Lab-0 补充

图形绘制技术

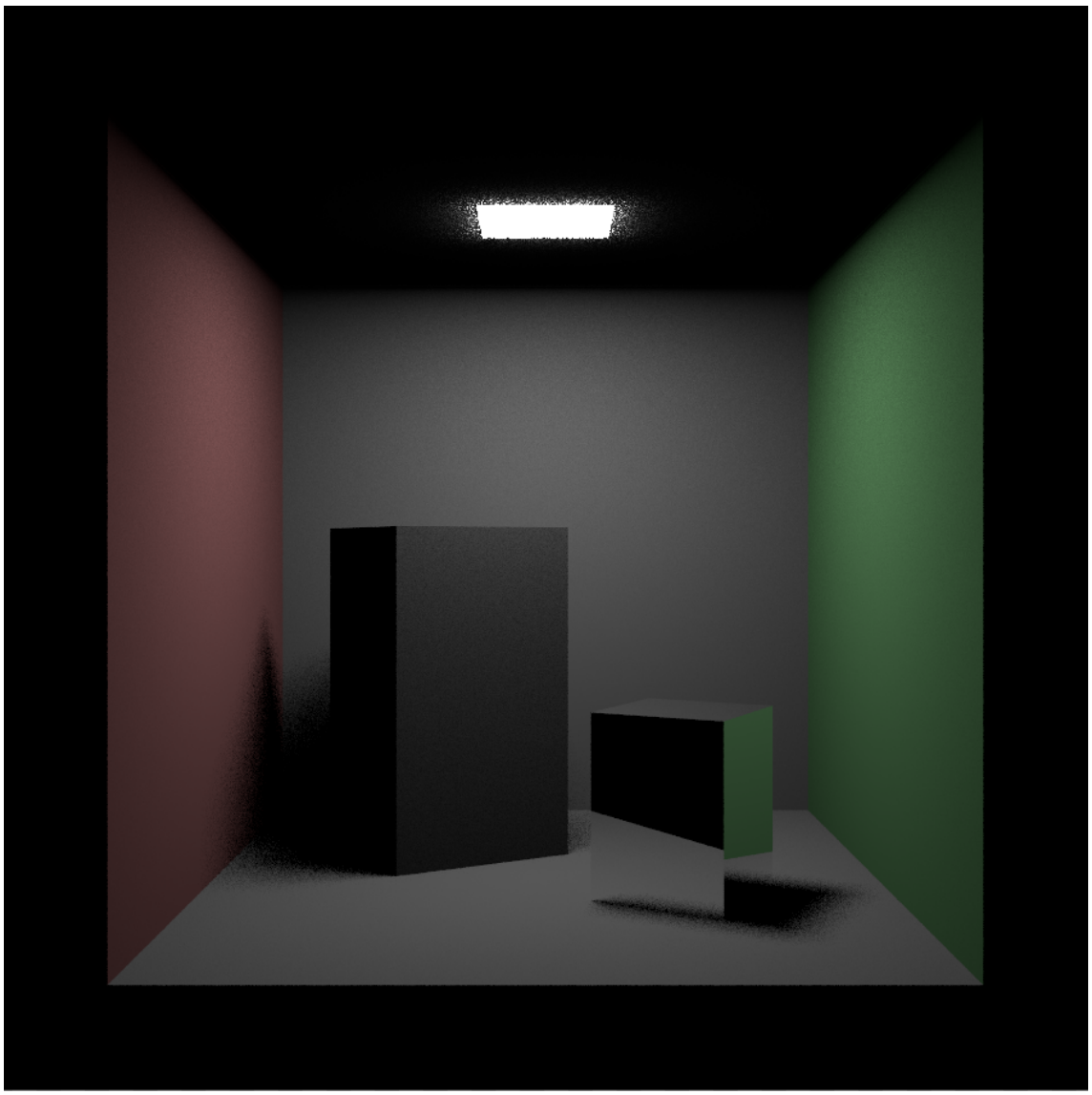
Introduction and Prerequisite

对于 Lab0 手册的补充

2023 春季学期

1. 如何向框架添加新的功能

目前我们又提供了一个新场景 examples/cornell-box，cornell-box 是渲染中最常见的测试场景之一。更新仓库中的代码并重新编译，运行该场景后，你应该能够得到如下的结果



该场景由一个 Box、两个 Cube 以及一个面光源组成，其中小 Cube 使用了新的材质 Mirror。为了支持该场景的渲染，我们需要在框架中增加以下功能：

- Cube 几何体
- Mirror 材质
- Whitted-style 积分器

该补充文档将以以上三个功能为例，详细介绍如何向框架中添加新的功能。

1.1 Cube 立方体

Cube 立方体是一个 shape 对象，因此它需要实现 rayIntersectShape 和 fillIntersection 两个方法，只要我们不使用 Cube 作为面光源，uniformSampleOnSurface 可以暂时不实现。

Cube 的几何描述

目前，我们只使用三个参数就可以描述一个 Cube 几何体

```
Transform transform;    // in base class
Point3f boxMin{-1.f, -1.f, -1.f},
        boxMax{ 1.f,  1.f,  1.f};
```

对于 Cube 相关的计算，我们始终在局部坐标系内进行，这样有一个明显的好处是可以减少求交计算以及几何描述的复杂度。在局部坐标系内，Cube 的中心始终位于原点，且每个棱都与某一个坐标轴平行。**这样的几何体有一个更准确的名称 Axis-aligned box，在 Lab1 的作业中我们将详细介绍 AABB（Axis-aligned bounding box）在渲染中的作用**。此处，我们将它当作一个和 sphere 一样，能够与光线进行求交，并计算交点处的各类信息的几何体即可。

Cube 的初始化

Cube 默认为中心在原点，每条棱长均为 2 的正方体。由于所有计算都只在 Cube 的局部坐标系内进行，因此对于 Transform，我们只使用其中的 scale 对 boxMin 和 boxMax 进行缩放。

```
// Default setting
boxMin = Point3f{-1.f, -1.f, -1.f};
boxMax = Point3f{1.f, 1.f, 1.f};

// Build AABB(axis-aligned bounding box)
// AABB is an axis-aligned box which completely bounds the shape
pMin = pMax = transform.toWorld(boxMin);
for (int i = 0; i < 8; ++i) {
  Point3f p;
  p[0] = (i & 0b100) ? boxMax[0] : boxMin[0];
  p[1] = (i & 0b010) ? boxMax[1] : boxMin[1];
  p[2] = (i & 0b001) ? boxMax[2] : boxMin[2];
  p = transform.toWorld(p);

  for (int j = 0; j < 3; ++j) {
    pMin[j] = std::min(pMin[j], p[j]);
    pMax[j] = std::max(pMax[j], p[j]);
  }
}

// Only apply scalation to the Cube
// We will apply the translation and rotation to the ray, to ensure the
// relative position remains consistent
// Here is an example for applying a transform to a point in homogeneous coordinate
Matrix4f scale = transform.scale;
vecmat::vec4f min{boxMin[0], boxMin[0], boxMin[0], 1.f},
    max{boxMax[0], boxMax[0], boxMax[0], 1.f};
min = scale * min, max = scale * max;
min /= min[3], max /= max[3];
boxMin = Point3f{min[0], min[1], min[2]};
boxMax = Point3f{max[0], max[1], max[2]};
```

rayIntersectShape

对于 rayIntersectShape 方法，我们在其中实现 shape 与 ray 的求交逻辑。如果 ray 与 shape 相交，那么返回 true 并且使用三个参数（primID, u, v）唯一地确定交点的位置。

首先，我们将 shape 的旋转变换和平移变换的**逆变换**应用到光线的起始点与方向。这样保证了 Cube 和 ray 的相对位置不变，计算结果与对 Cube 应用 transfrom 相同。

```
// begin Cube::rayIntersectShape

// apply rotation and translation to ray
Point3f origin = ray.origin;
Vector3f direction = ray.direction;
vecmat::vec4f o{origin[0], origin[1], origin[2], 1.f},
    d{direction[0], direction[1], direction[2], 0.f};
o = transform.invRotate * transform.invTranslate * o;
d = transform.invRotate * transform.invTranslate * d;
o /= o[3];
origin = Point3f{o[0], o[1], o[2]};
direction = Vector3f{d[0], d[1], d[2]};
```

之后，我们对 ray(origin, direction) 和 axis-aligned 的 Cube 进行求交

```
float tNear = ray.tNear, tFar = ray.tFar;

// Along each axis, we compute the distance for ray to reach the cube
for (int i = 0; i < 3; ++i) {
  float invDir = 1.f / direction[i];
  float t0 = (boxMin[i] - origin[i]) * invDir,
        t1 = (boxMax[i] - origin[i]) * invDir;
  if (t0 > t1)
    std::swap(t0, t1);
  tNear = std::max(tNear, t0);
  tFar = std::min(tFar, t1);

  if (tNear > tFar)
    return false;
}
```

e.g. 当 i==0 时，t0 和 t1 是光线到达 **Cube 与 x 轴垂直的两个面**所需要的距离，因此光线与 Cube 发生相交的距离一定在 [t0, t1] 之间。同理，我们对于每个轴进行相同计算，对最终的三个结果求交集，即可确定光线与 Cube 两个交点到光线起始点的距离。如果三个结果中有任意两个求交是空集，那么说明光线与 Cube 不发生相交。

```
bool hit = false;
// Compute the tFar first
if (ray.tNear < tFar && tFar < ray.tFar) {
  Point3f hitpoint = origin + tFar * direction;
  compute(hitpoint, primID, u, v);
  *distance = tFar;
  hit = true;
}

// If tNear is the real hit distance, this will overwrite the tFar result
if (ray.tNear < tNear && tNear < ray.tFar) {
  Point3f hitpoint = origin + tNear * direction;
  compute(hitpoint, primID, u, v);
```

```
    *distance = tNear;
    hit = true;
  }
  return hit;
  // Cube::rayIntersectShape end
```

最终，我们需要将求得的 tNear 和 tFar 与光线的范围进行比较，计算出真正的交点。对于交点的参数化，我们使用了一个 lambda 函数进行计算

```
auto compute = [min = boxMin, max = boxMax](Point3f hitpoint, int *primID,
                                            float *u, float *v) {
  float minBias = FLT_MAX;

  // Use flag to identify hit which surface
  int flag = -1;
  for (int i = 0; i < 3; ++i) {
    if (float bias = std::abs(hitpoint[i] - min[i]); bias < minBias) {
      flag = 2 * i;
      minBias = bias;
    }
    if (float bias = std::abs(hitpoint[i] - max[i]); bias < minBias) {
      flag = 2 * i + 1;
      minBias = bias;
    }
  }

  *primID = flag;
  // u, v represent the hitpoint offset
  int axis = (flag / 2 + 1) % 3;
  *u = (float)(hitpoint[axis] - min[axis]) / (max[axis] - min[axis]);
  axis = (axis + 1) % 3;
  *v = (float)(hitpoint[axis] - min[axis]) / (max[axis] - min[axis]);
};
```

以上就是 Cube 与光线求交的所有逻辑。

fillIntersection

计算得到 primID、u 和 v 后，我们需要根据这三个参数计算出交点个各种信息，并填充至 Intersection 结构中。

```
  // begin Cube::fillIntersection
  intersection->shape = this;
  intersection->distance = distance;
  intersection->texCoord = Vector2f{u, v};

  // transform the local normal to the world coordinate
  vecmat::vec4f normal{.0f, .0f, .0f, .0f};
  normal[primID / 2] = (primID % 2) ? 1 : -1;
  normal = transform.rotate * normal;
  intersection->normal = normalize(Vector3f{normal[0], normal[1], normal[2]});

  // transform the local hitpoint to the world coordinate
  // compute local hitpoint
  vecmat::vec4f hitpoint;
  hitpoint[primID / 2] = (primID % 2) ? boxMax[primID / 2] : boxMin[primID / 2];
  int axis = (primID / 2 + 1) % 3;
```

```
  hitpoint[axis] = boxMin[axis] + u * (boxMax[axis] - boxMin[axis]);
  axis = (axis + 1) % 3;
  hitpoint[axis] = boxMin[axis] + v * (boxMax[axis] - boxMin[axis]);
  // transform it to world space
  hitpoint[3] = 1.f;
  hitpoint = transform.translate * transform.rotate * hitpoint;
  hitpoint /= hitpoint[3];
  intersection->position = Point3f{hitpoint[0], hitpoint[1], hitpoint[2]};

  // TODO compute the tangent and bitangent
  Vector3f tangent{1.f, 0.f, .0f};
  Vector3f bitangent;
  if (std::abs(dot(tangent, intersection->normal)) > .9f) {
    tangent = Vector3f(.0f, 1.f, .0f);
  }
  bitangent = normalize(cross(tangent, intersection->normal));
  tangent = normalize(cross(intersection->normal, bitangent));
  intersection->tangent = tangent;
  intersection->bitangent = bitangent;
  // Cube::fillIntersection end
```

目前切线和副切线的计算还没有完全实现，在不使用法线贴图等需要准确局部切空间的功能时，我们只需要保证 tangent 和 bitangent 与和 normal 一起构成一个正交空间系即可。

Register 注册

本系统中，我们使用了一个宏对各种可配置组件进行注册

```
REGISTER_CLASS(Cube, "cube")
```

在**.cpp** 文件中使用REGISTER_CLASS宏为当前类提供一个类型名称进行注册。之后创建该类时只需要在 json 中声明对应 type 即可。**需要进行注册的类都应当实现参数为 const Json & 的构造方法**。

1.2 Mirror 镜面反射材质

镜面反射材质只需要实现以下方法

```
  std::shared_ptr<BSDF>
  MirrorMaterial::computeBSDF(const Intersection &intersection) const {
    Vector3f normal, tangent, bitangent;
    computeShadingGeometry(intersection, &normal, &tangent, &bitangent);
    return std::make_shared<SpecularReflection>(normal, tangent, bitangent);
  }
```

intersection 中已经有了在求交时计算出的法线、切线与副切线。此处的 computeShadingGeometry 是在当前材质配置了法线贴图时起作用的，它会根据法线贴图提供的法线指重新计算一个局部正交切空间。**SpecularReflection BSDF** 每个材质都需要能够根据交点信息返回该点的 BSDF 描述。对于镜面反射材质，其交点处 BSDF 是镜面反射 BSDF

```
  class SpecularReflection : public BSDF {
public:
  SpecularReflection(const Vector3f &_normal, const Vector3f &_tangent,
                     const Vector3f &_bitangent)
```

```
    : BSDF(_normal, _tangent, _bitangent) {}

  virtual Spectrum f(const Vector3f &wo, const Vector3f &wi) const override {
    return Spectrum(.0f);
  }

  virtual BSDFSampleResult sample(const Vector3f &wo,
                                  const Vector2f &sample) const override {
    Vector3f woLocal = toLocal(wo);
    Vector3f wiLocal{-woLocal[0], woLocal[1], -woLocal[2]};
    return {Spectrum(1.f), toWorld(wiLocal), 1.f, BSDFType::Specular};
  }
};
```

SpecularReflection 的实现非常简单。其中 toWorld 和 toLocal 方法是 BSDF 基类提供的方法，它根据构建 BSDF 时提供的局部正交切空间完成向量在不同坐标系内的转换。由于镜面反射 BSDF 实际上是一个 delta 分布，因此在调用 f 计算任意两个方向对应的 BSDF 值时，我们始终返回 0——如果不对该 delta 分布做直接的采样，任何其他采样方法几乎不可能采样到有效的值——具体的来说，当我们任意的选择一个发光点并与 shadingPoint（材质为 Mirror）做连接时，该入射方向经过反射后恰好是 tracing 的出射方向的概率基本为 0。sample 时，我们直接计算出射方向经过反射后的方向即可得到入射方向，该过程也不涉及采样。

在之前的框架中，我们提供了两个用于计算直接光照的积分器，分别是 directSampleLight 和 directSampleBSDF。其中 directSampleLight 在采样时不考虑 shadingPoint 处材质的信息，而 directSampleBSDF 在采样时又不会考虑场景中光源的信息。因此它们所适用的场景都是有限的，例如 directSampleBSDF 无法渲染只有点光源的场景，因为点光源无法进行求交；而 directSampleLight 无法渲染有镜面反射材质的场景，因为采样光源得到的入射方向基本无法构建成一条完整的镜面反射光路。因此，我们提供了一种新的积分器 whittedIntegrator，实现了 whitted-style ray tracing。

同学们可以修改 cornell-box 中 scene.json 使用的积分器类型，看看 directSampleLight 和 directSampleBSDF 积分器的渲染结果如何。

1.3 WhittedIntegrator whitted 光追

Whitted-style ray tracing 的原理很简单。在 trace 到每一个 shadingPoint 时，如果该点的材质是漫反射材质，那么我们就通过采样光源的方法计算该点的光照值，然后停止光路；如果该点的材质是镜面材质，那么我们反射光线，延伸光路，然后循环该过程，直到光路停止。

```
// begin WhittedIntegrator::li
Spectrum spectrum(.0f), beta(1.0f);
Ray ray(_ray);

do {
  auto itsOpt = scene.rayIntersect(ray);

  // escape the scene
  if (!itsOpt.has_value()) {
    for (auto light : scene.infiniteLights) {
      spectrum += beta * light->evaluateEmission(ray);
    }
```

```
      break;
    }

    Intersection its = itsOpt.value();
    if (auto light = its.shape->light; light) {
      spectrum += beta * light->evaluateEmission(its, -ray.direction);
    }
    computeRayDifferentials(&its, ray);
    auto bsdf = its.shape->material->computeBSDF(its);

    auto bsdfSampleResult = bsdf->sample(-ray.direction, sampler->next2D());

    // If the surface is specular, spwan the ray
    if (bsdfSampleResult.type == BSDFType::Specular) {
      ray = Ray(its.position, bsdfSampleResult.wi);
      beta *= bsdfSampleResult.weight;
      continue;
    }
    // If the surface is not specular, sample the light
    else {
      // First, sample infinite light
      for (auto light : scene.infiniteLights) {
        auto lightSampleResult = light->sample(its, sampler->next2D());
        Ray shadowRay(its.position, lightSampleResult.direction, 1e-4f,
                      FLT_MAX);
        // Successfully connect the light source
        if (auto occlude = scene.rayIntersect(shadowRay);
            !occlude.has_value()) {
          Spectrum f = bsdf->f(-ray.direction, shadowRay.direction);
          float pdf = convertPDF(lightSampleResult, its);
          spectrum += beta * lightSampleResult.energy * f / pdf;
        }
      }

      float pdfLight = .0f;
      // Second, sample the light in scene
      auto light = scene.sampleLight(sampler->next1D(), &pdfLight);
      if (light && pdfLight != .0f) {
        auto lightSampleResult = light->sample(its, sampler->next2D());
        Ray shadowRay(its.position, lightSampleResult.direction, 1e-4f,
                      lightSampleResult.distance);
        // Successfully connect the light source
        if (auto occlude = scene.rayIntersect(shadowRay);
            !occlude.has_value()) {
          Spectrum f = bsdf->f(-ray.direction, shadowRay.direction);
          lightSampleResult.pdf *= pdfLight;
          float pdf = convertPDF(lightSampleResult, its);
          spectrum += beta * lightSampleResult.energy * f / pdf;
        }
      }
      break;
    }
  } while (1);

  return spectrum;
  // WhittedIntegrator::li end
```

1.4 编译新加入的文件

我们使用 CMake 对跨平台编译提供支持，因此项目的编译信息需要我们自行提供，这其中也包括了需要编译哪些源文件。

```
add_executable(Moer
  # previous
  ...

  # new source files
  src/FunctionLayer/Integrator/WhittedIntegrator.cpp
  src/FunctionLayer/Shape/Cube.cpp
  src/FunctionLayer/Material/Mirror.cpp
)
```

我们只需要将新添加的源文件加入 add_executable 命令的参数中即可。之后按照 Lab0 中的编译过程即可将添加的代码集成到系统中。