600086 Lab Book

Week 8 – Lab 8 A simple Particle animation in CUDA

Date: 24th Mar 2022

Exercise 1. Draw a box without front wall.

Question1: adjust the code from lab 7 so that you have an open box drawn to the sscreen

Solution:

I added in a set of 5 spheres that will make up the box they are very large making the curvature barely noticeable at our current scale.

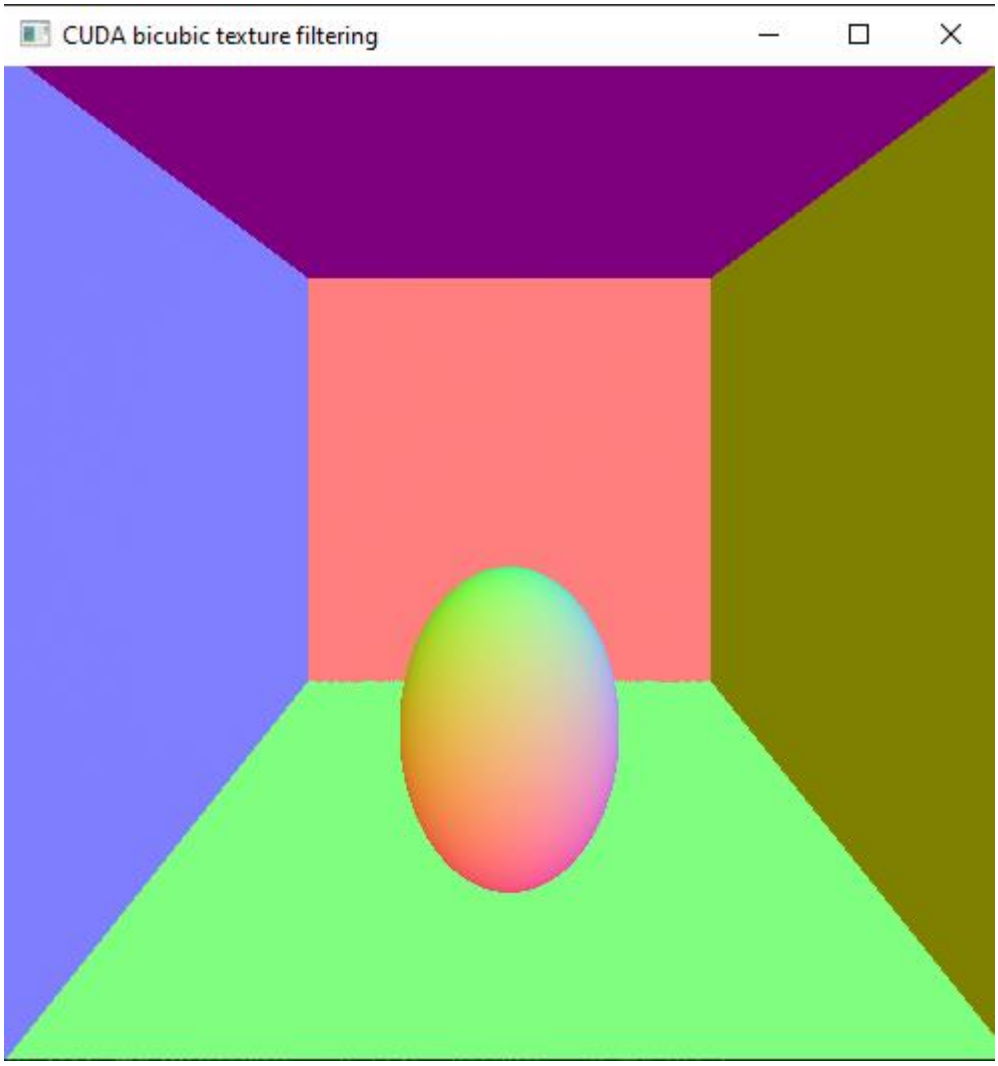
```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //Create the walls
        *(d_list + 0) = new sphere(vec3(-10002.0, 0, -3), 10000);
        *(d_list + 1) = new sphere(vec3(10002.0, 0, -3), 10000);
        *(d_list + 2) = new sphere(vec3(0, -10002.0, -3), 10000);
        *(d_list + 3) = new sphere(vec3(0, 10002.0, -3), 10000);
        *(d_list + 4) = new sphere(vec3(0, 0, -10001.0), 10000);
        //Create the balls
        *(d_list + 5) = new sphere(vec3(0, 0, 1), 0.2);


        *d_world = new hitable_list(d_list, 6);
    }
}
```

Test data:

N/A

Sample output:



Reflection:

none

Metadata:

Further information:

Exercise 2. Free motion animation

Question1: implement code to allow the particle to rotate about the centre of the box

Solution:

Added a device wide global variable named tick

```
__device__ static int ticks = 1;
```

Changed the sphere to be drawn to the following

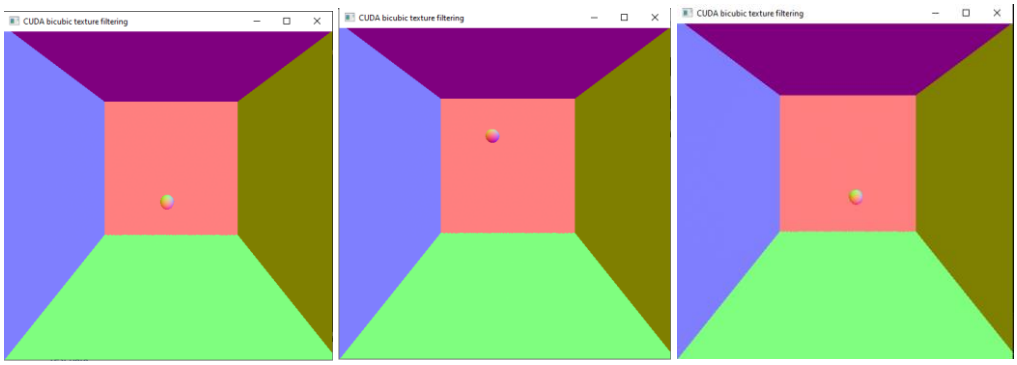
```
//Create the balls
*(d_list + 5) = new sphere(vec3(cos(0.01 * (float)ticks++), sin(0.01 * (float)ticks++), -1), 0.2);
```

This moves the ball every time the image is rendered due to the tick variable being incremented

Test data:

n/a

Sample output:



Reflection:

This was fairly perfunctory and following the lab made sense

Metadata:

Exercise 3. Ball-box walls collision animation

Questions:

1. Modify the previous code to give the sphere a velocity and bounce off of the walls of the box
2. Implement a code change that will make the ball change colour after a collision.
3. Implement code change to allow multiple balls to move at the same time;

Solution:

1. I first defined some global variables for the sphere to track its movement

```
__device__ static int ticks = 1;
__device__ static double x_position = 0.0;
__device__ static double y_position = 0.0;
__device__ static double x_velocity = 0.02;
__device__ static double y_velocity = 0.03;
```

I then modified the create world kernel to move the ball and check for collisions with the walls

```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //define vectors
        vec3 left   = vec3(-10002.0,        0,        0);
        vec3 right  = vec3( 10002.0,        0,        0);
        vec3 bottom = vec3(       0, -10002.0,        0);
        vec3 top    = vec3(       0,  10002.0,        0);
        vec3 back   = vec3(       0,        0, -10001.0);
        int wall_size = 10000;
        //Create the walls
        *(d_list + 0) = new sphere(left, wall_size);
        *(d_list + 1) = new sphere(right, wall_size);
        *(d_list + 2) = new sphere(bottom, wall_size);
        *(d_list + 3) = new sphere(top, wall_size);
        *(d_list + 4) = new sphere(back, wall_size);
        //Create the balls
        *(d_list + 5) = new sphere(vec3( x_position += x_velocity, y_position += y_velocity, 0), 0.2);
        if (x_position - 0.2 <= left.x() + wall_size || x_position + 0.2 >= right.x() - wall_size)
        {
            x_velocity *= -1;
        }
        if (y_position - 0.2 <= bottom.y() + wall_size || y_position + 0.2 >= top.y() - wall_size)
        {
            y_velocity *= -1;
        }

        //move balls
        *d_world = new hitable_list(d_list, 6);
    }
}
```

The collision is calculated by working out if the distance between the spheres position adjusted for the radius is intersecting with the wall position adjusted for the wall radius.

2. In order to achieve this:

I modified the Sphere class so that it contained a vec3 named colour to store its rgb values and modified the hit class so that hit_record had a colour variable which would be set when the ray hits the sphere as shown below

```
class sphere : public hitable {
public:
    __device__ sphere() {}
    __device__ sphere(vec3 cen, vec3 col, vec3 vec, float r) : center(cen), radius(r), colour(col), vector(vec) {};
    __device__ virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    vec3 center;
    vec3 colour;
    vec3 vector;
    float radius;
};

__device__ bool sphere::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - a * c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            rec.colour = colour;
            return true;
        }
        temp = (-b + sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            rec.colour = colour;
            return true;
        }
    }
    return false;
}
```

Modified the cast ray function so that when setting the colour, it applies the colour from the sphere that was hit to the shader.

```
__device__ vec3 castRay(const ray& r, hitable** world) {
    hit_record rec;

    if ((*world)->hit(r, 0.0, FLT_MAX, rec)) {
        vec3 result = 0.5f * vec3(rec.normal.x() + 1.0f , rec.normal.y() + 1.0f, rec.normal.z() + 1.0f);
        return result * (rec.colour/255);
    }
    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);
    }
}
```

The results of this can be seen in sample output section 2

3. To achieve multiple spheres at the same time I modified the static device variable to store the following values

```
__device__ static int colour_index = 1;
__device__ static vec3 sphere_centres[3];
__device__ static vec3 sphere_velocitys[3];
__device__ static vec3 sphere_colours[3];
```

Then in the create world kernel if they are null then they are initialised and used to store the persistent sphere values the create world kernel now looks as follows

```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0)
    {
        //define wall vec3s
        vec3 lef = vec3(-10002.0,        0,        0);
        vec3 rig = vec3( 10002.0,        0,        0);
        vec3 bot = vec3(       0, -10002.0,        0);
        vec3 top = vec3(       0,  10002.0,        0);
        vec3 bac = vec3(       0,        0, -10001.0);

        //define the colours
        vec3 blank = vec3(255, 255, 255);
        int number_of_colours = 6;
        vec3 colour_list[6];
        colour_list[0] = vec3(066, 245, 242);
        colour_list[1] = vec3(245, 066, 224);
        colour_list[2] = vec3(245, 242, 066);
        colour_list[3] = vec3(000, 255, 000);
        colour_list[4] = vec3(000, 000, 255);
        colour_list[5] = vec3(255, 000, 000);
```

```
    //initialise sphers statics
    if (sphere_velocitys[0].x() == 0 && sphere_velocitys[0].y() == 0 && sphere_velocitys[0].z() == 0) { sphere_velocitys[0] = vec3(0.01, 0.05, 0); sphere_colours[0] = vec3(150, 150, 150); }
    if (sphere_velocitys[1].x() == 0 && sphere_velocitys[1].y() == 0 && sphere_velocitys[1].z() == 0) { sphere_velocitys[1] = vec3(-0.03, -0.01, 0); sphere_colours[1] = vec3(150, 150, 150); }
    if (sphere_velocitys[2].x() == 0 && sphere_velocitys[2].y() == 0 && sphere_velocitys[2].z() == 0) { sphere_velocitys[2] = vec3(0.05, -0.02, 0); sphere_colours[0] = vec3(150, 150, 150);
    }

    int wall_size = 10000;
    //Create the walls
    *(d_list + 0) = new sphere(lef, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 1) = new sphere(rig, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 2) = new sphere(bot, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 3) = new sphere(top, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 4) = new sphere(bac, blank, vec3(0, 0, 0), wall_size);
```

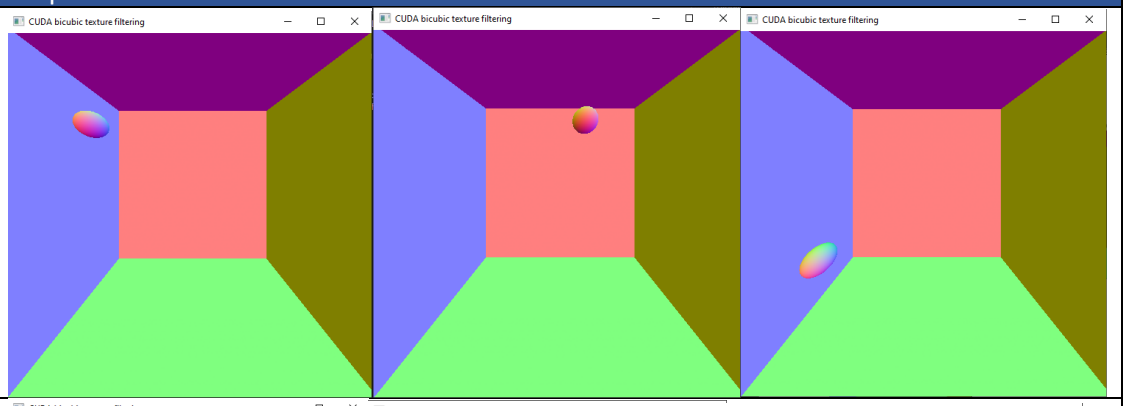
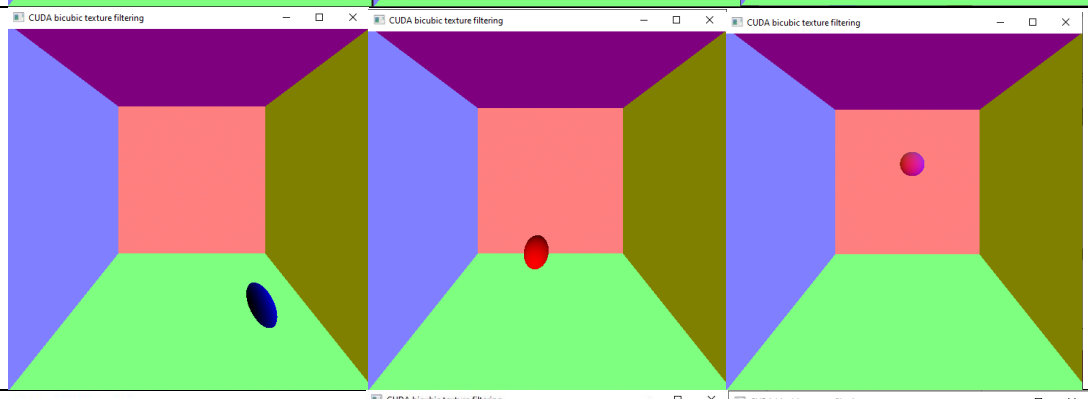
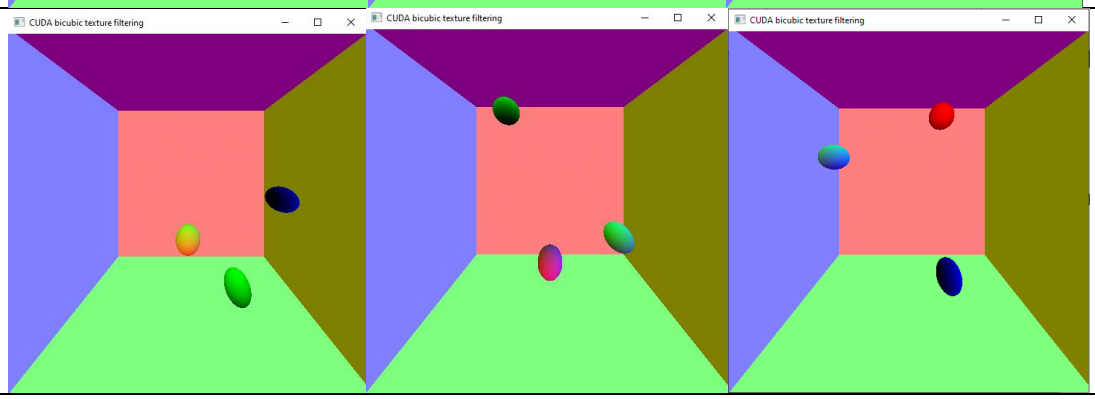
```
        //Modify balls
        for (int i = 0; i < 3; i++)
        {
            //move balls
            sphere_centres[i] += sphere_velocitys[i];
            //Test Collisions
            if (sphere_centres[i].x() - 0.2 <= lef.x() + wall_size || sphere_centres[i].x() + 0.2 >= rig.x() - wall_size)
            {
                sphere_colours[i] = colour_list[colour_index];
                colour_index++;
                if (colour_index > 5)
                {
                    colour_index = 0;
                }
                sphere_velocitys[i] = vec3(sphere_velocitys[i].x() * -1, sphere_velocitys[i].y(), sphere_velocitys[i].z());
            }
            if (sphere_centres[i].y() - 0.2 <= bot.y() + wall_size || sphere_centres[i].y() + 0.2 >= top.y() - wall_size)
            {
                sphere_colours[i] = colour_list[colour_index];
                colour_index++;
                if (colour_index > 5)
                {
                    colour_index = 0;
                }
                sphere_velocitys[i] = vec3(sphere_velocitys[i].x(), sphere_velocitys[i].y() * -1, sphere_velocitys[i].z());
            }
            *(d_list + i + 5) = new sphere(sphere_centres[i], sphere_colours[i], sphere_velocitys[i], 0.2);
        }
        *d_world = new hitable_list(d_list, 8);
    }
}
```

The above code will create 3 moving balls within the walls and the results can be seen in sample output ref 3.

Test data:

N/A

Sample output:

Ref	output
1	
2	
3	

Reflection:

Getting the shading to apply to the coloured balls lead to some difficulty as I had not considered applying the colour to the normal value to achieve the shading results.

Metadata: