Ray Tracer for complex scene
Final Report

<jg4006> Jiakang Guo

1. Motivation

Ray tracing is a common technique for generating realistic lighting and reflection effects. Most related algorithms are parallel in nature in order to run on GPUs, which makes it suitable to be the final project of this course.

As opposed to the majority of ray tracer projects that only support basic geometries such as cube, plane and sphere, placed in a simple scene, this project is intended for complex scenes, which means 1) multiple objects and lights are placed in different pose, 2) objects are defined with triangle mesh instead of simple parametric functions, and each mesh contains thousands of triangles.

To deal with a complex scene, it's not a good idea to use custom scene definition formats (like .xml), if there's no real-time visualization tools for that format. For example, manually calculating the pose of all objects is hard, and we have to run our renderer frequently to verify any adjustment. Thus, I make use of the Unity Engine, where I can edit the scene before exporting the generated scene definition (.unity) to our ray tracer. On the other hand, I have to ensure that my parsing and transformation result would generate the exact same scene as I see in the Unity editor.

To support large number of triangles (usually >10k in total) given by the mesh, it's obviously unacceptable if we check every one of them for a ray-triangle intersection test when doing ray casting. Instead, we have to build a space partitioning data structure, such as BVH, octree or k-d tree, to discard most unnecessary checks. Kd-tree is my choice for this project, which provide logarithm time complexity as opposed to linear in case of brute force.

2. What I implemented and & How

2.1 Input handling

Parsing the scene definition and performing correct geometric transformation is the first step towards scene construction. The input for the scene is a .unity file with description of each object's pose (position & rotation represented by euler angles) , scale (in x,y,z axis), and mesh name. After knowing what mesh we would be needing, we can read them from corresponding .obj files. For simplicity, I use the name of a *GameObject* in Unity to represent the file name, and also ask the user to provide a directory where he stores the .obj file. The parser I wrote for .obj only supports triangle faces, so any .obj with polygon faces have to be triangulated with softwares such as MAYA and Blender.

The triangles defined in the .obj file represents their position in a local coordinate. We have to transform it to the global coordinate based on its pose and scale in the scene. The point and normalized vector transformation are implemented in transformPoint and transformVectorNormal function. Note that Unity uses a left-handed coordinate system which is the opposite to the convention in most mathematics courses, so here I swapped the Y and Z axis and turned it into a right-handed coordinate system. It makes no difference in the rendering result as the scene itself is not changed.

At the end of this step and before performing ray trace, the scene is represented with a list of triangles, with material information attached to them as well.

2.2 Ray Tracing

The ray tracing algorithm is implemented as follows:

For each pixel on the screen,

1. Shoot a ray from the view point to the center of that pixel. If the ray doesn't hit any surface, return background_color. Otherwise, go to 2.
2. Calculate

diffuse_color = sum {light_color * dot(surface_normal, light_dir) * surface_diffuse} for each *visible light*, where surface_normal is the interpolated normal inside a triangle, light_dir is the direction from the hit point to the light, and *visible light* means there's nothing that blocks the light from reaching the hit point. To determine if a light is *visible*, we have to do a ray cast from the light and check the first hit.

3. Calculate the reflected ray based on incoming ray and surface normal. Cast the reflected ray. If it reaches maximum recursion count or doesn't hit any surface, return **diffuse_color.** Otherwise, go to 2 and recursively calculate the **reflected_color**. Return **diffuse_color +** surface_reflection * **reflected_color.**

This approach is a basic version for ray tracing, but it's good enough to demonstrate some reflection effects as well as shadows.

Each pixel can be run in parallel since there's no dependency in between.

2.3 K-d tree

The k-d tree needs to be built before starting ray tracing, to accelerate the ray cast, i.e. ray-triangle intersection query. It is a binary search tree that stores a group of triangles. Each node is split on the center of mass (C.M.) of all triangles underneath: all triangles completely lying on the left of the C.M. go to the left branch, all triangles completely lying on the right go to the right branch, while the rest of them (i.e. spanning across both sides) are stored right in this node. Level 1,4,7.. split on the X axis, level 2,5,8.. split on the Y axis, and level 3,6,9.. split on the Z axis. Its construction can be done recursively as described above. I build it in a single-threaded manner as it usually takes up less than 10% of the overall running time.

To check the *first/nearest* intersection between a ray (defined by its origin and direction) and a group of triangles stored in a k-d tree, it can be easily done recursively for each tree node. Here we assume the ray goes from left to right with regard to the particular axis of that level, since the opposite case is symmetric.

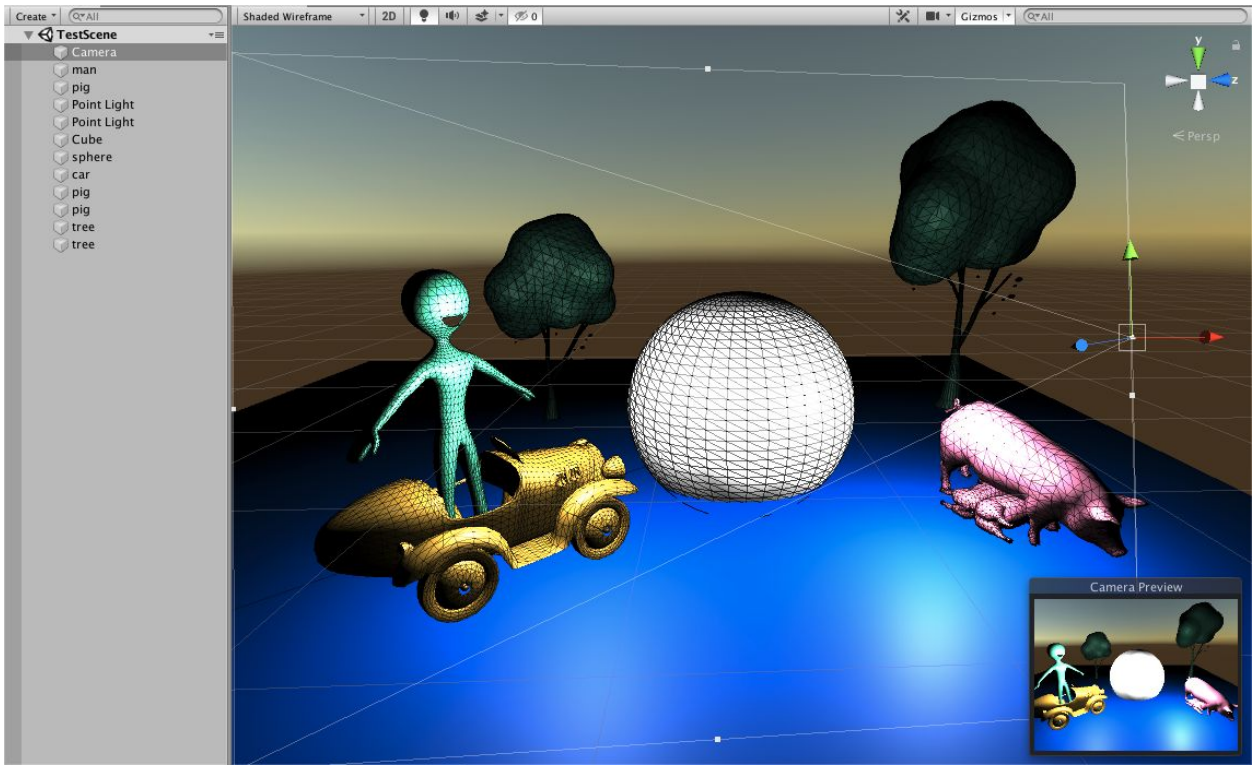
1.Iterate upon all triangles stored in that node and store all hits. If this is a leaf node, just return the nearest one.
2.If the axis-aligned bounding box (AABB) of the left child intersects with the ray, run the algorithm for the left child. If the result is not empty, go to 4.
3.If the AABB of the right child intersects with the ray, run the algorithm for the right child.
4.Combine the outcome of step 1,2 and 3 and return the nearest one.

We make use the AABB to drop out a branch if its bounding box doesn't even intersect with the ray. We also adopt a trick in step 2 since any intersection in step 3, i.e. with the right child, is further than what we get from the left child, so there's no point doing step 3 in this case.

Utilizing the k-d tree provides a huge performance improvement for this scenario, while the trick mentioned above provides another 10%~15% speed up. It will take **days** to render the baseline scene in the next section without using k-d tree, based on my observation over a much smaller scene.

3. Baseline scene

We would use a consistent scene below for demonstration and efficiency analysis. The scene is edited and previewed inside the Unity editor, as shown in the screenshot below. Note that Unity uses a different lighting model so what we care about here is the geometry relationship rather than the exact color.



The mesh of all objects are downloadable for free on the Internet. Here's a summary of what exactly the scene contains:

Name	Triangles contained	Number	Material
Camera	/	1	/
Point Light	/	2	/
sphere	6,240	1	100% Reflective
Cube (used as floor)	12	1	80% Reflective 40% Diffusion, Blue
man	4,704	1	Diffusion, Green
car	10,385	1	Diffusion, Yellow
pig	2,454	3	Diffusion, Pink
tree	6,565	2	Diffusion, Green
Total	**41,833**	**12**	**/**

Here's the result of the ray tracer two resolutions 1) 1024*768 2) 320*240, and trace depth = 2(At most 2 reflections are considered for each ray). The first resolution is for evaluating the peak image quality, while the second is for doing tests and analysis since the total time is far shorter.

Resolution = 1024 * 768. Total Time = 6504s, GC = 1068s on 8 threads

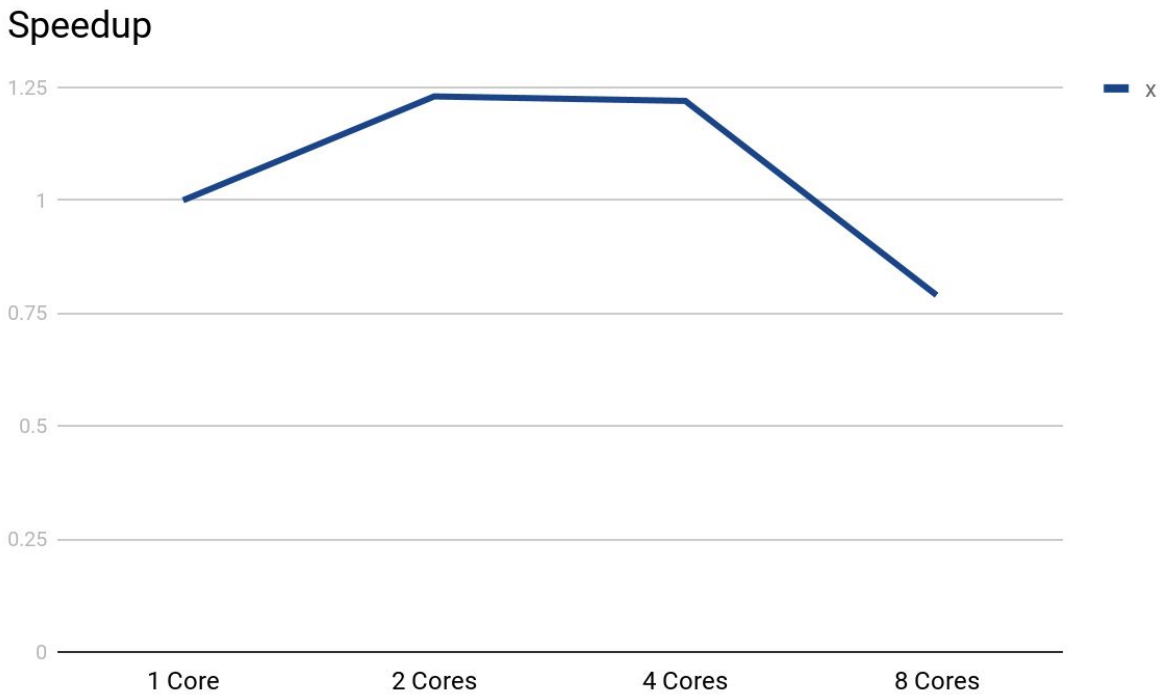
Resolution = 320 * 240. Total Time = 133s, GC = 48s on 8 threads

We can see from the first image that the ray tracer produces correct reflection and shadow effects, while geometric relationship of all objects is the same as what we see in the Unity Editor.

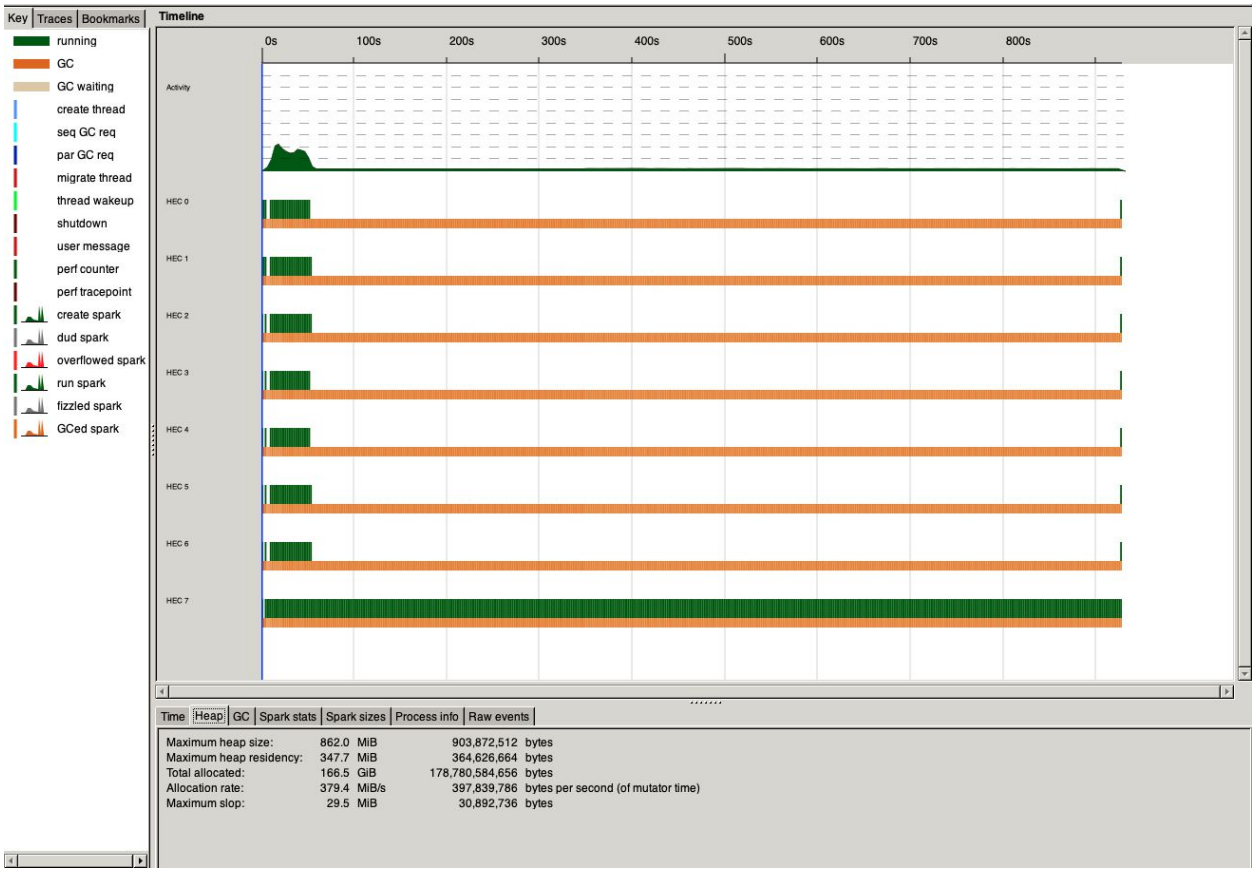
4. Parallelism Analysis

4.1 Parallelism on pixels

The first attempt I made was to evaluate each pixel parallelly, i.e. we would have 320*240 sparks when rendering a 320*240 image. I simply used parList API with deepseq strategy, to fully evaluate every pixel's color.



Resolution = 320 * 240

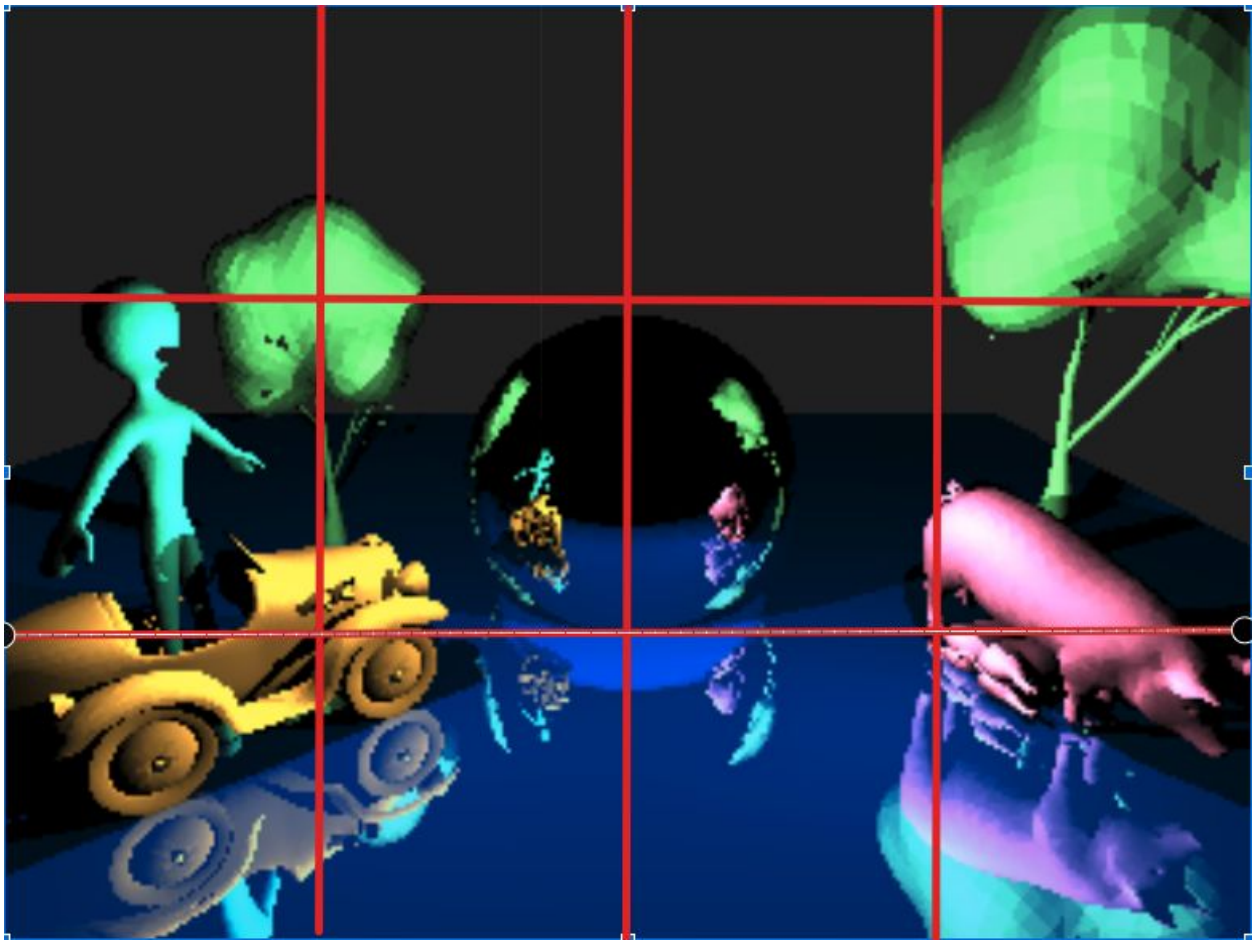


Resolution = 320 * 240, Time = 935s on 8 threads

We can see that multi-thread execution does provide some speedup, but apparently not to the degree we want. The number of sparks is way more than the extra threads we have, and the task distribution is heavily unbalanced.

4.2 Parallelism on blocks

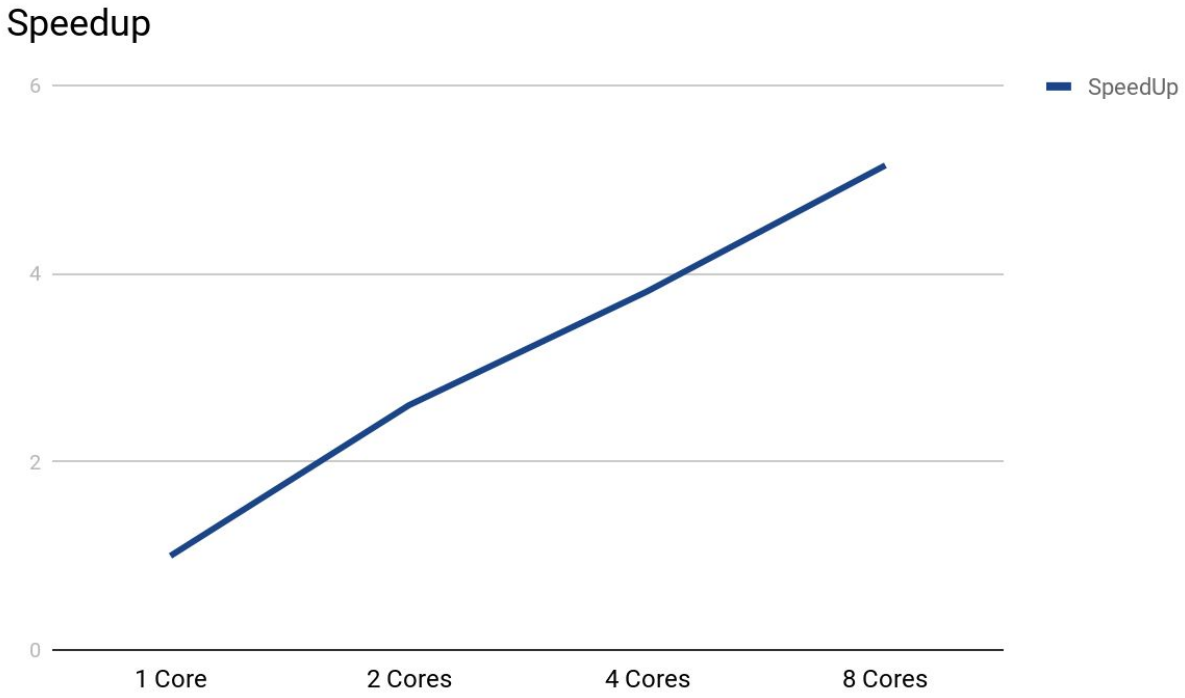
To avoid creating too many sparks, it's natural to divide the pixels into blocks and evaluate all pixels inside the same block sequentially. The figure below shows the result of dividing the image into 12 blocks so that we can evaluate them with 12 threads.



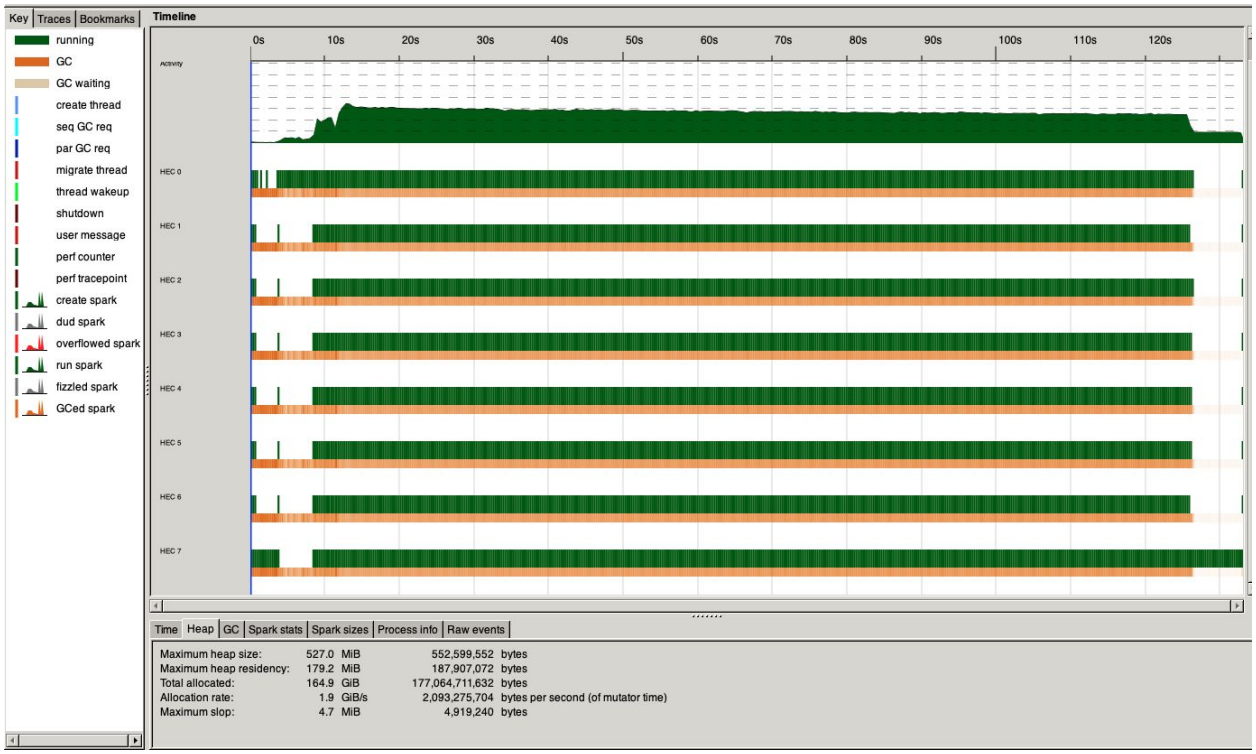
However, there's a pitfall in this case that the first 3 blocks barely need any execution time, as there's almost nothing in that area. As a result, the task to be heavily unbalanced and lead to notably longer execution time.

To avoid this pitfall, we just have to break the sequential order of pixels inside a block. The way I adopt is doing a permutation to the list of all pixels that places

ith element to slot (p*i) % length, where p is a large prime number and length is the length of the list. This way we can ensure that no two elements go to the same slot, i.e. we won't evaluate the same ray twice *(Proof: assume p*i = p*j + k*length => length | p(i-j) => length | (i - j) given p > length and p is prime => i = j)*. Once we permute the list of rays, there's a far better chance that task will be distributed equally among all threads. To retrieve the final result, we just have to do the inverse mapping that brings a pixel back to its original position.



Resolution = 320 * 240



Resolution = 320 * 240, Time = 133s on 8 threads

The speedup now steadily increases as we use more threads. Also, we get a much more even distribution of tasks on all threads.

If we compare the two approaches closer, we can see that there's no significant time difference if using a single thread, while the latter is 7 times faster when it comes to 8 threads. Honestly it's a bit strange to me what makes the difference so huge.

5. Summary

This project exploits the power of the data structure (k-d tree) and parallelism in accelerating ray tracing algorithm, and produces a deliverable that enables an efficient workflow with the help of the Unity Editor.

There's still plenty of room for improvement. For instance, we can see from the threadscope screenshot that the construction of k-d tree takes up the first 5~10% running time, on a single thread. There are some parallel k-d tree construction papers suggested by the TA, but I didn't have enough time for that.

The file raytracer.hs contains all the code I wrote for this project. All codes are written by myself, while any reference is shown below as well as inside a comment.

6. References

Htrace, Haskell RayTracer http://www.nobugs.org/developer/htrace/index.html

Notes on efficient ray tracing, Solomon Boulos, University of Utah

http://www.cs.utah.edu/~awilliam/box/box.pdfb