135



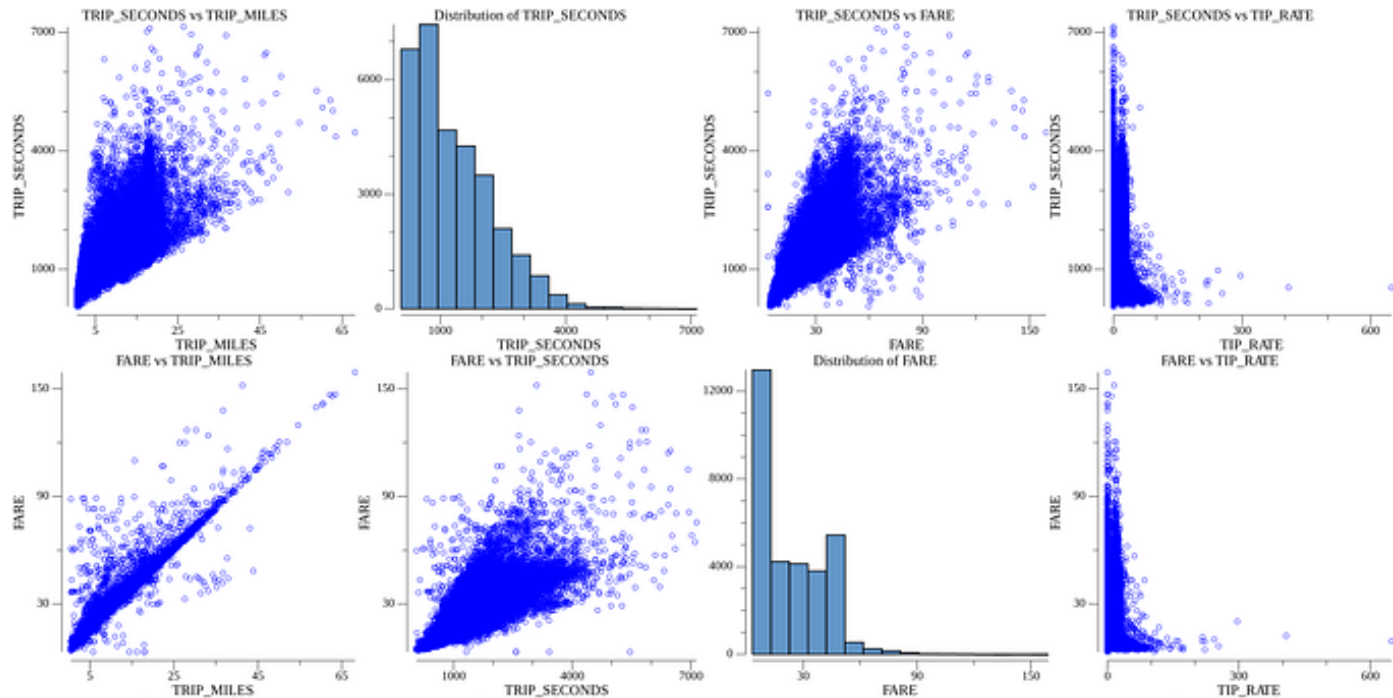
 Ari Joury, PhD

## Understanding The “Why”: 10 Techniques for Causal Inference

With the right tools you can get some pretty deep insights

★ Dec 23, 2024 🖱 353 💬 5





In Level Up Coding by Davide Marro

# Machine Learning in Go—Part 1

When we think of machine learning, Python dominates the conversation. Its extensive libraries and frameworks, such as TensorFlow, PyTorch...

Dec 23, 2024 56



See more recommendations