50.017 Graphics and Visualization

Assignment 4 – Ray Tracing

Handout date: 2022.07.07

Submission deadline: 2022.07.18, 11:59 pm

**Late submissions are not accepted**

A picture containing ball

Description automatically generatedA picture containing light

Description automatically generatedA picture containing knife

Description automatically generatedA picture containing object, clock

Description automatically generatedA close up of an animal

Description automatically generatedA picture containing sitting, food

Description automatically generated

Figure 1. From left to right and then top to bottom: expected rendering outputs of scene 1-6.

In this assignment, you will implement a basic ray tracer. As seen in lecture, a ray tracer sends a ray for each pixel and intersects it with all the objects in the scene. Your ray tracer will support perspective cameras as well as several primitives (spheres, planes, and triangles). You will also have to support Phong lighting.

**1. User Interface**

In this assignment, there is no longer an interactive window for manipulating and rendering 3D models. Instead, the program takes a scene file that describes the camera, lights and objects information as input. The output is a rendered image saved at a user-specified path.

Your program should take a number of command line arguments to specify the input file, output image size, and output file. To run your program on all the six test cases, you can type commands as follows:

*YourProgramName -input scene01\_plane.txt -size 200 200 -output scene\_1.bmp*

*YourProgramName -input scene02\_cube.txt -size 200 200 –output scene\_2.bmp*

*YourProgramName -input scene03\_sphere.txt -size 200 200 -output scene\_3.bmp*

*YourProgramName -input scene04\_axes.txt -size 200 200 -output scene\_4.bmp*

*YourProgramName -input scene05\_bunny\_1k.txt -size 200 200 -output scene\_5.bmp*

*YourProgramName -input scene06\_shine.txt -size 200 200 -output scene\_6.bmp*

**2. Basic Ray Tracer**

You will use object-oriented design to make your ray tracer flexible and extendable. A generic Object3D class will serve as the parent class for all 3D primitives. You will derive subclasses (such as Sphere, Plane, Triangle, Group, Transform, Mesh) to implement specialized primitives. Also, this assignment requires the implementation of a material class for Phong lighting computation.

**2.1 Ray Tracing Pipeline**

**Main**. The main function in main.cpp reads the scene (using the SceneParse class), loops over the pixels in the image plane, generates a ray using your camera class, intersects it with the high-level Group that stores the objects of the scene, and writes the color of the closest intersected object.

Note that SceneParse is completely implemented for you. Use it to load the camera, background color and objects of the scene from scene files.

**2.2 Ray Generation**

**Camera**. In PerspectiveCamera class that inherits Camera, the field of view is specified with an angle (as shown in Figure 2). Here *up* and *direction* are not necessarily perpendicular. The , , vectors are computed using cross products. , , . The camera does not know about screen resolution. Image resolution should be handled in your main loop.

![A close up of a logo

Description automatically generated]()

Figure 2. Perspective virtual camera.

The Camera class has two pure virtual methods:

virtual Ray generateRay( const Vector2f& point ) = 0;

virtual float getTMin() const = 0;

The first is used to generate rays for each screen-space coordinate, described as a Vector2f. The getTMin() method will be useful when tracing rays through the scene. For a perspective camera, the value of tmin will be zero to correctly clip objects behind the viewpoint (already implemented for you).

**2.3 Scene Objects**

**Object3D**. Look at the virtual Object3D class. It only provides the specification for 3D primitives, and in particular the ability to be intersected with a ray via the virtual method:

virtual bool intersect(const Ray& r, Hit& h, float tmin) = 0;

Since this method is pure virtual for the Object3D class, the prototype in the header file includes ‘= 0’. This ‘= 0’ tells the compiler that Object3D won’t implement the method, but that subclasses derived from Object3D must implement this routine. An Object3D stores a pointer to its Material type. The Object3D class has:

* a default constructor and destructor
* a pointer to a Material instance
* a pure virtual intersection method

**Group**. Fill in Group, also a subclass of Object3D, that stores a list of pointers to Object3D instances. For example, it will be used to store all objects in the entire scene. The bool intersect( const Ray& r , Hit& h , float tmin ) function of Group loops through all these instances, calling their intersection methods. The Group constructor should take as input the number of objects under the group. The group also includes a method to add objects.

**3. Your Tasks**

**3.1 Ray Intersection**

**Sphere**. Fill in Sphere, a subclass of Object3D, that additionally stores a center point and a radius. The Sphere constructor will be given a center, a radius, and a pointer to a Material instance. The Sphere class implements the virtual intersect method mentioned above (but without the ‘= 0’):

virtual bool intersect(const Ray& r, Hit& h, float tmin);

With the intersect routine, we are looking for the closest intersection along a Ray, parameterized by t.tmin is used to restrict the range of intersection. If an intersection is found such that t > tmin and t is less than the value of the intersection currently stored in the Hit data structure, Hit is updated as necessary. Note that if the new intersection is closer than the previous one, t, Material, and Normal must be modified. It is important that your intersection routine verifies that t >= tmin. Note that tmin is not modified by the intersection routine.

**Plane**. Implement Plane, an infinite plane primitive derived from Object3D. The constructor is assumed to be:

Plane( const Vector3f& normal, float d, Material\* m );

*d* is the offset from the origin, meaning that the plane equation is .

Implement the virtual intersect method mentioned above (but without the ‘= 0’), and remember that you also need to update the normal stored by Hit, in addition to the intersection distance and color.

virtual bool intersect(const Ray& r, Hit& h, float tmin);

**Triangle**. Fill in triangle primitive which also derives from Object3D. The constructor takes 3 vertices:

Triangle( const Vector3f& a, const Vector3f& b, const Vector3f& c, Material\* m );

Use the method of your choice to implement the ray-triangle intersection, preferably using barycentric coordinates. Suppose we have barycentric coordinates , , and vertex normals , , , the interpolated normal can be computed as .

**Transform**. Fill in subclass Transform from Object3D. Similar to a Group, a Transform will store a pointer to an Object3D (but only one, not an array). The constructor of a Transform takes a matrix as input and a pointer to the Object3D modified by the transformation:

Transform( const Matrix4f& m, Object3D\* o );

The intersect routine will first transform the ray, then delegate to the intersect routine of the contained object. Make sure to correctly transform the resulting normal according to this rule: if we transform a vertex by a homogeneous transformation matrix **M**, its normal should be transformed by the matrix . A discussion of why this is the case appears in this wiki page: https://en.wikipedia.org/wiki/Normal\_(geometry) (see Transforming Normals section).

You may choose to normalize the direction of the transformed ray or leave it un-normalized. If you decide not to normalize the direction, you might need to update some of your intersection code.

**3.2 Lighting Computations**

**Material**. Implement the diffuse and specular components of Phong lighting in Material class. You may refer to Lecture 11 for the equations of these two components.

Vector3f Shade( const Ray& ray, const Hit& hit, const Vector3f& dirToLight, const Vector3f& lightColor )

**Light**. We provide the pure virtual Light class and two subclasses: directional light and point light. Scene lighting can be accessed with

SceneParser::getLight() and

SceneParser::getAmbientLight() methods.

Use the Light method:

void getIllumination( const Vector3f& p, Vector3f& dir, Vector3f& col);

to find the illumination at a particular location in space. *p* is the intersection point that you want to shade, and the function returns the normalized direction toward the light source in dir and the light color and intensity in col.

**3. Hints**

• Incremental debugging. Implement and test one primitive at a time. Test one lighting component at a time: ambient, diffuse, and specular.

• Use a small image size for faster debugging. pixels is usually enough to realize that something might be wrong.

• As usual, don’t hesitate to print as much information as needed for debugging, such as the direction vector of the rays, the hit values, etc.

• Use assert() to check function preconditions, array indices, etc. See cassert.

• The “very large” negative and positive values for *t* used in the Hit class and the intersect routine can simply be initialized with large values relative to the camera position and scene dimensions. However, to be more correct, you can use the positive and negative values for infinity from the IEEE floating point standard.

• Parse the arguments of the program in a separate function. It will make your code easier to read.

• Implement the normal visualization and Phong lighting before the transformations.

• Use the various rendering modes (normal, diffuse, distance) to debug your code. This helps you locate which part of your code is buggy.

**4. Grading**

Each part of this assignment is weighted as follows:

* Ray Intersection: 70%
  + Sphere: 15%
  + Plane: 15%
  + Triangle: 20%
  + Transform: 20%
* Lighting Computations: 30%

**5. Submission**

A .zip compressed file renamed to AssignmentN\_Name\_I.zip, where N is the number of the current assignment, Name is your first name, and I is the number of your student ID. It should contain only:

* The **source code** project folder (the entire thing).
* A **readme.txt file** containing a description of how you solved each part of the assignment (use the same titles) and whatever problems you encountered. If you know there are bugs in your code, please provide a list of them, and describe what do you think caused it if possible. This is very important as we can give you partial credit if you help us understand your code better.
* Six **screenshots** clearly showing that you can render images of the six test scenes with ray tracing.

Upload your zipped assignment to e-dimension. Late submissions receive 0 points!