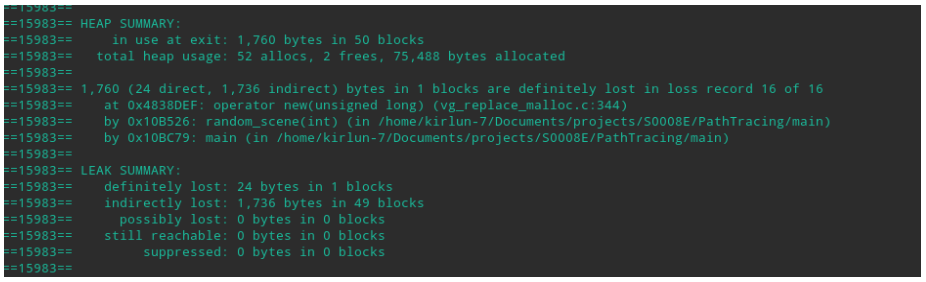
Optimizing a Path Tracer

Kristoffer Lundgren

January 17, 2020

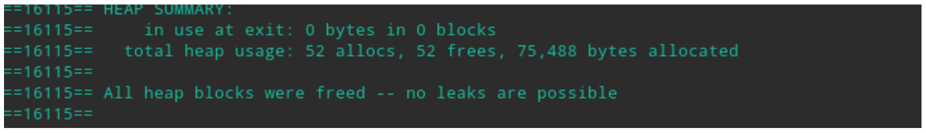
1 Valgrind Static Code Analysis



```
==15983==
==15983== HEAP SUMMARY:
==15983==     in use at exit: 1,760 bytes in 50 blocks
==15983==   total heap usage: 52 allocs, 2 frees, 75,488 bytes allocated
==15983==
==15983== 1,760 (24 direct, 1,736 indirect) bytes in 1 blocks are definitely lost in loss record 16 of 16
==15983==    at 0x4838DEF: operator new(unsigned long) (vg_replace_malloc.c:344)
==15983==    by 0x10B526: random_scene(int) (in /home/kirlun-7/Documents/projects/S0008E/PathTracing/main)
==15983==    by 0x10BC79: main (in /home/kirlun-7/Documents/projects/S0008E/PathTracing/main)
==15983==
==15983== LEAK SUMMARY:
==15983==    definitely lost: 24 bytes in 1 blocks
==15983==    indirectly lost: 1,736 bytes in 49 blocks
==15983==      possibly lost: 0 bytes in 0 blocks
==15983==    still reachable: 0 bytes in 0 blocks
==15983==         suppressed: 0 bytes in 0 blocks
==15983==
```

Figure 1: First Analysis

The first Valgrind analysis revealed a lot of memory leaks but that was to be expected I had made no effort to prevent memory leaks. I then added a deconstructor in the hitableList class which in turn loops through all its contents and calls their destructors.



```
==16115== HEAP SUMMARY:
==16115==     in use at exit: 0 bytes in 0 blocks
==16115==   total heap usage: 52 allocs, 52 frees, 75,488 bytes allocated
==16115==
==16115== All heap blocks were freed -- no leaks are possible
==16115==
```

Figure 2: After Adding deconstructors

2 Optimizing

In order to optimize my path tracer i implemented simple multithreading since that provides the best performance impact. The threads each run a function that calculates a pixel, when a pixel is complete it calls a function to ask for another pixel.

```
void calcPixel(int ns, hitable * world, int nx, int ny, camera cam)
{
    while(1)
    {
        pixel* currentPixel = cont.getPixel();
        if(currentPixel == nullptr)
            break;
        Vector4D col(0,0,0,1);
        for (int i = 0; i < ns; ++i)
        {
            float u = float(currentPixel->x + xorShift())/float(nx);
            float v = float(currentPixel->y + xorShift())/float(ny);
            ray r = cam.getRay(u, v);
            Vector4D p = r.pointAtParameter(2.0);
            col = col + color(r, world, 0);

        }
        col = col / float(ns);
        col = Vector4D(sqrt(col[0]), sqrt(col[1]), sqrt(col[2]), 1);
        currentPixel->r = int(255.99*col[0]);
        currentPixel->g = int(255.99*col[1]);
        currentPixel->b = int(255.99*col[2]);
    }
}
```

This code snippet starts six threads that all run the function seen above.

```
std::thread threads[6];
for (int m = 0; m < 6; ++m)
{
    threads[m] = std::thread(calcPixel, ns, world, nx, ny, cam);
}
for (int j = 0; j < 6; ++j)
{
    threads[j].join();
}
```