アドバンスト CG
第 3 回レポート

学籍番号：201811411
所属：情報学群情報メディア創成学類
氏名：加藤虎之介

2021 年 5 月 6 日

1　実行環境

1.1　実行に用いた OS

macOS Big ver11.3

1.2　プログラム起動時に表示される情報

OpenGL version: 2.1 ATI-4.4.17
GLSL version: 1.20
Vendor: ATI Technologies Inc.
Renderer: AMD Radeon Pro 5300M OpenGL Engine

2　課題結果

課題 1

プログラム

Code 1　PathTracer.cpp

```cpp
#include <GL/glew.h>
#include "PathTracer.h"
#include "GLFW/glfw3.h"
#include "arcball_camera.h"
#include "Scene.h"
#include "HitRecord.h"
#include <iostream>
#include <chrono>
```

```
#include "DiffuseMaterial.h"
#include "BlinnPhongMaterial.h"
#include "PerfectSpecularMaterial.h"
#include "SpecularRefractionMaterial.h"

using namespace std;
using namespace glm;

float PathTracer::s_Gamma = 2.2f;
int PathTracer::s_MaxRecursionDepth = 32;
int PathTracer::s_MinRecursionDepth = 5;
int PathTracer::s_NumSamplesPerPixel = 100;
int PathTracer::s_NumSamplesPerUpdate = 5;

extern GLFWwindow *g_pWindow;
extern ArcballCamera g_Camera;
extern Scene g_Scene;
extern int g_WindowWidth, g_WindowHeight;
extern mat4 g_ProjMatrix;
extern bool g_KeepTracing;

void PathTracer::renderScene()
{
        const auto tStart = chrono::system_clock::now();

        vec3 xAxis, yAxis, zAxis, eye;
        g_Camera.getEyeCoordinateSystem(xAxis, yAxis, zAxis, eye);

        m_FrameBuffer.allocate(g_WindowWidth, g_WindowHeight);

        const float halfWidth = 0.5f * g_WindowWidth;
        const float halfHeight = 0.5f * g_WindowHeight;
        const float screenDist = halfHeight * g_ProjMatrix[1][1];

        int nRemainingSamples = s_NumSamplesPerPixel;

        while (nRemainingSamples > 0)
        {
                const int nSamplesDone = s_NumSamplesPerPixel - nRemainingSamples;
                const int nNewRemainingSamples = std::max(nRemainingSamples -
                  s_NumSamplesPerUpdate, 0);
                const int nNewSamples = nRemainingSamples - nNewRemainingSamples;
                const int nNewSamplesDone = nSamplesDone + nNewSamples;

#pragma omp parallel for
```

```
        for (int yi = 0; yi < g_WindowHeight; ++yi)
        {
            for (int xi = 0; xi < g_WindowWidth; ++xi)
            {
                vec3 pixelColor = float(nSamplesDone) * m_FrameBuffer(xi, yi);

                for (int si = 0; si < nNewSamples; ++si)
                {
                    const vec3 dir = (xi + frand() - halfWidth) * xAxis + (yi + frand() - halfHeight) * yAxis - screenDist * zAxis;
                    pixelColor += traceRec(Ray(eye, glm::normalize(dir)), 0);
                }

                m_FrameBuffer(xi, yi) = pixelColor / float(nNewSamplesDone);
            }
        }

        nRemainingSamples = nNewRemainingSamples;

        renderIntermediateFrame();

        const auto tNow = chrono::system_clock::now();
        const auto elapsed = chrono::duration_cast<chrono::milliseconds>(tNow - tStart).count() / 1000.f;
        cerr << __FUNCTION__ << ": " << nNewSamplesDone << "/" << s_NumSamplesPerPixel << " samples (" << elapsed << " sec)" << endl;
    }
}

void PathTracer::renderFrame()
{
    if (!m_FrameBufferTexID)
        return;

    if (!m_pGammaShader)
        initShader();

    const float uOffset = (m_isNVIDIADriver) ? -0.5f / g_WindowWidth : 0.f;
    const float vOffset = (m_isNVIDIADriver) ? -0.5f / g_WindowHeight : 0.f;

    glBindTexture(GL_TEXTURE_2D, m_FrameBufferTexID);

```

```
	m_pGammaShader->use();
	m_pGammaShader->sendUniform1i("tex", 0);
	m_pGammaShader->sendUniform1f("gamma", s_Gamma);
	m_pGammaShader->sendUniform2f("offset", uOffset, vOffset);

	glBegin(GL_QUADS);
	glTexCoord2f(0, 0);
	glVertex2f(-1, -1);
	glTexCoord2f(1, 0);
	glVertex2f(1, -1);
	glTexCoord2f(1, 1);
	glVertex2f(1, 1);
	glTexCoord2f(0, 1);
	glVertex2f(-1, 1);
	glEnd();

	m_pGammaShader->disable();

	glBindTexture(GL_TEXTURE_2D, 0);
}

void PathTracer::initShader()
{
	if (!m_pGammaShader)
		m_pGammaShader = new GLSLProgramObject();

	m_pGammaShader->attachShaderCodeString(
		R"(#version 120
                        void main(void)
                        {
                                gl_TexCoord[0] = gl_MultiTexCoord0;
                                gl_Position = gl_Vertex;
                        }
                )", GL_VERTEX_SHADER);

	m_pGammaShader->attachShaderCodeString(
		R"(#version 120
                        uniform sampler2D tex;
                        uniform float gamma;
                        uniform vec2 offset;
                        void main(void)
                        {
                                gl_FragColor = vec4(pow(texture2D(tex, gl_TexCoord[0].st + offset).rgb, vec3(1.0 / gamma)), 1.0);
                        }
                )", GL_FRAGMENT_SHADER);
```

```
        m_pGammaShader->link();

        if (!m_pGammaShader->linkSucceeded())
        {
                cerr << __FUNCTION__ << ": gamma correction shader" << endl;
                m_pGammaShader->printProgramLog();
        }

        // NVIDIA driver requires -0.5 offsets in texture fetch in order to match the path-traced result
        m_isNVIDIADriver = strncmp((const char *)glGetString(GL_VENDOR), "NVIDIA", sizeof("NVIDIA") - 1) == 0;
}

glm::vec3 PathTracer::traceRec(const Ray &ray, int recursionDepth)
{
        if (recursionDepth > s_MaxRecursionDepth)
                return g_Scene.getBackgroundColor(ray);

        const float tEpsilon = 0.01f;
        const float tInfinity = 1.0e+10f;

        HitRecord record;
        record.m_ParamT = tInfinity;

        bool hitObject = false;

        for (int oi = 0; oi < g_Scene.getNumObjects(); oi++)
        {
                GeometricObject *o = g_Scene.getObject(oi);

                HitRecord tmpRec;
                const bool isHit = o->hit(ray, tEpsilon, tInfinity, tmpRec);

                if (isHit && record.m_ParamT > tmpRec.m_ParamT)
                {
                        record = tmpRec;
                        hitObject = true;
                }
        }

        if (!hitObject)
                return g_Scene.getBackgroundColor(ray);

        const Material::Material_Type matType = record.m_pMaterial->getMaterialType();
```

```cpp

        if (matType == Material::Pseudo_Normal_Color_Type)
        {
                return 0.5f * record.m_Normal + vec3(0.5f);
        }
        else if (matType == Material::Diffuse_Type)
        {
                // TODO: implement Lambert (diffuse) reflection with BRDF importance sampling
                // 拡散反射係数を取得
                const vec3 &diffuseCoeff = ((DiffuseMaterial *)record.m_pMaterial)->getDiffuseCoeff();
                // 再帰の深さが最小値より小さければ閾値を 1.0にする。
                const float russianRouletterProbability = (recursionDepth > s_MinRecursionDepth) ? std::max(diffuseCoeff.x, std::max(diffuseCoeff.y, diffuseCoeff.z)) : 1.f;
                // 閾値より大きければ計算を打ち切る
                if (frand() >= russianRouletterProbability){
                        // return g_Scene.getBackgroundColor(ray);
                        return vec3(0.f);
                }

                // 追跡するレイの方向を決めるために、局所座標系を定義する。
                // HINT: local coordinate system can be defined using the following function:
                vec3 xLocal, yLocal, zLocal;
                calcLocalCoordinateSystem(record.m_Normal, ray.getUnitDir(), xLocal, yLocal, zLocal);

                // 乱数に基づいてθとφの値を決め、局所座標系でレイの追跡方向を決定する。
                // 乱数でサンプリング
                const float xi1 = frand();
                const float xi2 = frand();
                // 局所座標系でのレイの追跡方向を決定
                const float phi = 2.f * pi<float>() * xi1;
                const float theta = acos(sqrt(xi2));
                // 通常座標系でのレイの追跡方向
                const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta), sin(phi) * cos(theta));
                const vec3 traceDir = normalize(traceDirLocal);

                // 積分計算と、再帰呼び出しの返値であるレイの追跡結果の色とを、RGBの各成分に乗算してリターンする。
                const vec3 weight = diffuseCoeff / russianRouletterProbability;
                // const vec3 weight = diffuseCoeff;
```

```
            return weight * traceRec(Ray(record.m_HitPos, traceDir), recursionDepth
              + 1);
        }
        else if (matType == Material::Blinn_Phong_Type)
        {
            // TODO: implement Blinn-Phong reflection with BRDF importance sampling
            // 鏡面反射係数を取得
            const vec3 &diffuseCoeff = ((BlinnPhongMaterial *)record.m_pMaterial)->
              getDiffuseCoeff();
            const vec3 &specularCoeff = ((BlinnPhongMaterial *)record.m_pMaterial)->
              getSpecularCoeff();
            const float shiness = ((BlinnPhongMaterial *)record.m_pMaterial)->
              getShininess();
            // 再帰の深さが最小値より小さければ閾値を 1.0にする。
            const float russianRouletterProbability = std::max(diffuseCoeff.x, std::
              max(diffuseCoeff.y, diffuseCoeff.z));
            const vec3 dsCoeff = diffuseCoeff + specularCoeff;
            const float russianRouletterProbability2 = (recursionDepth >
              s_MinRecursionDepth) ? std::max(dsCoeff.x, std::max(dsCoeff.y, dsCoeff
              .z)) : 1.f;
            // 閾値より場合分け
            const float val = frand();

            // 追跡するレイの方向を決めるために、局所座標系を定義する。
            vec3 xLocal, yLocal, zLocal;
            calcLocalCoordinateSystem(record.m_Normal, ray.getUnitDir(), xLocal,
              yLocal, zLocal);

            if (val < russianRouletterProbability) {
                // ****拡散反射の計算****
                // 乱数に基づいてθとφの値を決め、局所座標系でレイの追跡方向を決定する。
                // 乱数でサンプリング
                const float xi1 = frand();
                const float xi2 = frand();
                // 局所座標系でのレイの追跡方向を決定
                const float phi = 2.f * pi<float>() * xi1;
                const float theta = acos(sqrt(xi2));
                // 通常座標系でのレイの追跡方向
                const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta
                  ), sin(phi) * cos(theta));
                const vec3 traceDir = normalize(traceDirLocal);

                // 積分計算と、再帰呼び出しの返値であるレイの追跡結果の色とを、
                  RGB の各成分に乗算してリターンする。
                // const vec3 weight = 1.f / diffuseCoeff;
                const float weight = 0.8f;
```

```
                // const vec3 weight = diffuseCoeff / russianRouletterProbability;
                return weight * traceRec(Ray(record.m_HitPos, traceDir), recursionDepth + 1);

            } else if (russianRouletterProbability <= val && val < russianRouletterProbability2){
                // ****鏡面反射の計算****
                // 乱数に基づいてθとφの値を決め、局所座標系でレイの追跡方向を決定する。
                const float xi1 = frand();
                const float xi2 = frand();
                float phi = 2.f * pi<float>() * xi1;
                float theta = acos(pow(xi2, 1.f/(shiness+1)));
                // 通常座標系でのレイの追跡方向
                const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta), sin(phi) * cos(theta));
                const vec3 traceDir = normalize(traceDirLocal);

                // ハーフベクトルの計算(これは局所座標系で計算されているのか？)
                const vec3 half = (-ray.getUnitDir() + traceDir) / length(-ray.getUnitDir() + traceDir);

                // const vec3 weight = (specularCoeff * ((shiness + 2.f) / (shiness + 1.f)) * 4.f * std::max(glm::dot(traceDir, half), 0.f));
                // const vec3 weight = (specularCoeff * ((shiness + 2.f) / (shiness + 1.f)) * glm::dot(4.f * traceDir, half));
                const vec3 weight = (specularCoeff * ((shiness + 2.f) / (shiness + 1.f)) * 4.f * glm::dot(traceDir, half));
                return weight * traceRec(Ray(record.m_HitPos, traceDir), recursionDepth + 1);
            } else {
                // ****計算しない****
                // return g_Scene.getBackgroundColor(ray);
                return vec3(0.f);
            }
        }
        else if (matType == Material::Perfect_Specular_Type)
        {
            const vec3 &specularCoeff = ((PerfectSpecularMaterial *)record.m_pMaterial)->getSpecularCoeff();
            const float russianRouletteProbability = (recursionDepth > s_MinRecursionDepth) ? std::max(specularCoeff.x, std::max(specularCoeff.y, specularCoeff.z)) : 1.f;

            if (frand() >= russianRouletteProbability)
                return g_Scene.getBackgroundColor(ray);
```

```

        const vec3 reflectDir = normalize(reflect(ray.getUnitDir(), record.m_Normal));

        const vec3 incomingRadiance = traceRec(Ray(record.m_HitPos, reflectDir), recursionDepth + 1);
        const vec3 weight = specularCoeff / russianRouletteProbability;

        return weight * incomingRadiance;
    }
    else if (matType == Material::Specular_Refraction_Type)
    {
        const vec3 &specularCoeff = ((SpecularRefractionMaterial *)record.m_pMaterial)->getSpecularCoeff();
        const float russianRouletteProbability = (recursionDepth > s_MinRecursionDepth) ? std::max(specularCoeff.x, std::max(specularCoeff.y, specularCoeff.z)) : 1.f;

        if (frand() >= russianRouletteProbability)
            return g_Scene.getBackgroundColor(ray);

        const float _dot = dot(ray.getUnitDir(), record.m_Normal);

        // is the ray entering or outgoing the ball?
        const bool isEntering = _dot < 0.f;

        const float eta = ((SpecularRefractionMaterial *)record.m_pMaterial)->getRefractionIndex();
        const float relativeIndex = isEntering ? 1 / eta : eta;

        // Schlick's Fresnel approximation

        const float R0 = ((eta - 1.f) * (eta - 1.f)) / ((eta + 1.f) * (eta + 1.f));

        const float c = 1.f - fabsf(_dot);
        const float c2 = c * c;
        const float Re = R0 + (1 - R0) * c2 * c2 * c;
        const float Tr = (1.f - Re) * relativeIndex * relativeIndex;

        const vec3 normal = isEntering ? record.m_Normal : -record.m_Normal;

        const vec3 refractVec = refract(ray.getUnitDir(), normal, relativeIndex);
        const vec3 reflectVec = reflect(ray.getUnitDir(), normal);

```

```
        if (refractVec == vec3(0.f)) // total reflection
        {
            const vec3 incomingRadiance = traceRec(Ray(record.m_HitPos,
              reflectVec), recursionDepth + 1);
            const vec3 weight = specularCoeff / russianRouletteProbability;

            return weight * incomingRadiance;
        }

        if (recursionDepth <= 2)
        {
            const vec3 incomingRadiance = Re * traceRec(Ray(record.m_HitPos,
               reflectVec), recursionDepth + 1) + Tr * traceRec(Ray(record.
              m_HitPos, refractVec), recursionDepth + 1);

            const vec3 weight = specularCoeff / russianRouletteProbability;

            return weight * incomingRadiance;
        }
        else
        {
            // apply russian roulette for Fresnel reflection

            const float reflectionProbability = Re;

            if (frand() < reflectionProbability)
            {
                const vec3 incomingRadiance = Re * traceRec(Ray(record.
                  m_HitPos, reflectVec), recursionDepth + 1);
                const vec3 weight = specularCoeff / (
                  reflectionProbability * russianRouletteProbability);

                return weight * incomingRadiance;
            }
            else
            {
                const vec3 incomingRadiance = Tr * traceRec(Ray(record.
                  m_HitPos, refractVec), recursionDepth + 1);
                const vec3 weight = specularCoeff / ((1.f -
                  reflectionProbability) * russianRouletteProbability);

                return weight * incomingRadiance;
            }
        }
    }

```

```
        return vec3(0.f);
}

void PathTracer::updateFrameBufferTexture()
{
        if (!m_FrameBufferTexID)
                glGenTextures(1, &m_FrameBufferTexID);
        glBindTexture(GL_TEXTURE_2D, m_FrameBufferTexID);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, g_WindowWidth, g_WindowHeight, 0, GL_RGB, GL_FLOAT, m_FrameBuffer.getData());
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glBindTexture(GL_TEXTURE_2D, 0);
}

void PathTracer::renderIntermediateFrame()
{
        updateFrameBufferTexture();

        if (!g_KeepTracing)
        {
                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                renderFrame();
                glfwSwapBuffers(g_pWindow);
        }
}

void PathTracer::calcLocalCoordinateSystem(const vec3 &normal, const vec3 &inDir, vec3 &xLocal, vec3 &yLocal, vec3 &zLocal) const
{
#if 0
        yLocal = dot(normal, inDir) < 0.f ? normal : -normal;
        xLocal = normalize(cross(fabsf(normal.x) > 0.001f ? vec3(0, 1, 0) : vec3(1, 0, 0), normal));
        zLocal = cross(xLocal, yLocal);
#else
        yLocal = normal;

        if (fabsf(glm::dot(normal, inDir)) < 0.9f)
                xLocal = glm::normalize(glm::cross(inDir, normal));
        else
                xLocal = glm::normalize((fabsf(normal.y) > 0.1f) ? glm::vec3(normal.y, -normal.x, 0.f) : glm::vec3(normal.z, 0.f, -normal.x));
```

```

        zLocal = glm::cross(xLocal, yLocal);
#endif
}
```

実行結果

以下の条件で実行をした。

- 最大再帰深度：32
- 最小再帰深度：5
- １ピクセルあたりのサンプル数：5

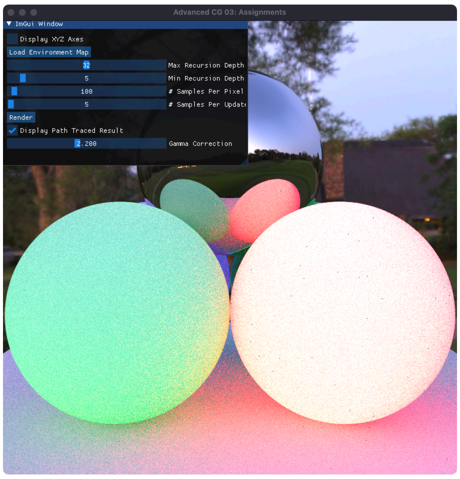

図 1　実行結果