Milestone

Abstract -

To design a python-based render engine that performs raytracing and outputs image files. Using multiprocessing library of python, multiple images are rendered and the time taken is compared when rendering without multiprocessing. External GPU (CUDA cores) has been integrated to test the efficiency of distributed computing. In the distributed architecture few of the tasks are shared between CPU and GPU to increase performance.

Introduction -

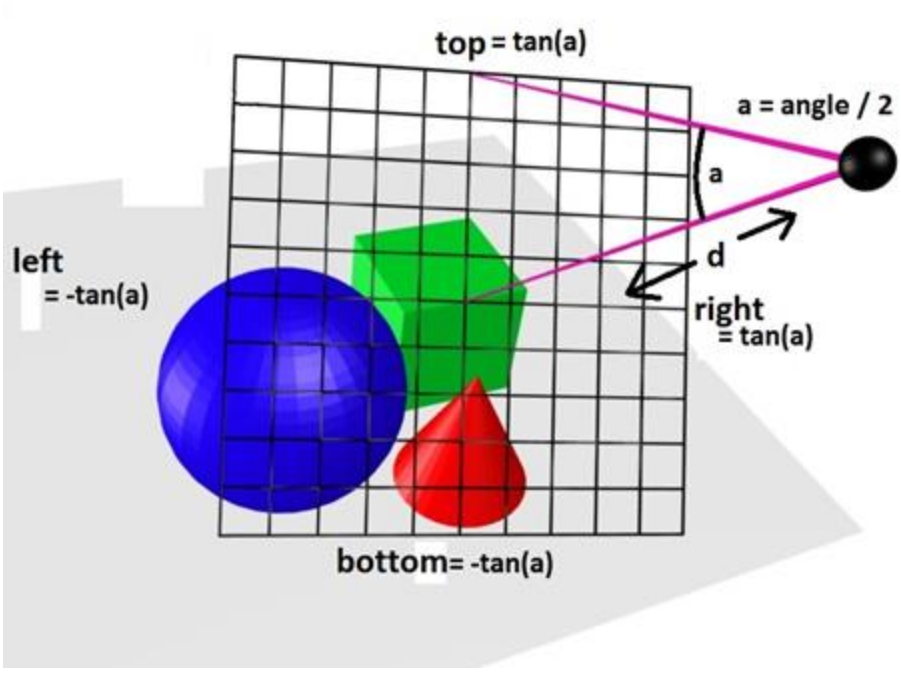
Rendering engines have always intrigued me, in order to render a scene as close to reality means that human has understood everything about light and materials. This paper focuses on implementing a small render engine that takes shapes(object) as coordinates in a 3d space, defines material properties to it applies light interaction on it to render the output.

The main goal of this project is to run the render code in 3 different ways, first we see how the render works without using parallelism, secondly, we see how running the code on multiple CPU cores increase speed and Thirdly we make use of an external GPU to further increase the efficiency.

In computer graphics, ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion.

Background -

Objects in the scene are defined by polygons, this is called mesh and the quality of the render highly depends on the details of the mesh, in many applications a high polygon mesh is compressed to low polygon mesh to decrease the processing time taken. For the scope of this project, objects are defined by coordinates.



Methodology -

Used volume ray casting then applied raytracing to get the RGB values onto the defined screen. These RGB values are then used in creating a png file. A total of around 30 image files are created which are then used to create an animated video. In the first case these image files are created sequentially, in the second case, the image file creation process is run parallelly. In the third case the same parallel code is used but one of the calculation is run on GPU.

Experiments/Analysis -

The core render program was then tested using concurrent programming with different inputs. Using multiprocessing library has shown 50% increase in efficiency. The efficiency still majorly depends on the number of objects and the distance from camera.

When there are no objects in the scene, the render time is almost close to 0, to simulate this result the position of the camera is changed to a point in the scene where there are no objects, similarly when there are more objects in the scene, the time taken to render increases exponentially. To test this scenario, more objects were added to the scene and in this case more spheres.

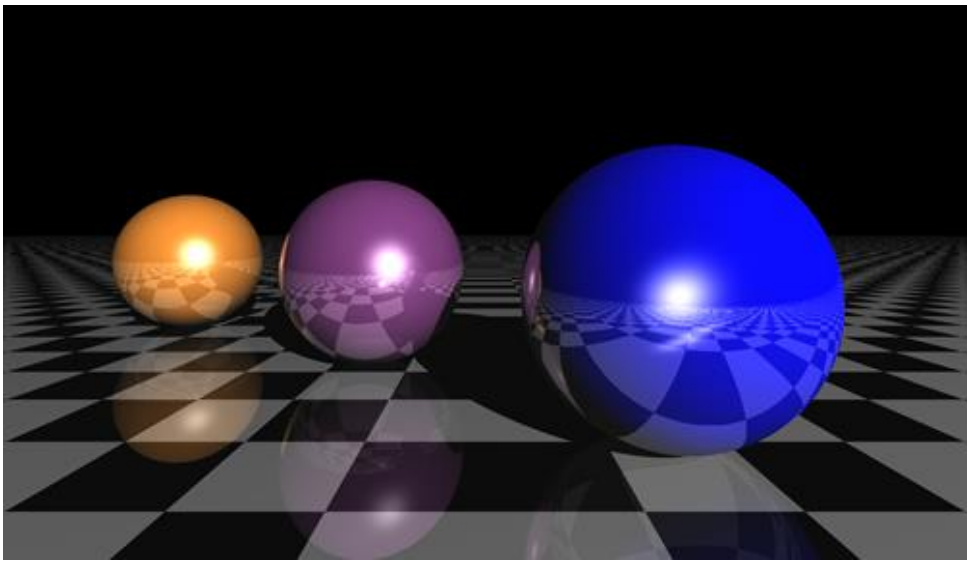
In order to get the external GPU working with the python code, Pycuda has been used. Anaconda python interpreter provides easier way to use Pycuda. Using Anaconda python interpreter many of the basic operations can be performed in GPU.

GPU are known for processing high volumes of data in parallel, but on the contrary when used alongside CPU for small calculation actually reduced the performance of the rendering due to 2 main reasons.

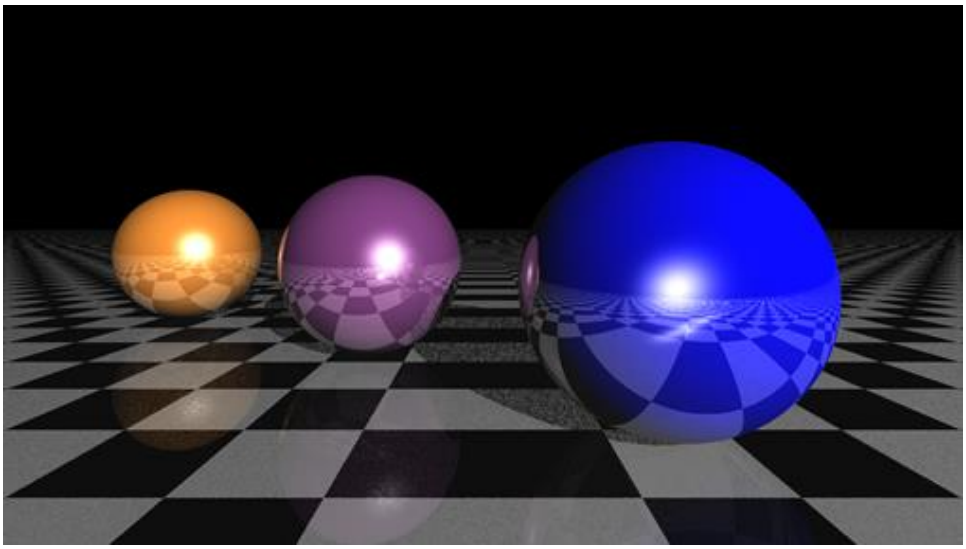
1) GPU has strict data type, when sending a portion of data to GPU the program had to convert it manually every time. The same had to be done while retrieving the output.

2) The calculation's that the GPU has to perform resulted in complex128 format which when converted to float32, lost precision.

Due to the conversion between data type, using GPU has encountered loss factor. Below diagram shows difference between actual output vs output with loss.



Actual output without loss



Output with loss, mostly observable at the shadows

Although GPU could run the squareroot on huge arrays faster than CPU when experimented.

Conclusion-

Raytracing technique is best to create high quality images but due to its heavy computations it is not efficient for real time rendering.

Unlocking global interpreter of python can increase the efficiency of the programs but is not thread safe which means the execution of the programs when run parallelly cannot be managed efficiently. Pythons multiprocessing library has been used to achieve parallel processing. This has significantly increased performance.

GPU are designed to work efficiently for rendering, when GPU has to be used its best to use GPU for the entire processing than partially sharing the load. Working with CUDA requires optimization and dedicated programs.

References -

Volume ray casting -

Volume Ray Casting part-1

Volume Ray casting part-2

Ray Tracing -

Ray Tracing

Implementation Reference

GIT HUB