Improving an Image Rendering Program with Multiprocessing in Python

Zach Dunn

December 17, 2023

Abstract

Developing an efficient solution to the problem of rendering images based on a 3D scene is a complicated task that many are trying to achieve. Due to the amount of computation involved in rendering an image it has the potential to take a long time especially as the demands for higher quality images and faster response times increase. Addressing this issue will positively impact the graphics community by providing a way to speed up the rendering pipeline resulting in faster image generation. This project aims to utilize parallel processing embedded in the rendering pipeline to increase the speed at which images can render by enabling the rendering pipeline to calculate the data for multiple objects in a scene at once.

Background and Introduction

The current technological landscape for image rendering is vast with many different approaches for generating an image. One of these approaches involves representing a scene through a series of 3D objects. Additional information provided alongside these objects includes transformations to define orientation, a camera, and a light source. In this method, the rendering pipeline must take this information and convert it to a 2D image with the proper orientation of all the objects and accurate shading. As a result, the process to produce a final image is quite heavy in computation. Since the scenes to render can become complex in nature, with multiple objects, light sources, and shading techniques, it is important to optimize the computations so the image can be produced as quickly as possible. To address this, I propose that each set of computations be done in parallel to increase efficiency. In practice, this would mean that in each step of the rendering pipeline the calculations would be split into groups and processed in parallel. The groups will be determined by what object (mesh) the vertices to transform are a part of. After that additional threads will be created to process the vertices in each of the following transformation steps. The steps that would be parallelized include the geometry processing and rasterization. Geometry processing is where the object transformations take place. The computations here transform the coordinates of vertices to different space representations, i.e. object space, world space, camera space, screen space, and pixel space. During rasterization the objects that are now in pixel space need to be projected to the screen. Here the computations include finding where a point is on the 2D screen, then determining its depth value. The expected final result of this implementation is that the time to render an image will be significantly reduced.

If this does not turn out to be the case, as will be shown later, another method of rendering images will be used to test if parallel processing can be successfully implemented for image rendering. The other method that will be tested is ray-traced rendering. The majority of the rendering algorithm will stay the same as outlined above, however the geometry processing and rasterization will look a little different. In raytracing instead of iterating through each mesh and each face therein, the render loop will iterate through each pixel of the screen and project a ray. The ray will then be evaluated to see if it collides with an object that is in the scene. If it does, a shadow ray is generated and used to see if the surface the first ray collided with is in shadow or not. At this point if the surface is not in shadow a color is calculated based on a pre-determined shading technique and the pixel is colored the resulting color. This type of rendering differs from the first method because the vertices for an object only ever need to be converted to world space so there is less geometry processing. Additionally, for rasterization there is no need to convert a point from pixel space back to screen space because we already know the depth value thanks to the ray that was calculated earlier. However, this does not mean it is more efficient. During the evaluation to determine if a ray has collided with something, every face of every object must be tested for a collision. This takes much longer than the previous rendering method, so it is a good choice for parallelization. To parallelize the raytracing rendering loop I propose the set of pixels to be looped over be split up into groups so that different chunks of the screen can be rendered at the same time.

Ideally, the total computation time would be cut in half or more. Due to the fact that there is notable overhead for creating new python processes, if a simple scene was given to the renderer there will be no significant increase in speed. However, as the scenes become more complex the execution time of the parallel solution should stay relatively small compared to the sequential one. This is significant because by developing more efficient methods for generating images it allows for more detailed imagery in media while not sacrificing time or computational resources.

Literature Review

Parallel Rendering Mechanism for Graphics Programming on Multicore Processors

In this study the researchers sought to understand how implementing the rendering pipeline on a multicore processor could increase computation speed. The method they used to study this involved providing a system that allowed OpenGL programs to be run on multicore processors. In addition to this the researchers utilized "an OpenMP API for the thread scheduling of parallel task[s]" (Chickerur, Satyadhyan, et al.) and Intel VTune Performance Analyzer to see how parallelizing the rendering of graphics over multiple cores affected the performance. The main graphics program was written in C/C++ which amounted to an animation of a rotating object. To assess whether parallelism improved performance the achieved frame rate during execution was used to measure performance gains. The main conclusion the researchers came to was that graphical performance can be increased using a multicore processor to execute the rendering pipeline, however the question of how much parallelism to use arose. The researchers were able to show a decrease in performance by using an excessive amount of parallelism but were unable to determine an optimal level for all cases.

Evaluation

The results shown above are promising for improving the speed of the rendering pipeline. The researchers' results demonstrated how graphics computation could be sped up by parallelizing the whole process. In the solution I have outlined above the parallelization would be a little more subtle. Instead of high-level parallelization where everything in the pipeline is done on separate threads my solution will only split up the individual calculations that take up the most execution time. Additionally, to address the issue of how many threads spawn during computation my solution will start by spawning threads based on how many objects are in a certain scene. For the raytracing solution multiple tests will be performed to find the optimal number of threads.

A Sorting Classification of Parallel Rendering

This paper covers fully parallel rendering algorithms in which both the geometry processing and rasterization stages are done in parallel. The goal of the analysis is to classify the different types of fully parallel rendering algorithms and to identify what applications best suit each one. The authors identify three main types of algorithms: sort-first, sort-middle, and sort-last which differ mainly in when the sorting of the data happens, i.e. how the data is distributed among the parallel processes and transformed to the screen. Each approach offers its own unique set of benefits and disadvantages. Sort-first involves taking the render data and doing enough transformations to determine what region of the screen the data lies in, from there the data is redistributed to the appropriate parallel renderer to finish computation. The authors note that at first this seems impractical, however, it produces “less communication bandwidth than the other approaches, particularly if primitives are tessellated or if frame-to-frame coherence can be exploited” (Molnar, Steven, et al.). Next, sort-middle does what is considered sensible, sorting the data after geometry processing and before rasterization taking advantage of the natural break in computation. The data can be sorted and sent to the appropriate renderer with little complexity. Lastly, sort-last waits until the data has been converted to pixels before sorting them into the correct region for display and determining if they are visible. The authors note that there are two main ways of doing this, SL-full and SL-sparse, each with their own advantages. Overall, sort-last is said to be the most used of the methods outlined above.

Evaluation

My proposed solution falls outside any of the classifications outlined in the paper above. Instead of taking the render data as a whole and splitting it up at a certain point in the rendering pipeline then merging it back later, my solution will take the data as it comes and spawn a new thread for each mesh that is sent to the renderer. After this, inside each thread for the meshes additional threads can be created as needed to optimize the transformation of the vertices for a specific object. The ray-traced solution also does not fit into the classification defined above because it does not have specific geometry processing and rasterization stages.

Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU

In this proposal the authors present a solution to efficient rendering of vector graphics in parallel using a novel representation for the primitives. The authors defined their new representation as a CPatch which is “similar to the use of edge equations in polygon rasterization” (Dokter, Mark, et al.). To render an image using the author’s new approach they used a hierarchical rasterization method that tiled the screen into segments, focusing on tiles that landed near edges of the geometry. At each level of the hierarchy “the hierarchical rasterization removes irrelevant curves” (Dokter, Mark, et al.) and processes the rest in parallel. Once the last level of the hierarchy has been reached, they then evaluate the remaining curves and color the pixels. After implementing this approach, the authors analyzed its performance compared to previous methods and found that it was comparable to the current state of the art and even achieved speedups and avoided making any approximations.

Evaluation

In the paper outlined above the researchers proposed a new representation for the input data that defines a scene. They used a hierarchical approach to rasterization which, combined with their new data structure, led to comparable performance to what other methods have shown. In my proposed solution I also use a hierarchical system, however, it is hierarchical in its parallelization, not rasterization. The goal is to achieve significant speedup without needing to define a new complex datatype that could potentially be a new source of overhead causing a loss in performance.

Solution Design and Implementation

I have implemented two different methods of rendering, both of which are set up the same way. The rendering pipeline is separated into multiple modules, each designed to complete a single task. Tasks include creating a mesh object based on a provided STL file, taking the final image and showing it to the screen, transforming a vertex to a different space, defining a camera and its functions, defining a light source, and running the rendering loop to process the data.

The differences in the two methods of rendering become apparent when looking at the renderer module. In here is where the main render loop takes place. For the regular method of rendering, the render loop iterates through each mesh and each face within. Conversely, in the ray-traced renderer module, each pixel is iterated over and within each loop all of the faces of the objects are iterated over as well.

Additionally, within the renderer modules is where the parallelization takes place. For the regular renderer, a new process is spawned for each mesh that is looped over. In conjunction with this in each process for a mesh a pool of threads was created to parallelize the work done for each face. A similar method was used in the ray-traced renderer, where the first loop for iterating over the pixels was broken up into groups and run on multiple processes. To determine how big the groups should be and by extension how many processes should be made, tests were conducted to find the optimal size. Initially, within each process for a group of pixels a thread pool was created to help parallelize the consumption of all the meshes and their faces, however, preliminary tests showed that this was slower than only having parallelization for the pixels, so it was removed.

To implement this, I used python and a few key python modules including Numpy, Multiprocessing, Time, and Concurrent Futures. Numpy was used for all of the calculations involved with transforming objects and calculating the lighting. The multiprocessing library was used to create the python processes as well as manage any shared variables that were needed by all of the processes. Concurrent Futures was used to create a pool of threads inside each of the processes created by the multiprocessing library. Finally, the Time library was used to record how long the execution took. The code follows an object orientated approach where each task outlined above is encompassed in its own class.

Results

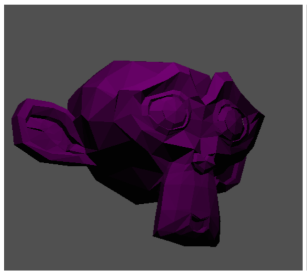
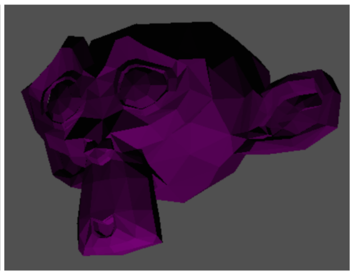
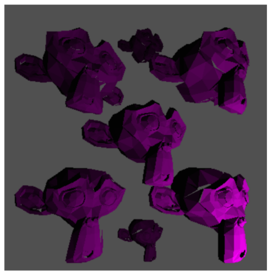
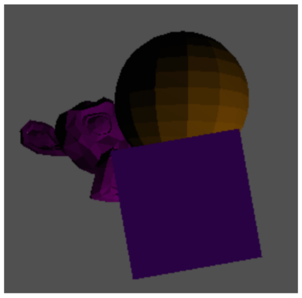
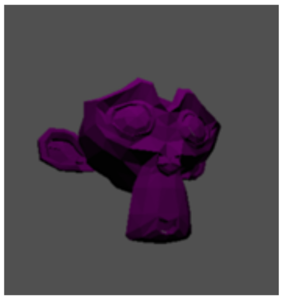
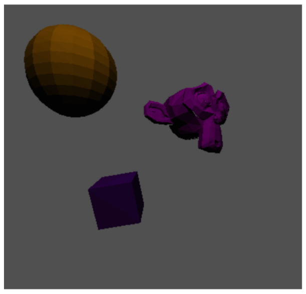
To test both rendering algorithms I created multiple scenes with different numbers of objects, varying orientations, and different shading methods. All executions were performed on the same hardware consisting of:

- Windows 11 Pro, 64-bit operating system, x64-based processor
- Intel Core i9-9900K CPU @ 3.60GHz processor
- 8 Cores, 16 logical processors
- 32.0 GB ram

Each picture corresponds to a test performed, laid out in order of the tests in each table.

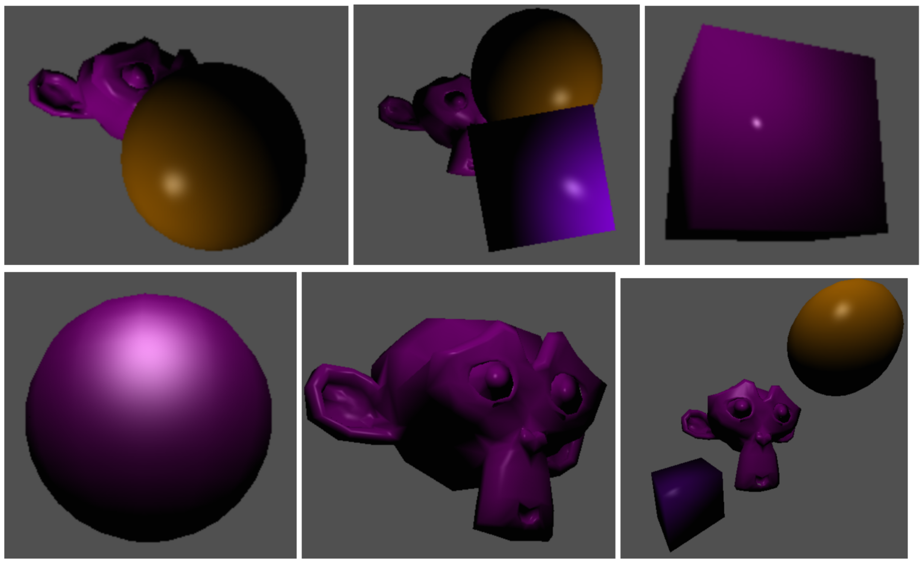
Flat shading, original rendering algorithm (non-raytraced)

Test	Time Parallel (seconds)	Time Serial (seconds)
flat_ortho_1	3.304	2.261
flat_ortho_2	3.505	2.176
flat_depth_1	5.394	4.074
flat_depth_2	2.472	1.250
flat_perspective_1	2.009	0.931
flat_perspective_2	2.644	1.102

Phong shading, original algorithm (non-raytraced)

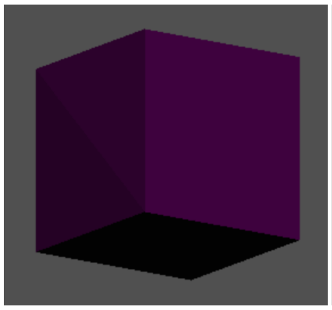
Test	Time Parallel (seconds)	Time Serial (seconds)
phong_depth_1	7.522	6.797
phong_depth_2	8.234	9.726
phong_highphongexp_1	5.709	3.528
phong_lowphongexp_1	16.872	11.910
phong_ortho_1	35.936	26.888
phong_perspective_1	16.327	17.703



Flat shading, raytracing algorithm

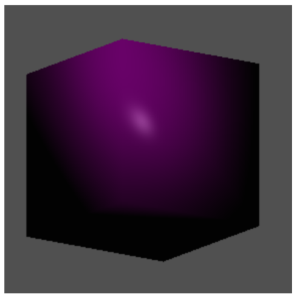
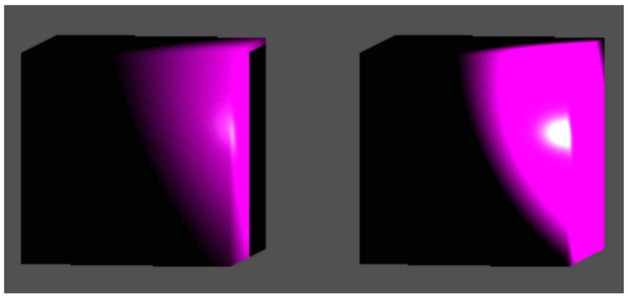
Test	Time Parallel (seconds)	Time Serial (seconds)
flat_ortho_1_raytrace (10 rows at a time)	46.647	332.977
flat_ortho_1_raytrace (20 rows at a time)	39.278	
flat_ortho_1_raytrace (25 rows at a time)	45.826	
flat_ortho_1_raytrace (50 rows at a time)	45.748	
flat_ortho_2_raytrace (10 rows at a time)	96.192	484.905

Test	Time Parallel (seconds)	Time Serial (seconds)
flat_ortho_2_raytrace (20 rows at a time)	90.085	
flat_ortho_2_raytrace (25 rows at a time)	91.355	
flat_ortho_2_raytrace (50 rows at a time)	96.239	



Phong shading, raytracing algorithm

Test	Time Parallel (seconds)	Time Serial (seconds)
phong_ortho_1_raytrace (10 rows at a time)	54.105	287.482
phong_ortho_1_raytrace (20 rows at a time)	51.162	
phong_ortho_1_raytrace (25 rows at a time)	52.653	
phong_ortho_1_raytrace (50 rows at a time)	54.431	
phong_ortho_2_raytrace (10 rows at a time)	102.669	516.464
phong_ortho_2_raytrace (20 rows at a time)	94.580	
phong_ortho_2_raytrace (25 rows at a time)	93.626	
phong_ortho_2_raytrace (50 rows at a time)	98.306	

Evaluation

The initial results for the parallelization of the basic rendering algorithm do not support the theory that by processing the objects for a scene in parallel the execution time will be reduced. As seen in the first table, the execution time for the parallel code was always greater than its sequential counterpart. This remains the case even as more objects are added to the scene. However, when the shading method is changed to a more computationally heavy technique, phong shading, the results start to become more promising. Here we can see that as more objects are added to the scene the parallel renderer provides a better execution time than the sequential one, although not by much. This changes when we start looking at the raytracing rendering algorithm. Here, even for flat shading there are large improvements in execution time for the parallel renderer over the sequential one. On average, the parallel raytraced rendering algorithm is five times faster than the non-parallel algorithm. However, since the method for rendering the scene is different, I was unable to implement the parallelism for each object in the scene and instead split up the work by groups of pixels on the screen. This brought up the question of how to split up the groups, i.e. how many rows of pixels should be given to each process. To find the answer to this question I tested multiple sizes of groups, each able to divide the screen size of 500 by 500 pixels evenly. Based on the test results above I estimate that the optimal group size is between 20 and 25 rows per process.

Conclusion

While rendering images based on 3D object inputs is a common task, finding efficient solutions is a difficult endeavor that requires thoughtful research. By implementing multiple rendering algorithms and testing their effectiveness in both sequential and parallel environments this study has established that parallel processing can be an effective technique to improve the efficiency of an image rendering program. However, this is not always the case as is true of the basic rendering algorithm that was tested. Here for the scenes that were tested the overhead in creating the parallel processes outweighed the potential benefits of computing the details for multiple objects at the same time. Although this does not support the research goal, parallel processing still proved to be effective in the ray-traced rendering algorithm by providing much faster execution times. Future research into parallel processing for image rendering algorithms should focus on how to further implement more multiprocessing into the rendering pipeline. Perhaps in other modules or in addition to what has already been implemented in the renderer module. Furthermore, it would be valuable to find other methods of multiprocessing that do not incur high overhead costs.

Works Cited

Chickerur, Satyadhyan, et al. “Parallel Rendering Mechanism for Graphics Programming on Multicore Processors.” International Journal of Grid and High Performance Computing, vol. 5, no. 1, 2013, pp. 82–94, https://doi.org/10.4018/jghpc.2013010106.

Dokter, Mark, et al. “Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU.” Computer Graphics Forum, vol. 38, no. 2, 2019, pp. 93–103, https://doi.org/10.1111/cgf.13622.

Molnar, Steven, et al. "A Sorting Classification of Parallel Rendering." University of North Carolina at Chapel Hill and Princeton University, 1994. https://www.cs.unc.edu/~fuchs/publications/SortClassify_ParalRend94.pdf

Git Repo

https://github.com/zodunn/Public_715_Final