コンピュータグラフィクスレンダリング課題

05242628 三田村彰大

概要　Shadertoy を用いてレイトレーシングを実装した。具体的には、球とレイの交点を計算し、Lambertian BRDF による拡散反射及びフレネル反射を用いた再帰的な鏡面反射の計算を行いシェーディングを行った。ただし反射については完全な鏡面反射のみを実装している。また、表面の粗さや metallicity といった材質の特性によってシェーディング結果が異なるような実装を行い、その結果の違いを観察した。他にも、再帰の回数を増やすと動作が若干重くなることなどを観察した。また課題全体において AI を用いたコーディングを行った。

※Shadertoy URL：　https://www.shadertoy.com/view/scc3RH

※AI との会話ログ：　https://chatgpt.com/share/6a4e0b40-c574-83e8-89a1-20b740caaff4

1　はじめに

物体をコンピュータ上に描画する手法としてレイトレーシングという手法が用いられている。これは、物体を反射してカメラに入ってくる光を逆にカメラ側から追跡することで各ピクセルにおいて描画すべき色を計算する手法である。

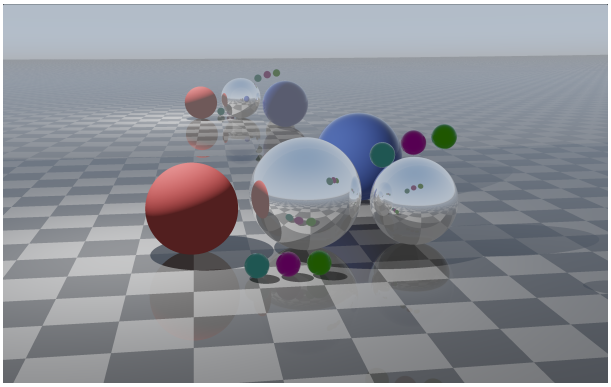

Figure1: レイトレーシングによるレンダリング。（今回の実装の結果を示している。）

今回、このレイトレーシングを授業で示された手法・計算式に則って実装する。具体的には、まずピンホールカメラを想定してカメラパラメタからレイの生成を行い、物体とレイの交差判定を計算する。ただし今回は最も単純な物体として球のみを考えることにする。その後、授業で示された Lambertian BRDF に則って物体表面を反射してカメラに入射する光の放射輝度を計算する。その際、物体の metallicity が小さいほど（すなわちマットな材質であるほど）この拡散反射が強くなるような実装を行う。

また、拡散反射以外の反射として鏡面反射の計算も行う。具体的には、レイと物体が交差した点から再帰的にレイを飛ばすことで最終的にカメラに入射する光の強さを計算する。ただし反射の強さとしてはフレネル反射を考え、Schlick の式（参考：Wikipedia）に基づいて反射率の計算を行う。また、この場合も先の拡散反射と同様物体の材質（この場合は roughness）によって反射の強さが変わるような実装を行う。ただし、反射による再帰的なレイの追跡は有限回で打ち切ることにする。

また、これらの反射以外にも環境光由来の光を考えることにする。すなわち仮に光源との間に物体が存在しており影と判定されるような点であっても完全に黒く描画することはせず、環境光によって若干色が見えるようにする。

※AI の利用について

今回、Shadertoy を用いた実装に際して AI を用いてコーディングを行った。具体的には、まず Shadertoy で簡易的なレイトレーシングを行うサンプルコードを AI に出力させ、それを授業に沿った形に書き換えながら適宜機能を追加・削除していくような形で実装を行った。ただし用いた LLM のモデルは ChatGPT 5.5-pro であり、AI との会話ログのリンクは冒頭に示した通りである。[*1]

また今回、レポートの作文は AI を用いず行なっているが、誤字脱字のチェックや内容の査読には AI を用いている。

2　理論及び実装

2.1　カメラレイの計算

今授業に則りワールド座標系におけるカメラの位置を $\vec{C}_{\text{from}}$、カメラが写すターゲットを $\vec{C}_{\text{to}}$、上方向を表すベクトルを $\vec{C}_{\text{up}}$ とする。この時、今 i, j 番目のピクセルの描画において追跡すべきレイは、i, j 番目のピクセルのワールド座標 $\vec{C}_{\text{pix}}$ とカメラの位置 $\vec{C}_{\text{from}}$ を用いて次のように表される。

$$\text{Origin}_{\text{ray}} = \vec{C}_{\text{from}} \tag{1}$$

$$\text{Direction}_{\text{ray}} = \frac{\vec{C}_{\text{from}} - \vec{C}_{\text{pix}}}{\left\| \vec{C}_{\text{from}} - \vec{C}_{\text{pix}} \right\|} \tag{2}$$

(Figure2参照。)

[*1] chatGPT-pro が優秀すぎてあまりやることがなかった。（今回のレポートにおいてはコーディングしている時間よりもコードを読んで理解しようとしている時間の方が長かった…）また、最初のサンプルコードの出力の時点においてマイクロファセット BRDF などの複雑な実装が行われてしまっており、もはやオーバースペックといった感じであった。~~果たして就活でエンジニアの道に進んで大丈夫なのだろうか…~~

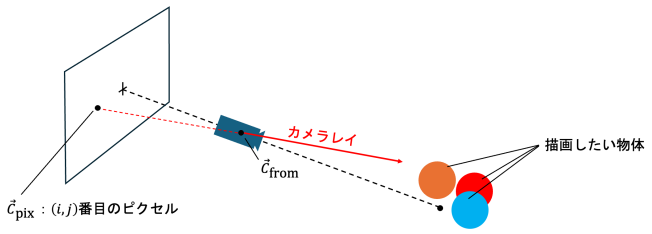

Figure2: カメラレイの向き。

よってカメラパラメタ $\vec{C}_{\text{from}}, \vec{C}_{\text{to}}, \vec{C}_{\text{up}}$ が定まっているとき、$\vec{C}_{\text{pix}}$ さえ計算できればカメラレイを計算することができる。今、ワールド座標系でのフィルム縦横の大きさ H, W とカメラまでの距離 D が定まっているとき、$\vec{C}_{\text{pix}}$ は次のように計算することができる。

$$\vec{C}_{\text{pix}} = \vec{C}_{\text{from}} + x\vec{u} + y\vec{v} + z\vec{w} \tag{3}$$

ただし

$$\vec{w} = \frac{\vec{C}_{\text{from}} - \vec{C}_{\text{to}}}{\left\|\vec{C}_{\text{from}} - \vec{C}_{\text{to}}\right\|} \tag{4}$$

及び

$$x = W * \left(1 - 2\frac{i + 0.5}{\text{resolutionWidth}}\right) \tag{5}$$

$$y = H * \left(1 - 2\frac{j + 0.5}{\text{resolutionHeight}}\right) \tag{6}$$

$$z = D \tag{7}$$

である。(Figure3参照。)

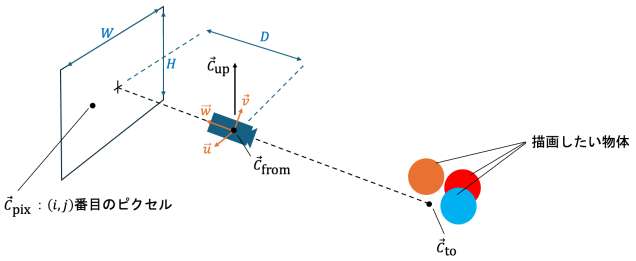

Figure3: $\vec{C}_{\text{pix}}$ の計算に用いられる量。

2.1.1 実装

まずカメラの位置 $\vec{C}_{\text{from}}$ を決定する必要がある。今回はカメラをマウスで動かせるようにすることを考え、$\vec{C}_{\text{to}}$ を中心とする球面上をカメラが動くようにする。よって Shadertoy の出力画面上の位置に応じて球面座標の天頂角・方位角を決定する。(詳しくは generateCameraRay 関数を参照。)[*2]

その上で、実装においては次のような Ray 構造体を定義する。

```
struct Ray {
    vec3 origin; // レイ原点
    vec3 direction; // レイの向き
};
```

これに対し、ピクセル座標 i, j を受け取り上でまとめた式に従って Ray を計算しているのが generateCameraRay 関数である。

[*2] その他、フィルムのカメラまでの距離は描画したい視野角をもとに計算する。またフィルムのワールド座標系での大きさは適当に定めることにする。

2.2 レイと物体の交差判定

レイが計算できたら、次はレイと物体の交差判定を行う必要がある。今求めたいのはレイが物体にぶつかる点 $\vec{p}$ と、(のちの計算で使う) $\vec{p}$ での物体表面の法線 $\vec{n}$ である。特に $\vec{p}$ については

$$\vec{p} = \text{Origin}_{\text{ray}} + t * \text{Direction}_{\text{ray}} \tag{8}$$

と表されるので、実質的に計算したいのは t の値である。

さて、今回は物体としては球のみを考えるので、t は授業で述べられたような二次方程式を解くことで計算することができる。すなわち今レイとの交差判定を行いたい球の中心を $\vec{c}$、球の半径を r とするとき、交点は

$$\begin{aligned} &\|\vec{p} - \vec{c}\|^2 = r^2 \\ \Leftrightarrow\ &\vec{d}\cdot\vec{d}t^2 + 2\vec{d}\cdot(\vec{o} - \vec{c})\,t + \left\{(\vec{o} - \vec{c})\cdot(\vec{o} - \vec{c}) - r^2\right\} = 0 \end{aligned} \tag{9}$$

を解くことによって計算することができる。(ただし $\vec{o} = \text{Origin}_{\text{ray}}$ 及び $\vec{d} = \text{Direction}_{\text{ray}}$ である。)

ただしこのとき、t が実解を持たない場合や、t が解を持ってもいずれも負になってしまうような場合はレイは(カメラより先に)交差点を持たないので、$t = \infty$ と定めることにする。また t の解が二つ以上現れた場合レイが物体と 2 点で交わることを意味するが、交差点として考えたいのは「レイと物体が最初にぶつかる点」であるからより小さい方の t を採用することにする。(Figure4参照。)

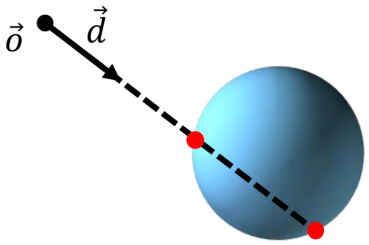

Figure4: レイが物体と 2 点以上で交わる場合。このとき、レイが物体と最初に交わる点(画像では二つの赤点のうち左のほう)を交差点 $\vec{p}$ として採用する。画像は授業スライドより抜粋。

また今回は物体としては球のみを考えているが、地表面もまた物体でありこれに対してもレイとの交差判定を行うべきである。今地表面を $y = y_0$ によって表すことにすると、これは次のように簡単に計算できる。

$$t = \frac{y_0 - \vec{o}_y}{\vec{d}_y} \tag{10}$$

また今回、描画する物体として二つ以上の球が存在しているような場合を考えたい。よって (9) (及び (10)) によって各球(と地表面)との交差判定を行うとき、一番 t が小さくなるような(すなわち一番最初にレイがぶつかるような)物体を交差物体として認識するべきである。

逆に、レイがどの物体ともぶつからないような場合(すなわち $t = \infty$ となるような場合)についても考えておかなければならない。これについては今回は系が屋外にあると考え、レイの指す高度(すなわち $\vec{d}_y$)に対応した空の色を描画することにする。[*3]

[*3] ただし空の描画(色の定め方や上空に行くにつれてのグラデーションなど)は AI の提示したものをそのまま用いている。

2.2.1 実装

物体の交差に関わる量を構造体として次のようにまとめる。

```
struct Hit {
    float t;
    vec3  p; // 交差点の座標
    vec3  n; // 交差点における物体の法線方向
    int   mat;
};
```

ただし t, p, n については先に述べたとおりである。また int mat は交差する物体の材質に関する情報を保持する量であり、例えば mat= 1 なら金属、mat= 2 なら非金属…というようにのちのシェーディングに必要となる情報を参照するための引数としての役割を持つ。

また、プログラム中で具体的に球や地表面との交差判定を行う intersectScene 関数の構造を簡略化して書くと次のようになる。

```
Hit intersectScene(ray) {
    for sphere in spheres {
        Hit h = hitSphere(ray,sphere);
        if (h.t < firstHit.t) {
            firstHit = h;
        }
    }
    Hit h = hitplane(ray,plane);
    if (h.t < firstHit.t) {
        firstHit = h;
    }
    return firstHit;
}
```

今上のコードにおいて、hitSphere 関数と hitplane 関数がそれぞれ (9) 及び (10) の球／地表面との交差計算に相当する。また、if 文によって t が最も小さくなるような交差点を選び取っている。[*4]

2.3 シェーディング

物体との交差判定が計算できたら、その交差点においてどのような色が反射されカメラに入射するのかを計算する必要がある。今回はその計算のために、拡散反射・鏡面反射・環境光による光の反射を考えることにする。

2.3.1 拡散反射

今、$\vec{\omega}_i$ 方向から位置 $\vec{p}$ に放射輝度 $L(\vec{p}, \vec{\omega}_i)$ の光が入射している状況を考える。このとき $\vec{p}$ において $\vec{\omega}_o$ 方向に反射される光の放射輝度 $L(\vec{p}, \vec{\omega}_o)$ は BRDF を $f(\vec{p}, \vec{\omega}_i, \vec{\omega}_o)$ として次のレンダリング方程式に従う。(ただし自然発光はのぞく。)

$$L(\vec{p}, \vec{\omega}_o) = \int_\Omega f(\vec{p}, \vec{\omega}_i, \vec{\omega}_o) L(\vec{p}, \vec{\omega}_i) \cos\theta d\vec{\omega}_i \tag{11}$$

ただし θ は入射方向 $\vec{\omega}_i$ と物体表面の法線 $\vec{n}$ のなす角である。

特に拡散反射 (Figure5参照) を考える場合においては、物体表面における反射率を k_d として BRDF が次の Lambertian BRDF に従うと考えることができる。

$$f(\vec{p}, \vec{\omega}_i, \vec{\omega}_o) = \frac{K_d}{\pi} \tag{12}$$

よってこの場合の反射光は

[*4] 上で示しているのはプログラム中の intersectScene 関数を簡略化して書き表したものである。実際のプログラムの intersectScene 関数においては hitSphere→intersectSphere/intersectPlaneY 関数が呼び出され、それぞれが (9) 及び (10) 式に従って t の計算を行なっている。

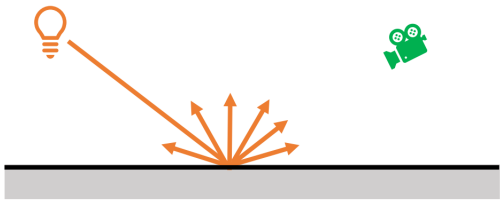

Figure5: 拡散反射の模式図。授業スライドより。

$$L^{拡}(\vec{p}, \vec{\omega}_o) = \frac{K_d}{\pi} \int_\Omega L(\vec{p}, \vec{\omega}_i) \cos\theta d\vec{\omega}_i \tag{13}$$

$$= \frac{K_d}{\pi} E(\vec{p}) \tag{14}$$

となる。特に今、$E(\vec{p})$ は位置 $\vec{p}$ における放射照度であり、今回は点光源が一つだけ存在しているような場合を考えることにすると、これは点光源の放射束 Φ、$\vec{p}$ から点光源までの距離 r、$\vec{p}$ から点光源の向きに進むベクトル $\vec{l}$ に対し次のように計算できる。

$$L^{拡}(\vec{p}, \vec{\omega}_o) = \frac{K_d}{\pi} E(\vec{p}) \tag{15}$$

$$= \frac{K_d}{\pi} \frac{\Phi \times \left(\vec{n} \cdot \vec{l}\right)}{4\pi r^2} \tag{16}$$

その上で、今回は物体ごとにその表面がどの程度 metallic であるかを metallicity という変数によって表すことにし、metallicity が小さい (すなわち物体が非金属的) 時に拡散反射が強くなるよう調整する。

また、$\vec{p}$ と点光源との間に物体が存在している場合は $\vec{p}$ は影であると判断し $L^{拡} = 0$ と考える。よって最終的に拡散反射によってカメラに届く光の強さは次のように計算される。

$$L^{拡}(\vec{p}, \vec{\omega}_o) \tag{17}$$

$$= \frac{K_d}{\pi} \frac{\Phi \times \left(\vec{n} \cdot \vec{l}\right)}{4\pi r^2} \times (1 - \text{metallicity}) \times \text{shadow} \tag{18}$$

ただし shadow は $\vec{p}$ が影の時 0、それ以外の場合 1 を取る変数である。プログラム中ではこれは shade 関数の次の部分に相当する。

```
vec3 shade(ray, Hit hit) {
    ...
     // Lambert diffuse BRDF
    vec3 diffuseBRDF = albedo/PI;
    vec3 Li = lightCol/max(dist2, 0.1);
    vec3 direct = diffuseBRDF*(Li*NoL/(4.0*PI))*(1.0-
        metallic)*shadow;
    ...
}
```

ただし上のコードでは direct が $L^{拡}$ の RGB 三次元ベクトルに対応する。また albedo は物体の反射率 (すなわち物体の色) であり、球を配置する時に決定しておく。

2.3.2 鏡面反射

物体表面における反射は拡散反射だけではない。今、物体表面で鏡面反射しカメラに入射してくるような光 $L^{鏡}$ も考えるべきである。この時、$L^{鏡}$ は再帰的にレイトレーシングを行うことで計算することができる。

まず、交差判定の際に計算した物体表面の向き $\vec{n}$ に従ってレイを反射させる。その後再びレイと物体の交差判定を行い、新しく得られた交差点 $\vec{p}'$ において反射される光を計算する。この時 $\vec{p}'$ におい

て鏡面反射が起こるのであれば、再度レイを反射させ、再びレイと物体の交差判定を行い…といったことを再帰的に繰り返していく。(Figure6参照。) ただしこの時、再帰は有限回で打ち切ることにする。

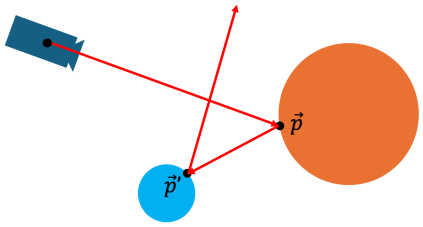

Figure6: 鏡面反射における再帰的なレイトレーシングの模式図。

この時、鏡面反射における反射率 K_s は反射角に依存し、レイの物体表面に対する角度が浅ければ浅いほど反射は強くなるはずである。今回はフレネル反射によりこの反射率を計算することを考え、以下の Schlick の式を使って K_s を求める。

$$K_s = K_s^0 + \left(1 - K_s^0\right)(1 - \cos\theta)^5 \tag{19}$$

ただし上式で θ は反射角であり、K_s^0 は物体正面（すなわち $\theta = 0$）における反射率 (< 1) で物体の材質ごとにあらかじめ決定しておくものである。(上の式を見ると確かに $\theta = \pi/2$ の時 $K_s = 1$ となり反射が最も強くなっている。)

また、今回は物体ごとにその表面がどの程度ざらついているかを roughness という変数によって表すことにし、roughness が小さいほど鏡面反射が強くなるよう調整する。

よって最終的に鏡面反射による反射光は次のように計算される。

$$L^{\text{鏡}} = L^{\text{再帰}} \times K_s \times (1 - \text{roughness}) \tag{20}$$

これを実装しているのが shade 関数の次の部分である。

```
vec3 shade(ray, Hit hit) {
    ...
    // 鏡面反射
    vec3 R = reflect(rd, N);
    vec3 reflected = traceSimple(hit.p + N * EPS * 6.0,
        R);
    vec3 F = F_Schlick(saturate(dot(N, V)), F0);
    vec3 mirror = reflected * F * (1.0 - roughness);
    ...
}
```

ただし上のコードでは mirror が $L^{\text{鏡}}$ の RGB 三次元ベクトルに対応する。また、tracesimple は $\vec{p}$ から鏡面反射方向に ray を飛ばし交差判定・反射光計算を行う関数である。traceSimple 関数の中で同様に鏡面反射が計算され、その際同様の traceSimple2 関数が呼び出される。その traceSimple2 関数の中ではさらに traceSimple3 関数が呼び出され…といった形で再帰が実現される。[*5] また、F, F_0 が先の式の K_s, K_s^0 に相当する。

2.3.3 環境光

最後に環境光について考える。今、物体との交差位置 $\vec{p}$ と光源の間に別の物体があった場合、(仮に鏡面反射がないとすると) $\vec{p}$ は影と判定され真っ黒になってしまう。しかし実際には点光源が届かなくても、うっすらと環境光が $\vec{p}$ に届いているはずであり、わずかに物体の色が現れるはずである。

そこで今回は簡単に、物体表面の色を一定の割合で弱めたものを $L^{\text{環}}$ として導入する。すなわち物体の反射率 K_d を用いて

$$L^{\text{環}} = K_d * \text{ambient\_ratio} \tag{21}$$

と計算することにする。ただし ambient_ratio については描画結果を見ながら適宜調整していく。

2.3.4 まとめ

よって最終的にカメラに入射する光は次のようになる。

$$L = L^{\text{拡}} + L^{\text{鏡}} + L^{\text{環}} \tag{22}$$

ただし $L^{\text{鏡}}$ については再帰的にレイを飛ばした先で (22) を $L \leftarrow L^{\text{鏡}}$ として再び計算する。

3 プログラムの概要

今回の実装における main 関数は次のようになっている。

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    Ray ray = generateCameraRay(fragCoord);
    Hit hit = intersectScene(ray.origin, ray.direction);
    vec3 col;
    if (hit.mat == 0) {
        col = environmentColor(ray.direction);
    } else {
        col = shade(ray.origin, ray.direction, hit);
    }
}
```

すなわち、各ピクセルに対して次の処理を行う。

1. generateCameraRay でカメラレイを生成する
2. intersectScene で Scene 内の全ての物体と交差判定を行い、一番近くの交差点を計算する
3. shade 関数によって交差点での反射光を計算し描画すべき色 col を求める。ただし交差点がない (hit.mat=0) 時は environmentColor 関数によって適切な空の色を返す。

また、実際のプログラムにおいてはトーンマッピングおよびガンマ補正を main 関数中で行っているが、今回はこれについては割愛する。[*6]

4 結果と考察

Figure7〜Figure10に今回の実装結果を示す。今 Figure7を見ると、影や鏡面における反射、材質ごと質感の違いなどが表現できており、ひとまずレイトレーシングを実装できていることがわかる。(Shadertoy のページではマウスを動かすことでカメラ位置を変えることができる。)

また、Figure8は ambient_ratio を 0 にすることで環境光を無くした時の描画結果である。この結果を見ると、環境光がないと点光源が届かない場所で影が非常に強く現れてしまっており、今回の場合のような空が明るい場合（すなわち環境光がある程度強いと考えられる場合）においては不自然になってしまっていることがわかる。

また、Figure9は鏡面反射における再帰的なレイトレーシングの様子を見やすくしたものである。今この結果を見ると途中で再帰が打ち切られ鏡面反射がそれ以上計算されなくなっている様子を確認す

[*5] すなわち今回はゴリ押しで再帰を実装した。(所詮 3,4 回しか再帰しないので。)

[*6] トーンマッピングとガンマ補正は今回あまり本質的ではないので AI に丸投げしている。

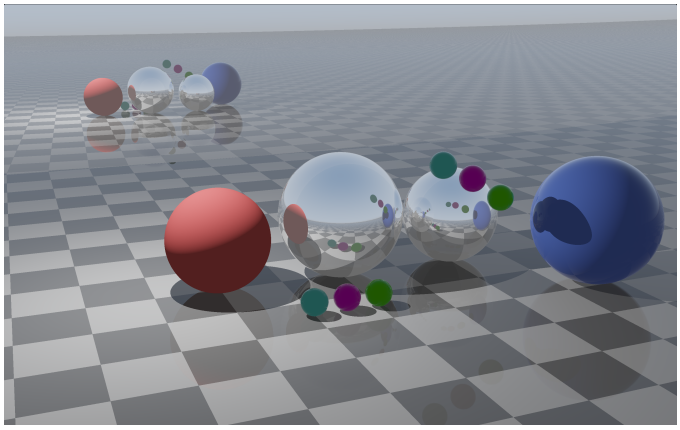

Figure7: レイトレーシングの実装結果。

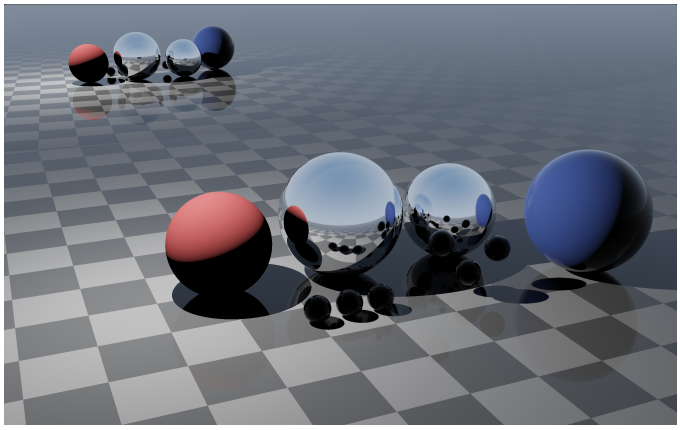

Figure8: 環境光を 0 にした場合の図。

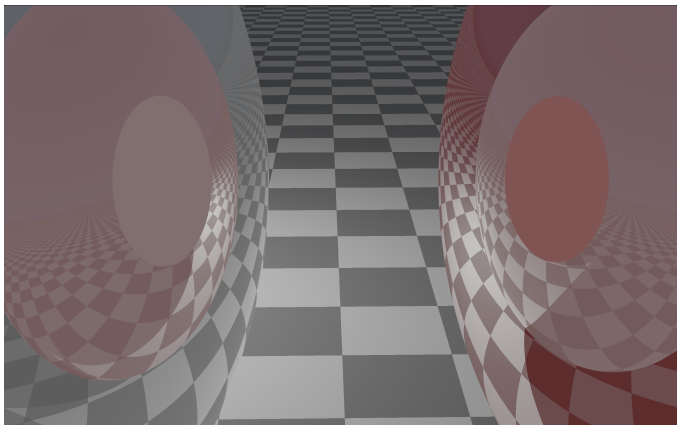

Figure9: 再帰的なレイトレーシングの図。

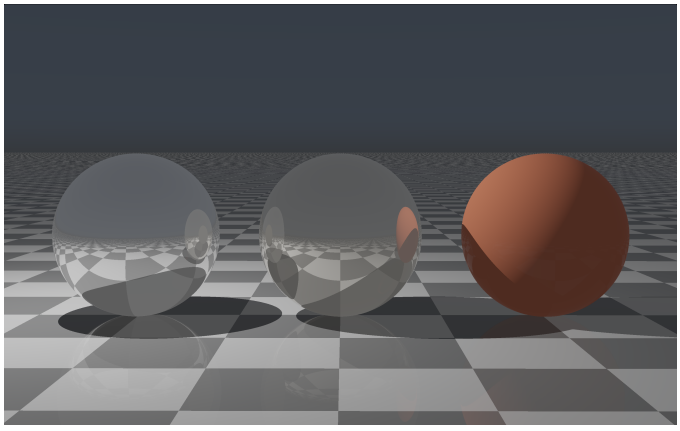

Figure10: 異なる材質の描画。

ることができる。また、今回再帰の回数を多くするほど Shadertoy の挙動が重くなる様子を確認することができた。[*7]

最後に Figure10に異なる材質を想定して色味や metallicity、roughness を調整した結果を示している。今、左から鏡面・金属・マットな非金属に対応している。この結果については厳密に材質を再現できているかを議論することは難しいが、おおよその質感を少ないパラメタ調整で達成することができることはわかった。

5　おわりに

今回は Shadertoy を用いたレイトレーシングの実装を通じてその仕組みを非常によく理解することができた。また、実際にレイトレーシングを行う前に比べて授業の理解度が格段に上がったように思う。[*8]~~今後とも隙をみて遊べたらなと思っている。~~

*7 気のせいかもしれないが…
*8 現時点で今学期で一番楽しい課題であった。