TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Ray tracing is a technique for rendering images from a three dimensional model of a scene by projecting it onto a two dimensional image plane. It works by calculating the direction of the ray that strikes each pixel of the image plane, and tracing that ray back into the scene to determine the interaction of lights and surfaces that produced that ray. Ray tracing is preferred over the traditional rasterization techniques of rendering images as there is better handling of shadows, reflections and blurs. Ray tracing is known to be computational intensive and costly when implemented. In this project, our main objective will be to reduce the time of the process while retaining the property of creating photorealistic images. With the current advent of parallel computing, it is possible to do so by using methods such as parallel computing, multiprocessor programming and vector intrinsics.

2. LITERATURE REVIEW

In [1], two sequential ray tracing applications i.e. POV-Ray(Persistence of Vision RayTracer) and PBRT(Physically based RayTracer) is used to compare the issues. They have done parallelization, debugging/verification to optimize the ray tracers. All their efforts led to the result that POV-Ray needed more efforts to parallelize and debug than PBRT, but less efforts to optimize. Both the application codes were parallized using pthreads library, in a shared memory programming model. POV-Ray was implemented in C and PBRT in C++. They used disparate tools such as Vtune and Thread Checker for the sole purpose of parallelizing the programs carefully. They did privatization i.e. protecting shared data structures with locks or accessing them with atomic operations. They also checked for data structure replication so that parallelization can be done easily.

They totally converted these two open source applications code to multithreaded code by taking many factors into account such as putting locks in shared data structures and optimizing the code along with parallelizing it. They checked the performance of the new applications using tools that are provided by Intel.

In article [2], it gives insight on how a demand driven parallelization of the ray tracing algorithm is presented.Integration of antialiasing into the load balancing is proposed.Every available processor keeps a permanent subset of the object database as well as a cache for a temporary storage of other objects.Application of some of the caching techniques like LRU policy reduces the number of requests for missing data to required minimum where object bounding boxes are put into effect along with the bounding hierarchy.This parallelization is robust and simply effective.Parallel Ray tracing algorithms can be divided into 2 types Image space subdivision and object space subdivision. The former encounters the problem of unequal workload in processors and rendering of large scenes with extensive memory usage. In the latter there is unequal workload and extensive communication as well between the processors and they’re quite laborious to implement.This paper involves integration of antialiasing as mentioned above with the help of process farming.

It was found that significant parallelization was unlocked while experimenting. There is still more room for perfect load balancing with the help of more tuning of the parameters of the process farming model and this development can be stepping stones for developing better algorithms.

Authors of [3] first highlight the problems with parallelising the basic ray tracing algorithms, such as, an optimization in ray tracing algorithm is to cache and check objects that were intersected by the ray from the neighbouring pixels. Since the rays from neighbouring pixels will have similar trajectories, the caching and checking of objects intersected by one ray first will help reduce time. This optimization can't be easily parallelised because this requires the scene data to be modified globally, which is difficult to do. Another technique regularly used in ray tracing is adaptive non-uniform sampling is a technique in which the image is non-uniformly sampled by initial trace rays. After that more rays are traced in locations where there is a need for high detail. This technique allows for efficient rendering since the areas of the image with high detail are traced heavily while the areas with low detail aren't. This also complicates the parallelisation process since each ray is not independent of every other ray.

Authors proposed an implementation based on OpenMP for parallelising the ray tracer. In this implementation, the authors maintain a global scene data structure that will contain the information about the scenes to render. The overhead of combining the subsection of the scene rendered by each processor is avoided by letting each processor to directly write into the different sections of the final image.

The OpenMP experiments were done on a mraz, 8 - core machine. A 64x64 image was rendered on each of the 8 processors on the machine to measure the average serial time to render the image. It was observed that OpenMP achieves nearly perfect speedup for 6 processors but plateaus at 7 and 8 processors. For the 6 processors, the authors achieved a speedup of 6 times over the serial implementation. This didn't increase by the other 2 cores. This may be due to the residual load on the testing machine.
To explore the effects of different scheduling in OpenMP, five experiments were conducted using five different scheduling parameters, static, static 1, dynamic and guided, the image size

of 128x64 were used to intensify the difference in runtime of different algorithms. For the scheduling experiment, static with chunk size of 1 was the fastest with 26.3865 seconds and guided was slowest with 30.89894 seconds.

In [4] authors suggest a parallel ray tracing on the distributed memory multicomputer and evaluate its performance based on test cases. The authors developed the images from a scattering decomposition of which the allocation unit is one pixel and each processor takes part in pixels scattered around the whole image space, which can make a load balance efficiently. In a scattered decomposition scheme, remote data accesses may be frequent since it can not utilize ray coherence sufficiently. To reduce the communication overhead due to this, the authors propose data prefetching by multicasting (DPM) where requested object data is also transferred to the processors dealing with the adjacent pixels if a processor requests a remote data. The experiment results show that the load imbalance is under 10% in most cases. They also observe that their data distribution scheme requires a relatively small cache memory and the overall performance is enhanced because it can reduce the remote data requests significantly.

The researchers in [5] have implemented an object oriented parallel ray tracer using C++ that can be compiled and run either sequentially or in parallel mode using MPI and OpenMP. Certain load balancing schemes were also implemented which permitted load distribution and favourable load balancing for particular scenes. These measures could be enabled using a compile time parameter. The current implementation only supports spherical objects however new shapes could be added by including a new class. The scene, once generated, has its configurations saved in an XML file. They compared the performance of both OpenMP and MPI integrated with load balancing techniques so that the rendering work is done in parallel wherein each thread renders only the segments allocated to it. The rendered image is collected to master thread of OpenMP using recursive doubling method or the MPI's "MPI_Gather" function and saved in TGA file format.
Experimental results show that the implemented static load balancing technique achieved linear speedup and optimal efficiency in both MPI and OpenMP especially in uneven object distributions. However, there is still a lot of scope for dynamic load balancing.

In [6], the authors have proposed an algorithm to increase the trace time of the ray tracing algorithm by increasing the quality of the BVH (Boundary volume hierarchy) constructed. Boundary Volume Hierarchies are one of the most efficient spatial data structures for organising scene primitives. BVH in ray tracers reduces the number of intersections that are needed to be performed. The higher the quality of the BVH constructed, the less will be the number of intersections needed. BVH can be constructed in O(nlog(n)) time using a full sweep method based on surface area heuristics.

Authors in this paper [6] revisit and extend the build-from-hierarchy method using the scene graph structure to accelerate kD-tree construction to create a BVH instead of kD-tree. They use a triangle soup instead of a scene graph. Starting from a triangle soup, the authors build an auxiliary BVH using a fast BVH builder. This gives a coarse description of the spatial distribution of the primitives. Then a new adaptive method is used to access the auxiliary BVH while the final BVH is constructed. To improve the quality of the constructed BVH for ray tracing, a fast way of integrating spatial splits in the BVH construction process is proposed.

LBVH algorithm is used to create an auxiliary BVH to provide a scalable description of the scene structure. Algorithm will then progressively refine the structure of the auxiliary BVH to get the final BVH. First the initial cuts of the auxiliary BVH are gathered. Then progressively the nodes on this cut are examined and the partition of these nodes in two sets are searched. To do so, a full sweep SAH algorithm is used in all three directions.
The best cut is identified and then nodes are partitioned into two different cuts. These cuts are then refined according to a new threshold corresponding to the reached subdivision depth. The algorithm continues by applying the method recursively in the new subtrees with corresponding refined cuts.

The parallelisation method proposed by the authors is to use a shared work queue which contains entries representing unconstructed parts of the tree. Each entry contains an index of a node in the new BVH and an array representing the corresponding cut in the auxiliary BVH. Threads pop an entry from this queue, find a subdivision of the cut and insert the right cut back into the queue.The left cut is then processed immediately by the thread if it is not a leaf node.

The proposed method was implemented in C++ using STL library and no explicit SIMD constructions were used. A series of tests were performed using different BVH construction algorithms on 9 test scenes. The measurements were performed on a 16 thread PC with 2x Intel Xeon E5-2620. All the BVH were then used by a single ray tracer to construct a 1024 x 1024 image. In terms of the quality of image rendered, PHR - HQ was the best in 2 test cases and close to the best in 5 test cases. The build times were second slowest for PHR algorithms proposed in this paper but the trace time was second fastest. This makes this method suitable for interactive ray tracing.

In [7] the authors state that the Ray Tracers, ray shooting is an operation that generally involves traversing hierarchical acceleration such as kd-tree or bounding volume hierarchy (BVH). The researchers have preferred stackless traversal algorithms because for ray tracing acceleration structures require significantly less storage per ray than ordinary stack based ones. In ray tracing there are many rays in flight so stackless traversal algorithms will save us quite a lot of storage. On SIMD architectures, a commonly used acceleration structure is the MBVH. In this paper they have introduced stackless traversal algorithms for MBVHs with up to four-way branching. Their algorithm has a low computational overhead and supports dynamic ordered traversal.

They have presented a novel and efficient stackless ray traversal algorithm for the MBVH acceleration structure. They introduced two variations of the algorithms, one for four-way trees and one for binary trees. According to them this is the first published stackless method for wide BVHs. On their current architecture their algorithm performs competitively to stack based approaches. The main advantage is that their algorithm has a much smaller traversal state because of which it can significantly enhance the efficiency of advanced, massively parallel in-core or out-of-core ray tracing schemes.

In [8] paper a hybrid scheduling approach is presented that combines demand driven and data parallel techniques. Which tasks to process are demand driven and which data parallel, is decided by the data intensity of the task and the amount of data locality (coherence) that will be

present in the task. A suitable data distribution is to split the object space into equal sized voxels exploiting local object coherence. Each processor is assigned a voxel and its objects. Ray tracing is now performed in a master-slave setup.

In this article [9], the researchers proposed a highly parallel, linearly scalable technique of kd-tree construction for ray tracing. The kd-tree was used with high performance algorithms such as MLTRA. A kd-tree is an axis-aligned BSP tree that splits the scene space using a cost function(in this case SAH) for the split position. The researchers attempted to parallelize the construction of this data structure whose performance scales linearly with the number of threads.To do this, they generated split plane candidates and evaluated the cost of each using SAH. They picked the optimal candidate having the lowest cost and further split it into 2 child nodes. This is repeated recursively. Certain clustering algorithms were also used for quick partitioning of the domain and the clusters obtained were in turn used to build local kd-trees.
The technique proposed allows for ray tracing of complex animated models at a performance,the researchers claim, is far better than any ray tracer for dynamic models known.

This literature [10] explores the possibility of introduction of a parallel architecture. One of the main challenges in the Computer Graphics field is to produce photorealistic images from a three-dimensional model, especially in real time speed. However, such a goal is still far from being real. In most cases, both requirements are mutually exclusive. It is very difficult to achieve real time rendering of photorealistic images, particularly in illumination rendering models, such as Ray Tracing. An algorithm which is based on a global illumination model is capable of rendering very high realistic images, since the interaction of each light source against the whole scene accounts for the production of every single image pixel. In this proposal, put forward is a parallel architecture for Ray Tracing which employs a spatial subdivision technique known for its regularity in order to reduce the number of ray-object intersection tests. The data structure model is subdivided into regions of equal size,where each contains a piece of scene data and only those that are in the direction of a given ray are selected for intersection testing therefore avoiding further unnecessary testing.

Although intersection tests are being performed in parallel. There is still more untapped potential to unleash more parallelism when more than one rays are forced into parallelism processing with different processing entities.

In [11] the authors have presented a stream programming oriented traversal algorithm that processes streams of rays in SIMD fashion, the algorithm that they have presented is motivated by breadth-first ray traversal. Their algorithm re-orders streams of rays on the fly by removing deactivated rays after each traversal step. This will improve SIMD efficiency in the presence of complex scenes and diverging packets. To achieve quite high performance in CPUs requires efficient utilization of SIMD units. They are trying to take the full advantage of the SIMD width offered and to not waste SIMD instructions on low utilization cases.

They successfully presented a new approach to SIMD ray tracing that yields higher SIMD efficiency than traditional packet tracing schemes which ensures that SIMD can be used very efficiently with the CPUs. The key strengths of their algorithm is that it can re-order the rays on the fly so the order in which the rays is passed is completely irrelevant, the re-ordering is done on the geometry hierarchy itself and it also supports arbitrary SIMD widths so that it is applicable for a wide range of future hardware architectures.

In this undertaking [12], the researcher modified a currently available ray tracer implementation in a way that it would work in parallel on a cluster of workstations. The ray tracer used by the author was Physically Based Ray Tracer(PBRT). The Message Passing Interface (MPI) was used to communicate between the processes running on different workstations. The Modification of only a small no.of environment files was required to allow the ray tracer to run in parallel. A sample plugin(stratified) was also used to reduce noise variance in distribution ray tracing. A square number of workstations(1,4,9,16..) were used as the assignment of screen space to the workstations was done by tiling the screen into square blocks. The workstations used were Pentium 3 550 MHz PCs running Linux with 384 MB of RAM. Once all the required plugins were installed three test scenes were rendered each with 1,4,9 and 16 workstations.

It was found that the computationally most expensive part of a ray tracer(“rendering computations”) was reduced by more than 90%. The time saved would open doors towards using even more sophisticated algorithms to improve the quality of image even further.

In [13] two different ray tracing techniques are compared, and their performance is evaluated. Ray tracing in its serial form is computational intensive and hence the authors of this paper have implemented a parallel ray tracer using openMP. The loop-level parallelism implemented with OpenMP made it possible to divide the intensive ray-geometry intersection among all participating threads. Each thread created by the process is allocated a task, they execute their own part of the code, and return with the result.

In this paper [14] a new parallel algorithm has been proposed using a processor farming model.The parallel algorithm was developed based on parallelism characteristics that can be executed in parallel, not related with the number of processors or the image size. Thus, it keeps good scalability for different images. Performance was measured in terms of speed-up for various numbers of processors and data granularity.The ray tracing algorithm traces the path of light of every ray that can be computationally expensive and intensive as well.In this algorithm proposed using process farming model one of the processor is given higher rank compared to the other the one with the zero rank is set as the master whereas all the others are workers.Image info and work distribution info lies with the master.Every worker processor has 2 buffers so as to compute alternatively. Parallelism is achieved here as the worker nodes do not consider the other worker processors in order to synchronize their work.

Even though it implements a process farming model to reduce computational power by many folds it can be further optimized by making sure the worker processor who doesn't have any communication with other worker processors keep asking for more work from the master processor moments becomes idle.

In this paper [15] authors describe the software architecture of Manta interactive ray tracer. Authors propose a two piece programming model to take advantage of the hardware design. First piece is a multithreaded scalable parallel pipeline. Second is a collection of software mechanisms and data structures based on ray packets for extended parallelism and performance optimisation. This model will allow for both scalability on multicore processors and also use hardware designs that enable higher degrees of parallelism such as SIMD. Manta is composed of two groups of modular components, a pipeline and a rendering stack. Components in the pipeline group form a front-end that implements the parallel pipeline model and drives the backend components arranged in a stack.The rendering stack samples pixels, generates rays, computes intersections and shaded colors, and finally produces colors for each pixel. The API of each component defines how data moves through the pipeline and between components in the rendering stack. Ray packets are the primary data structure used to communicate ray tracing data between components in the rendering stack, scene intersection, texturing and shading.Ray packet sizes were limited to the max of 64 bytes to optimally use the L1 cache memory. The data layout is designed to purposely take advantage of SIMD registers and Intel's SSE instructions. Manta's parallel pipeline components are responsible for controlling thread activity and synchronization. The pipeline is used to overlap asynchronous image display with image rendering to reduce the overhead of single-threaded tasks. Each thread asynchronously executes the modular rendering stack during the rendering stage of the pipeline. These components are responsible for dividing the frame up into tiles, determining sample points and finally performing ray tracing. Loosely coupled components are neighboring code modules with no dependencies on the others' implementation. These components increase flexibility in Manta because they increase code reuse by allowing software from one configuration to be placed in another.

Authors in this article provide a software architecture for an interactive ray tracer that is much more flexible than single purpose renderers and can be used for many different scientific applications, such as massive model visualisation in the fields of engineering and anatomy. The architecture is designed with parallelism in mind and has many decisions revolving around this fact, such as the loosely coupled multi-component system for achieving better independence

between threads. Other techniques such as SIMD, optimally using L1 - cache and Intel SSE registers of CPU were also used.

3. PROBLEM FORMULATION

3.1 Brief Description Of Idea

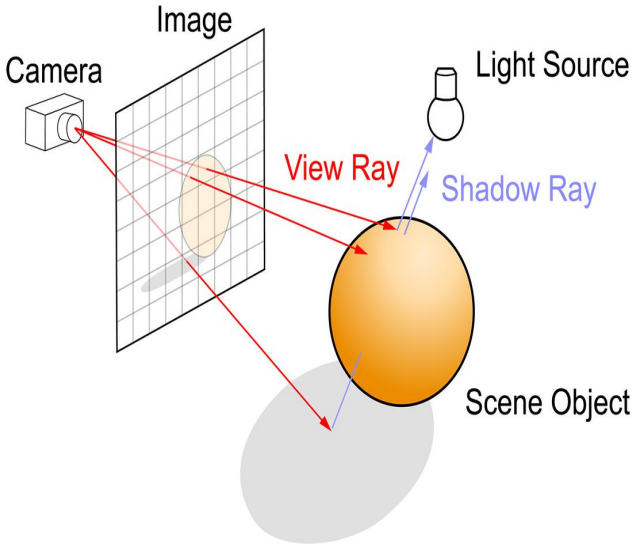

Fig1. Simple Ray Tracer

Ray Tracing is a technique for modeling light transport to use in a variety of rendering algorithms for generating the digital images.

Due to the high computational cost and visual fidelity mainly in path tracing, it becomes slower and there is higher fidelity. The speed of the ray tracer is really significant due to the rendering of each frame. That is the reason why reducing the time is of significance here. We are going to parallelize the whole path tracing process to reduce the time significantly.

3.2 Flowchart

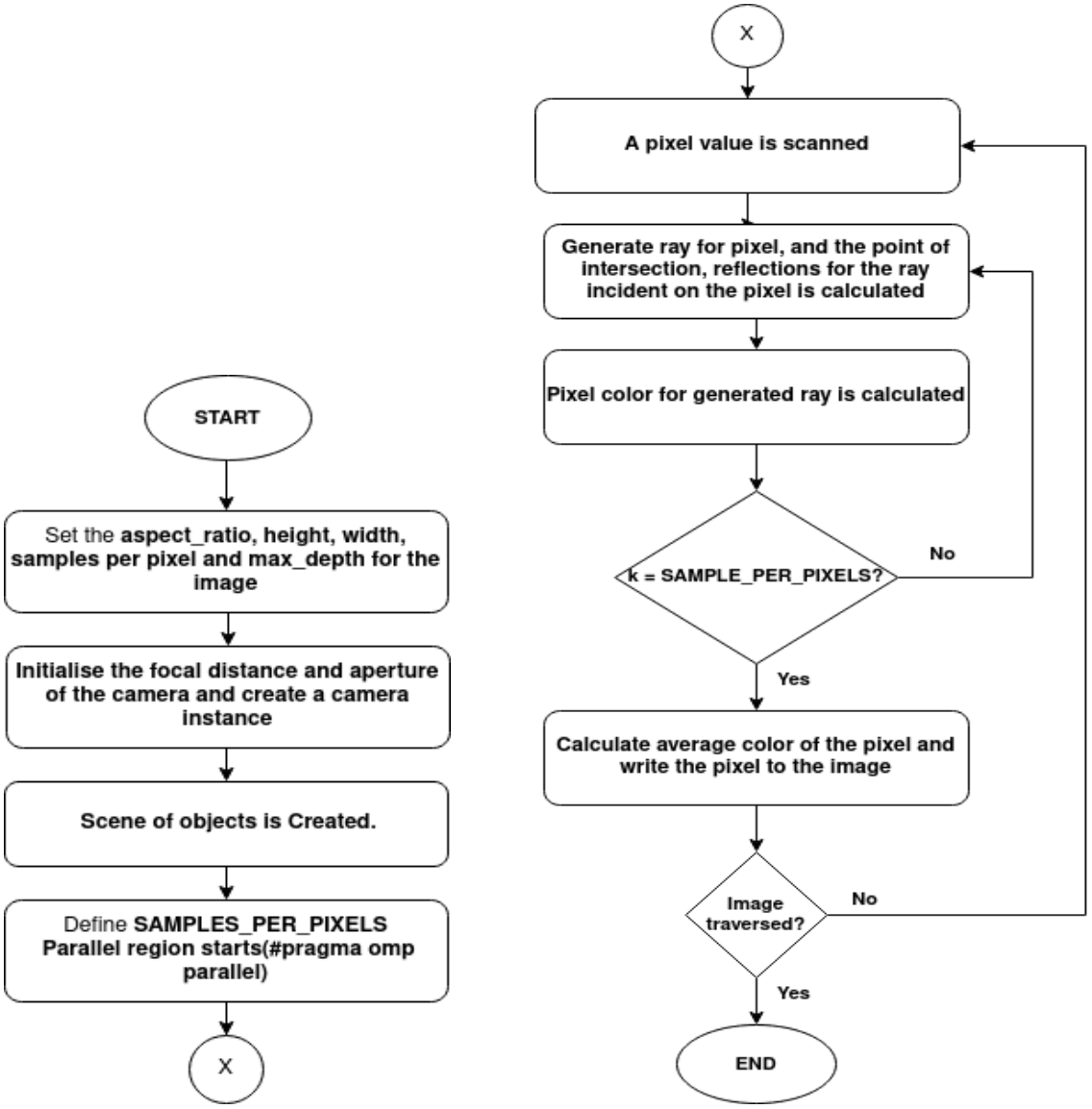

Fig.2 Flow Chart of Ray Tracer Algorithm

3.3 Use Cases

Some of the applications of ray tracing are discussed below:

- **Architectural Rendering:** Ray tracing can be used by architects to achieve realistic renderings accompanying their designs. Backwards ray tracing, combined with radiosity techniques, is suggested to be the most useful method for architectural rendering as even CAD)programs which accurately model objects are unable to model light accurately.

- **Theater and Television Lighting Design:** Stage and television productions require hundreds of individual lights that need to be positioned, aimed, filtered, redirected and dimmed while a production is actually in progress. Ray tracing allows engineers to develop and visualize complex lighting setups in advance and with relative ease.

- **Ray Tracing in Animation:** Ray tracing can be used to effects such as advanced reflection, shadowing, and specularity to animations. Graphical technology is also capable of rendering photorealistic images that would be nearly impossible to produce without computerized ray tracing.

- **Ray Tracing as a Tool for Engineers:** Ray tracing is capable of considering all of the light in a given environment. This concept of global illumination can be used by lighting designers, solar energy researchers, and mechanical engineers to predict illumination levels, luminance gradients, and visual performance criteria while creating any applications.

- **Geophysical modeling and presentation:** Seismic tomography is a major research topic on geophysics and concerns the reconstruction of the Earth's interior. The purpose of seismic modeling is to provide the seismic interpreter with a tool to aid in the interpretation of difficult geological structures. In raypath modeling, ray tracing is carried out for models of multilayered folded structures so as to generate ray diagrams and synthetic time sections. The growing need for fast and accurate prediction of high-frequency wave properties in complex subterranean structures has spawned the requirement of a ray tracer to compute the trajectory of the path corresponding to wave front normals.

4. METHODOLOGY

4.1 Implementation of Ray Tracer

Every Ray Tracer has a ray class and a computation of what color is seen along a ray. A ray can be represented as a function P(t) = A + tb. Here P is a 3D position along a line in 3D, A is ray origin, b is the ray direction and t is the ray parameter. Since C does not have the concept of classes, we used header files(.h files) and an implementation file(.c file).

At the core, the ray tracer sends rays through pixels and computes the color seen in the direction of those rays. The involved steps are:

(1) calculate the ray from the eye to the pixel

(2) determine which objects the ray intersects

(3) compute a color for that intersection point

The recommended aspect ratio for the image is 16:9. In addition to setting up the pixel dimensions for the rendered image, a virtual viewport was also set up through which to pass our scene rays. To remove the aliasing effect(jagged edges in rasterized image), a technique called Antialiasing was used.

In the ray tracer we implemented, one can add multiple objects to a scene each of which may have varied properties(reflective or non reflective). The equation of the sphere in vector form is: (P-C).(P-c) = r^2. if our ray P(t) = A + tb ever hits the sphere anywhere. If it does hit the sphere, there is some t for which P(t) satisfies the sphere equation. So we looked for any t where this is true:

$(P(t)-C) \cdot (P(t)-C)=r^2$. The equation of the sphere is solved with the ray so as to get the "nearest" point of incidence on the scene.

In order to get the shading and reflective effects in the scene, we used the concept of surface normals. In addition, it also helps in identifying the surface where the light is incident(inner or outer). Also, to encompass N number of spheres within the scene we have implemented a list

of "hittables" which also had, type of material as a parameter(Lambertian,Metal, Dielectrics and Glass).

For reflective surfaces we have used the laws of reflections for calculating the direction of reflected rays and similarly Snell's law for dielectric services. A number of factors had to be taken into consideration while calculating the resultant direction of the reflected ray like TIR(Total Internal Reflection).

We have implemented many object types like Diffuse objects that don’t emit light merely take on the color of their surroundings, but they modulate that with their own intrinsic color. Light that reflects off a diffuse surface has its direction randomized. So, we had to use random spheres to predict the direction of the reflected ray and the corresponding child rays which could be controlled using the "depth" attribute.

In order to get different views of the scene, we also implemented a positionable camera. We had to use a thin lens approximation to use defocus blur. The reason we defocus blur in real cameras is because they need a big hole (rather than just a pinhole) to gather light. This would defocus everything, but if we use a lens in the hole, there will be a certain distance where everything is in focus.

4.2 Linear Algebraic Functions Implemented Using SIMD Library

We made separate math linear algebraic functions library for a vector of 3D and 4D to carry out linear algebraic operations necessary for parallel ray tracing in the most efficient way possible

Same common functions were implemented separately for both 3D and 4D vectors. The parallel mechanism used was OpenMP vectorization often referred to as SIMD (Single Instruction Multiple Data) parallelism. SIMD provides data parallelism at the instruction level where a single instruction operates upon multiple data elements. The best part of SIMD is that the instructions are almost as fast as their scalar counterparts which effectively reduces processor time spent by many folds.

This library primarily made use of only the SIMD loop parallel constructs with some clauses like the reduction clause and schedule clause for a variety of reason which shall be explained below:

4.2.1 pragma omp parallel for simd schedule (simd : static)

This directive was used several times when it came to calculation of variables, assigning the values of a vector with other scalar or values of other vectors. Each loop has a certain amount of work associated with it. If the number of threads is increased, the amount of work performed per thread is reduced. Adding SIMD parallelism to this does not necessarily improve the efficiency also especially in the SIMD loops, where each of the threads focusses on reduction in length as and when the thread count increases.

The loop schedule of the work-sharing construct hampers any chance for vectorization. For good performance this must be taken into account. If the static schedule is used with a small chunk size, or if the dynamic, or guided, schedule is used, the SIMD optimal efficiency may not be attained. The reason is that the number of iterations per thread may be too small for SIMD to prove any good to us and moreover the compiler may also fail to generate the most efficient SIMD code due to loop tails and poor memory access patterns.

To address these problems the SIMD schedule modifier was taken into account where it takes care of the problem of the chunk size of a given vector with a definite length. The implementation of a ‘parallel’ construct mixed with ‘for simd’ construct proves to be

gamechanger without an issue or problem arising out of load balancing topic and a point to note is that ‘simd:static’ clause is among those efficient for use.

The above reasons and explanation as well as some literature surveys lead us to implement this directive with schedule clause and a parallel construct especially when iteration is required over a small number. The vectors used are of either 3D or 4D where it involves iteration over three or four of its respective coordinates and this necessity is perfectly met by the above mentioned clauses and work sharing constructs.

4.2.2 pragma omp simd reduction (+ : sum)

This directive was helpful in processing functions related to calculating the sum of the vectors and their values. The reduction clause for OpenMP works similarly in the context of the SIMD construct. The reduction clause takes a list of variables and the reduction operator as its arguments. For each variable in the list, a private instance is used during the execution of the SIMD loop. The partial results are accumulated into the private instance. The reduction operator is applied to combine all partial results, such that the final result is returned in the original variable and made available after the SIMD loop.

This function was instrumental in algebraic calculation of dot products of vectors.

5. RESULTS AND DISCUSSION

5.1 Images Rendered By Our Ray Tracer

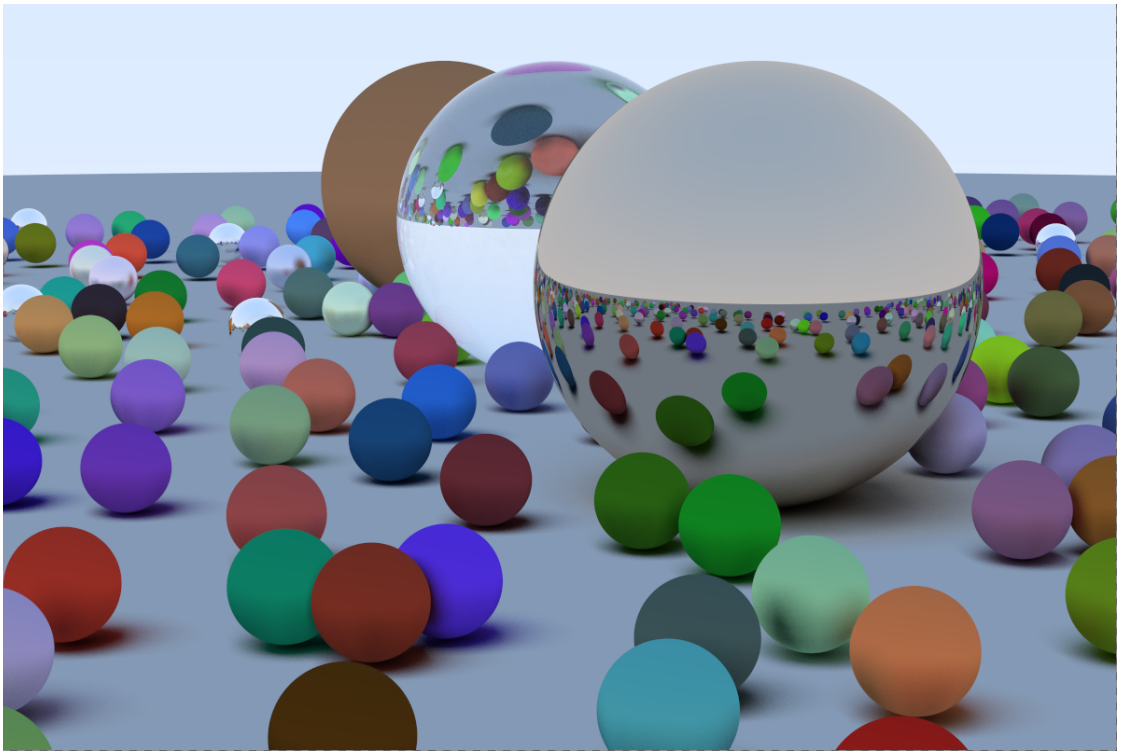

Fig 3: Procedurally generated scene

Fig 4: UV texture mapping

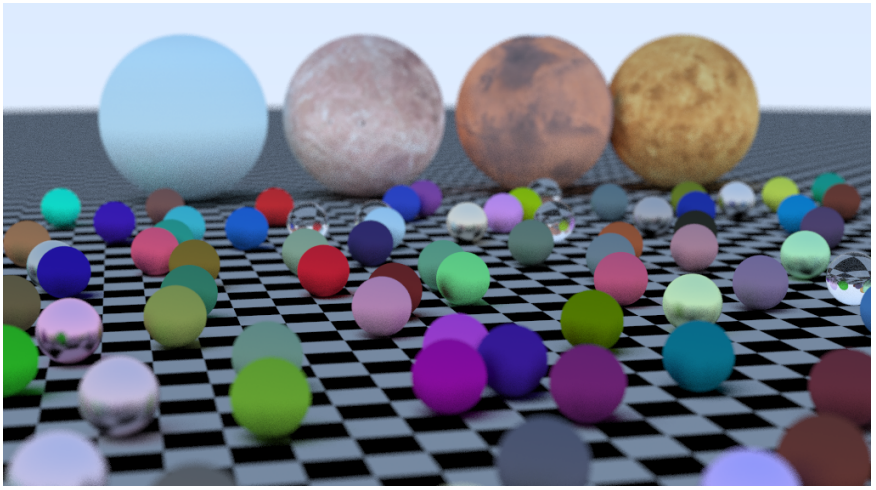

Fig 5: Procedural scene generation with uv texture mapping

5.2 Discussion

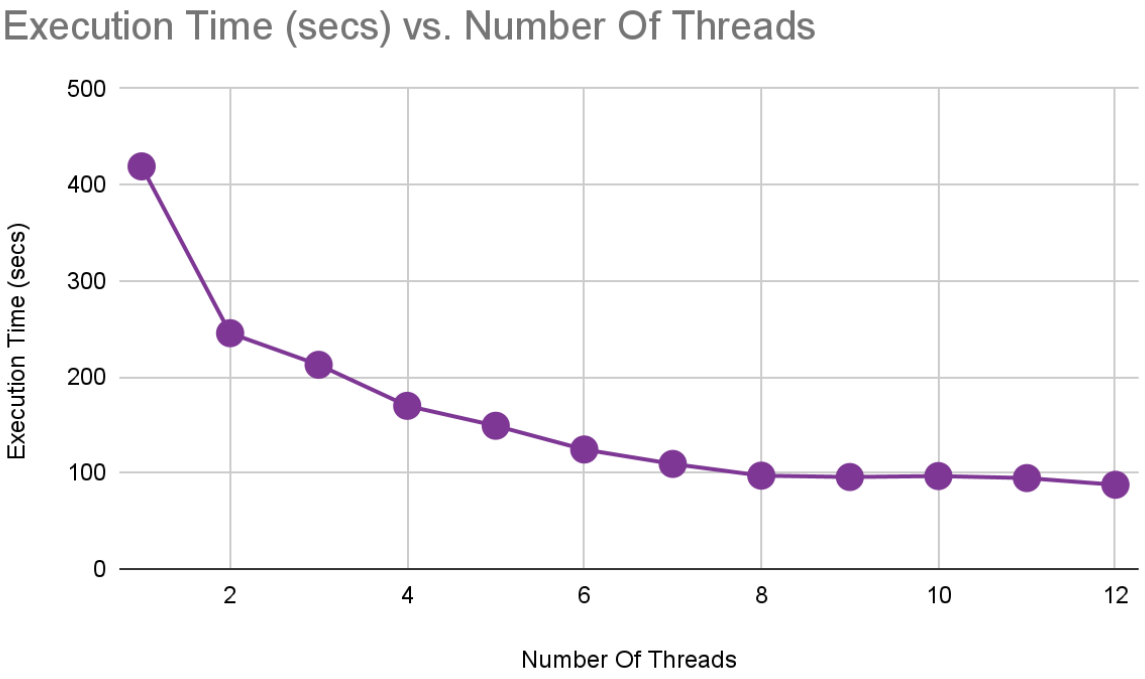

Fig 6: Execution Time vs Number of Threads

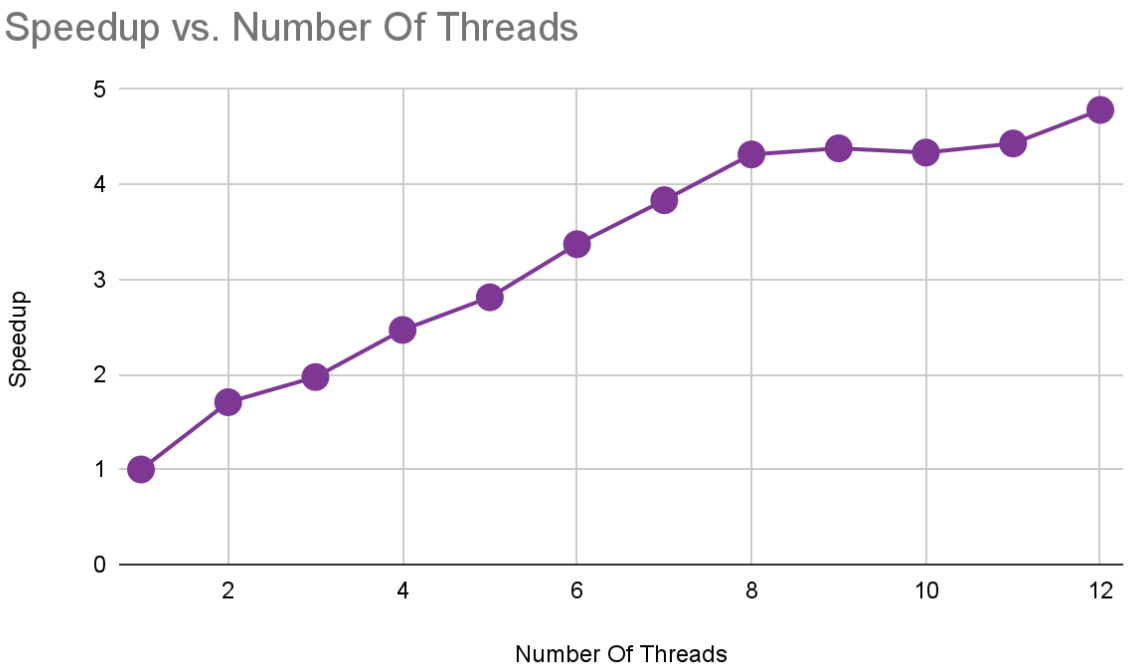

Fig 7: Speedup vs Number of Threads

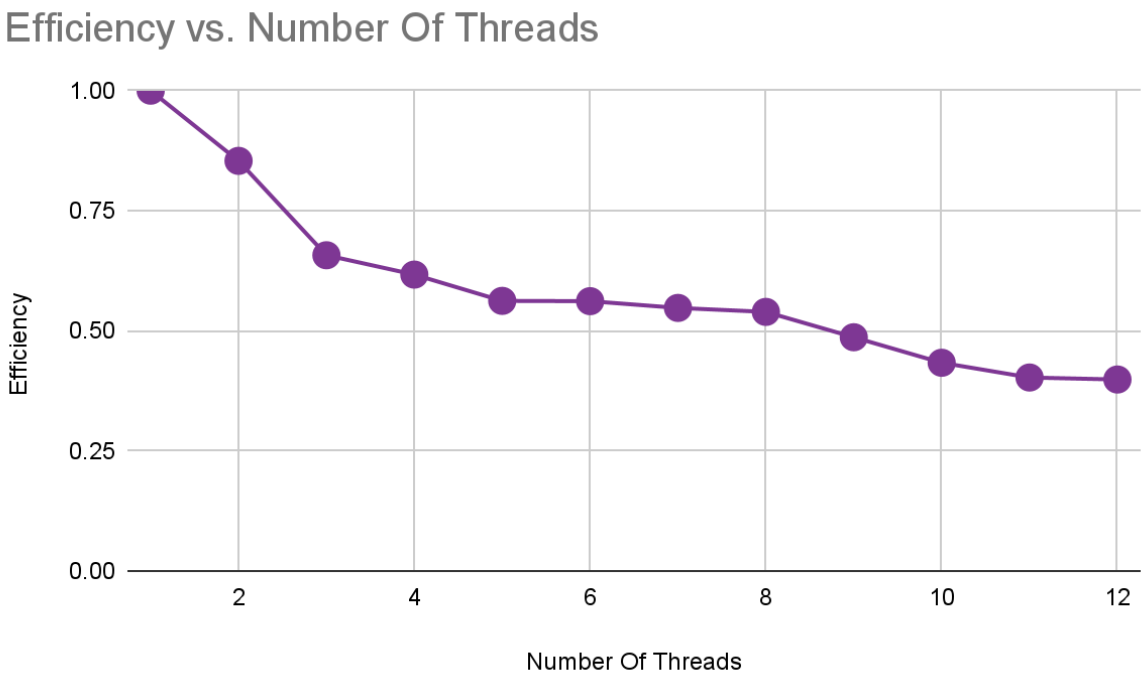

Fig 8: Efficiency vs Number of Threads

Fig 6, 7 and 8 above are produced by rendering the same image on multiple threads. Fig 6 is the graph of execution time vs the number of threads, we can observe that there is a steep decrease in the execution time when going from a single thread to eight threads. After eight threads the decrease is not noticeable.

In the graph of speedup vs number of threads (Fig 7) we can observe a linear speedup going from one thread to 8 threads and after that the speedup increases slowly. The efficiency (Fig 8) is around 0.50 for 3 to 8 threads and it decreases to 0.45 when more threads are added. By observing the trends in the above graphs (Fig 6, 7, and 8), we conclude that the number of threads optimal for our application is around 8.

6. CONCLUSION AND FUTURE WORK

Ray tracer as an image rendering technique has gained a lot of attention in recent years due to its realistic rendering of shadows and textures. Serial implementation of Ray Tracing algorithms can take hours to render one scene. To solve this problem , we have proposed and implemented a ray tracing algorithm that makes use of the vector features of SIMD model and the parallel processing features of OpenMP to reduce the running time of this algorithm. While the serial ray tracing algorithm took -- seconds to execute, the parallel algorithm achieved the same results in -- seconds. While the serial ray tracing algorithm took 801.42 seconds to execute, the parallel algorithm achieved the same results in 245.60 seconds.Hence we can conclude that we were successfully able to parallelize a Ray Tracing algorithm. In the future we can add more features to the ray tracer such as adding a cornell box, we can also add smoke/fog/mist. We will also try to improve the load balancing when the new features are added.

7. REFERENCES

1. Yang, C., Chen, Y., Fu, X., Lim, C.C. and Ju, R., “A comparison of parallelization and performance optimizations for two ray-tracing applications.”, Proceedings of HPC&S (2006), 6, pp.321-330.
2. Plachetka T., “Perfect Load Balancing for Demand- Driven Parallel Ray Tracing.” Lecture Notes in Computer Science (2006), 410–419.
3. Kazeka, Alexander and John Stevens, "Parallel ray tracing." (2007)
4. Yoon, Hyeon-Ju, Seongbae Eun, and Jung Wan Cho. “Image parallel ray tracing using static load balancing and data prefetching.” *Parallel Computing* 23.7 (1997): 861-872.
5. Kadir, S.A. and Khan, T., “Parallel ray tracing using mpi and openmp” Project Report, Introduction to High Performance Computing (2008), Royal Institute of Technology, Stockholm, Sweden
6. Hendrich, J., Meister, D. and Bittner, J., May., “Parallel BVH construction using progressive hierarchical refinement.” In Computer Graphics Forum (2017) (Vol. 36, No. 2, pp. 487-494).
7. Áfra, A.T. and Szirmay‐Kalos, L., “Stackless Multi‐BVH Traversal for CPU, MIC and GPU Ray Tracing”. *Computer Graphics Forum* February 2014 (Vol. 33, No. 1, pp. 129-140).
8. Reinhard, E. and Jansen, F.W., “Rendering large scenes using parallel ray tracing. Parallel Computing” (1997), 23(7), pp.873-885.
9. Shevtsov, M., Soupikov, A. and Kapustin, A., “Highly parallel fast KD‐tree construction for interactive ray tracing of dynamic scenes.”, Computer Graphics Forum (Vol. 26, No. 3, pp. 395-404). February 2014, Oxford, UK: Blackwell Publishing Ltd.
10. Nery, A.S., Nedjah, N. and França, F.M., “A parallel architecture for ray-tracing.” First IEEE Latin American Symposium on Circuits and Systems (LASCAS). February 2010 (pp. 77-80). IEEE.
11. Wald, I., Gribble, C.P., Boulos, S. and Kensler, A., “Simd ray stream tracing-simd ray traversal with generalized ray packets and on-the-fly re-ordering.”, 2007, Informe Técnico, SCI Institute, 2.
12. Scott, M.R., A Parallel Ray Tracer. (2005)
13. Adewale, Ayobami Ephraim. "Performance Evaluation of Monte Carlo Based Ray Tracer." *Journal of Computational Science* 12.1 (2021).
14. Lee, H.J. and Lim, B.H., “Parallel ray tracing using processor farming model.” *Proceedings International Conference on Parallel Processing Workshops* (pp. 59-63). IEEE.
15. Bigler, J., Stephens, A. and Parker, S.G., 2006, September. Design for parallel interactive ray tracing systems. In 2006 IEEE Symposium on Interactive Ray Tracing (pp. 187-196). IEEE.

APPENDIX A.

Project repository link : https://github.com/kaustubh0201/RayTracer

A.1 SIMD Library Using using OpenMP Vectorization (SIMD):

```
#include <stdio.h>
#include<omp.h>

static HYP_INLINE HYP_FLOAT HYP_ABS(HYP_FLOAT value)
{
	return (value < 0.0f) ? -value : value;
}


short scalar_equals_epsilonf(const HYP_FLOAT f1, const HYP_FLOAT f2, const HYP_FLOAT epsilon)
{
	if ((HYP_ABS(f1 - f2) < epsilon) == 0) {
		return 0;
	}

	return 1;
}

short scalar_equalsf(const HYP_FLOAT f1, const HYP_FLOAT f2)
{
	return scalar_equals_epsilonf(f1, f2, HYP_EPSILON);
}

#define scalar_equals scalar_equalsf


struct vector3 {
	union {
		HYP_FLOAT v[3];
		struct {
			HYP_FLOAT x, y, z;
		};
		struct {
			HYP_FLOAT yaw, pitch, roll;
		};
	};
};

struct vector4 {
	union {
		HYP_FLOAT v[4];
		struct {
			HYP_FLOAT x, y, z, w;
		};
	};
};
struct vector3 *vector3_zero(struct vector3 *self){
```

```
    return vector3_setf3(self, 0.0f, 0.0f, 0.0f);
}

struct vector3 *vector3_set(struct vector3 *self, const struct vector3 *vT)
{
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
        (self->v[i])=(vT->v[i]);
   }
    return self;
}

struct vector3 *vector3_setf3(struct vector3 *self, HYP_FLOAT xT, HYP_FLOAT yT, HYP_FLOAT zT){
    self->x = xT;
    self->y = yT;
    self->z = zT;
    return self;
}

struct vector3 *vector3_negate(struct vector3 *self){
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
        (self->v[i])=-(self->v[i]);
   }
    return self;
}

struct vector3 *vector3_add(struct vector3 *self, const struct vector3 *vT)
{
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
        (self->v[i]) += (vT->v[i]);
   }
    return self;
}

struct vector3 *vector3_addf(struct vector3 *self, HYP_FLOAT fT){
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
            (self->v[i]) += fT;
    }
    return self;
}

struct vector3 *vector3_subtract(struct vector3 *self, const struct vector3 *vT)
{
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
            (self->v[i]) -= (vT->v[i]);
    }
    return self;
}

struct vector3 *vector3_subtractf(struct vector3 *self, HYP_FLOAT fT)
{
```

```
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
		(self->v[i])+=fT;
	}
	return self;
}

struct vector3 *vector3_multiply(struct vector3 *self, const struct vector3 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
		(self->v[i])*=(vT->v[i]);
	}
	return self;
}

struct vector3 *vector3_multiplyf(struct vector3 *self, HYP_FLOAT fT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
		(self->v[i])*=fT;
	}
	return self;
}

struct vector3 *vector3_divide(struct vector3 *self, const struct vector3 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
		(self->v[i])/=(vT->v[i]);
	}
	return self;
}

struct vector3 *vector3_dividef(struct vector3 *self, HYP_FLOAT fT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
		(self->v[i])/=fT;
	}
	return self;
}

struct vector3 *vector3_normalize(struct vector3 *self)
{
	HYP_FLOAT mag;

	mag = vector3_magnitude(self);

	if (scalar_equalsf(mag, 0.0f)) {
		return self;
	}

	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<3 ; i++){
```

```
                (self->v[i])/=mag;
        }
        return self;
}

HYP_FLOAT vector3_magnitude(const struct vector3 *self)
{
        double sum=0.0;
        #pragma omp simd reduction(+:sum)
        for (int i=0 ; i<3 ; ++i){
            sum+=( (self->v[i])*(self->v[i]) );
    }
        return HYP_SQRT(sum);
}

HYP_FLOAT vector3_distance(const struct vector3 *v1, const struct vector3 *v2)
{
        double sum=0.0;
        #pragma omp simd reduction(+:sum)
        for (int i=0 ; i<3 ; ++i){
            sum+=( ( (v2->v[i])-(v1->v[i]) )*( (v2->v[i])-(v1->v[i]) ) );
    }
        return HYP_SQRT(sum);
}

HYP_FLOAT vector3_dot_product(const struct vector3 *self, const struct vector3 *vT)
{
        double sum=0.0;
        #pragma omp simd reduction(+:sum)
        for (int i=0 ; i<3 ; ++i){
            sum+=((self->v[i])*(vT->v[i]));
    }
    return sum;
}

struct vector3 *vector3_cross_product(struct vector3 *vR, const struct vector3 *vT1, const struct vector3 *vT2)
{
        #pragma omp parallel for simd schedule(simd:static)
        for(int i=0 ; i<3 ; i++){
                vR->v[i]=((vT1->v[(i+1)%3] * vT2->v[(i+2)%3]) - (vT1->v[(i+2)%3] * vT2->v[(i+1)%3]));
        }
        return vR;
}

HYP_FLOAT vector3_angle_between(const struct vector3 *vT1, const struct vector3 *vT2)
{
        HYP_FLOAT c;
        c = vector3_dot_product(vT1, vT2) / (vector3_magnitude(vT1) * vector3_magnitude(vT2));
        return 2.0f * HYP_ACOS(c);
}

struct vector3 *vector3_find_normal_axis_between(struct vector3 *vR, const struct vector3 *vT1, const struct
vector3 *vT2)
{
        vector3_cross_product(vR, vT1, vT2);
```

```
	vector3_normalize(vR);
	return vR;
}

#define vector3_length(v) vector3_magnitude(v)

int vector4_equals(const struct vector4 *self, const struct vector4 *vT)  //UNABLE TO APPLY SIMD
{
	return  HYP_ABS(self->x - vT->x) < HYP_EPSILON &&
		HYP_ABS(self->y - vT->y) < HYP_EPSILON &&
		HYP_ABS(self->z - vT->z) < HYP_EPSILON &&
		HYP_ABS(self->w - vT->w) < HYP_EPSILON;
}

struct vector4 *vector4_setf4(struct vector4 *self, HYP_FLOAT xT, HYP_FLOAT yT, HYP_FLOAT zT,
HYP_FLOAT wT)
{
	self->x = xT;
	self->y = yT;
	self->z = zT;
	self->w = wT;
	return self;
}

struct vector4 *vector4_zero(struct vector4 *self)
{
	return vector4_setf4(self, 0.0f, 0.0f, 0.0f, 0.0f);
}

struct vector4 *vector4_negate(struct vector4 *self)
{
	#pragma omp parallel for simd schedule(simd:static)
	for (int i = 0; i <4 ; i++){
		self->v[i] = -self->v[i];
	}
	return self;
}

struct vector4 *vector4_set(struct vector4 *self, const struct vector4 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<4 ; i++){
	    (self->v[i])=(vT->v[i]);
   }
	return self;
}

struct vector4 *vector4_add(struct vector4 *self, const struct vector4 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i=0 ; i<4 ; i++){
	    (self->v[i])+=(vT->v[i]);
   }
	return self;
}
```

```
struct vector4 *vector4_addf(struct vector4 *self, HYP_FLOAT f)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		self->v[i]+=f;
	}

	return self;
}

struct vector4 *vector4_subtract(struct vector4 *self, const struct vector4 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		self->v[i]-=vT->v[i];
	}
	return self;
}

struct vector4 *vector4_subtractf(struct vector4 *self, HYP_FLOAT f)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		self->v[i]-=f;
	}
	return self;
}

struct vector4 *vector4_multiply(struct vector4 *self, const struct vector4 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		self->v[i]*=vT->v[i];
	}
	return self;
}

struct vector4 *vector4_multiplyf(struct vector4 *self, HYP_FLOAT f)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		self->v[i]*=f;
	}
	return self;
}

struct vector4 *vector4_divide(struct vector4 *self, const struct vector4 *vT)
{
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		(self->v[i])/=(vT->v[i]);
	}
	return self;
}
```

```
struct vector4 *vector4_dividef(struct vector4 *self, HYP_FLOAT fT){
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		(self->v[i])/=fT;
	}
	return self;
}

HYP_FLOAT vector4_magnitude(const struct vector4 *self)
{
	double sum=0.0;
	#pragma omp simd reduction(+:sum)
	for (int i=0 ; i<4 ; ++i){
	    sum+=( ( (self->v[i])*(self->v[i]) ) );
   }
	return HYP_SQRT(sum);
}

struct vector4 *vector4_normalize(struct vector4 *self)
{
	HYP_FLOAT mag;
	mag = vector4_magnitude(self);
	if (scalar_equalsf(mag, 0.0)) {
		return self;
	}
	#pragma omp parallel for simd schedule(simd:static)
	for(int i = 0; i<4 ; i++){
		(self->v[i])/=mag;
	}
	return self;
}


HYPAPI HYP_FLOAT vector4_distance(const struct vector4 *v1, const struct vector4 *v2)
{
	double sum=0.0;
	#pragma omp simd reduction(+:sum)
	for (int i=0 ; i<4 ; ++i){
	    sum+=(((v2->v[i])-(v1->v[i]))*((v2->v[i])-(v1->v[i])));
   }
	return HYP_SQRT(sum);
}

HYP_FLOAT vector4_dot_product(const struct vector4 *self, const struct vector4 *vT)
{
	double sum=0.0;
	#pragma omp simd reduction(+:sum)
	for (int i=0 ; i<4 ; ++i){
	    sum+=((self->v[i])*(vT->v[i]));
   }
   return sum;
}
```

```
struct vector4 *vector4_cross_product(struct vector4 *vR, const struct vector4 *vT1, const struct vector4 *vT2)
{
    #pragma omp parallel for simd schedule(simd:static)
    for(int i=0 ; i<3 ; i++){
            vR->v[i]=((vT1->v[(i+1)%3] * vT2->v[(i+2)%3]) - (vT1->v[(i+2)%3] * vT2->v[(i+1)%3]));
    }
    vR->v[3] = (vT1->v[3] * vT2->v[3]) - (vT1->v[3] * vT2->v[3]);
    return vR;
}
```

A.2 Procedural Scene Generation Code

```
void randomSpheres2(ObjectLL * world, DynamicStackAlloc * dsa, int n, Image imgs[n], int * seed){

  LambertianMat* materialGround = alloc_dynamicStackAllocAllocate(dsa,
                    sizeof(LambertianMat),
                    alignof(LambertianMat));
  SolidColor * sc1 = alloc_dynamicStackAllocAllocate(dsa,
            sizeof(SolidColor), alignof(SolidColor));

  SolidColor * sc = alloc_dynamicStackAllocAllocate(dsa,
            sizeof(SolidColor), alignof(SolidColor));

  Checker * c = alloc_dynamicStackAllocAllocate(dsa, sizeof(Checker), alignof(Checker));

  sc1->color = (RGBColorF) {.r = 0.0, .b = 0.0, .g = 0.0};
  sc->color = (RGBColorF){.r = 0.4, .b = 0.4, .g = 0.4};

  c->even.tex = sc1;
  c->even.texType = SOLID_COLOR;
  c->odd.tex = sc;
  c->odd.texType = SOLID_COLOR;

  materialGround->lambTexture.tex = c;
  materialGround->lambTexture.texType = CHECKER;

  /*materialGround->albedo.r = 0.5;
  materialGround->albedo.g = 0.5;
  materialGround->albedo.b = 0.5;*/
```

```
obj_objLLAddSphere(world, (Sphere){
   .center = {.x = 0, .y = -1000, .z = 0}, .radius = 1000, .sphMat = MAT_CREATE_LAMB_IP(materialGround)
});

for (int a = -2; a < 9; a++){
   for (int b = -9; b < 9; b++){
      CFLOAT chooseMat = lcg(seed);
      vec3 center = {
         .x = a + 0.9 * lcg(seed),
         .y = 0.2,
         .z = b + 0.9 * lcg(seed)
      };

      if(chooseMat < 0.8){
         // diffuse
         RGBColorF albedo = {
               .r = lcg(seed) * lcg(seed),
               .g = lcg(seed) * lcg(seed),
               .b = lcg(seed) * lcg(seed),

         };

         LambertianMat* lambMat = alloc_dynamicStackAllocAllocate(dsa,
                        sizeof(LambertianMat),
                        alignof(LambertianMat));

         SolidColor * sc = alloc_dynamicStackAllocAllocate(dsa,
                  sizeof(SolidColor), alignof(SolidColor));

         sc->color = albedo;

         lambMat->lambTexture.tex = sc;
         lambMat->lambTexture.texType = SOLID_COLOR;

         obj_objLLAddSphere(world, (Sphere) {
            .center = center,
            .radius = 0.2,
```

```
        .sphMat = MAT_CREATE_LAMB_IP(lambMat)
    });

}else if(chooseMat < 0.95){
    // metal
    RGBColorF albedo = {
        .r = lcg(seed)/2 + 0.5,
        .g = lcg(seed)/2 + 0.5,
        .b = lcg(seed)/2 + 0.5
    };
    CFLOAT fuzz = lcg(seed)/2 + 0.5;

    MetalMat* metalMat = alloc_dynamicStackAllocAllocate(dsa,
                        sizeof(MetalMat),
                        alignof(MetalMat));

    metalMat->albedo = albedo;
    metalMat->fuzz = fuzz;

    obj_objLLAddSphere(world, (Sphere) {
        .center = center,
        .radius = 0.2,
        .sphMat = MAT_CREATE_METAL_IP(metalMat)
    });
}else{
    DielectricMat * dMat = alloc_dynamicStackAllocAllocate(dsa,
                         sizeof(DielectricMat),
                         alignof(DielectricMat));
    dMat->ir = 1.5;
    obj_objLLAddSphere(world, (Sphere) {
        .center = center,
        .radius = 0.2,
        .sphMat = MAT_CREATE_DIELECTRIC_IP(dMat)
    });

}
```

```
    }

}

LambertianMat* material2 = alloc_dynamicStackAllocAllocate(dsa,
                                sizeof(LambertianMat),
                                alignof(LambertianMat));



material2->lambTexture.tex = &imgs[0];
material2->lambTexture.texType = IMAGE;
/*material2->albedo.r = 0.4;
material2->albedo.g = 0.2;
material2->albedo.b = 0.1;
*/

obj_objLLAddSphere(world, (Sphere){
    .center = {.x = -4, .y = 1, .z = 0},
    .radius = 1.0,
    .sphMat = MAT_CREATE_LAMB_IP(material2)
});


material2 = alloc_dynamicStackAllocAllocate(dsa,
                                sizeof(LambertianMat),
                                alignof(LambertianMat));



material2->lambTexture.tex = &imgs[1];
material2->lambTexture.texType = IMAGE;
/*material2->albedo.r = 0.4;
material2->albedo.g = 0.2;
material2->albedo.b = 0.1;
*/
```

```
 obj_objLLAddSphere(world, (Sphere){
    .center = {.x = -4, .y = 1, .z = -2.2},
    .radius = 1.0,
    .sphMat = MAT_CREATE_LAMB_IP(material2)
 });

material2 = alloc_dynamicStackAllocAllocate(dsa,
                                sizeof(LambertianMat),
                                alignof(LambertianMat));



 material2->lambTexture.tex = &imgs[2];
 material2->lambTexture.texType = IMAGE;
 /*material2->albedo.r = 0.4;
 material2->albedo.g = 0.2;
 material2->albedo.b = 0.1;
 */

 obj_objLLAddSphere(world, (Sphere){
    .center = {.x = -4, .y = 1, .z = +2.2},
    .radius = 1.0,
    .sphMat = MAT_CREATE_LAMB_IP(material2)
 });

material2 = alloc_dynamicStackAllocAllocate(dsa,
                                sizeof(LambertianMat),
                                alignof(LambertianMat));



 material2->lambTexture.tex = &imgs[3];
 material2->lambTexture.texType = IMAGE;
 /*material2->albedo.r = 0.4;
 material2->albedo.g = 0.2;
 material2->albedo.b = 0.1;
 */
```

```
    obj_objLLAddSphere(world, (Sphere){
        .center = {.x = -4, .y = 1, .z = -4.2},
        .radius = 1.0,
        .sphMat = MAT_CREATE_LAMB_IP(material2)
    });
}
```