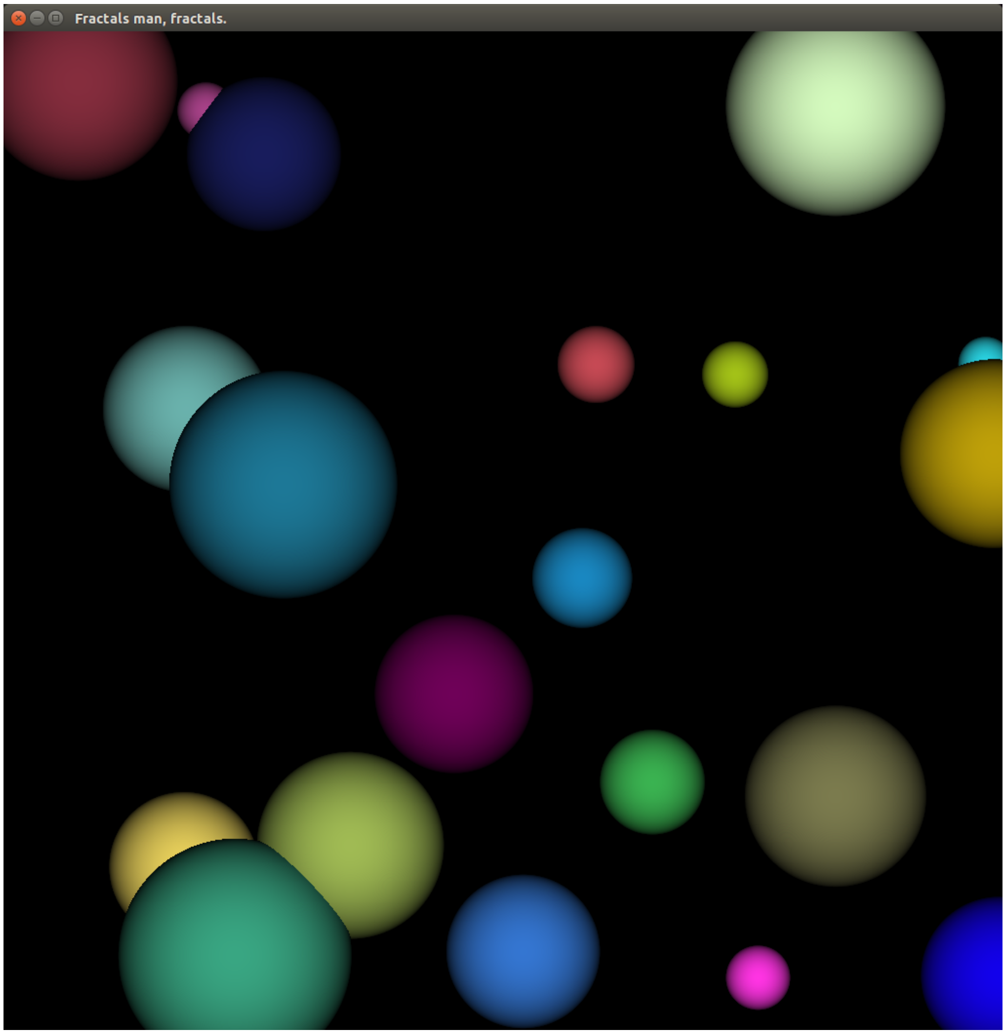
Barker, Mary

Mathematical Modeling, Spring 2017

Homework 10

Ray Tracing on GPU with Constant Memory


Fractals man, fractals.

```
/*
        Mary Barker HW 10
        Ray Tracing with constant memory

        to compile and run:
                nvcc Barker10.cu -lm -lGL -lGLU -lglut
                ./a.out
*/
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#define INF 2e10f
#define SPHERES 20
#define rnd( x ) (x * rand() / RAND_MAX)
#define MIN(x,y) (x< y) ? x : y
#define xmin -50
#define xmax  50
#define ymin -50
#define ymax  50

struct Sphere {
        float r, g, b;
        float x, y, z;
        float radius;
};

Sphere s[SPHERES];
__constant__ Sphere GPUs[SPHERES];

// arrays to hold pixels
float * pixels = NULL;
float * GPUpixels = NULL;

//thread format for screen display
unsigned int window_width = 1024;
unsigned int window_height = 1024;
float stepSizeX = (xmax - xmin)/((float)window_width - 1.0);
float stepSizeY = (ymax - ymin)/((float)window_height - 1.0);
dim3 nthreads = MIN(window_width, 1024);
dim3 nblocks = (window_width*window_height - 1) / nthreads.x + 1;

__device__ float hit(float x, float y, float z, float radius, float ox, float oy,
    float *n ) {
        float dx = ox - x;
        float dy = oy - y;

        if (dx*dx + dy*dy < radius*radius) {
                float dz = sqrtf( radius * radius - dx*dx - dy*dy );
                *n = dz / sqrtf( radius * radius );
                return dz + z;
        }
        return -INF;
}
```

```
__global__ void trace_rays(float * pix, float dx, float dy, int nx, int ny) {
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        float xx, yy, maxz=-INF;

        if(i < nx * ny) {

                float rr = 0, gg = 0, bb = 0;

                xx = (xmin + threadIdx.x * dx);
                yy = (ymin +  blockIdx.x * dy);

                for(int j = 0; j < SPHERES; j++){

                        float n, t = hit(GPUs[j].x, GPUs[j].y, GPUs[j].z, GPUs[j].
                            radius, xx, yy, &n);

                        if(t > maxz){
                                rr = n * GPUs[j].r;
                                gg = n * GPUs[j].g;
                                bb = n * GPUs[j].b;
                                maxz = t;
                        }
                }
                pix[3*i+0] = rr;
                pix[3*i+1] = gg;
                pix[3*i+2] = bb;
        }
}

void allocate_memory() {
        pixels = (float*)malloc(3*window_width*window_height * sizeof(float));
        cudaMalloc(&GPUpixels, 3*window_width*window_height * sizeof(float));
        for(int i = 0; i < SPHERES; i++){
                s[i].x =                rnd(100.0f) - 50;
                s[i].y =                rnd(100.0f) - 50;
                s[i].z =                rnd(100.0f) - 50;
                s[i].r =                rnd(1.0f);
                s[i].g =                rnd(1.0f);
                s[i].b =                rnd(1.0f);
                s[i].radius =           rnd(10.0f) + 2;
        }
        cudaMemcpyToSymbol(GPUs, s, SPHERES*sizeof(Sphere));
}

void display(void) {
        allocate_memory();

        trace_rays<<<nblocks,nthreads>>>(GPUpixels, stepSizeX, stepSizeY, window_width
            , window_height);

        cudaMemcpy(pixels, GPUpixels, 3*window_width*window_height*sizeof(float),
            cudaMemcpyDeviceToHost);

        glDrawPixels(window_width, window_height, GL_RGB, GL_FLOAT, pixels);
        glFlush();
```

```
}

int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
        glutInitWindowSize(window_width, window_height);
        glutCreateWindow("Fractals␣man,␣fractals.");
        glutDisplayFunc(display);
        glutMainLoop();
}
```