

# 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

```
1 #include <GL/glew.h>
2 #include "PathTracer.h"
3 #include "GLFW/glfw3.h"
4 #include "arcball_camera.h"
5 #include "Scene.h"
6 #include "HitRecord.h"
7 #include <iostream>
8 #include <chrono>
```

```

9
10 #include "DiffuseMaterial.h"
11 #include "BlinnPhongMaterial.h"
12 #include "PerfectSpecularMaterial.h"
13 #include "SpecularRefractionMaterial.h"
14
15 using namespace std;
16 using namespace glm;
17
18 float PathTracer::s_Gamma = 2.2f;
19 int PathTracer::s_MaxRecursionDepth = 32;
20 int PathTracer::s_MinRecursionDepth = 5;
21 int PathTracer::s_NumSamplesPerPixel = 100;
22 int PathTracer::s_NumSamplesPerUpdate = 5;
23
24 extern GLFWwindow *g_pWindow;
25 extern ArcballCamera g_Camera;
26 extern Scene g_Scene;
27 extern int g_WindowWidth, g_WindowHeight;
28 extern mat4 g_ProjMatrix;
29 extern bool g_KeepTracing;
30
31 void PathTracer::renderScene()
32 {
33     const auto tStart = chrono::system_clock::now();
34
35     vec3 xAxis, yAxis, zAxis, eye;
36     g_Camera.getEyeCoordinateSystem(xAxis, yAxis, zAxis, eye);
37
38     m_FrameBuffer.allocate(g_WindowWidth, g_WindowHeight);
39
40     const float halfWidth = 0.5f * g_WindowWidth;
41     const float halfHeight = 0.5f * g_WindowHeight;
42     const float screenDist = halfHeight * g_ProjMatrix[1][1];
43
44     int nRemainingSamples = s_NumSamplesPerPixel;
45
46     while (nRemainingSamples > 0)
47     {
48         const int nSamplesDone = s_NumSamplesPerPixel - nRemainingSamples;
49         const int nNewRemainingSamples = std::max(nRemainingSamples -
            s_NumSamplesPerUpdate, 0);
50         const int nNewSamples = nRemainingSamples - nNewRemainingSamples;
51         const int nNewSamplesDone = nSamplesDone + nNewSamples;
52
53 #pragma omp parallel for

```

```

54     for (int yi = 0; yi < g_WindowHeight; ++yi)
55     {
56         for (int xi = 0; xi < g_WindowWidth; ++xi)
57         {
58             vec3 pixelColor = float(nSamplesDone) * m_FrameBuffer(xi
59                                     , yi);
60
61             for (int si = 0; si < nNewSamples; ++si)
62             {
63                 const vec3 dir = (xi + frand() - halfWidth) *
64                                     xAxis + (yi + frand() - halfHeight) * yAxis -
65                                     screenDist * zAxis;
66                 pixelColor += traceRec(Ray(eye, glm::normalize(
67                                         dir)), 0);
68             }
69
70             m_FrameBuffer(xi, yi) = pixelColor / float(
71                 nNewSamplesDone);
72         }
73     }
74
75     nRemainingSamples = nNewRemainingSamples;
76
77     renderIntermediateFrame();
78
79     const auto tNow = chrono::system_clock::now();
80     const auto elapsed = chrono::duration_cast<chrono::milliseconds>(tNow -
81         tStart).count() / 1000.f;
82     cerr << __FUNCTION__ << ": " << nNewSamplesDone << "/" <<
83         s_NumSamplesPerPixel << " samples " << elapsed << " sec" << endl;
84 }
85
86 void PathTracer::renderFrame()
87 {
88     if (!m_FrameBufferTexID)
89         return;
90
91     if (!m_pGammaShader)
92         initShader();
93
94     const float uOffset = (m_isNVIDIAIDriver) ? -0.5f / g_WindowWidth : 0.f;
95     const float vOffset = (m_isNVIDIAIDriver) ? -0.5f / g_WindowHeight : 0.f;
96
97     glBindTexture(GL_TEXTURE_2D, m_FrameBufferTexID);
98
99

```

```

93     m_pGammaShader->use();
94     m_pGammaShader->sendUniform1i("tex", 0);
95     m_pGammaShader->sendUniform1f("gamma", s_Gamma);
96     m_pGammaShader->sendUniform2f("offset", uOffset, vOffset);
97
98     glBegin(GL_QUADS);
99     glTexCoord2f(0, 0);
100    glVertex2f(-1, -1);
101    glTexCoord2f(1, 0);
102    glVertex2f(1, -1);
103    glTexCoord2f(1, 1);
104    glVertex2f(1, 1);
105    glTexCoord2f(0, 1);
106    glVertex2f(-1, 1);
107    glEnd();
108
109    m_pGammaShader->disable();
110
111    glBindTexture(GL_TEXTURE_2D, 0);
112 }
113
114 void PathTracer::initShader()
115 {
116     if (!m_pGammaShader)
117         m_pGammaShader = new GLSLProgramObject();
118
119     m_pGammaShader->attachShaderCodeString(
120         R"(#version_120
121         void main(void)
122         {
123             gl_TexCoord[0] = gl_MultiTexCoord0;
124             gl_Position = gl_Vertex;
125         }
126         ", GL_VERTEX_SHADER);
127
128     m_pGammaShader->attachShaderCodeString(
129         R"(#version_120
130         uniform sampler2D tex;
131         uniform float gamma;
132         uniform vec2 offset;
133         void main(void)
134         {
135             gl_FragColor = vec4(pow(texture2D(tex, gl_TexCoord[0].st +
136                 offset).rgb, vec3(1.0/gamma)), 1.0);
137         }
138         ", GL_FRAGMENT_SHADER);

```

```

138
139     m_pGammaShader->link();
140
141     if (!m_pGammaShader->linkSucceeded())
142     {
143         cerr << __FUNCTION__ << ":\gamma_correction_shader" << endl;
144         m_pGammaShader->printProgramLog();
145     }
146
147     // NVIDIA driver requires -0.5 offsets in texture fetch in order to match the path
148     -traced result
149     m_isNVIDIADriver = strcmp((const char *)glGetString(GL_VENDOR), "NVIDIA",
150                               sizeof("NVIDIA") - 1) == 0;
151 }
152
153 glm::vec3 PathTracer::traceRec(const Ray &ray, int recursionDepth)
154 {
155     if (recursionDepth > s_MaxRecursionDepth)
156         return g_Scene.getBackgroundColor(ray);
157
158     const float tEpsilon = 0.01f;
159     const float tInfinity = 1.0e+10f;
160
161     HitRecord record;
162     record.m_ParamT = tInfinity;
163
164     bool hitObject = false;
165
166     for (int oi = 0; oi < g_Scene.getNumObjects(); oi++)
167     {
168         GeometricObject *o = g_Scene.getObject(oi);
169
170         HitRecord tmpRec;
171         const bool isHit = o->hit(ray, tEpsilon, tInfinity, tmpRec);
172
173         if (isHit && record.m_ParamT > tmpRec.m_ParamT)
174         {
175             record = tmpRec;
176             hitObject = true;
177         }
178     }
179
180     if (!hitObject)
181         return g_Scene.getBackgroundColor(ray);
182
183     const Material::Material_Type matType = record.m_pMaterial->getMaterialType();

```

```

182
183     if (matType == Material::Pseudo_Normal_Color_Type)
184     {
185         return 0.5f * record.m_Normal + vec3(0.5f);
186     }
187     else if (matType == Material::Diffuse_Type)
188     {
189         // TODO: implement Lambert (diffuse) reflection with BRDF importance
190         sampling
191         // 拡散反射係数を取得
192         const vec3 &diffuseCoeff = ((DiffuseMaterial *)record.m_pMaterial)->
193             getDiffuseCoeff();
194         // 再帰の深さが最小値より小さければ閾値を 1.0にする。
195         const float russianRouletterProbability = (recursionDepth >
196             s_MinRecursionDepth) ? std::max(diffuseCoeff.x, std::max(diffuseCoeff.
197             y, diffuseCoeff.z)) : 1.f;
198         // 閾値より大きければ計算を打ち切る
199         if (frand() >= russianRouletterProbability){
200             // return g_Scene.getBackgroundColor(ray);
201             return vec3(0.f);
202         }
203
204         // 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。
205         // HINT: local coordinate system can be defined using the following
206         function:
207         vec3 xLocal, yLocal, zLocal;
208         calcLocalCoordinateSystem(record.m_Normal, ray.getUnitDir(), xLocal,
209             yLocal, zLocal);
210
211         // 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。
212         // 乱数でサンプリング
213         const float xi1 = frand();
214         const float xi2 = frand();
215         // 局所座標系でのレイの追跡方向を決定
216         const float phi = 2.f * pi<float>() * xi1;
217         const float theta = acos(sqrt(xi2));
218         // 通常座標系でのレイの追跡方向
219         const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta), sin(
220             phi) * cos(theta));
221         const vec3 traceDir = normalize(traceDirLocal);
222
223         // 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、
224         RGB の各成分に乘算してリターンする。
225         const vec3 weight = diffuseCoeff / russianRouletterProbability;
226         // const vec3 weight = diffuseCoeff;

```

```

219         return weight * traceRec(Ray(record.m_HitPos, traceDir), recursionDepth
220             + 1);
221     }
222     else if (matType == Material::Blinn_Phong_Type)
223     {
224         // TODO: implement Blinn-Phong reflection with BRDF importance sampling
225         // 鏡面反射係数を取得
226         const vec3 &diffuseCoeff = ((BlinnPhongMaterial *)record.m_pMaterial)->
227             getDiffuseCoeff();
228         const vec3 &specularCoeff = ((BlinnPhongMaterial *)record.m_pMaterial)->
229             getSpecularCoeff();
230         const float shininess = ((BlinnPhongMaterial *)record.m_pMaterial)->
231             getShininess();
232         // 再帰の深さが最小値より小さければ閾値を 1.0にする。
233         const float russianRouletterProbability = std::max(diffuseCoeff.x, std:::
234             max(diffuseCoeff.y, diffuseCoeff.z));
235         const vec3 dsCoeff = diffuseCoeff + specularCoeff;
236         const float russianRouletterProbability2 = (recursionDepth >
237             s_MinRecursionDepth) ? std::max(dsCoeff.x, std:::max(dsCoeff.y, dsCoeff
238             .z)) : 1.f;
239         // 閾値より場合分け
240         const float val = frand();
241
242         // 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。
243         vec3 xLocal, yLocal, zLocal;
244         calcLocalCoordinateSystem(record.m_Normal, ray.getUnitDir(), xLocal,
245             yLocal, zLocal);
246
247         if (val < russianRouletterProbability) {
248             // ****拡散反射の計算****
249             // 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。
250             // 乱数でサンプリング
251             const float xi1 = frand();
252             const float xi2 = frand();
253             // 局所座標系でのレイの追跡方向を決定
254             const float phi = 2.f * pi<float>() * xi1;
255             const float theta = acos(sqrt(xi2));
256             // 通常座標系でのレイの追跡方向
257             const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta)
258             ), sin(phi) * cos(theta));
259             const vec3 traceDir = normalize(traceDirLocal);
260
261             // 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、
262             RGB の各成分に乘算してリターンする。
263             // const vec3 weight = 1.f / diffuseCoeff;
264             const float weight = 0.8f;

```

```

255         // const vec3 weight = diffuseCoeff / russianRouletteProbability;
256         return weight * traceRec(Ray(record.m_HitPos, traceDir),
                                   recursionDepth + 1);
257
258     } else if (russianRouletteProbability <= val && val <
                russianRouletteProbability2){
259         // ****鏡面反射の計算****
260         // 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。
261         const float xi1 = frand();
262         const float xi2 = frand();
263         float phi = 2.f * pi<float>() * xi1;
264         float theta = acos(pow(xi2, 1.f/(shiness+1)));
265         // 通常座標系でのレイの追跡方向
266         const vec3 traceDirLocal = vec3(cos(phi) * cos(theta), sin(theta)
                                           ), sin(phi) * cos(theta));
267         const vec3 traceDir = normalize(traceDirLocal);
268
269         // 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?)
270         const vec3 half = (-ray.getUnitDir() + traceDir) / length(-ray.
                                                                    getUnitDir() + traceDir);
271
272         // const vec3 weight = (specularCoeff * ((shiness + 2.f) / (
273             shiness + 1.f)) * 4.f * std::max(glm::dot(traceDir, half), 0.f
274             ));
275         // const vec3 weight = (specularCoeff * ((shiness + 2.f) / (
276             shiness + 1.f)) * glm::dot(4.f * traceDir, half));
277         const vec3 weight = (specularCoeff * ((shiness + 2.f) / (
278             shiness + 1.f)) * 4.f * glm::dot(traceDir, half));
279         return weight * traceRec(Ray(record.m_HitPos, traceDir),
                                   recursionDepth + 1);
280     } else {
281         // ****計算しない****
282         // return g_Scene.getBackgroundColor(ray);
283         return vec3(0.f);
284     }
285 }
286
287 else if (matType == Material::Perfect_Specular_Type)
288 {
289     const vec3 &specularCoeff = ((PerfectSpecularMaterial *)record.
290                                 m_pMaterial)->getSpecularCoeff();
291     const float russianRouletteProbability = (recursionDepth >
292                                               s_MinRecursionDepth) ? std::max(specularCoeff.x, std::max(
293                                                 specularCoeff.y, specularCoeff.z)) : 1.f;
294
295     if (frand() >= russianRouletteProbability)
296         return g_Scene.getBackgroundColor(ray);

```



```

289
290     const vec3 reflectDir = normalize(reflect(ray.getUnitDir(), record.
      m_Normal));
291
292     const vec3 incomingRadiance = traceRec(Ray(record.m_HitPos, reflectDir),
      recursionDepth + 1);
293     const vec3 weight = specularCoeff / russianRouletteProbability;
294
295     return weight * incomingRadiance;
296 }
297 else if (matType == Material::Specular_Refraction_Type)
298 {
299     const vec3 &specularCoeff = ((SpecularRefractionMaterial *)record.
      m_pMaterial)->getSpecularCoeff();
300     const float russianRouletteProbability = (recursionDepth >
      s_MinRecursionDepth) ? std::max(specularCoeff.x, std::max(
      specularCoeff.y, specularCoeff.z)) : 1.f;
301
302     if (frand() >= russianRouletteProbability)
303         return g_Scene.getBackgroundColor(ray);
304
305     const float _dot = dot(ray.getUnitDir(), record.m_Normal);
306
307     // is the ray entering or outgoing the ball?
308     const bool isEntering = _dot < 0.f;
309
310     const float eta = ((SpecularRefractionMaterial *)record.m_pMaterial)->
      getRefractionIndex();
311     const float relativeIndex = isEntering ? 1 / eta : eta;
312
313     // Schlick's Fresnel approximation
314
315     const float R0 = ((eta - 1.f) * (eta - 1.f)) / ((eta + 1.f) * (eta +
      1.f));
316
317     const float c = 1.f - fabsf(_dot);
318     const float c2 = c * c;
319     const float Re = R0 + (1 - R0) * c2 * c2 * c;
320     const float Tr = (1.f - Re) * relativeIndex * relativeIndex;
321
322     const vec3 normal = isEntering ? record.m_Normal : -record.m_Normal;
323
324     const vec3 refractVec = refract(ray.getUnitDir(), normal, relativeIndex
      );
325     const vec3 reflectVec = reflect(ray.getUnitDir(), normal);
326

```

```

327         if (refractVec == vec3(0.f)) // total reflection
328         {
329             const vec3 incomingRadiance = traceRec(Ray(record.m_HitPos,
330                 reflectVec), recursionDepth + 1);
331             const vec3 weight = specularCoeff / russianRouletteProbability;
332             return weight * incomingRadiance;
333         }
334
335         if (recursionDepth <= 2)
336         {
337             const vec3 incomingRadiance = Re * traceRec(Ray(record.m_HitPos,
338                 reflectVec), recursionDepth + 1) + Tr * traceRec(Ray(record.
339                 m_HitPos, refractVec), recursionDepth + 1);
340
341             const vec3 weight = specularCoeff / russianRouletteProbability;
342             return weight * incomingRadiance;
343         }
344         else
345         {
346             // apply russian roulette for Fresnel reflection
347
348             const float reflectionProbability = Re;
349
350             if (frand() < reflectionProbability)
351             {
352                 const vec3 incomingRadiance = Re * traceRec(Ray(record.
353                     m_HitPos, reflectVec), recursionDepth + 1);
354                 const vec3 weight = specularCoeff / (
355                     reflectionProbability * russianRouletteProbability);
356                 return weight * incomingRadiance;
357             }
358             else
359             {
360                 const vec3 incomingRadiance = Tr * traceRec(Ray(record.
361                     m_HitPos, refractVec), recursionDepth + 1);
362                 const vec3 weight = specularCoeff / ((1.f -
363                     reflectionProbability) * russianRouletteProbability);
364                 return weight * incomingRadiance;
365             }
366         }
367     }
368 }

```

```

366         return vec3(0.f);
367     }
368
369     void PathTracer::updateFrameBufferTexture()
370     {
371         if (!m_FrameBufferTexID)
372             glGenTextures(1, &m_FrameBufferTexID);
373         glBindTexture(GL_TEXTURE_2D, m_FrameBufferTexID);
374         glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
375         glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
376         glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
377         glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
378         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, g_WindowWidth, g_WindowHeight, 0,
379                     GL_RGB, GL_FLOAT, m_FrameBuffer.getData());
380         glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
381         glBindTexture(GL_TEXTURE_2D, 0);
382     }
383
384     void PathTracer::renderIntermediateFrame()
385     {
386         updateFrameBufferTexture();
387
388         if (!g_KeepTracing)
389         {
390             glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
391             renderFrame();
392             glfwSwapBuffers(g_pWindow);
393         }
394     }
395
396     void PathTracer::calcLocalCoordinateSystem(const vec3 &normal, const vec3 &inDir, vec3 &
397         xLocal, vec3 &yLocal, vec3 &zLocal) const
398     {
399         #if 0
400             yLocal = dot(normal, inDir) < 0.f ? normal : -normal;
401             xLocal = normalize(cross(fabsf(normal.x) > 0.001f ? vec3(0, 1, 0) : vec3(1,
402                 0, 0), normal));
403             zLocal = cross(xLocal, yLocal);
404         #else
405             yLocal = normal;
406
407             if (fabsf(glm::dot(normal, inDir)) < 0.9f)
408                 xLocal = glm::normalize(glm::cross(inDir, normal));
409             else
410                 xLocal = glm::normalize((fabsf(normal.y) > 0.1f) ? glm::vec3(normal.y,
411                     -normal.x, 0.f) : glm::vec3(normal.z, 0.f, -normal.x));
412         #endif
413     }

```

```
408
409         zLocal = glm::cross(xLocal, yLocal);
410 #endif
411 }
```

---

## 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

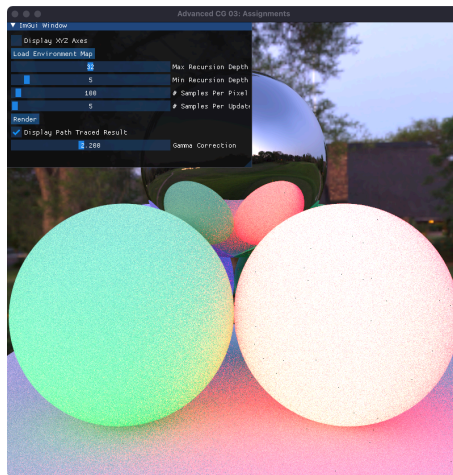


図 1 実行結果