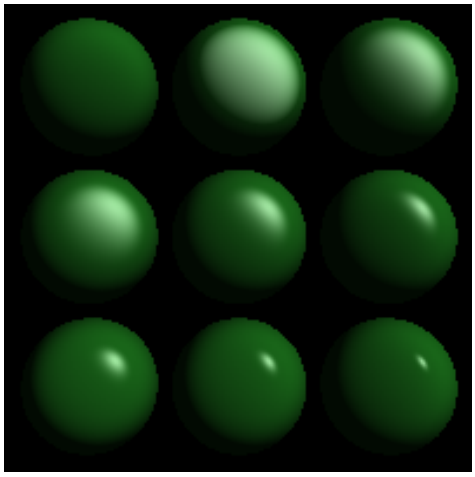
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

ISTD 50.017 Graphics & Visualization

Spring Semester 2015 (Term 7)



Assignment 4 : Ray Casting

In this assignment, you will implement a ray caster. This will be the basis of your final assignment, so proper code design is quite important. As seen in class, a ray caster sends a ray for each pixel and intersects it with all the objects in the scene. Your ray caster will support perspective cameras as well as several primitives (spheres, planes, and triangles). You will also have to support phong shading and texture mapping.

1. Getting Started

Note that this assignment is non-trivial. Please start as early as possible. One significant way in which this assignment differs from previous assignments is that you start off with a lot less starter code.

Run the sample solution **a4soln** from the starter code (Xcode build) as follows:

```
./a4soln -input scene01_plane.txt -size 200 200 -output output01.bmp -depth 8 12 depth01.bmp
```

These will generate an image named output01.bmp with different effects. We will describe the rest of the command-line parameters later. When your program is complete, you will be able to render this scene as well as well as the other test cases given below.

2. Summary of Requirements

This section summarizes the core requirements of this assignment. There are a lot of them and you should start early. Let's walk through them.

You will use object-oriented design to make your ray caster 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. Similarly, this assignment requires the implementation of a general **Camera** class with perspective camera subclasses.

You will implement Phong shading with texture mapping. We will focus on diffuse and specular shading. Diffuse shading is our first step toward modeling the interaction of light and materials. Specular shading will be explained later when you get the basic components working. Given the direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$ we can compute the diffuse shading as a clamped dot product:

$$d = \begin{cases} \mathbf{L} \cdot \mathbf{N} & \text{if } \mathbf{L} \cdot \mathbf{N} > 0 \\ 0 & \text{otherwise} \end{cases}$$

If the object has diffuse color $k_d = (r, g, b)$ (in case the object has texture, just use the texture color instead), and the light

source has color $c_{light} = (L_r, L_g, L_b)$, then the pixel color is $c_{pixel} = (rL_rd,\ gL_gd,\ bL_bd)$. Multiple light sources are handled by simply summing their contributions. We can also include an ambient light with color $c_{ambient}$, which can be very helpful for debugging. Without it, parts facing away from the light source appear completely black. Putting this all together, the formula is:

$$c_{pixel} = c_{ambient} * k_a + \sum_i \left[clamp(\mathbf{L}_i \cdot \mathbf{N}) * c_{light} * k_d \right]$$

Color vectors are multiplied term by term. Note that if the ambient light color is $(1, 1, 1)$ and the light source color is $(0, 0, 0)$, then you have constant shading.

You may optionally implement two visualization modes. One mode will display the distance t of each pixel to the camera. The other mode is a visualization of the surface normal. For the normal visualization, you will simply display the absolute value of the coordinates of the normal vector as an (r, g, b) color. For example, a normal pointing in the positive or negative z direction will be displayed as pure blue $(0, 0, 1)$. You should use black as the color for the background (undefined normal).

Your code will be tested using a script on all the test cases below. Make sure that your program handles the exact same arguments as the examples below.

3. Starter Code

(As always, you can add files or modify any files. You can even start from scratch.)

The **Image** class is used to initialize and edit the RGB values of images. Be careful—do not try to read or write to pixels outside the bounds of the image. The class also includes functions for saving simple **.bmp** image files (and **.tga** files).

For linear algebra, you should use the **vecmath** library that you are familiar with from previous assignments.

We provide you with a **Ray** class and a **Hit** class to manipulate camera rays and their intersection points, and a skeleton **Material** class. A **Ray** is represented by its origin and direction vectors. The **Hit** class stores information about the closest intersection point, normal, texture coordinates, the value of the ray parameter t and a pointer to the **Material** of the object at the intersection. The **Hit** data structure must be initialized with a very large t value (try **FLT_MAX**). It is modified by the intersection computation to store the new closest t and the **Material** of intersected object.

Your program should take a number of command line arguments to specify the input file, output image size and output file. Make sure the examples below work, as this is how we will test your program. A simple scene file parser for this assignment is provided. Several constructors and the **Group::addObject** method you will write are called from the parser (and will be a source for many compilation errors initially). Look in the **scene_parser.cpp** file for details.

If you're interested, the scene description file grammar used in this assignment is included in the source file distribution.

4. Implementation Notes

This is a very large assignment. We can't repeat this enough. Here's a suggested recipe to follow to get as far as possible, as quickly as possible.

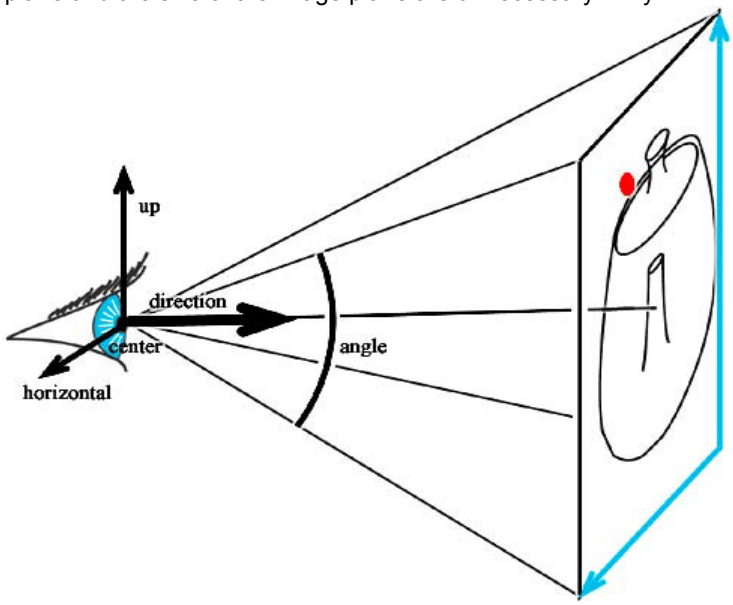
1. Look at the virtual **Object3D** class (a virtual class in C++ is like an abstract class in Java). It only provides the specification for 3D primitives, and in particular the ability to be intersected with a ray via the virtual method:
   virtual bool intersect(const Ray& r, Hit& h, float tmin) = 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

2. 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 intersect** method mentioned above (but without the **'=0';**): **virtual bool intersect(const Ray& r, Hit& h, float tmin);**
   With the intersect roWith 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, both t and **Material** must be modified. It is important that your intersection routine verifies that t >= **tmin**. **tmin**is not modified by the intersection routine.

3. 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. You'll need to write the intersect method of **Group** which 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 should include a method to add objects: **void addObject(int index, Object3D* obj);**

4. Look at the pure virtual **Camera** class (in Java parlance, an interface). 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 provided).

5. Fill in **PerspectiveCamera** class that inherits **Camera**. Choose your favorite internal camera representation. The scene parser provides you with the center, direction, and up vectors. The field of view is specified with an angle (as shown in the diagram). **PerspectiveCamera(const Vector3f& center, const Vector3f& direction, const Vector3f& up, float angle);** Here up and direction are not necessarily perpendicular. The **u, v, w** vectors are computed using cross products. $\mathbf{w} = direction$, $\mathbf{u} = \mathbf{w} \times up$, $\mathbf{v} = \mathbf{u} \times \mathbf{w}$. The camera does not know about screen resolution. Image resolution should be handled in your main loop.
Hint: In class, weHint: In class, we often talk about a "virtual screen" in space. You can calculate the location and extents of this "virtual screen" using some simple trigonometry. You can then sample points on the virtual screen. Direction vectors can then be calculated by subtracting the camera center point from the screen point. Don't forget to normalize! In contrast, if you iterate over the camera angle to obtain your direction vectors, your scene will look distorted -especially for large camera angles, which will give the appearance of a fisheye lens. Note: the distance to the image plane and the size of the image plane are unnecessary. Why?



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

7. Write the **main** function that reads the scene (using the parsing code provided), 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. Up to this point, you may choose to render some spheres with a single color. If there is an intersection, use one color and if not, use another color.

8. (Optional, but good for debugging) Implement an alternative rendering style to visualize the depth t of objects in the scene. Two input depth values specify the range of depth values which should be mapped to shades of gray in the visualization. Depth values outside this range should be clamped.

9. Update your sphere intersection routine to pass the correct normal to the **Hit**.

10. (Optional) Implement normal visualization. Add code to parse an additional command line option **-normals <normal_file.bmp>** to specify the output file for this visualization.

11. Implement diffuse shading in **Material** class, ignoring textures for now. We provide the pure virtual **Light** class and two subclasses: directional light and point light. Scene lighting can be accessed with the **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**.

12. Implement **Plane**,an infnite plane primitive derived from **Object3D**. Use the representation of your choice, but 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 $\mathbf{P} \cdot \mathbf{n} = d$. You can also implement other constructors(e.g., using 3 points). Implement **intersect**, and remember that you also need to update the normal stored by **Hit**, in addition to the intersection distance t and color.

13. Fill in triangle primitive which also derives from **Object3D**. The constructor takes 3 vertices:
Triangle(const VeTriangle(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 λ_0, λ_1, λ_2 and vertex normals $\mathbf{n}_0$, $\mathbf{n}_1$, $\mathbf{n}_2$, the interpolated, the interpolated, the interpolated normal can be computed as
$$\lambda_0\mathbf{n}_0 + \lambda_1\mathbf{n}_1 + \lambda_2\mathbf{n}_2$$
Texture coordinates can be interpolated in the same way.

14. 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 4 × 4 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 the rule seen in lecture. 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. Instancing.

15. Implement specular component in the Phong shading model. This is as simple as adding another formula. The intensity c_s depends on four q depends on four quantities: shininess s, ray direction $\mathbf{d}$, direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$. We can first compute the direction $\mathbf{R}$ of the reflected ray using $\mathbf{d}$ and $\mathbf{N}$. The specular sha. The specular shading intensity is another clamped dot product:
$$c_s = \begin{cases} (\mathbf{L} \cdot \mathbf{R})^s & \text{if } \mathbf{L} \cdot \mathbf{R} > 0 \\ 0 & \text{otherwise} \end{cases}$$
If the object has specular color $k_s = (s_r, s_g, s_b)$, and the light source has color $c_{light} = (L_r, L_g, L_b)$, then the pixel color is $c_{pixel} = (s_r L_r c_s, s_g L_g c_s, s_b L_b c_s)$ Combining this with diffuse and ambient, the formula is:
$$c_{pixel} = c_{ambient} * k_a + \sum_i \left[clamp(\mathbf{L}_i \cdot \mathbf{N}) * c_{light} * k_d + clamp(\mathbf{L}_i \cdot \mathbf{R})^s * c_{light} * k_s \right].$$

16. Texture mapping. This is the last item. The scene parser will load texture and coordinates for you. We also provide a **Texture** class that facilitates texture look-up. All that remains are interpolating texture coordinate in **Triangle**'s intersect function, and looking up texture coordinates in **Material**'s Shade function. If the material has valid texture indicated by **t.valid()**, then simply use the texture color instead of k_d. The texture color can be retrieved by
Vector3f color = t(u,v);
where **u, v** is the texture coordinate.

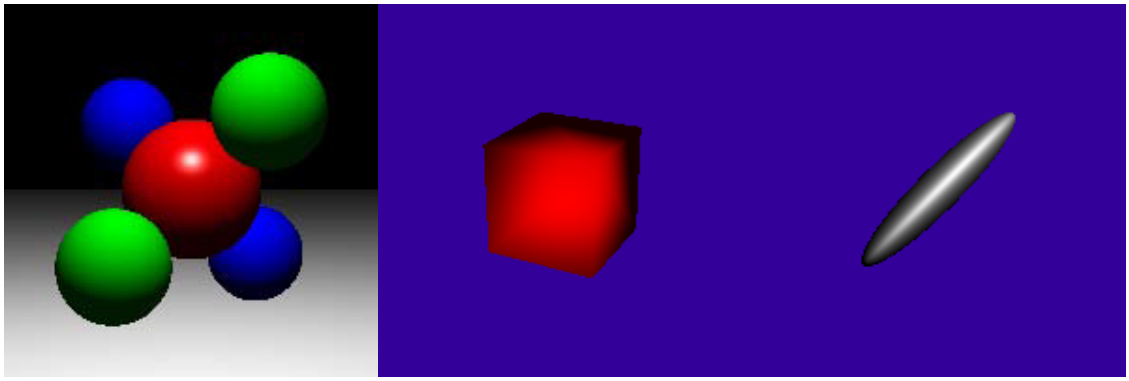
5. Test Cases

Your assignment will be graded by running a script that runs these examples below. Make sure your ray caster produces the same output (up to visual perception).

./a4 -input scene01_plane.txt -size 200 200 -output 1.bmp

./a4 -input scene02_cube.txt -size 200 200 -output 2.bmp

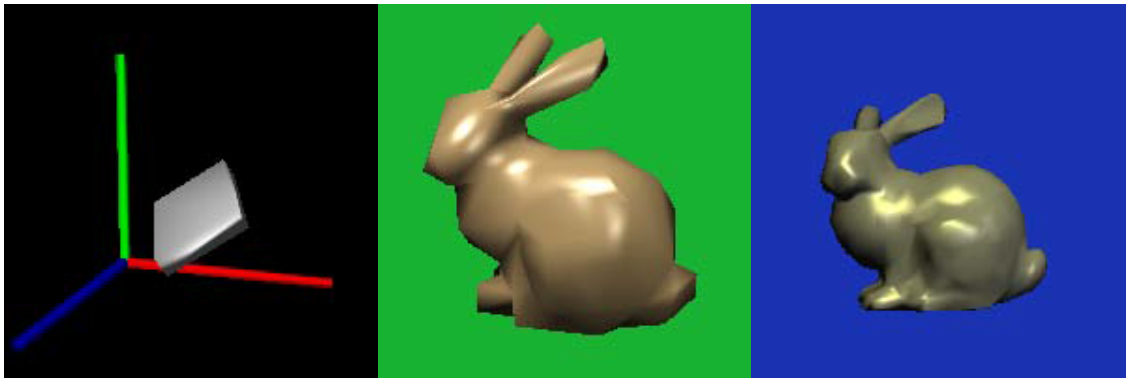
./a4 -input scene03_sphere.txt -size 200 200 -output 3.bmp



./a4 -input scene04_axes.txt -size 200 200 -output 4.bmp

./a4 -input scene05_bunny 200.txt -size 200 200 -output 5.bmp

./a4 -input scene06_bunny 1k.txt -size 200 200 -output 6.bmp



./a4 -input scene07_shine.txt -size 200 200 -output 7.bmp

./a4 -input scene08_c.txt -size 200 200 -output 8.bmp

./a4 -input scene09_s.txt -size 200 200 -output 9.bmp

6. Hints

- Incremental debugging. Implement and test one primitive at a time. Test one shading a time. Ambient, diffuse, specular, and then texture.
- Use a small image size for faster debugging. 64 × 64 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. When you get weird results, don't hesitate to use simple cases, and do the calculations manually to verify your results. Perhaps instead of casting all the rays needed to create an image, just cast a single ray (and its subsequent recursive rays).
- Modify the test scenes to reduce complexity for debugging: remove objects, remove light sources, change the parameters of the materials so that you can view the contributions of the different components, 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 diffuse shading 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.

7. Extra Credits (ISTD)

Note that there isn't much extra credit for this assignment. That's because we want you to focus on a good design so that your code will survive not only this assignment but the next one as well.

7.1 Easy

Add simple fog to your ray tracer by attenuating rays according to their length. Allow the color of the fog to be specified by the user in the scene file.

Add other types of simple primitives to your ray tracer, and extend the file format and parser accordingly. For instance, how about a cylinder or cone? These can make your scenes much more interesting.

Add a new oblique camera type (or some other weird camera). In a standard camera, the projection window is centered on the z-axis of the camera. By sliding this projection window around, you can get some cool effects

7.2 Medium


Implement a torus or higher order implicit surfaces by solving for t with a numerical root finder.


Implement texture mapping for spheres. Render a bunch of planets in some space scene for example.


Implement environment mapping for objects (sphere maps or cube maps).


Load more interesing textured models and put them into new scenes. Note that the starter code only loads **.bmp** image files. The **Mesh** utility only loads **.obj** files with a single texture map.

7.2 Hard


Add normal mapping (aka bump mapping).


Bloom: render multiple passes and do some blurring.


Depth of field blurring. The camera can focus on some distance and objects out of focus are blurred depending on how far it is from the focal plane. It doesn't have to be optically correct, but it needs to be visually pleasing.

8. Submission

You are to write a README.txt (or optionally a PDF) that answers the following questions:

- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.

- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. *This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.*
- Did you do any extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what and how you did it.
- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

Submit your assignment online. Please submit a single archive (.zip or .tar.gz) containing:

- Your source code.
- A compiled executable built from your code name `a4`.
- Any additional files that are necessary
- The README file.