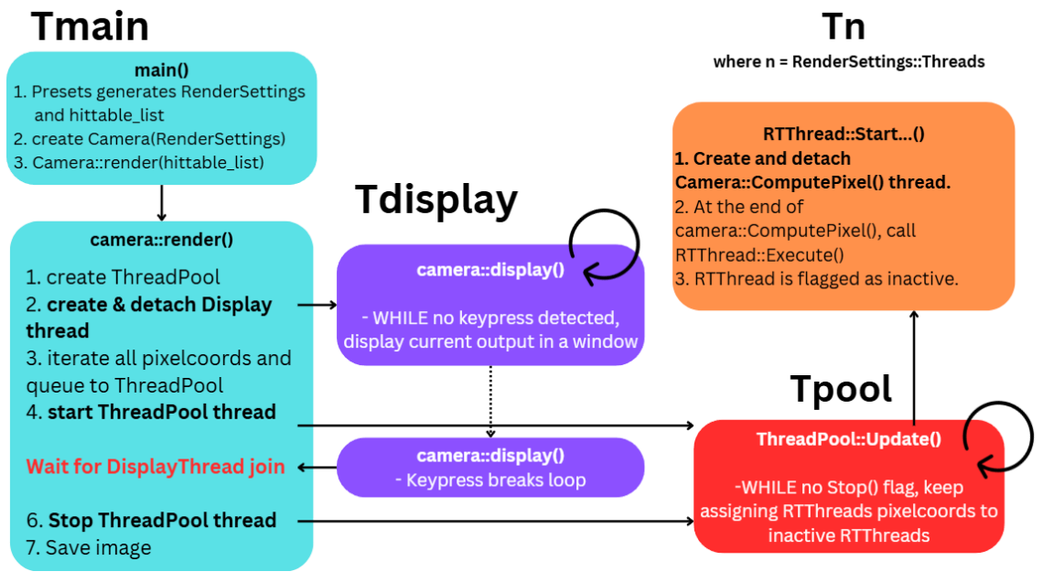
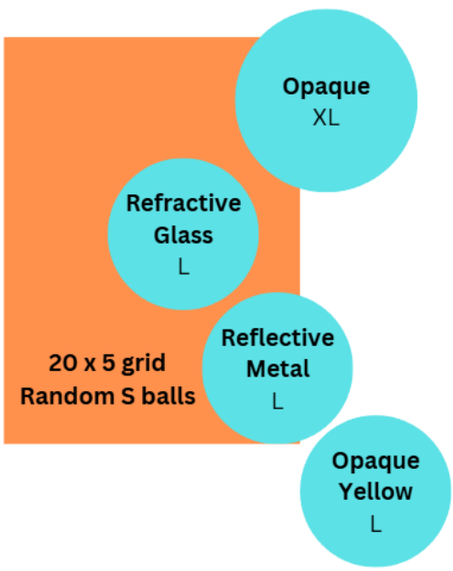
Technical Report
Raytracing in One Weekend Multithreading

This is a multithreaded extension of the Raytracing in One Weekend (Shirley, etal. 2024).



Multithreading generates 3 Active threads and N amount of RTThread worker threads..

1. First is the main thread which generates the world objects list, rendersettings and camera. Eventually it creates the Display thread and ThreadPool thread, and will then WAIT.
2. Display thread is generated and is simply a visual representation of progress. It needs to terminate first before main thread can proceed to saving.
3. The ThreadPool thread endlessly checks if it can assign available RTThreads any work.
4. Last are the N worker threads which call Camera::ComputePixel() before flagging itself as available upon finishing ComputePixel().



For testing, spheres were laid out with different materials assigned/randomized.

In testing with a 4 core (8 processor) device, creating 4 worker threads generated 90+% CPU usage; 5 threads used 100%.

The render method differs as it is by pixel assignment instead of line. Another difference is that to avoid race conditions in ThreadPool's queue, all pixelcoords are first added before ThreadPool's update is started to avoid a push/pop conflict. Last difference is that only pixelcoords data is queued, not threads since race conditions are an unlikely concern inside camera's functions, then multiple threads can use the same camera instance.

For future improvements, consider extracting the Camera::ComputePixel() method into a thread method instead. This would avoid context switching overhead since they're sharing the same camera instance.

The reason this wasn't performed was because ComputePixel() was heavily reliant on private variables within Camera. There would be an extreme overlap that it's more sensible to turn the Camera class entirely into a thread class instead.

Source code: https://github.com/CharlesO55/GDPARCM