600086 Lab Book

Week 7 – Lab 6 A simple CUDA ray caster

Date: 24th Mar 2022

Exercise 1. Drawing based on a canvas of size [-1, 1]x[-1, 1]

Question1: implement a solution using GPU processing to solve the problem

Solution:

1. Draw the image based on pixel coordinates defined in float type variables in [-1, 1]x[-1,1]

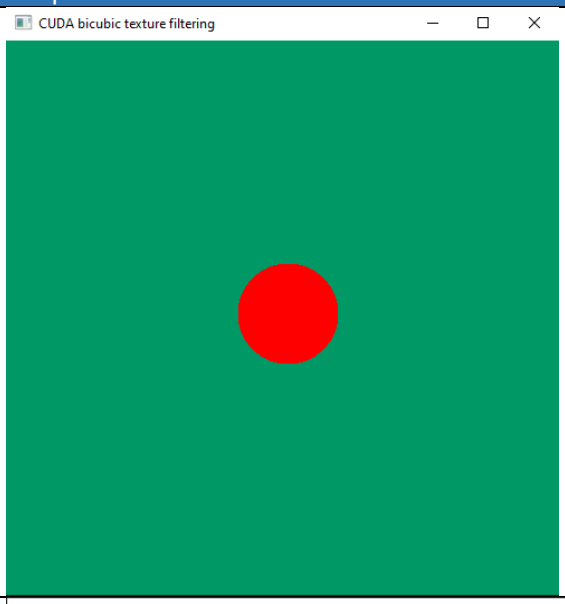
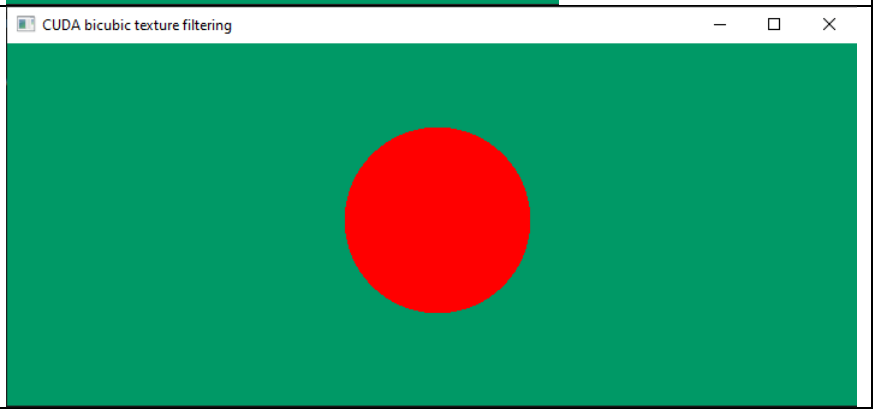
```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;

    uint c = 255;
    float r = 0.5;
    if ((x < width) && (y < height))
    {
        float dist = sqrtf(powf(u - (0) ,2) + powf(v - (0),2));

        if(dist<r)
        {
            d_output[i] = make_uchar4(0x00 , 0x00 , 0xff , 0);
        }
        else
        {
            d_output[i] = make_uchar4(0x66, 0x99, 0x00, 0);
        }
    }
}
```

Added in a translation for the pixel location represented by u and v and then added a scale translation to ensure the resultant image matched the aspect ratio of the window to prevent distortion. See Sample Output Ref 1 and 2 for the results.

Test data:

N/A

Sample output:

REF	Output
2	
2	

Reflection:

none

Metadata:

Further information:

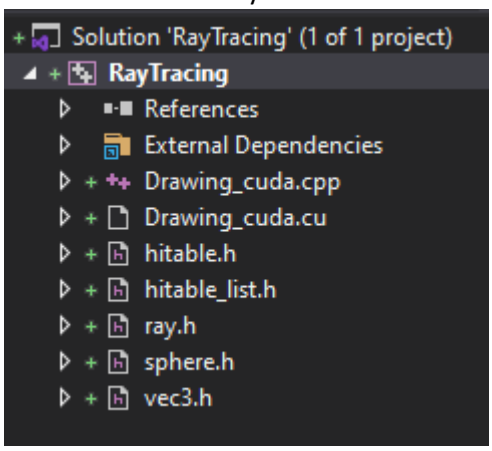
none

Exercise 2. Write a simple ray caster

Question1: implement ray casting based on raycasting in a weekend repo

Solution:

1. Add the necessary header files to the project



2. Change the variable names in the ray class to make them more meaningful for the current implementation

```
class ray
{
public:
    __device__ ray() {}
    __device__ ray(const vec3& a, const vec3& b) { O = a; Dir = b; }
    __device__ vec3 origin() const { return O; }
    __device__ vec3 direction() const { return Dir; }
    __device__ vec3 point_at_parameter(float t) const { return O + t * Dir; }

    vec3 O;
    vec3 Dir;
};
```

Changed the A and B variables to be O for origin and Dir for direction.

3. Implement the following functions cuda_check_error, castRay, create_world and free_world

```
#define checkCudaErrors(val) check_cuda( (val), #val, __FILE__, __LINE__ )
void check_cuda(cudaError_t result, char const* const func, const char* const file, int const line) {
    if (result) {
        std::cerr << "CUDA error = " << static_cast<unsigned int>(result) << " at " <<
            file << ":" << line << " '" << func << "' \n";
        // Make sure we call CUDA Device Reset before exiting
        cudaDeviceReset();
        exit(99);
    }
}
```

```
__device__ vec3 castRay(const ray& r, hitable** world) {
    hit_record rec;
    if ((*world)->hit(r, 0.0, FLT_MAX, rec)) {
        return 0.5f * vec3(rec.normal.x() + 1.0f, rec.normal.y() + 1.0f, rec.normal.z() + 1.0f);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5f * (unit_direction.y() + 1.0f);
        return (1.0f - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
    }
}
```

```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list) = new sphere(vec3(0, 0, -1), 0.5);
        *(d_list + 1) = new sphere(vec3(0, -100.5, -1), 100);
        *d_world = new hitable_list(d_list, 2);
    }
}
```

```
__global__ void free_world(hitable** d_list, hitable** d_world) {
    delete* (d_list);
    delete* (d_list + 1);
    delete* d_world;
}
```

4. Modify the d_render method so that it will raycast and render the image

```
__global__ void d_render(uchar4* d_output, uint width, uint height, hitable** d_world)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    float u = x / (float)width;
    float v = y / (float)height;

    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;

    vec3 eye = vec3(0, 0.5, 1.5);
    float distFromEyeToImg = 1.0;
    if ((x < width) && (y < height))
    {
        vec3 pixelPos = vec3(u, v, eye.z() - distFromEyeToImg);
        ray r;
        r.O = eye;
        r.Dir = pixelPos - eye;

        vec3 col = castRay(r, d_world);
        float red = col.x();
        float green = col.y();
        float blue = col.z();
        d_output[i] = make_uchar4(red * 255, green * 255, blue * 255, 0);
    }
}
```

5. Modify the render() method so that it will create a sphere and pass it into the d_render method for casting

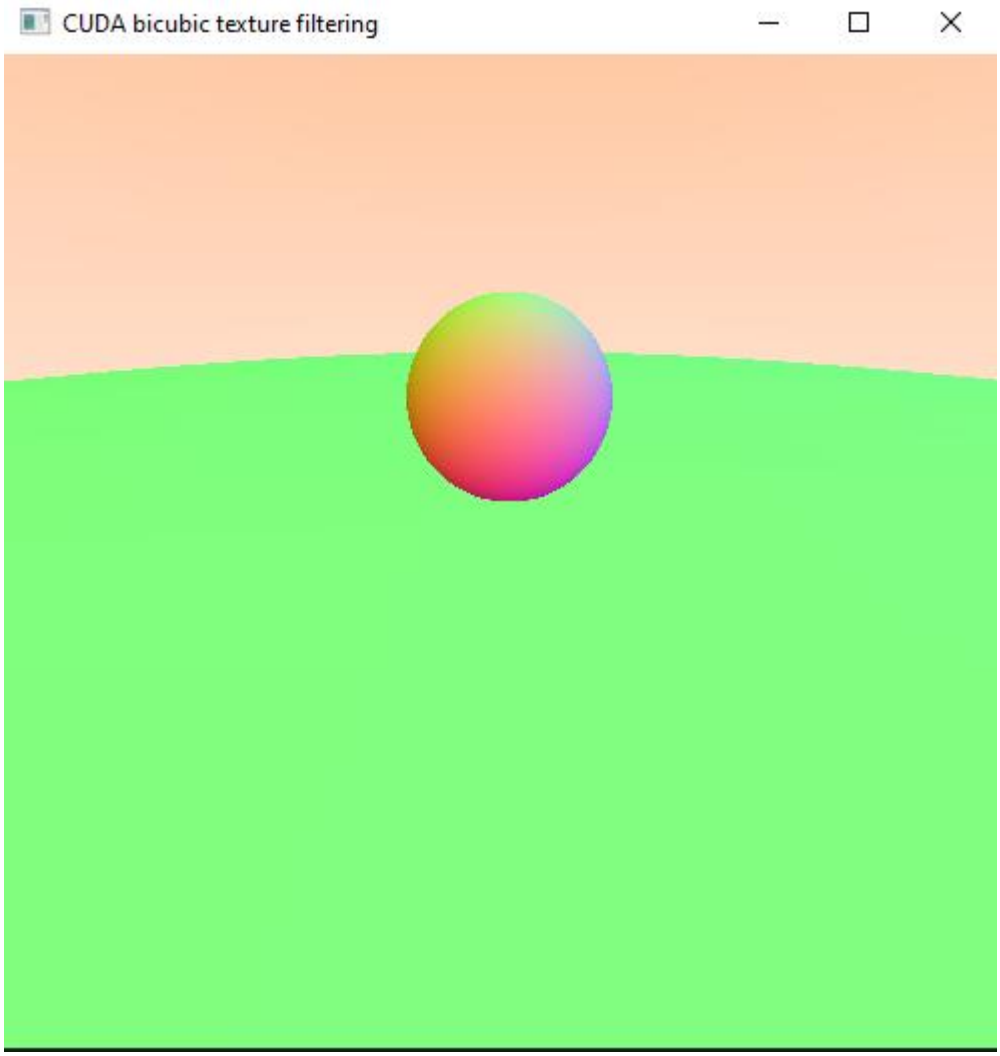
```
// render image using CUDA
extern "C"
    void render(int width, int height,  dim3 blockSize, dim3 gridSize, uchar4 * output)
{
    hitable **d_list;
    checkCudaErrors(cudaMalloc((void **)&d_list, 2 * sizeof(hitable*)));
    hitable **d_world;
    checkCudaErrors(cudaMalloc((void**)&d_world, sizeof(hitable*)));
    create_world << <1, 1 >> > (d_list, d_world);
    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaDeviceSynchronize());
    d_render << <gridSize, blockSize >> > (output, width, height, d_world);
    getLastCudaError("kernel failed");
}
```

Running the solution at this point renders the image shown in the sample out put section.

Test data:

none

Sample output:



Reflection:

The task above was fairly easy to follow unsure why my red and blue values seem to have switched making my sky tinged red rather than the blue in the example.

Metadata:

Exercise 3. Adding multiple spheres to the ray caster

Question1: modify the previous code to render 10 spheres on the screen

Solution:

```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list)       = new sphere(vec3(1, 1, 0), 0.2);
        *(d_list + 1)   = new sphere(vec3(1, 0.5, 0), 0.2);
        *(d_list + 2)   = new sphere(vec3(1, 0, 0), 0.2);
        *(d_list + 3)   = new sphere(vec3(0, 1, 0), 0.2);
        *(d_list + 4)   = new sphere(vec3(0, 0.5, 0), 0.2);
        *(d_list + 5)   = new sphere(vec3(0, 0, 0), 0.2);
        *(d_list + 6)   = new sphere(vec3(-1, 1, 0), 0.2);
        *(d_list + 7)   = new sphere(vec3(-1, 0.5, 0), 0.2);
        *(d_list + 8)   = new sphere(vec3(-1, 0, 0), 0.2);
        *(d_list + 9)   = new sphere(vec3(-0.5, 0.5, -1), 0.5);
        *(d_list + 10)  = new sphere(vec3(0.5, 0.5, -1), 0.5);
        *(d_list + 11)  = new sphere(vec3(0, -100.5, -1), 100);

        *d_world = new hitable_list(d_list, 12);
    }
}
```

Added in the 10 extra spheres to form a 3*3 grid of spheres with two larger spheres in the background further back. See sample output for resulting render.

Test data:

none

Sample output: