Lab 2 (12%)

This lab will allow you to gain further understanding of PBRT shapes and ray-object intersection.

Objectives

- Develop hands-on experience by constructing a new shape class in PBRT.
- Gain familiarity with the basic concepts of ray-object intersection.

Step 0: Background and Requirements

Download PBRT:

- Make sure you have Version 3 of PBRT downloaded and configured on your machines.

Familiarize with PBRT's Directory Structure:

- Key directories of PBRT's source code include `src/core`, `src/shapes`, `src/integrators`, `src/cameras`, `src/lights`, etc.
- In `src/shapes`, PBRT has a small number of simple shapes, namely triangles and various simple quadrics. More complicated shapes can be approximated by building them out of these simple primitives.

Ray object intersection:

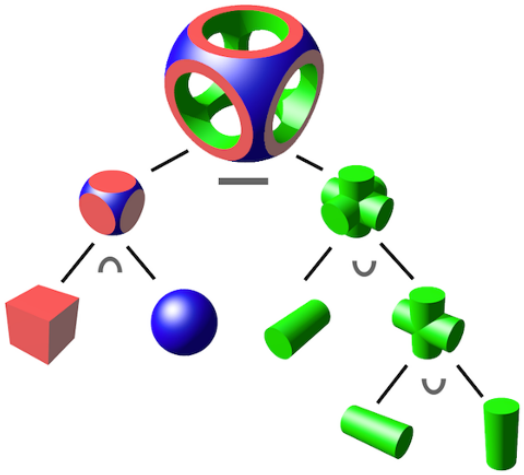
- Each time the camera generates a ray, the first task of the renderer is to determine which object, if any, that ray intersects first and where the intersection occurs.
- Read the example in our Intro slides or in Version 3 Chapter 1.2.2 Ray–Object Intersections

In this lab, we will extend PBRT to support a composite shape.

Requirements:

- Implement a new shape subclass.
- Use the new class directives in the `.pbrt` file to render this shape.
- Use your shapes and other PBRT shapes to construct a scene.

In this lab, **the new composite shape should consist of the union of three or more different types of PBRT objects**. An example is shown in the image below. Note that the creation of shapes through primitive difference and intersection is not required for this lab.



From "Constructive solid geometry" on Wikipedia

CSG is built on 3 primitive operations: intersection (∩), union (∪), and difference (−).

Object Union:

- For the union of two objects A and B, a ray must be considered to intersect the union if it intersects either A, B, or both.
- Here's a strategy to implement this:
  - **Cast the Ray**: Check for intersection with both A and B.
  - **Evaluate Intersections**: Determine the nearest intersection point, if any. This point can be from either A, B, or the closest of the two if both are intersected.
  - **Result**: If the ray intersects either object, use the nearest intersection point for further shading or calculations.

Step 1: Plan Your Composite Shape

- Decide on the composite shape you want to create. For example, I create a snowman consisting of two spheres.
- Your composite shape should consist of at least **three different** shape types, and you will utilize the Object Union for this design.

Step 2: Create the Composite Shape Class (6%)

- Extend PBRT with a new Shape subclass, for example, `SnowMan`. Unzip `lab2.zip`, move `snowman.pbrt` scene file into folder `scenes/`, and move `snowman.cpp` and `snowman.h` into `src/shapes/`.

- `SnowMan` is an example here, you should declare and implement your new class in: `shapes/your_class_name.h` and `shapes/your_class_name.cpp`.

- Start by taking a closer look to the existing shape classes in PBRT, such as `Sphere` and `Cylinder`. Understand how these classes inherit from the base `Shape` class and how they implement the `Intersect()`, `IntersecP()` and `ObjectBound()` method.

- Create a new C++ class that inherits from the `Shape` class in PBRT. For example, `SnowMan`. Add member variables to hold the instances of the basic shapes that compose your new shape.

- Implement the constructor for the new class, initializing the basic shapes with the appropriate transformations to position them.

- Implement four pure virtual functions from the `Shape` Class (`ObjectBound()`, `Intersect()`, `IntersectP()` and `Area()`), the rest have acceptable default implementations. We will also stub out the `Sample()` function: it is only needed for using the shape as an area light, which we will not do in this project.

    - `Intersect()` returns geometric information about a single ray–shape intersection corresponding to the first intersection, if any, in the `(0, tMax)` parametric range along the ray. If an intersection is found, its parametric distance along the ray should be stored in the `tHit` pointer. Information about an intersection is stored in the `SurfaceInteraction` structure, which captures the local geometric properties of a surface.
    - `IntersectP()` is used by PBRT when casting shadow rays. Since shadow rays only need to know if there was any hit before their `tMax` value, `IntersectP()` does not need the `tHit` value or a `SurfaceInteraction` describing the hitpoint.
    - `ObjectBound()` returns an axis-aligned bounding box in the shape's object space. Axis-aligned bounding boxes require only six floating-point values to store and fit many shapes well. Each `Shape` must implement an axis-aligned bounding box represented by a `Bounds3f`.
    - `Area()` function calculates the surface area of a shape. It is important for Light Sampling (affects the distribution of emitted light), Monte Carlo Integration (determine the probability distribution functions for sampling), etc. For simplicity, you can just sum the areas of all individual shapes and ignore any overlaps that might occur.

- For example, `SnowMan` is a union of two spheres. We can implement ray-object intersection in `SnowMan::Intersect()` by calling the `Intersect()` method of each sphere instance to determine if the ray intersects with any part of the composite shape.

    - If the ray hits both objects, the intersection would be the closest of the two. If it only has one intersection, that would be the point of intersection.
    - `Intersect()` method should return `true` if there is an intersection with any part of the composite shape, and `false` otherwise.
    - Remember to update `SurfaceInteraction *isect`.

- When you finish `Intersect()`, make sure to implement `IntersectP()`, `Area()`, and `ObjectBound()`. You don't need a real implementation of `Sample()`, youc can make one that returns the default `Interaction()` (like `Cone`'s implementation).

- To utilize the new class directives `snowman` in the `.pbrt` file, you should implement `CreateSnowManShape()` function by mirroring The `CreateSphereShape()` function (see `sphere.h` and `sphere.cpp`), and modify `MakeShapes()` in `api.cpp`

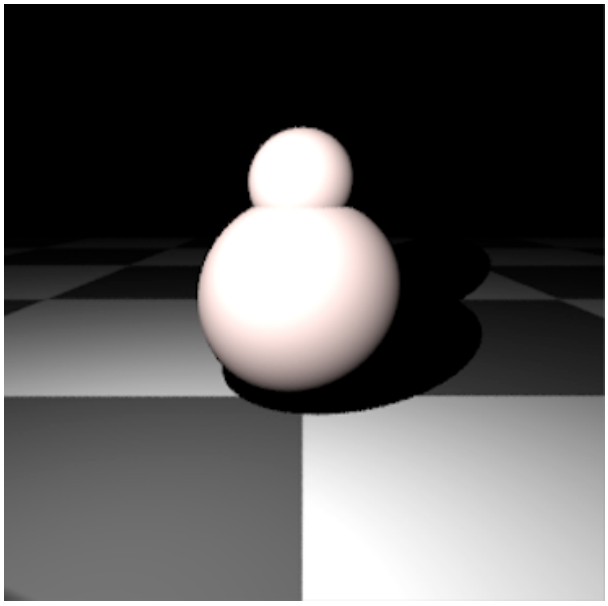
    - For your own composite shape class, you should implement `Create_your_class_name_Shape()` and modify `MakeShapes()` as well.

- Compile the PBRT source code with your new composite shape class. Create a scene file that includes the composite shape, for example in `scenes/snowman.pbrt`:

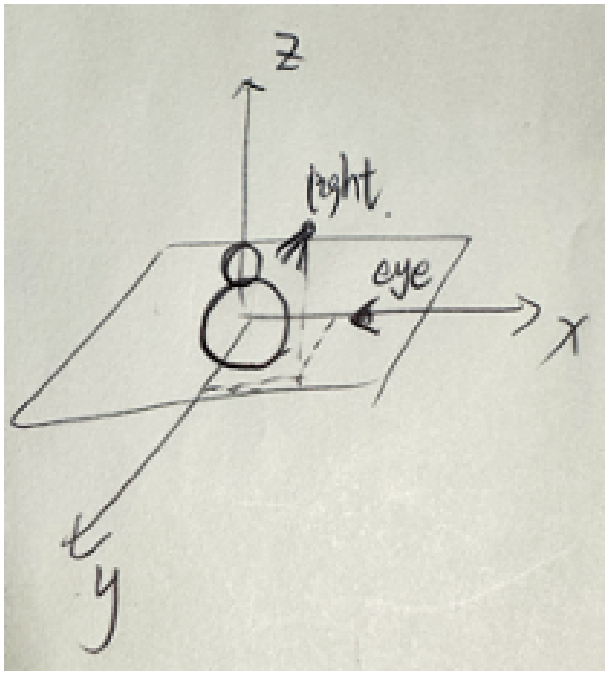
```
AttributeBegin
Material "matte" "spectrum Kd" [0.8 0.8 0.8]
Shape "snowman"
AttributeEnd
```

Step 3: Create your own scene (3%)

Create a scene where you will place your shape(s) at two different locations at least, among other objects of your choice. As an example, if you render the scene in `scenes/snowman.pbrt`, you will get the following image of a SnowMan. Your grade will depend on how much effort you put in creating and rendering the scene.



Create a diagram of your chosen composite shape(s), clearly illustrating how the basic shapes, light and camera are positioned relative to each other in your scene. Please submit this diagram with your code. Ensure your rendered image aligns with this diagram.



Step 4: Profilling (3%)

The goal of this step is to deepen your understanding of ray-object intersections. You will add code to PBRT to count the numbers of ray-object and ray-bounding boxes intersections.

Specifically, you should implement features to track the following metrics:

- The total number of rays cast for the whole scene.
- The number of rays that intersect with bounding boxes.
- The number of rays that intersect with any object.
- The number of rays that hit each single object within the composite shape.
- The number of rays that intersect with any object in each image tile.

For your information, `STAT_COUNTER` in PBRT can perform profiling, but you will need to implement it yourself by adding integer counters in proper locations of your revised PBRT and print them. The profiling statistics should be printed out right after the rendering is completed.

To make sure that your profiling code runs correctly, we will also use the `snowman.pbrt` file to test your executable. It is expected that your program should output the correct number for each of the categories for the standard snowman scene.

Step 5: Submission

- Submit a zip file containing:
  - A diagram of your composite shape in the third person perspective, illustrating the XYZ-axes, camera pose, positions of light, and the composite shape.
  - The scene file `your_class_name.pbrt` of your composite class, corresponding to your diagram.
  - The new composite class `your_class_name.h` and `your_class_name.cpp` files, any modified files such as `api.cpp`.
  - A rendered image of the scene
  - A report detailing the profiling analysis conducted in step 4.
  - Be sure to submit any comments or remarks in a `README.txt` file. `README.txt` should include your scene description and how to run your code. Also add information about any issues you encountered.
  - Points will be deducted for scenes that lack creativity or complexity.

Tips:

- Rerun `cmake` whenever you change the source code.
- Use debug prints in your `Intersect()` method to understand the flow of data.
- Pay attention to transformations. By default, a shape will be placed at the origin. To position it elsewhere, you must apply the appropriate transformation.