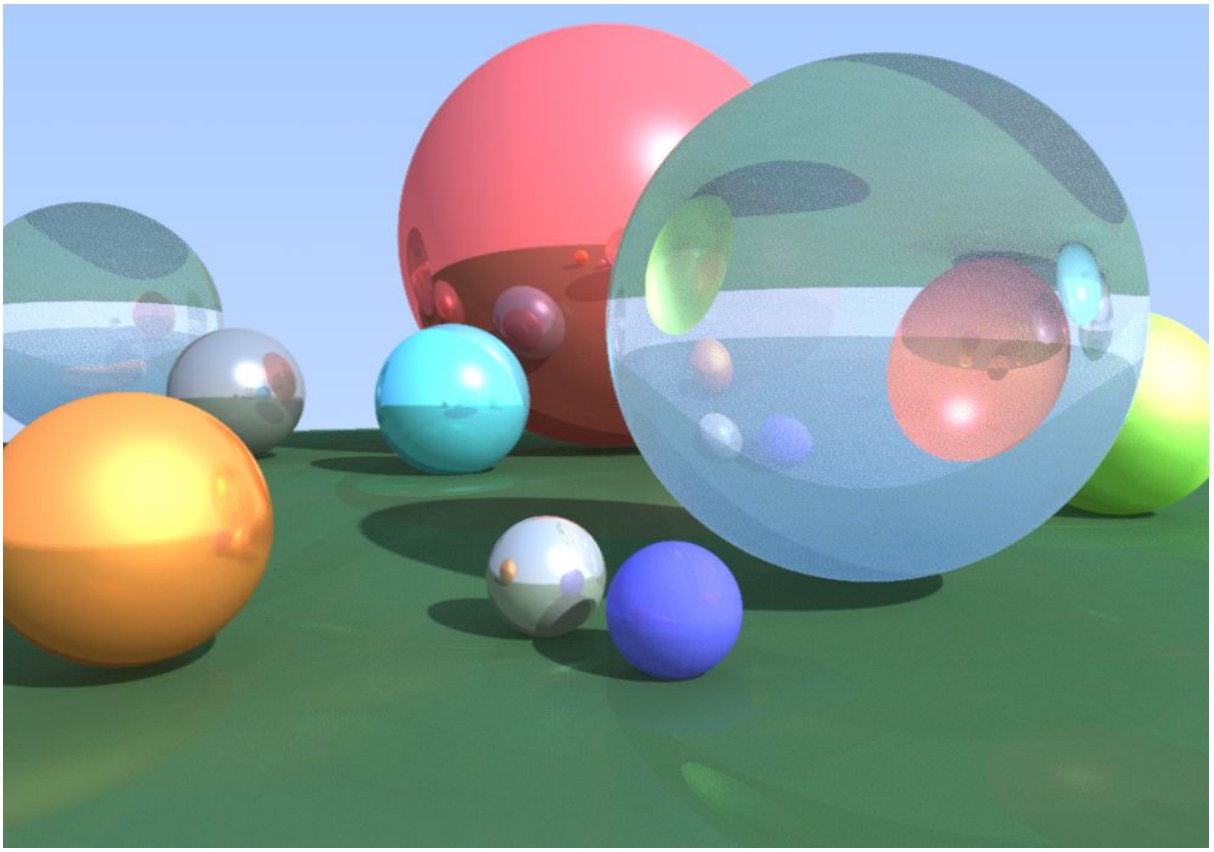
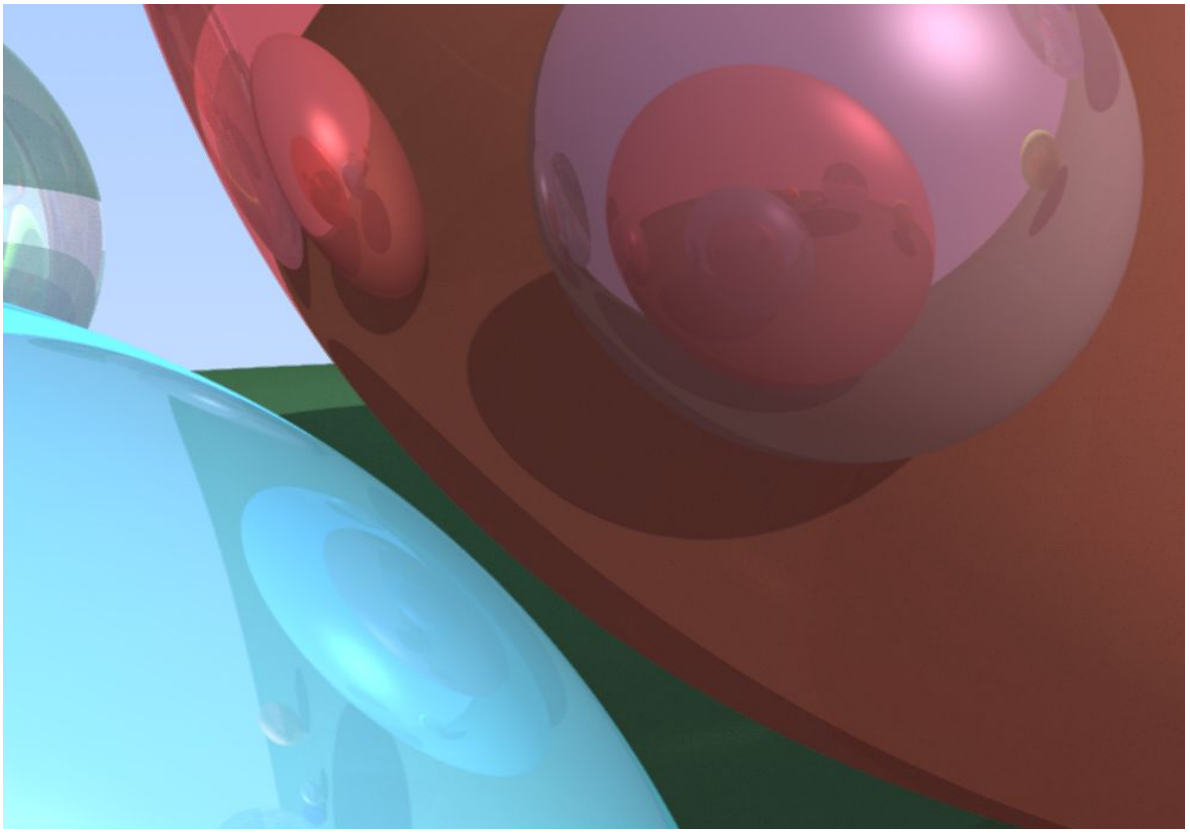
HW3 Report

鄭伯俞 0610021

成果介紹

本次做業中我使用WebGL，用shader達成Realtime Tracing，過程中最大的阻礙是設定WebGL的環境，以及用shader實做Ray tracer時比較難debug。成果已經host在github.io上，可以用支援WebGL的瀏覽器觀看。網址為：https://nemo1999.github.io/





上圖是範例成果，包含玻璃球（透明），鏡面球（紅色），霧面球（橘色），與純粹diffuse的球（紫色與地板）。由於是Real Time產生，我加上探索的機制，滑鼠拖拉可以改變視角，滑鼠滾動可以前後移動相機(Bonus)。如果想要將場景除存為png檔：對視窗按左鍵 → 選擇『另存新檔』 → 選擇存位置與檔名 → 按確定。

程式碼介紹

本專案的程式碼都放在 https://github.com/Nemo1999/WebGL-Raytracer
較特殊的地方有寫上註解，主要的檔案與功能簡述如下：

1. index.html
    a. 定義網頁架構
    b. 引用gl-matrix.js函式檔
    c. 引用index.js主程式
    d. 定義canvas元素
2. index.js
    a. 開啟webgl2背景環境
    b. 引入並編譯shader檔案（共有tracer 與 render 兩組 shader program）
    c. 用gameState物件紀錄所有場景資訊，其中包含
        i. 球的位置大小，顏色，材質
        ii. 相機的位置朝向
        iii. 光源的位置與大小
        iv. 網頁開始到目前時間，用Uniform傳入tracerProgram，作為random number generator 的seed

v. textureWeight, 由SceneData.frameCount算出，告訴tracer_frag.glsl, texture[0]紀錄的顏色要用多少比重混入當前計算的結果。

d. 用sceneData紀錄webgl的物件資訊，其中包含
   i. 除存目前場景的sceneData.frameBuffer
   ii. 兩個texture, sceneData.textures[0] , 與 sceneData.textures[1], (使用ping pong shading)來平均每次計算出的畫面
   iii. sceneData.frameCount 紀錄目前場景被渲染的次數
   iv. sceneData.mousePressed 紀錄滑鼠是否被按著

e. 每25毫秒執行一次update()函數，按照順序進行
   i. 根據相機資訊計算viewProjection Matrix
   ii. 根據viewProjection Matrix計算四個角落的ray
   iii. 講四個角落的ray方向，相機位置與gameState中的場景資訊放入tarcer_program
   iv. bind textures[0] 和 frambuffer到目前的state machine上面
   v. bind textures[1] 到 frambuffer 上
   vi. bind vertexArrayObject （只包含一個全畫面的方形mesh）
   vii. 執行tracer_program，(成果經framebuffer存入textures[1])
   viii. 講textures[0] 與 textures[1]互換，以便之後render到canvas，與提供下次tracer_program計算平均
   ix. 增加framcount，以更新下次tracer_Program採用texture做平均時的權重

f. 執行完update() 之後用render_program將textures[0]除存的目前場景畫到default framebuffer, 也就是canvas上面

g. 滑鼠移動時，觸發handeler更新相機轉向資訊，並將sceneData.frameCount歸零，重新計算場景

h. 滑鼠滾動時，觸發handeler更新相機位置資訊，並將sceneData.frameCount歸零，重新計算場景

3. tracer_vert.glsl
   將輸入的四個角落的ray方向根據目前pixel座標做平均，傳給tracer_frag.glsl

4. tracer_frag.glsl
   將varying ray的方向用亂數模糊化，以配合ping pong shading用多次計算的平均產生subpixel的取樣效果。
   用findColor進行迴圈式的ray-tracing，目前最大深度為6，過程包含
   a. 迴圈開始前，vec3 colorMask = (1.0, 1.0, 1.0), vec3 accumColor = (0.0,0.0,0.0)，目前ray方向為initialRayBlur, 起點為相機位置
   b. 迴圈開始，用intersectObjects尋找目前射線的焦點，並找出焦點物體的位置大小顏色材質等資訊，若無焦點則用background()找出背景顏色並結束函數。
   c. 將所交物體資訊放入materialBounce函數，根據物體材質，找到該點亮度，與下一個ray的起點與方向
   d. 計算shadow ray是否可以觸及光源
   e. 將colorMask乘上所交物體的顏色，將accumColor加上『colorMask乘以物體亮度* shadow』

f. 重複迴圈，結束時回傳accumColor的值

5. render_vert.glsl, render_frag.glsl
    a. vertext Shader用gl_Coord計算textureCoord並傳給fragment shader
    b. fragment Shader 採樣 texture, 並回傳所採的texture color

```
function setUniforms(gl, program, data){
    for(var name in data){
        var value =  data[name];
        var location = gl.getUniformLocation(program, name);
        if(location == null) continue;
        if(typeof(value) == 'number'){
            gl.uniform1f(location, value);
        }
        else if(value instanceof Array){
            if(value.length % 3 == 0)
                gl.uniform3fv(location, value);
            else if( value.length % 4 == 0)
                gl.uniform4fv(location, value);
            else
                gl.uniform1fv(location, value);

        }
        else if(value instanceof Float32Array){
            if(value.length == 3)
                gl.uniform3fv(location, value);
            else if(value.length == 4)
                gl.uniform4fv(location, value);
            else
            {continue;}
        }
        else{
            {continue;}
        }
    }

}
```

上圖為index.js中 setUniform函數的截圖，得益於javascript的特性，可以直接判斷gameState內物件是否是Uniform，然後再根據該物件的動態型別資訊等決定要如何將其送入shader中。

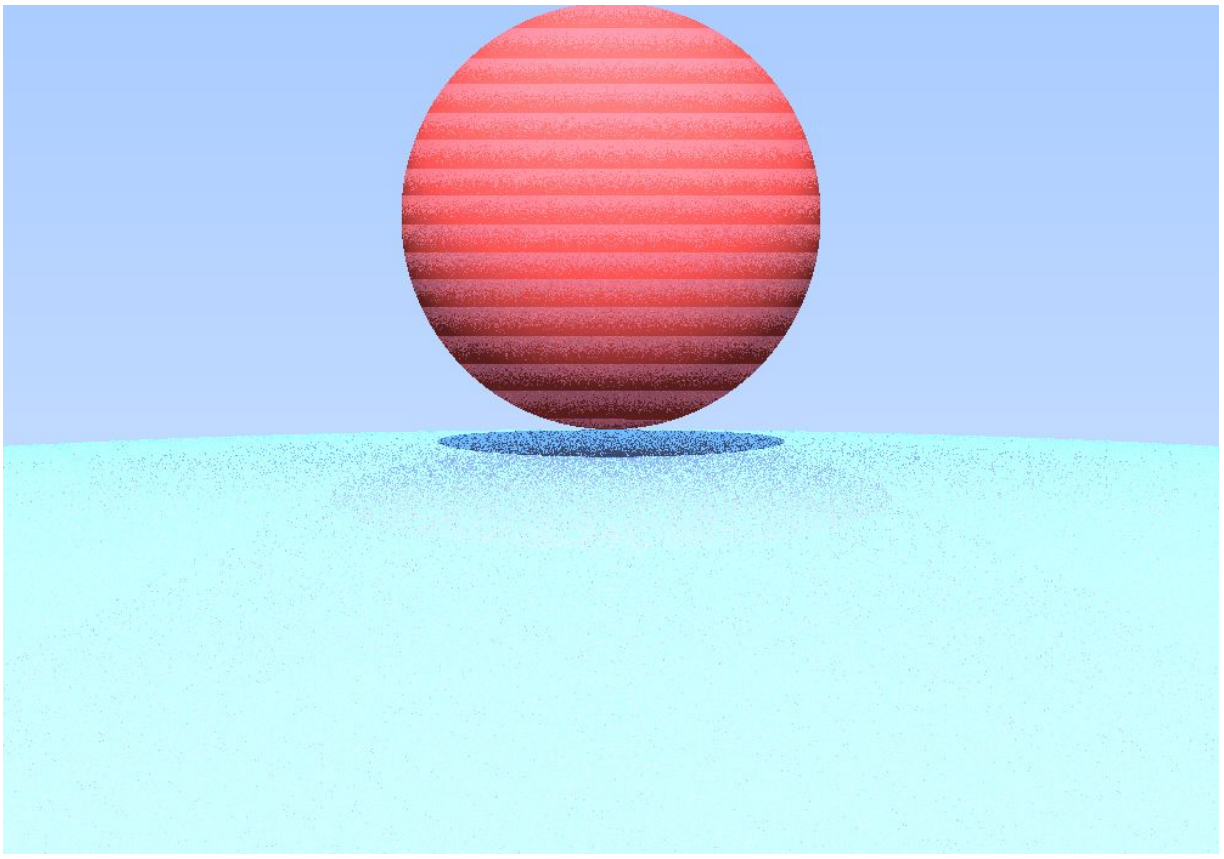
```
//find pixel color iteratively
vec3 findColor(vec3  origin,vec3 dir ){
  vec3 o = origin;
  vec3 d = dir;
  vec3 colorMask = vec3(1.0,1.0,1.0);
  vec3 accumColor = vec3(0.0,0.0,0.0);
  bool breakEarly = false;
  for(int i=0;i < maxDepth ;i++){
    vec4 hitObjCenterRadius;
    float hitObjMaterial;
    vec3 hitObjColor;
    //record the nearest hit so far
    float t = intersectObjects(o, d, hitObjCenterRadius, hitObjMaterial, hitObjColo

    float surfaceLight;
    if(t == inf){
      surfaceLight = 1.0;
      hitObjColor = findBackGround(o,d);
      breakEarly = true;
    }
    else{
      materialBounce(o, d, surfaceLight, t, hitObjMaterial, hitObjCenterRadius, i);
    }
    colorMask *= hitObjColor;
    accumColor += colorMask * surfaceLight;
    if(breakEarly) break;
  }
  return accumColor;
}
```

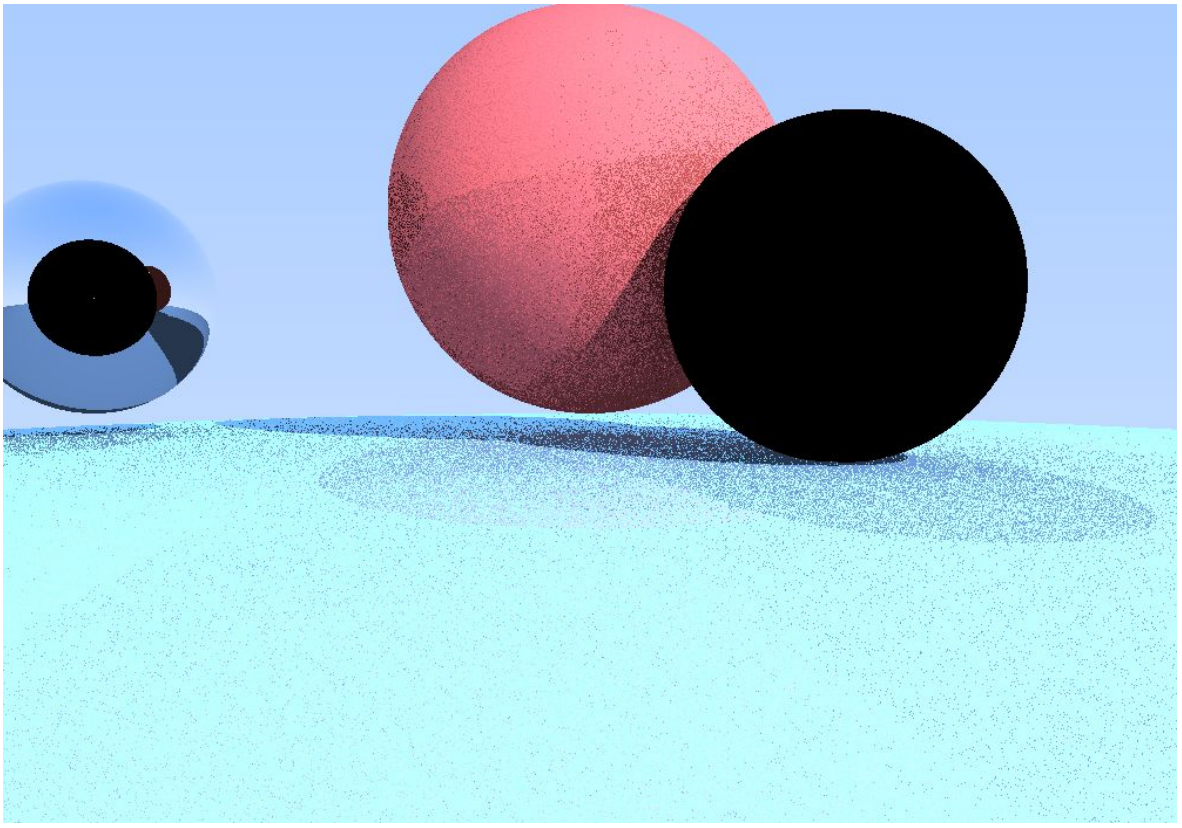
上圖為tracer_frag.glsl中findColor函數的截圖，其中使用的許多函數使用inout的功能進行宣告，已達成在函數內修改參數，起到多個回傳值的作用。例如materialBounce函數，會修改o和d，提供下一個迴圈的射線方向與起始位置。

問題與解法

以下簡述遇到的兩個問題（附圖片）



上圖所示是一開始測試球相交時的情形，當ray-trace深度大於1時會出現條文，原因我找了好久，後來發現是因為浮點數的誤差導致有時候相焦點會跑到球的內部，造成有些本來該反射的光線被困在球內部，形成暗紋，解決方法就是把起始點往球外部移一小段距離。origin = hitPoint + epsilon * surfaceNormal;
注意這個解決方式在遇到下一個問題的時候要再修改，因為surfaceNormal的方向正負號可能會是錯誤的。



上圖為實做玻璃材質的過程中遇到的情形，左球為折射律小於一的球（現實中不會出現），可以看到，反射的情形沒有問題。右球為折射率大於一的球，整顆都變成黑色，穿過球的光線不知怎麼都困住了。後來發現是surfaceNormal的方向導致的，原本的noraml都假設是由球心到球外的方向，因此當射線從球內部出來時，就會出錯。解方式就是增加判對設限是否在球內的判斷式

bool inCircle = dot(surfaceNormal , dir) > 0.0 ;

並由此找出正確的法向量

vec3 refractSurfaceNormal = (inCircle) ? - surfaceNormal : surfaceNormal;

最後在送進glsl內建的refract函數，得到新的射線方向

refract(normalize(dir), refractSurfaceNormal, refractionRatio);