Tutorial 2
OSPRay – Classic Ray Tracing

Iluminação e Visualização II

Luís Paulo Santos, April 2021

Throughout this tutorial you will develop a Renderer using the classical (Whitted) ray tracing algorithm.

Introduction

Classical ray tracing, as initially proposed by Whitted[1] and depicted in the figure below, is completely deterministic. At each intersection point three light transport phenomena are taken into account: i) direct lighting, ii) ideal specular reflection and iii) ideal specular transmission.

In order for you to have access to all the materials required for this and next tutorials a new `git` branch, called `Tutorial`, has been created. Make sure you move to the directory where you installed OSPRay and write:

```
> git checkout Tutorial
```

In order to build the new version of `CGViewer2021` execute:

```
> cd ospray-vi2/build
```

---

[1] Turner Whitted. 1998. An improved illumination model for shaded display. In Seminal graphics. ACM, New York, NY, USA 119-125.

```
> cmake .. -DCMAKE_INSTALL_PREFIX=../local-thrparty/install

> make
```

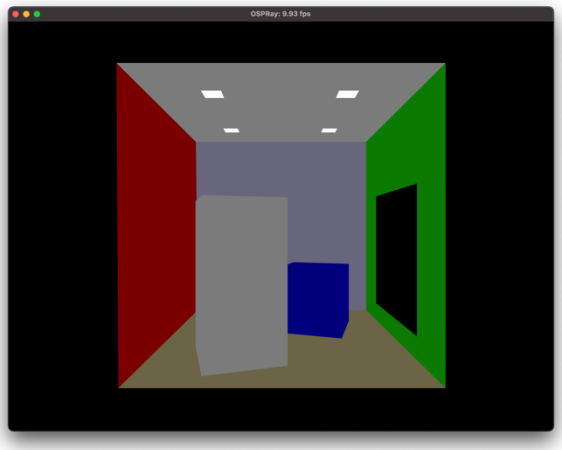
Now run the viewer for the cornell_VI2 scene:

```
> ./cornell_VI2
```

You can also run it for the conference scene:

```
> ./conference
```

You must have obtained the following images, which result from using ambient lighting:



Basic Renderer

The initial version of the renderer does only ambient lighting. Open the file `MyRenderer-T2.ispc`. There you will see the method used to shade each primary ray:

```
void MyRenderer_renderSample_RT(uniform Renderer *uniform _self,
                                uniform FrameBuffer *uniform fb,
                                uniform World *uniform model,
                                void *uniform perFrameData,
                                varying ScreenSample &sample)
{
  uniform MyRenderer *uniform self = (uniform MyRenderer *uniform) _self;

  // get the primary ray
  Ray ray = sample.ray;

  // call recursive classicRT
```

```
  vec3f color = recursive_classicRT (self, model, ray, 0);

  sample.rgb = color;
  sample.alpha = 1.f;
  sample.z = ray.t;
}
```

Basically it calls a recursive function (`recursive_classicRT`) where we will code our shader.

The code of this function is now explained. Please note that some parts are commented out. These commented parts will not be described yet: they will be described later in the tutorial, basically when we uncomment them.

`recursive_classicRT()` starts by tracing the primary ray, i.e., intersecting it with the geometry:

```
// trace the ray
traceRay(model, ray);
```

The ray is then intersected with each light source. If it intersects a light source at a shorter distance than the intersection with the geometry, then IT IS THE LIGHT SOURCE that is visible, NOT THE GEOMETRY. The following code is that which allows the light sources to be directly visible (otherwise they would not appear in the images); these are LE paths.

```
for (uniform int i = 0; self->lights && i < self->numLights; i++) {
          const uniform Light *uniform l = self->lights[i];

          const Light_EvalRes l_eval = l->eval(l, dg, ray.dir, 0.f, ray.t);
          if (ne(l_eval.radiance, ZeroVec3)) {
             // returned radiance is not zero: LIGHT SOURCE INTERSECTED
             return(l_eval.radiance);
          }
      }
```

If no light source is visible than it is checked whether a geometric primitive was intersected. If not the background color is returned:

```
if (ray.geomID < 0) {  // No intersection
  color = make_vec3f(self->super.bgColor.x,self->super.bgColor.y,self->super.bgColor.z);
  return (color);
}
```

If an intersection was found, various parameters are computed (related to the intersection, such as normal, materials ID, texture coordinates, etc.) which are stored into the `DifferentialGeometry dg` structure. Then it retrieves some of the intersected material's properties, such as `Kd` (diffuse reflection

coefficient), `Ks` (specular reflection coefficient), `Ns` (specular reflection sharpness) and `d` (dissolved coefficient, which amounts to 1-transparency):

```
	postIntersect (model, &self->super, dg, ray, DG_NG | DG_NS | DG_NORMALIZE | DG_FACEFORWARD
| DG_TANGENTS | DG_COLOR | DG_TEXCOORD);
	MyMaterial *materials = (MyMaterial *) dg.material;
	if (materials) {
		foreach_unique (mat in materials) {
		  Kd = (mat ? mat->kd : make_vec3f (1.0f));
		  Ks = (mat ? mat->ks : make_vec3f (0.f));
		  ns = (mat ? mat->ns : 1.f);
		  d = (mat ? mat->d : 1.f);
		}
	} else {
		  Kd = make_vec3f (0.8f,0.5f,0.2f);
		  Ks = make_vec3f (0.f);
		  ns = 1.f;
		  d = 1.f;
	}
```

Then the ambient light `ambColor` is multiplied by `Kd` and the `color` for that primary rays is computed:

```
    // compute the ambient light contribution
    vec3f ambient_contribution;
    ambient_contribution = Kd * ambColor;
    color = color + ambient_contribution;
    return (color);
```

No recursion up to now. This is a really basic, ambient lighting only renderer.

Direct Lighting

Defining the light sources

On a really generic renderer the light sources should probably be loaded from some file. On this working version of the renderer the light sources are programmatically defined. Open the file `MyRenderer.cpp`. You can see that the `commit()` method calls a different function to define the light sources for each of the `cornell_VI2` or the `conference` scenes:

```
void MyRenderer::commit()
{
    vec3f ambColor = vec3f(0.2f);

    Renderer::commit();

    // define the lights according to the scene
    defineLightsCornell(this);
    //defineLightsConference(this);

    // now set lightPtr as the pointer to the lights
    // buffer that will be set via ispc::VI2Renderer_set
    void **lightPtr = lightArray.empty() ? nullptr : &lightArray[0];

    ispc::MyRenderer_set(getIE(), lightPtr, lightArray.size(), (ispc::vec3f&)ambColor);
}
```

Depending on the scene you want to render, you should comment/uncomment the appropriate lines above, such that you get the correct set of light sources. You must, obviously, rebuild (`make`) the solution afterwards.

Direct light: no shadows

Computing each light source contribution for a given intersection point implies selecting a point (sample) on the light source surface, evaluating that sample RGB weight and then multiplying this weight by the material's diffuse reflection coefficient and the cosine of the angle formed by the normal and the direction to the light source:

$$\text{light\_contrib} = \text{light\_weight} * K_d * \cos(\vec{N}, \vec{L})$$

Within `MyRenderer-T2.ispc` there is a direct illumination method. For each light source, it deterministically selects a sample on the light's surface center, computes the direction to that point and the sample's weight and multiplies by K_d and the cosine. Uncomment the invocation of this method in `recursive_classicRT()`:

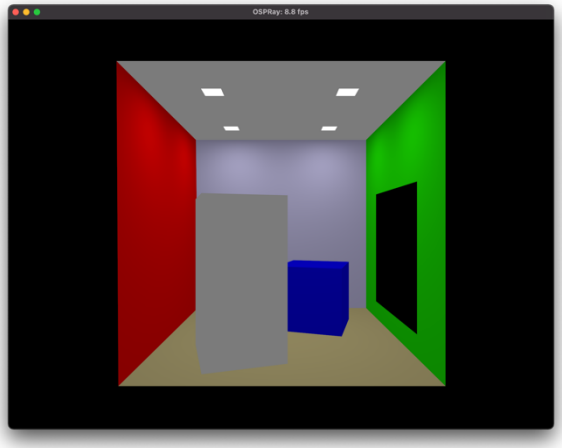
```
// direct illumination
color = color + direct_illumination (self, model, dg, Kd, ray.time);
```

Let's have a look at the direct illumination method:

```
vec3f direct_illumination (uniform MyRenderer *uniform self,
                           World *uniform model,
                           const DifferentialGeometry &dg,
                           const vec3f Kd,
                           const float Raytime) {

  vec3f color = make_vec3f(0.f);

  //calculate shading for all lights
  for (uniform int i = 0; self->lights && i < self->numLights; i++) {
    const uniform Light *uniform l = self->lights[i];
    const vec2f s = make_vec2f(0.0f); // sample center of area lights

    const Light_SampleRes light = l->sample(l, dg, s);

    if (reduce_max(light.weight) > 0.f) {   // any potential contribution?
       float cosNL = dot(light.dir, dg.Ns);  // cosine between lighting dir and normal
       if (cosNL < 0.0f) continue;  // light on the other side

       const vec3f diffuse = Kd * cosNL;
       const vec3f light_contrib = diffuse * light.weight;
       color = color + light_contrib;
    }  // light weight > 0
  } // for all lights
  return (color);
}
```

Build the renderer and run it for the Cornell scene.

Then change the definition of the lights in the `commit()` method in file `MyRenderer.cpp`, such that the lights appropriate for the conference scene are used. Rebuild the renderer and run it for the conference scene. You should get the following output:



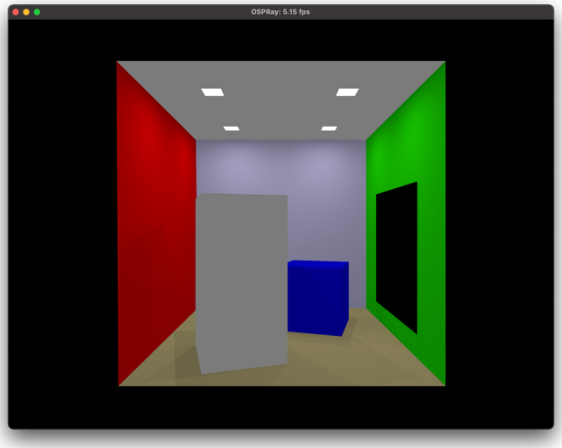
Direct light: enabling shadows

In order to include shadows the above direct lighting equation has to include the visibility between the point being shaded $\dot{p}$ and the point selected on the light source $\dot{l}$, $\text{visible}(\dot{p}, \dot{l})$. If there is a geometric primitive occluding $\dot{l}$ from $\dot{p}$ then visibility is 0, otherwise visibility is 1:

$$\text{light\_contrib} = \text{light\_weight} * K_d * \cos(\vec{N}, \vec{L}) * \text{visible}(\dot{p}, \dot{l})$$

Uncomment the appropriate lines on `direct_illumination()`; these are very clearly commented in the code. Note that a shadow ray is shot towards THE CENTER of each light source in order to assess its visibility.

Recompile the renderer and run the application for both scenes (do not forget to set the appropriate lighting in `MyRenderer.cpp` and rebuild the renderer in each case). You should get the following images:



Comment the resulting images:

- What happened to the frame rate compared to the no shadows version? Why?
- Why are all shadows hard shadows (in opposition to soft shadows, which include, besides the umbra, a penumbra)?

Specular Reflections

Specular reflections are simulated by spawning a secondary ray at each intersection point (referred to as reflection ray) and recursively calling the same function `recursive_classicRT()`. This recursive chain of calls must have some maximum allowable depth to ensure the algorithm terminates.

In `recursive_classicRT()` uncomment the lines below:

```
if ((depth+1) < self->super.maxDepth) {
  vec3f secondary = secondary_classic (self, model, dg, ray, Ks, d, depth+1);
     color = color + secondary;
}
```

Is it clear for you how this recursion process terminates?

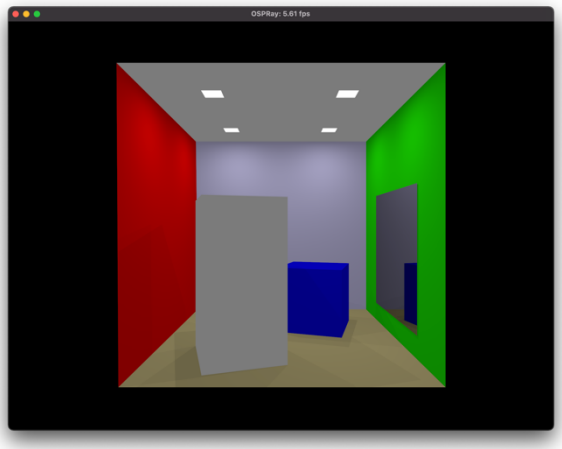
Now have a look at the `secondary_classic ()` function (repeated below). Note that if K_s is larger than 0 then it computes the reflection direction, sets up the reflection ray, recursively calls `recursive_classicRT()` for computing this ray's radiance, computes its contribution by multiplying by K_s and the respective cosine and then adds this value to color.

```
vec3f secondary_classic (uniform MyRenderer *uniform self, World *uniform model,
```

```
                    const DifferentialGeometry &dg, Ray &ray, vec3f Ks, int depth) {

        vec3f color = make_vec3f (0.f);
        // Specular reflection
        if (reduce_max(Ks) > 0.f) {   // any potential contribution from specular ray ?
               Ray specRay;
               // compute reflection direction
               vec3f wi = 0.f - ray.dir;
               vec3f Rs = 2.f * dot (dg.Ns, wi) * dg.Ns - wi;
               // set reflection ray
               setRay(specRay, dg.P, Rs, dg.epsilon, inf, ray.time);
               vec3f specularC = recursive_classicRT (self, model, specRay, depth);
               color = color + Ks * specularC * dot(dg.Ns, Rs) ;
        }  // spec ray
        return (color);
}
```

Rebuild and render both scenes (do not forget to set the appropriate lighting).

- Can you see the specular reflections?
- Do you think the materials are appropriately defined for the conference scene?
- What are the frame rates?



That's all, folks!

Universidade do Minho