2020 CG HW3 Report

一、 環境

完全遵照 HW0 所介紹的環境進行搭建。
OpenGL
Visual Studio 編寫 C++
32 bit freeglut

二、 作業內容（Ray Tracing 光線追蹤）

1. 渲染一顆擴散光表面的球
2. 渲染一顆折射光表面的球
3. 渲染一顆反射光表面的球
4. 渲染所有球的陰影
5. 將結果輸出至 ppm 檔案與一般圖檔格式（bmp/jpg/png）

三、 實作

1. 設定攝影機位置與畫面屬性（左下角座標與寬高）

```
// camera and projection plane
vec3 lower_left_corner(-4, -2.25, -1);
vec3 origin(0, 0, 0);
vec3 horizontal(8, 0, 0);
vec3 vertical(0, 4.5, 0);
```

2. 建造儲存所有球的 vector
   一顆地板球，一顆 diffuse 大球，一顆折射光大球，一顆反射光大球，48 顆隨機折射反射值的小球。（隨機部分過長，因此未提供原始碼）

```
vector<sphere> hitable_list;
hitable_list.push_back(sphere(vec3(0, -100.5, -2), 100)); //ground default color (1.0, 1.0, 1.0)
hitable_list.push_back(sphere(vec3(0, 0, -2), 0.5, vec3(1.0f, 1.0f, 1.0f), 0.0f, 0.9f)); // refracted ball
hitable_list.push_back(sphere(vec3(1, 0, -1.75), 0.5, vec3(1.0f, 1.0f, 1.0f), 0.9f, 0.0f)); // reflected ball
hitable_list.push_back(sphere(vec3(-1, 0, -2.25), 0.5, vec3(1.0f, 0.7f, 0.3f), 0.0f, 0.0f)); // diffuse ball
```

Sphere class 中儲存中心座標 center、球體半徑 radius、擴散光 kd 值（代表顏色）、折射率 w_t、反射率 w_r。

```
class sphere : public hitable {
public:
    sphere() {}
    sphere(vec3 c, float r, vec3 _kd = vec3(1.0, 1.0, 1.0), float w_ri = 0.0f, float w_ti = 0.0f):
        center(c), radius(r), kd(_kd), w_r(w_ri), w_t(w_ti) {};
    virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    vec3 center;
    float radius;
    vec3 kd;
    float w_r; //reflected
    float w_t; //transmitted
};
```

3. 計算每個 pixel 點的顏色

使用迴圈將最終輸出畫面的所有 pixel 都跑過一次，計算該 pixel 的座標值。利用攝影機位置與該 pixel 的座標作為參數創建一道 ray，再將此 ray 傳入 trace function，以得到該點的最終顏色。

```
for (int j = height - 1; j >= 0; j--) {
    for (int i = 0; i < width; i++) {
```

```
float u = float(i) / float(width);
float v = float(j) / float(height);

ray r(origin, lower_left_corner + u * horizontal + v * vertical);
vec3 color = trace(r, hitable_list, 0);
```

4. 儲存畫面

於前一步驟中的迴圈中，加入可以儲存所有 pixel 顏色的 array。三個用途：一個儲存到 data array 中供 glut 輸出視窗用，一個直接輸出到 ppm 檔案，一個丟到 pixels array 中供 png output 的 function 使用。

在這三者中，只有 OpenGL 的輸出是從左下角往右上角，其餘兩者（ppm 與一般圖檔）都是從左上角開始。

```
// for display window
int index = ((j) * width + i) * 3;
data[index + 0] = (GLbyte)(color.r() * 255);
data[index + 1] = (GLbyte)(color.g() * 255);
data[index + 2] = (GLbyte)(color.b() * 255);
```

5. 輸出 ppm 檔

```
fstream file;
file.open("../output.ppm", ios::out);
file << "P3\n" << width << " " << height << "\n255\n";
```

在迴圈中邊計算邊儲存

```
file << int(color.r() * 255) << " " << int(color.g() * 255) << " " << int(color.b() * 255) << "\n";
```

6. 輸出 png 檔

   將迴圈中儲存的 data 值存起來，使用 stb_image_write 函式庫，傳入需要的參數即可輸出 png 檔案。

```
int index_for_pixels = ((height - j) * width + i) * 3;
pixels[index_for_pixels + 0] = (uint8_t)(color.r() * 255);
pixels[index_for_pixels + 1] = (uint8_t)(color.g() * 255);
pixels[index_for_pixels + 2] = (uint8_t)(color.b() * 255);
```

```
stbi_write_png("output.png", width, height, 3, pixels, width * 3);
```

   3 代表 Channel 數，RGB 三色。

7. 顯示與設定視窗

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GL_DOUBLE);
glutInitWindowSize(width, height);
glutCreateWindow("Image Loading Test");
```

8. 繪製模型

```
glutReshapeFunc(ChangeSize);
glutDisplayFunc(RenderScene);
init(data);
glutMainLoop();
finish();
```

   重點說明 Ray Tracing 實作部分，請見第四大點

四、 繪製模型（Ray Tracing）

使用反向追蹤，掃描每一個像素點，以決定該像素點應該顯示什麼顏色。其中最關鍵的就是只顯示離相機最近的物體，因此需要知道該像素方向的 ray 究竟碰到了哪個物體為最近物體。

1. Trace 需要 ray、object list、depth 等參數

```
vec3 trace(const ray&r, const vector<sphere> &list, int depth) {
```

(1.) Ray 表示需要追蹤的一束光（實際上在反向追蹤中，Ray 應被稱為視線而非光線）Ray class 中只有儲存兩個屬性值，一為原點 O，二為射線方向 D。並提供一個函式，用以計算 O + tD，此為終點作標。

```
class ray {
public:
    ray() {}
    ray(const vec3& a, const vec3& b) { O = a; D = b; }
    vec3 origin() const { return O; }
    vec3 direction() const { return D; }
    inline vec3 point_at_parameter(float t) const {
        /*
        To-do:
            compute the position at t
        */

        vec3 pos = origin() + t * direction();

        return pos;
    }

    vec3 O; //center(origin) point
    vec3 D; //direction vector
};
```

(2.) Object list 中記錄著空間中的所有物體，用來知道 Ray 究竟穿過了哪些物體，以及哪個物體為最近物體。

(3.) Depth 用在折射與反射上，當經過一次的折射或反射，depth + 1，當 depth 過大則輸出背景圖案。

```
if (depth >= max_step) return skybox(r); // or return vec3(0,0,0);
```

2. Hit function

使用 hit function 來計算有無碰到物體，若有的話則將較近的點的屬性值存入 rec（rec 中儲存 t 值、t 值對應的終點座標、該點的 normal）。最後取得離得最近的物體得編號與 rec。

```
for (int i = 0; i < list.size(); i++) {
    if (list[i].hit(r, 0.01, 100, rec)) {

        float t_near = rec.t;

        if (t_near < closest) {
            closest = t_near;
            intersect = i;
            closest_rec = rec;
        }
    }
}
```

$$f(p) = ||p - c|| - r = 0$$

$$f(r(t)) = ||r(t) - c|| - r = 0$$

$$||o + td - c|| = r$$

$$t^2(d \cdot d) + 2t(d \cdot (o - c)) + (o - c) \cdot (o - c) - r^2 = 0$$

$$At^2 + Bt + C = 0$$

$$B^2 - 4AC \geq 0 \quad => hit = true$$

Hit 的計算方式如下：

(1.) 使用判斷式判斷射線有沒有任何值與球心距離小於半徑。

(2.) 若有的話則用一元二次方程式公式解計算剛好等於半徑處的 t 值，並取得值較小的那個。

(3.) 若 t < tmin 或 t >= tmax，則回傳 false

(4.) 若 tmin < t < tmax，則將紀錄屬性至 rec。

```
    vec3 v = r.origin() - center;
    float DdotV = dot(r.direction(), v);
    float DdotD = dot(r.direction(), r.direction());
    float discriminant = DdotV * DdotV - DdotD * (dot(v, v) - radius * radius);

    if (discriminant >= 0) {

        float near_t = (-DdotV - sqrt(discriminant)) / DdotD;

        if (near_t < tmin || near_t >= tmax) {
            return false;
        }

        rec.t = near_t;
        rec.p = r.point_at_parameter(rec.t);
        rec.nv = unit_vector(rec.p - center);

        return true;
    }

    return false;
}
```

3. 計算顏色
   顏色分為 3 部分：diffuse、reflect、transmit（refract），以下全部分開討論。
4. Diffuse
   Diffuse 可以說是最終顏色的主要來源，因為即使是反射或折射，最終都會光線都會落在一個 Diffuse 表面（或背景）。並且 Diffuse function 其實就需要把陰影也考慮進去，因此在本次作業中，Diffuse function 其實就名為 Shading。

```
vec3 diffuse = shading(lightPosition, lightIntensity, closest_rec, kd, list);
```

   Shading 需要的參數有光源位置、光源亮度、需求點的 rec、需求點的顏色（kd）、object list。

   (1.) 首先取得需求點往光源的方向向量 shadowRay。

```
vec3 L = unit_vector(lightsource - ht.p);
vec3 N = unit_vector(ht.nv);

ray shadowRay(ht.p, L);
```

   (2.) 做與 Trace 幾乎一樣的事情，呼叫 hit 來確認有沒有碰到物體。若 shadowRay 有碰到物體，則回傳陰影顏色。若沒有則計算 Diffuse 顏色。

```
for (int i = 0; i < list.size(); i++) {
    if (list[i].hit(shadowRay, 0.01, 1000, rec)) {

        float t_near = rec.t;

        if (t_near < closest) {
            closest = t_near;
            intersect = i;
        }
```

```
vec3 original_color = kd * intensity * MAX(0, dot(N, L));
```

   (3.) 此處我使用的陰影顏色，原始色與 Diffuse 顏色相同，但需要另外計算 shadowRay 與碰到物體的該點的 normal 做夾腳計算，若夾角愈小則顏色會愈深，直至全黑。若有多顆球同時造成了該處的陰影，則挑出顏色最深的來顯示。

```
float darkness = (1 + dot(L, rec.nv)) / 2;

if (darkness < darkest) {
    darkest = darkness;
}
```

```
if (intersect == -1) {
    return original_color;
} else {
    return original_color * darkest;
}
```

5. Reflect
   計算反射線的向量，並將該向量拿去做 trace，如此重複，直到碰到 Diffuse 表面或 Depth 值過大。

```
ray reflected_ray(closest_rec.p, reflect(r.direction(), closest_rec.nv));
vec3 reflected = trace(reflected_ray, list, depth + 1);
```

   而反射線的向量計算如下：

```
vec3 reflect(const vec3 &d, const vec3 &nv) {
    /*
    To-do:
        compute the reflect direction
    */

    vec3 reflect_direction = (1 * d) - (2 * (dot(d, nv)) * nv);

    return reflect_direction;
}
```

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$$

6. Transmit（Refract）
   與反射相同，計算折射線的向量，並將該向量拿去做 trace，如此重複，直到碰到 Diffuse 表面或 Depth 值過大。

```
ray transmitted_ray(closest_rec.p, refract(r.direction(), closest_rec.nv, 2));
vec3 transmitted = trace(transmitted_ray, list, depth + 1);
```

   折射線的向量計算如下：

```
vec3 refract(const vec3& d, const vec3& nv, float object_refraction_parameter) {
    /*
    To-do:
        compute the refracted(transmitted) direction
    */

    float air_reflection_parameter = 1;
    float refraction_parameter = air_reflection_parameter / object_refraction_parameter;

    auto cos_theta = fmin(dot((-1 * d), nv), 1.0);
    auto cos_phi = sqrt(1 - refraction_parameter * refraction_parameter * (1 - cos_theta * cos_theta));

    vec3 refracted_direction = refraction_parameter * (1 * d + cos_theta * nv) - cos_phi * nv;

    return refracted_direction;
}
```

$$\sin\phi = \frac{n}{n_t}\sin\theta$$

$$\mathbf{t} = \sin\phi\frac{\mathbf{d} + \mathbf{n}\cos\theta}{\sin\theta} + \cos\phi\cdot(-\mathbf{n})$$

$$= \frac{n(\mathbf{d} + \mathbf{n}\cos\theta)}{n_t} - \mathbf{n}\cos\phi$$

$$\cos^2\phi = 1 - \sin^2\phi = 1 - (\frac{n}{n_t}\sin\theta)^2$$

$$= 1 - \frac{n^2(1 - \cos^2\theta)}{n_t^2}$$

7. 計算最終顏色

   若該視線最終沒有碰到任何物體，則回傳背景顏色。

```
} else {
    return skybox(r);
}
```

   若只有 Diffuse，計算公式如下：

   $A = \text{Diffuse}$

   若只有 Diffuse 與 Reflect，公式如下：

   $B = (w_r * \text{Reflect}) + (1 - w_r) * A$

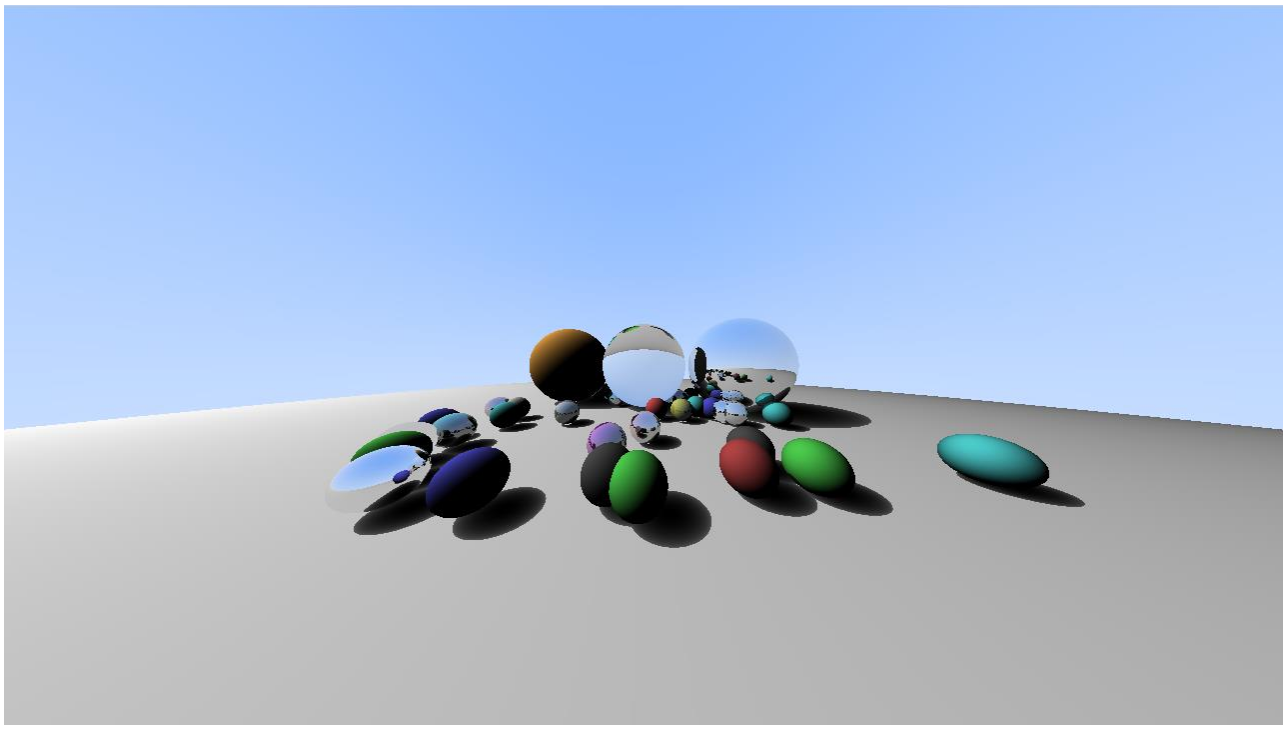
   若三項都考慮進來，公式則為：

   $C = (w_t * \text{Transmit}) + (1 - w_t) * B$

```
vec3 color = ((1 - w_t) * ((1 - w_r) * diffuse + w_r * reflected)) + (w_t * transmitted);

return color;
```

8. 最終輸出



五、 難點（疑問）與解法

1. 閱讀 code 架構
   本次作業光是要讀懂整個架構就花了不少時間，最後實在猜不出已經設定好的變數的用途，只好尋求助教幫忙。
2. Hit function 寫錯
   從網路上找到了有關 ray tracing 的文章，也照著打了我的 code，結果輸出的圖形完全不曉得發生什麼事。大約處理了半天左右才終於發現，該篇文章的一元二次方程式公式解完全打錯，不只描述公式的文字部分是錯的，連 code 本身也是錯的，不曉得該文章的作者此次作業被扣了多少分。
3. 不知道最後轉出 png 的 code 哪裡找
   儘管知道網路上會有已經建立好的 library 可以將顏色資訊輸出成圖檔，但始終找不到，最後也還是寄信問了助教才終於找到。並且該 library 名字為 stb_image，stb 甚至是作者本人的名字縮寫，因此對於需要這份工具的人來說，要找到它實在是非常痛苦。

六、 結語

本次作業基本上純粹是 C++對於演算法的實現，已經不是在熟悉 OpenGL 的過程了，因此實際的打 code 部分其實很迅速。時間花的最久的一個是剛開始的要看懂整份 code 結構，第二個就是找不到轉出圖檔的 library 了，非常幸運助教都十分好心又仁慈，不只提供了我所需要的答案，也順便解釋得更加詳細，讓整個 C++過程還算順暢。