**Title:** 2D Ray Tracer by Emily Zhang and Rahul Khandelwal
**URL:** https://exiaohuaz.github.io/15418-final-website/

**Summary:** We will create and observe the performance of 2D raytracing on scenes of input images and light sources. We will implement the image convolution and raycasting algorithms with various methods of parallelization, such as CUDA, ISPC, and OpenMP. We also intend to compare our results with the performance of pre-existing programs that execute 2D raytracing.

**Background:** Our project intends to accelerate the process of image convolution and raycasting. Image processing is widely used in many fields – for example feature extraction for machine learning, as well as image enhancement, restoration, encoding, and compression. The method by which many of these processes happen is by applying a convolution matrix to the image. Convolution is the process of adding each element of the image to its local neighbors, weighted by the convolution matrix.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy),$$

Wikipedia (https://en.wikipedia.org/wiki/Kernel_(image_processing)) supplies some sample code for applying a convolution matrix to an image:

```
for each image row in input image:
  for each pixel in image row:

    set accumulator to zero

    for each kernel row in kernel:
      for each element in kernel row:

        if element position  corresponding* to pixel position then
          multiply element value  corresponding* to pixel value
          add result to accumulator
        endif

    set output image pixel to accumulator
```

Raytracing is a technique for modeling light transport for use in a wide variety of rendering algorithms for generating digital images. It is used in any application that does rendering, such as game engines or CGI effects in movies. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it.

Ray casting qualifies as a brute force method for solving problems. The minimal algorithm is simple, particularly in consideration of its many applications and ease of use, but applications typically cast many rays. Millions of rays may be cast to render a single frame of an animated film. For this reason we might hope to see benefits from parallelism across rays. Additionally, larger image formats with more pixels could see benefits from parallelizing across pixels, in a similar manner to how the CUDA circle renderer was able to parallelize over pixels fairly well. Computer processing time increases with the resolution of the screen and the number of primitive solids/surfaces in the composition.

**The Challenge:**

Implementing the ray tracing algorithm from scratch will be tricky
Implementing various parallel versions of the ray tracing algorithms is a lot of work, and will be tricky.
Determining correctness of our algorithms will not be easy. At the minimum we could start with a sequential algorithm and ensure computed pixel values for the parallel algorithm are close to the sequential algorithm's outputs.

In terms of the communication to computation ratio, I would hypothesize this would become more prevalent in the animated or multiple frames/timesteps scenario. At each frame, the main processor that loads the image file would have to communicate out pixels and light blocking objects to other processors.
The number of bounded objects in a scene would affect the memory access pattern. Rays that are halted by any identified objects, also as the rays branch out, the computation of which rays intersect which pixels changes. Since there is no way to know beforehand which rays will propagate the furthest or which pixels will receive the most rays, a dynamic workload distribution may be needed here.

**Resources:**

Access to local, ghc, and psc machines would be sufficient for the purposes of this project. We would be starting from scratch but referencing algorithms online for raytracing and convolution techniques. This was an example of a project that was able to accomplish some of our goals using OpenGL: https://www.taylorpetrick.com/blog/post/2d-ray-tracing

**Goals and deliverables:**

Plan to achieve:

- Implement sequential image convolution and ray tracing alg
- Parallelize convolution and ray tracing alg with CUDA, ISPC, OpenMP
- Compare performance across methods, parallelizing image convolution and ray tracing separately will allow us to observe the individual performance gains in each)
- Achieve speed fast enough to support high fps gaming

- Present speedup graphs vs. implementations and processor counts

Hope to achieve:
- Find another program that does 2d ray tracing and figure out how to measure its performance against ours.
- Implement ray tracing using a quad tree accelerated approach and observe performance differences there.
- Have a demo app that takes screen pixels and updates lighting in real time.

**Platform Choice:**

Our algorithms would be written using C++. We are using CUDA, ISPC, and OpenMP to compare parallelization methods. We should run the algorithms mostly on an 8 core CPU because our local machines as well as the GHC machines have 8 cores and this would replicate a typical video game environment.

**Schedule:**

11/16: Our sequential convolution algorithm would be ready, and we would be part way through parallelizing this. We would begin writing a basic ray tracing algorithm.

11/23: Parallelized convolution would be finished on at least one framework, our basic ray tracing algorithm would be ready, and we would be starting our parallel approaches for the ray tracing algorithm.

11/30: The parallelized ray tracing algorithm would be ready on at least one framework. We would begin comparing results and analyzing data.

12/7: At this stage, we should have finished comparing results, hopefully for all the frameworks planned, and begin trying to achieve stretch goals or any unreached goals

12/14: Achieve stretch goals

12/17: Finalize Writeup Report

12/18: Finish up final touches on presentation