Faculty of Engineering
Universiti Teknologi Malaysia
81310 Skudai, Johor

"ASSIGNMENT 2 PROGRESS 2"

Lecturer:

Dr. Norhaida Mohd Suaib

Prepared by :

Diah Ayu Irawati | A17CS0248

Alfonsus Wisnu Echa Dewangga | X19EC3002

Masturina Hafizah Binti Ahmad Jailani | B17CS0012

Muhammad Farin Aieman Bin Rosman | A16CS0101

Rakesh A/L Ramesh | A17CS0299

March 2020

1. Output image

The first step is to setup the environment, the image produces pixels in a row. The code the image will be captured, and the output file will be in .ppm format.

```
#include <iostream>

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            float r = float(i) / float(nx);
            float g = float(j) / float(ny);
            float b = 0.2;
            int ir = int(255.99*r);
            int ig = int(255.99*g);
            int ib = int(255.99*b);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
   }
```

The code above consists of a pixel that is written in looping of row, from left right and top to bottom. The color will be defined by the convention of RGB color.

Code explanation
In the range of x and y of image size. Start from top left to the bottom right. Which shown color from 0 to 255.

2. The vec3 Class

The vec3 class stores and operates on a 3D vector = (x,y,z). It is used for storing geometric vectors and colors. A position, a movement direction, gravity, the points in some models geometry, particles or sometimes even rotations will depend on Vector3. It is vital that vec3 class is implemented as fast and lightweight as possible. Moreover, in our project vec3 class is used for colors, locations, directions and offsets. Consequently, vec3 is for (x,y,z),(r,g,b) and

operators of vec3 are 1. vec3+vec3, vec3- vec3 2. scalar * vec3 3. dot, cross 4. length, unit_vector.

Top part of the vec3 class:

```
#include <iostream>
#include <math.h>
#include <stdlib.h>

class vec3 {
public:
    vec3() {}
     vec3(float e0, float e1, float e2) { e[0] = e0; e[1] = e1; e[2] = e2;
}
    inline float x() const { return e[0]; }
    inline float y() const { return e[1]; }
    inline float z() const { return e[2]; }
    inline float r() const { return e[0]; }
    inline float g() const { return e[1]; }
    inline float b() const { return e[2]; }

    inline const vec3& operator+() const { return *this; }
    inline vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    inline float operator[](int i) const { return e[i]; }
    inline float& operator[](int i) { return e[i]; }

    inline vec3& operator+=(const vec3 &v2);
    inline vec3& operator-=(const vec3 &v2);
    inline vec3& operator*=(const vec3 &v2);
    inline vec3& operator/=(const vec3 &v2);
    inline vec3& operator*=(const float t);
    inline vec3& operator/=(const float t);

      inline float length() const { return sqrt(e[0]*e[0] + e[1]*e[1] +
e[2]*e[2]); }
     inline float squared_length() const { return e[0]*e[0] + e[1]*e[1] +
e[2]*e[2]; }
    inline void make_unit_vector();

    float e[3];
   };
```

Code explanation:

The code above allows us to reuse the vec3 type for all vectors and colors, and defines every operator one might need. For ray tracing, this is convenient,

since it keeps the code compact and simple. However, it also leads to code that is more error-prone, especially for larger projects. In the case of a Euclidean vector, a bound vector has a fixed starting and end point. It represents a fixed point in space. If we fix the starting point at the origin, this bound vector can represent the Cartesian coordinate of a point of relative to the origin. Furthermore, a free vector has no initial point, it only has direction and magnitude. Often, however, we need a way to represent *only* direction. A unit vector does exactly this - it is a free vector with its magnitude normalized to 1.0. Below part of the vec3 class:

```
inline std::istream& operator>>(std::istream &is, vec3 &t) {
    is >> t.e[0] >> t.e[1] >> t.e[2];
    return is;
}

inline std::ostream& operator<<(std::ostream &os, const vec3 &t) {
    os << t.e[0] << " " << t.e[1] << " " << t.e[2];
    return os;
}

inline void vec3::make_unit_vector() {
    float k = 1.0 / sqrt(e[0]*e[0] + e[1]*e[1] + e[2]*e[2]);
    e[0] *= k; e[1] *= k; e[2] *= k;
}

inline vec3 operator+(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] + v2.e[0], v1.e[1] + v2.e[1], v1.e[2] + v2.e[2]);
}

inline vec3 operator-(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] - v2.e[0], v1.e[1] - v2.e[1], v1.e[2] - v2.e[2]);
}

inline vec3 operator*(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] * v2.e[0], v1.e[1] * v2.e[1], v1.e[2] * v2.e[2]);
}

inline vec3 operator*(float t, const vec3 &v) {
    return vec3(t*v.e[0], t*v.e[1], t*v.e[2]);
}

inline vec3 operator*(const vec3 &v, float t) {
    return vec3(t*v.e[0], t*v.e[1], t*v.e[2]);
}

inline vec3 operator/(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] / v2.e[0], v1.e[1] / v2.e[1], v1.e[2] / v2.e[2]);
}

inline vec3 operator/(vec3 v, float t) {
    return vec3(v.e[0]/t, v.e[1]/t, v.e[2]/t);
}

inline float dot(const vec3 &v1, const vec3 &v2) {
```

```
    return v1.e[0]*v2.e[0]
         + v1.e[1]*v2.e[1]
         + v1.e[2]*v2.e[2];
}

inline vec3 cross(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[1] * v2.e[2] - v1.e[2] * v2.e[1],
                v1.e[2] * v2.e[0] - v1.e[0] * v2.e[2],
                v1.e[0] * v2.e[1] - v1.e[1] * v2.e[0]);
}

inline vec3& vec3::operator+=(const vec3 &v) {
    e[0] += v.e[0];
    e[1] += v.e[1];
    e[2] += v.e[2];
    return *this;
}

inline vec3& vec3::operator-=(const vec3& v) {
    e[0] -= v.e[0];
    e[1] -= v.e[1];
    e[2] -= v.e[2];
    return *this;
}

inline vec3& vec3::operator*=(const vec3 &v) {
    e[0] *= v.e[0];
    e[1] *= v.e[1];
    e[2] *= v.e[2];
    return *this;
}

inline vec3& vec3::operator*=(const float t) {
    e[0] *= t;
    e[1] *= t;
    e[2] *= t;
    return *this;
}

inline vec3& vec3::operator/=(const vec3 &v) {
    e[0] /= v.e[0];
    e[1] /= v.e[1];
    e[2] /= v.e[2];
    return *this;
}

inline vec3& vec3::operator/=(const float t) {
    float k = 1.0/t;

    e[0] *= k;
    e[1] *= k;
    e[2] *= k;
    return *this;
}

inline vec3 unit_vector(vec3 v) {
    return v / v.length();
    }
```

Code explanation:

Const-overloading is used to create an inspector and mutator for x,y,z. Composition is used rather than inheritance to prevent users from mutating any field in order to preserve its guarantees. Moreover, into unit vectors, they are not completely euclidean vectors, instead they are an abstraction over free vectors that guarantee a length of 1.0. Vec3 only defines inspector methods for x(), y(), and z(). This ensures that the fields cannot be changed, preserving our guarantees while keeping the API consistent.

3. Rays, a Simple Camera, and Background

The function used to compute the ray function is the p(t) = A + t*B, where p is the 3D position along a line in 3D, 'A' is the ray origin and 'B' is the ray direction. By applying this function, the point at parameter can be calculated.

```
#ifndef RAYH
#define RAYH
#include "vec3.h"

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

        vec3 A;
        vec3 B;
};

   #endif
```

Based on the coding above, vec3 origin ( ) and vec3 direction ( ) are the function respectively to get the origin and direction value while vec3 point_at_parameter (float t) is called to calculate the point at parameter value.

The next process is to calculate the ray tracer. The function of ray tracer is to send rays through pixels and calculate what color that can be seen in the direction of those rays. This form of calculation can be calculated by calculating the ray that goes from eye to a pixel, ray intersection and the colour of the intersection point. After initializing those coordinates, the pixel colour that is printed on the screen is calculated by calculating the point that intersects between the screen and ray that is directed from the origin coordinate.

```
#include "ray.h"

#include <iostream>

vec3 color(const ray& r) {
    vec3 unit_direction = unit_vector(r.direction());
    float t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
}

int main() {
    int nx = 200;
    int ny = 100;
        std::cout << "P3\n" << nx << " " << ny << "\n255\n";
   vec3 lower_left_corner(-2.0, -1.0, -1.0);
    vec3 horizontal(4.0, 0.0, 0.0);
    vec3 vertical(0.0, 2.0, 0.0);
        vec3 origin(0.0, 0.0, 0.0);
   for (int j = ny-1; j >= 0; j--) {
            for (int i = 0; i < nx; i++) {
           float u = float(i) / float(nx);
            float v = float(j) / float(ny);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical);
                vec3 col = color(r);
           int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);

            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
    }
```

Based on the coding above, vec3 lower_left_corner( ), horizontal( ), vertical( ) and origin( ) respectively are called to insert the coordinate of the lower left corner or the size of the screen, horizontal, vertical and origin. Inside the multi-dimensional loop, the pixel colour is printed by

calculating the point that intersects between the screen and ray that is directed from the origin coordinate. Then, the colour of the pixel is added by using the color( ) function. The colour bended in color( ) function is between white and blue. If the ‘r’ value is 0, the pixel printed with white colour but if the ‘r’ value is 1, the pixel printed with colour red. The Figure 1 shows the initial result of the image displayed.

Figure 1 : Initial result of the screen

Initially, the colour white and blue is blended between up and down. By changing the unit direction in color( ) function from y to x, the colour white and blue is blended between left and right.

Figure 2 : Unit direction in side color( ) function changed from y to x

4. Adding a Sphere

The sphere function is called to add the sphere object inside the scene. In order to display the sphere on the screen, the intersection between the sphere pixels and the ray directed from the origin is calculated. If the ray does not intersect with the sphere with 0 root, the screen will not print the sphere pixel. If the ray intersects with the sphere with 1 root, the screen will print the sphere and the pixel color displayed on the screen based on the sphere's pixel color that intersects with ray. If the ray intersects with the sphere with 2 roots, the screen will print the sphere and the pixel color displayed on the screen based on the latest sphere's pixel color that intersects with ray. The other intersection on the same ray will be ignored.

```
bool hit_sphere(const vec3& center, float radius, const ray& r) {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0 * dot(oc, r.direction());
    float c = dot(oc, oc) - radius*radius;
    float discriminant = b*b - 4*a*c;
    return (discriminant > 0);
}
```

```
vec3 color(const ray& r) {
    if (hit_sphere(vec3(0,0,-1), 0.5, r))
            return vec3(1, 0, 0);
   vec3 unit_direction = unit_vector(r.direction());
    float t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
```

Based on the coding above, the intersection between the sphere pixels and the ray directed from the origin is added in color( ) function. The initial result of the image as shown in Figure 3 below.

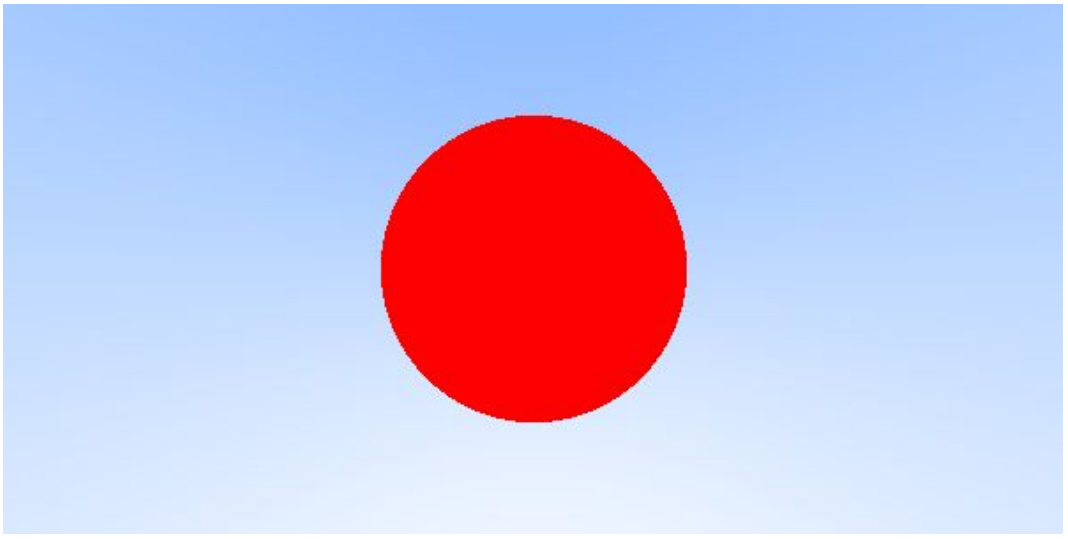


Figure 3 : Initial result of image

The next result test was done by changing the color of sphere pixels from red to yellow as shown in Figure 4.

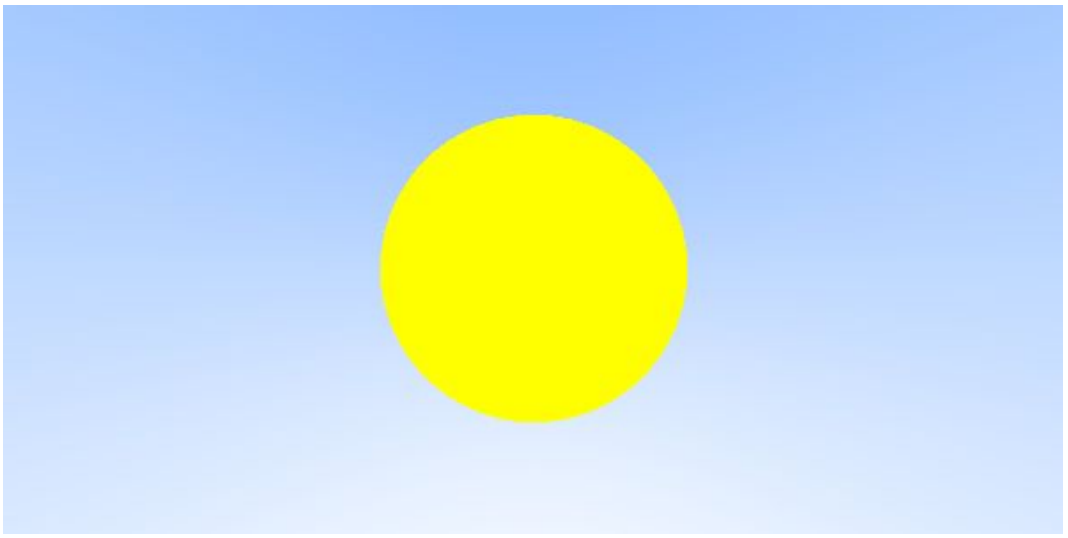


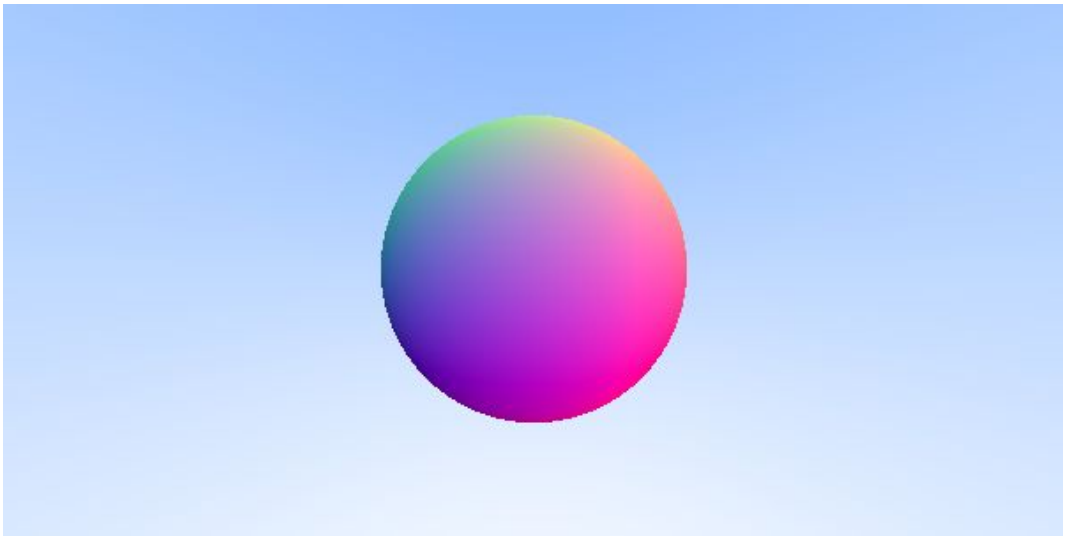
Figure 4 : Changing the sphere colour from red to yellow

6. Surface Normals and Multiple Objects

To get shade, we need a surface normal. A common trick used for visualizing normals (because it's easy and somewhat intuitive to assume N is a unit length vector — so each component is between −1 and 1) is to map each component to the interval from 0 to 1, and then map x/y/z to r/g/b. For the normal, we need the hit point, not just whether we hit or not. Let's assume the closest hit point (smallest t).

This is the result:

Second result after changing some color parameters:

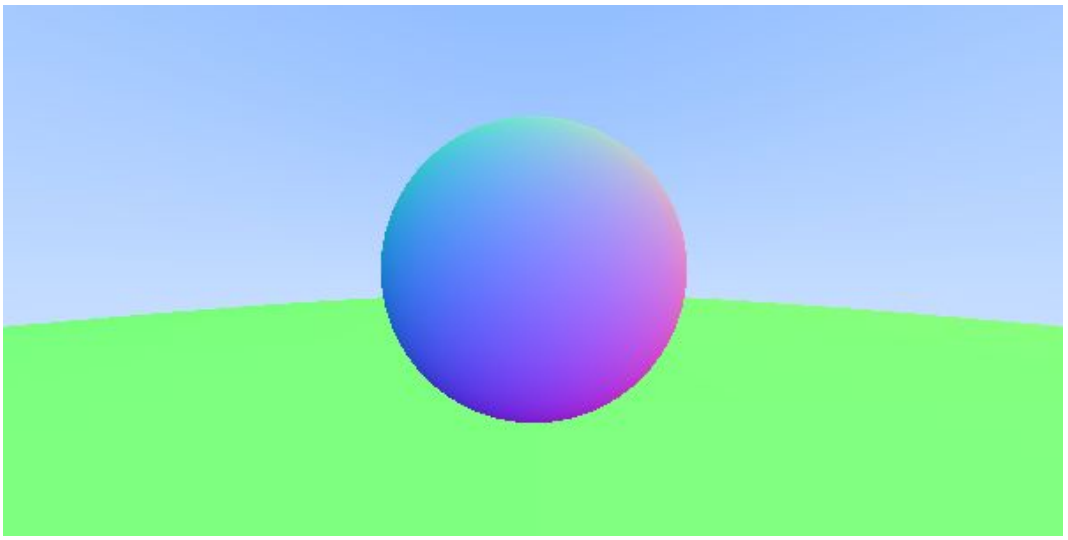


To make multiple sphere, we use a “hittable” abstract class.

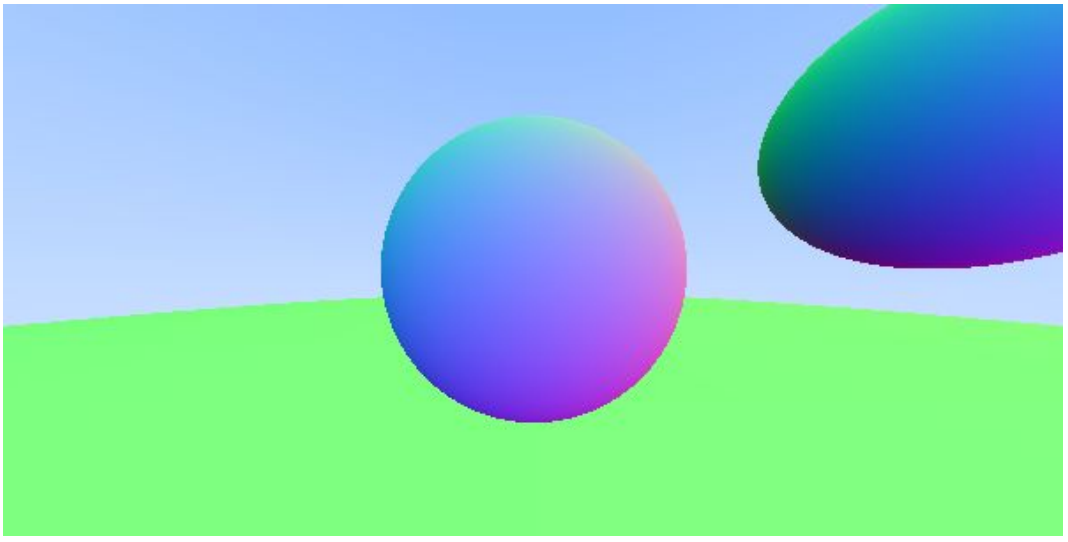
This hittable abstract class will have a hit function that takes in a ray. Most ray tracers have found it convenient to add a valid interval for hits tmin to tmax, so the hit only “counts” if tmin<t<tmax. For the initial rays this is positive t.

“hittable” class:

Here’s the result:



Adding more spheres:



7. Antialiasing