CS 7490, Spring 2016

Homework 3: Ray Tracing Acceleration

Due: 11:55pm, Friday, March 18, 2016

Objective

The goal of this project is to make your ray tracer more capable by adding an acceleration method. You may choose one approach out of three different methods of performing acceleration: 3D grids, kD-trees, or automatically generated bounding volume hierarchies. In addition to this, you will also implement object instancing and lists of objects. Finally, you will create a scene of your own that contains bunnies.

Scene Description Language Extensions

Below are the new commands that you will add to your ray tracer in order to give it new capabilities.

- **box xmin ymin zmin xmax ymax zmax**
  Create an axis aligned box. In addition to being able to render this box (just like a sphere or a triangle), you should also use such boxes as bounding boxes to enclose other geometry.

- **begin_list**
  Mark the beginning of a collection of objects that are to be placed in a list of objects. All objects that come after this command and before one of the end-of-list commands will go into this new list. There are two ways to end a list: 1) using "end_list", in which case these objects go into a flat list, and 2) using "end_accel", where the objects should be placed in a ray tracing acceleration data structure, such as a 3D grid, a bounding hierarchy, or a kD-tree.

- **end_list**
  Mark the end of a collection of objects that are to be placed in a list. Each such list should have a bounding box associated with it that surrounds all of the objects in the list. This means that rays that miss the bounding box do not get tested against the objects that are in the list.

- **named_object <name>**
  Gives the name of an object that may be instanced. The last object that was created prior to this command is the object that is named. You should allow naming of any type of object, including a triangle, a sphere, a list of objects, and an acceleration structure full of objects. This object is taken out of the global list of scene objects, and should be put in another location, waiting to be instanced. Only if this object is instanced should it be visible.

- **instance <name>**
  Create an instance of a given named object. The instance object should include a transformation matrix that is the inverse of the top of the stack. When you intersect a ray against this instance, first transform the ray by this stored matrix. After you find the

intersection between the transformed ray and the object being instanced, you should apply the adjoint of the matrix to the surface normal before you return it. It is important that you do **not** make a copy of all of the geometry inside the list that is being instanced. You should be able to define hundreds of instances of a complex object, and not incur a large storage cost.

- **end_accel**
  Mark the end of a list of objects, very similar to "end_list". Instead of creating a flat list of objects, however, this command will cause all of the objects to be placed in a ray tracing acceleration data structure. This may be a 3D grid, an automatically generated bounding hierarchy, or a kD-tree. There is no "begin_accel" command -- we will use the "begin_list" command to signify the start of both kinds of lists (flat or acceleration).

- **reset_timer**
  Reset a global timer that will be used to determine how long it takes to render a scene. Use the code snippet included in timer.txt for this.

- **print_timer**
  Print the time elapsed since reset_timer was called (in seconds). Use the code snippet included in timer.txt for this.

Create a Scene with Bunnies

The one final part of this assignment is for you to create your own scene that contains multiple bunnies. The description of your scene should be in one or more .cli files. The main .cli file for your scene should be called t10.cli, and your program should render this scene when the user types the zero key "0". Be sure to include a .png file with the resulting multiple-bunny image for this scene in your turned in assignment.

Your scene must include at least two visible full-sized bunnies (bun69k.cli). This is the only requirement of your scene. If you like, you can include more than two bunnies -- as many as you like. We will be viewing these images in class. In fact, we will have an informal competition for how many bunnies you can put in your scene that are visible. The winner gets bragging rights (no extra credit). Your classmates are also likely to respond better to an aesthetically pleasing image.

You can make your scene as complex or as simple as you like. So long as your scene shows two bunnies and uses proper .cli syntax you will get full credit for the scene (5% of this assignment grade).

Resources

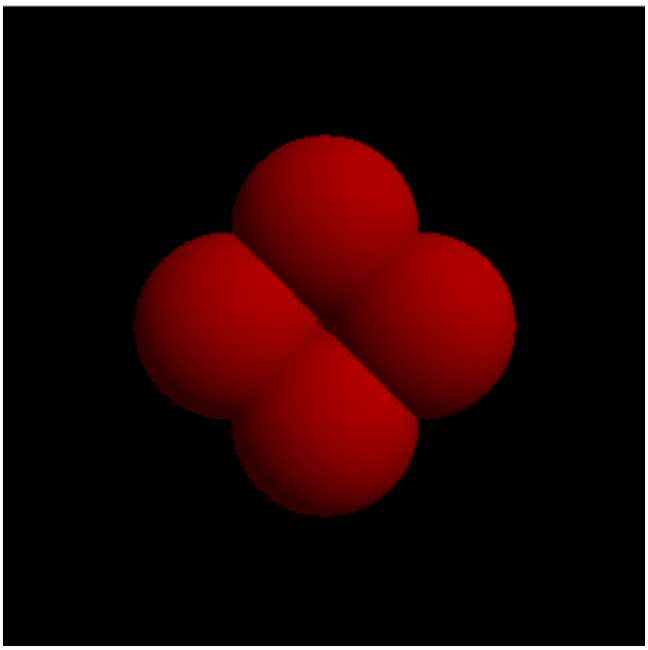
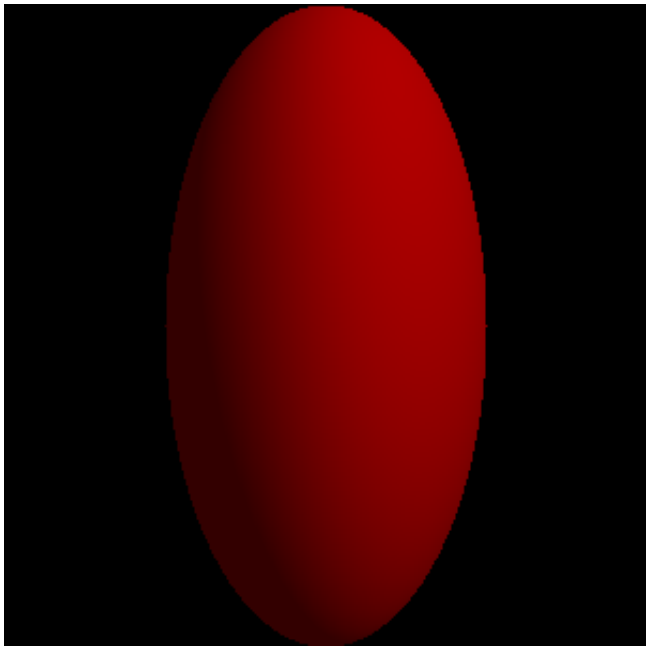
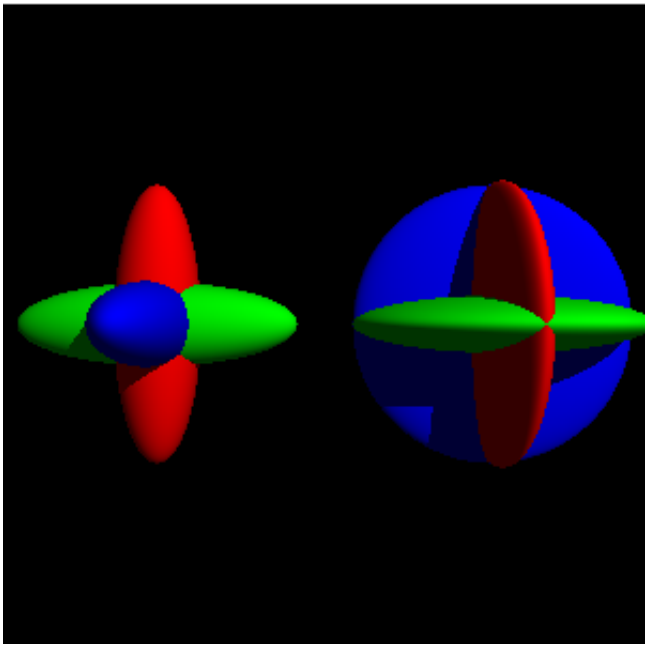
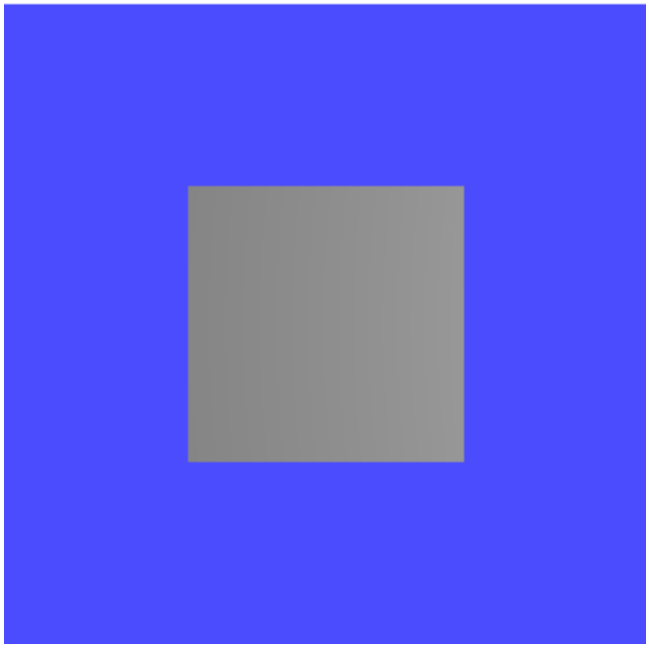
I recommend that you consult Chapter 4 of the book known as PBRT for detailed information about ray tracing acceleration structures. The link to this chapter is here.

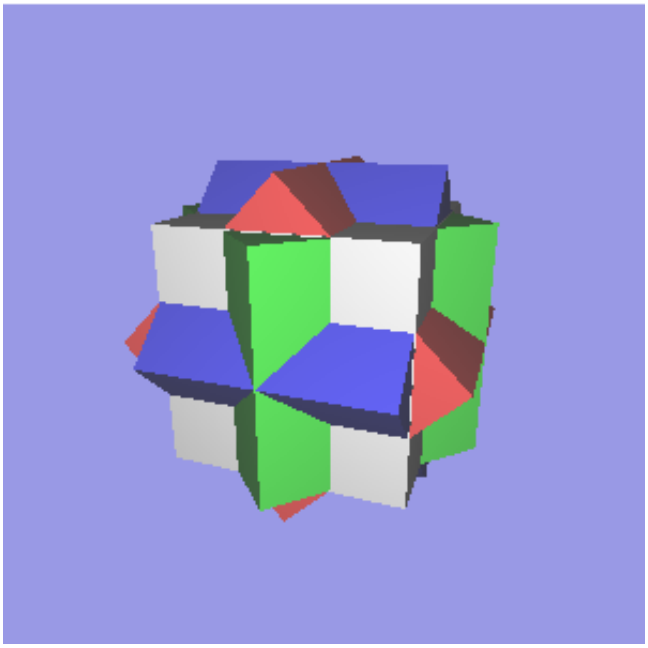
Another good reference is the Ray Tracing News, including the index of acceleration methods.

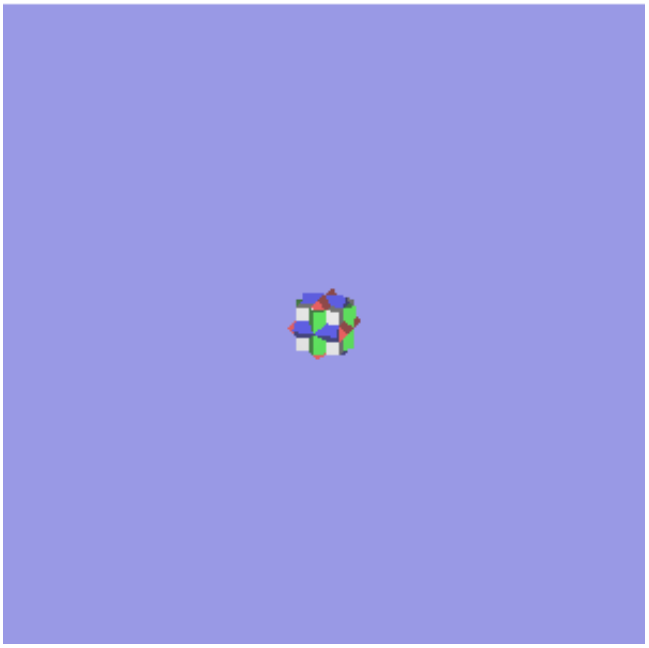
Scene Files

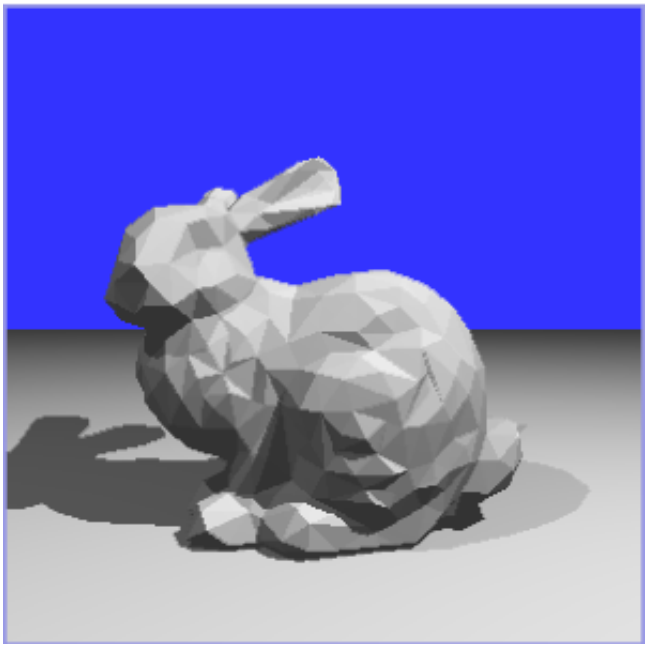
In the directory "**data**" are several test scenes that are described by .cli files. Also in that directory are the images that should be created by these scene files. The provided CLI and image files are in a zip file. Pressing keyboard keys 1-9 calls these test CLI files.

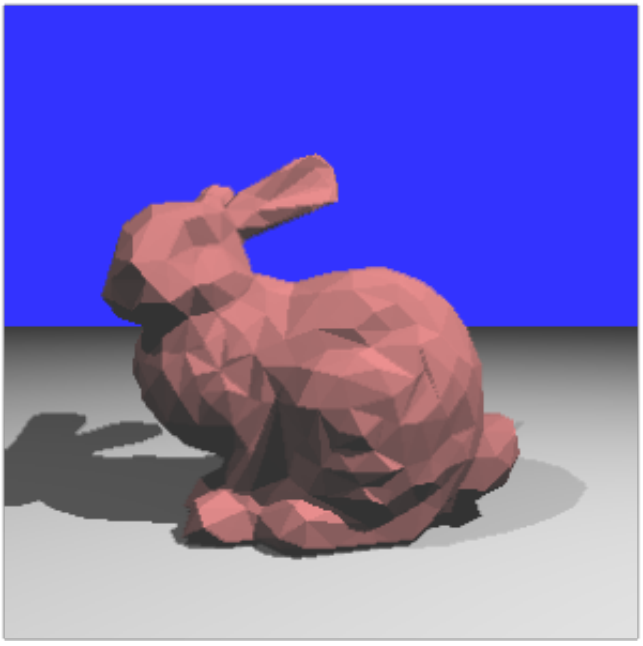
Sample Results:

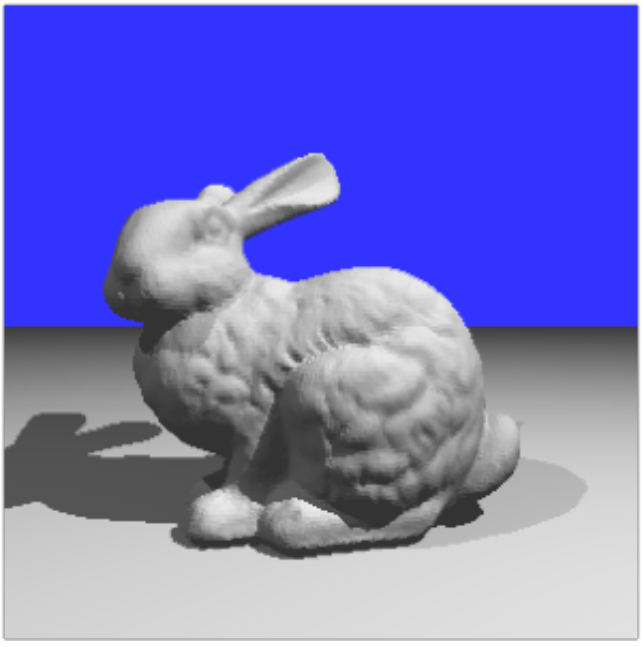















Suggested Approach

It is important for this assignment that you make full use of object-oriented programming. You should already have some parent class that has triangles and spheres as sub-classes. Let us call this parent class "Geometry". You will find it useful to have each sub-class of this class to be able to perform ray intersection with the given object. It will also be useful to have each sub-class return a bounding box for the given object. Also, you will probably want a bounding box to be a sub-class of Geometry. Moreover, lists of Geometry objects should also be sub-classes of Geometry. By allowing list of objects to be just another Geometry object, we can create lists of lists, lists of lists of lists, and so on. An instance of an object should also be a sub-class of Geometry. Finally, which ever acceleration structure you implement should be a sub-class of Geometry (3D grid, kD-tree, bounding volume hierarchy). Using this approach, you can mix and match different ways of putting together objects.

I recommend starting by working on instancing. One aspect of this is to keep all of the named objects in a global list, so that you can find the matching name for a given instance. When the "named_object" is called, you should remove this object from the collection of scene objects and move it to this named object list. Note that a named object is NOT drawn unless it is instanced. For a given instance of a named object, you will have to transform any incoming ray to this instance, and use the adjoint of the transformation to transform the surface normal. You can use the matrix inverse and transpose routines of Processing's Matrix3D class to help you get the adjoint (which is the inverse-transpose of the matrix). Test out the named object and instancing commands using the first three .cli files. If you implement ray transformations properly, you should be able to achieve non-uniform scaling.

Next, you should create an axis-aligned bounding box class, and make sure that it is just like another kind of Geometry. You should be able to intersect a ray with the bounding box, and return the nearest hit point. You can test out this using the file t04.cli, which contains a single box.

Next, you should start to implement lists of objects. To do this, you will probably want a global list of the current objects, and keep a stack pointers that indicate where the start of each list is within this current list. You start a new index on the stack of pointers when you see a "begin_list", and you do not actually copy the elements onto a new list until you see "end_list". Then this new list just becomes another object on the global list. The reason to use a stack of indices to the global list is so that you can create lists of lists (nested lists). Don't forget to associate a bounding box with each object list. That way, if a given ray does not hit the bounding box, you do not need to test the ray against any elements in this list. If you put all of the triangles and spheres from a scene into one big list, this is likely to speed up the rendering of most scenes. Why? Because some of the eye rays will entirely miss the bounding box for this one list, and thus will not be checked against any of the objects in the list. When properly implemented, you should find that scene t06 renders faster than scen t05, for precisely this reason.

The final piece of this project to work on is the ray tracing acceleration data structure. Note that the "end_accel" command is very similar to "end_list", except that it should move the objects on the list into your acceleration data structure (3D grid, bounding hierarchy, or kD-tree). Of course you must build your acceleration structure, and make use of it when you ray trace this collection. Using such a structure should give you a clearly noticeable speed-up when you render 100 objects or more. Your data structure should be able to handle large collections of objects, such as the triangles that make up the bunny model in the examples. The bunny in t09 is large enough that it is painful to wait for it to render if you do not use an acceleration structure (many minutes).

You will want to consider the ease of programming versus the likely speed benefit of the given

acceleration method. 3D grids are the easiest to implemenet. Bounding volume hierarchies are probably the next easiest. kD-trees are the most difficult to implement. The choice is yours which method you want to implement. All of them will be graded out of a possible 100%.

Authorship Rules

The code that you turn in must be entirely your own. You are allowed to talk to other members of the class and to the instructor about high-level questions about the assignment. You may not, however, use code that anyone other than yourself has written. Code that is explicitly **not** allowed includes code taken from the Web, from books, or from any source other than yourself. The only exception to this rule is that you should use the parser routines that we provide. You should not show your code to other students. If you need help with the assignment, seek the help of the instructor.

Development Environment

You must use the Processing language which is built on Java. Be sure that you are using Processing version 3.0 or higher. The best resource for Processing language questions is the online or offline Processing language API (found in the "reference" subdirectory of the Processing release).

What To Turn In

Compress the whole folder for this project (not merely the files within the folder) into a zip archive submit them to T-square. The zip archive should be included as an attachment. When unzipped, this should produce a folder containing all of your .pde files and a directory "data".
