

ROBOSIM GEOMETRY MODULE

by

RYAN SHELTON

B.S., Kansas State University, 2003

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2005

Approved by:

Major Professor
William H. Hsu, Ph.D.

ABSTRACT

The Geometry Module for RoboSim was made to fill the solid geometry needs of the simulator. RoboSim is a 3-D environment simulator for robotic application testing. To properly simulate environment objects, a robust collision detection package was required that could accurately and efficiently provide precise movement for robots and other objects. Although it was developed to fulfill a need in RoboSim, the geometry module has been designed to be general enough to meet rendering and animation requirements for future 3-D visualization environments. The geometry module is implemented with a bounding volume hierarchy using axis-aligned and oriented bounding volumes. Bounding volumes provide an efficient way to rule out non-collisions and find collisions within certain bounds. Using the Geometry Module, RoboSim is now ready for simulation with more complex robots, objects, and movements. The Geometry Module provides all the necessary support for creating the realistic environments that will make RoboSim an accurate simulator.

TABLE OF CONTENTS

Table of Contents.....	i
List of Figures.....	ii
Chapter 1: Introduction.....	1
Chapter 2: Geometry Concepts.....	4
Collision Detection.....	4
Spatial partitioning Trees.....	9
<i>Coldet</i> – A Collision detection Library.....	10
Chapter 3: Geometry Module Design and Implementation.....	12
Module Structure.....	13
Module Functions.....	14
Data Objects.....	15
Collision Detection.....	17
RoboSim Interface.....	18
Module Testing.....	19
Chapter 4: Conclusion.....	22
Bibliography.....	24
Appendix 1: Module Class Diagrams.....	25

LIST OF FIGURES

Figure 1.1. RoboSim Architecture.....	2
Figure 2.1. Sphere Intersection Test.....	6
Figure 2.2. Axis Aligned Bounding Box Intersection Test.....	7
Figure 3.1. Geometry Module Overview.....	13
Figure 3.2. Geometry Module Data Objects.....	16
Figure 3.3. The 2-D Geometry Viewer.....	20

CHAPTER 1: Introduction

The geometry module is a solid geometry API that can be used for collision detection and rendering. The module was created to fulfill a need in the RoboSim robotics simulation group, but it is a stand-alone package that can be used for a variety of applications. The module is designed to be flexible enough to use for real-time applications or high-precision simulations. The API has multiple modes that allow support for different end user needs. 3-D collision detection and ray casting are currently supported, but there are possibilities for expansion to include rendering with ray tracing and physically realistic movement.

RoboSim Overview

RoboSim is a software system for robot simulation developed at Kansas State University to provide a test bed for multi-agent systems. The project was started over 2 years ago as a collaborative effort by a group of professors and students. The system has progressed through many versions, but still is lacking the accuracy needed to simulate real world conditions. The RoboSim application exists as a collection of different modules that communicate over TCP/IP sockets. This provides a concurrent system that allows the application to run at a reasonable speed for more complex environments than a single threaded system. The central component of RoboSim is the environment module, which coordinates communication between all the modules. Previously, the environment was responsible for more of the computations such as collision detection and robot communication, but much of this has been off-loaded to other modules in the current version.

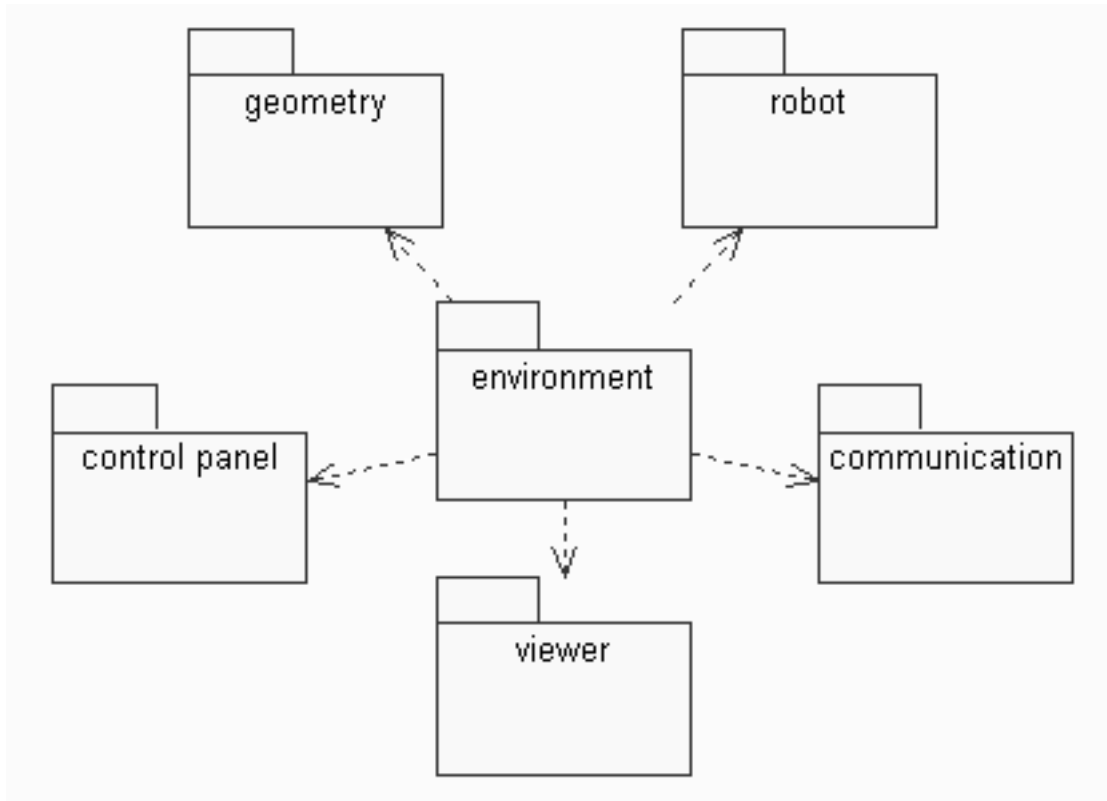


Figure 1.1. RoboSim Module Architecture

Figure 1.1 shows the separate modules and how they communicate with the environment module. The environment makes connections with all the other modules and handles message passing. The geometry module handles the representation and movement of objects as well as simulating sonar and laser range-finding devices. The robot module is used to emulate hardware robots and run the application code. The communication module provides a messaging system to allow robots to communicate with each other. The viewer module contains viewing applications that display the current state of the simulation. There is a simple 2-D viewer that shows a top-down view of the world and a 3-D viewer that has multiple camera angles and can show the world from a robot's

perspective. The control panel contains controls for setting the simulation parameters. In most simulations, only robots and the geometry are required to connect to the environment module to start a simulation. The communication module is not required unless the robots need the messaging system, and the simulation can run without a control panel or viewer if only console output is desired.

Once the robots connect to the environment they can request various actions. Robot actions include movement, sonar requests, and proximity sensors. The environment then queries the geometry module to obtain the results of the actions based on the current state of the world, returns those results to the robots, and updates the state of the viewer(s), if any are connected. Current RoboSim environments are three-dimensional, but the each robot's movement is limited to a single plane. Since only horizontal movement is used, full 3-D collision detection is not required at this point in the project. The geometry package runs in a 2-D mode where objects are specified in three dimensions, but movement calculations are done in two dimensions, which can greatly increase efficiency.

CHAPTER 2: Geometric Concepts

The geometry module makes use of several geometric concepts that are used to implement efficient collision detection and ray casting. First, there are the equations used for object-to-object intersection testing. These provide ways to determine the actual collisions between objects. Closely related to object-to-object testing is ray-to-object testing. Determining the intersection point of a ray with an object is integral to finding distances in the environment. To make all the operations execute in a timely manner, a space-partitioning scheme has been used to quickly narrow the scene down to only those objects that may be intersecting.

COLLISION DETECTION

Collision detection is an integral part of a believable simulated environment. Software can only render 2-D polygons based on vertex coordinates, so collision detection is required to give the illusion of solid objects. By determining when objects will collide with other objects during a simulation or interactive session, object intersections can be prevented that would be impossible with real-life objects. When an action will cause two objects in a scene to collide, the collision detection part of the program must recognize that a collision will occur and cause some kind of reaction that will prevent the objects from overlapping in the environment (Ericson, 2005).

The main problem with collision detection is producing a system that can run in a timely manner but still meets the application's demands for accuracy. Since different applications have different accuracy requirements, several ways to implement collision

detection have been developed over the years. All these implementations trade computation time for geometric accuracy and have several features in common. A widely used technique is to surround objects with simpler geometric shapes that are known as bounding volumes. A bounding volume can be a sphere, a box, a cylinder, a capsule, or any one of several other geometric volumes. Using bounding volumes simplifies collision computation because they involve simpler intersection tests than the actual object geometries. Bounding volumes usually enclose a larger volume than the object so sometimes a collision may be detected incorrectly.

Spherical Bounding Volumes

The simplest type of collision detection uses spherical bounding volumes. Spheres are the easiest to work with due to low memory requirements and simple intersection tests. A single point in the environment and the radius define a sphere, so only four values (radius, X coordinate, Y coordinate, Z coordinate) need to be stored to describe the volume. Spheres are also symmetric along all axes, so no rotations are required for intersection testing. Special considerations must be taken when determining a spherical bounding volume for an object, so the sphere fits around the object as tightly as possible. Even with tight fitting bounds, spheres will often encapsulate a much larger volume than the object, which leads to many false collisions being detected. Because of this, spherical bounding volumes are generally reserved for applications that use a large number of spherical objects or objects that are moving fast enough that the false collisions will not be noticeable to the user. Spherical volumes can also be used in groups to surround an

object. By having a collection of overlapping spheres be a bounding volume, you can achieve a much tighter fit while only increasing the computation by a linear factor.

```
boolean testIntersection(Sphere s1, Sphere s2) {  
  
    double distance = sqrt( s1.center.x * s2.center.x +  
                           s1.center.y * s2.center.y +  
                           s1.center.z * s2.center.z);  
  
    return distance <= s1.radius + s2.radius;  
  
}
```

Figure 2.1. Sphere Intersection Test

As shown in figure 2.1, the intersection equation for spheres is quite simple. By checking the distance between the centers with the sum of the radii, we can quickly determine if the spheres will intersect at any point in 3-D space. This simplicity makes spheres a highly desirable bounding volume from a computational standpoint.

Bounding Boxes

After spheres, boxes are next in the line of efficient bounding volumes. There are two types of algorithms that use bounding boxes for intersection testing. Axis-aligned bounding boxes (AABBs) are always aligned with the major axes, and oriented bounding boxes (OBBs) can be rotated in the environment to provide closer fitting bounds for objects (Ericson, 2005). A center point and three dimensions define AABBs. AABBs are often a much better fit than spherical volumes, and the computation is only slightly more complex.

```

boolean testIntersection(AABB b1, AABB b2) {

    double xdiff = abs( b1.center.x - b2.center.x );
    double ydiff = abs( b1.center.y - b2.center.y );
    double zdiff = abs( b1.center.z - b2.center.z );

    return (    xdiff <= ( b1.width + b2.width ) / 2 &&
              ydiff <= ( b1.height + b2.height ) / 2 &&
              zdiff <= ( b1.length + b2.length ) / 2 );

}

```

Figure 2.2. Axis Aligned Bounding Box Intersection Test

Figure 2.2 shows that finding AABB intersections is simply testing for separation along the three major axes. Even though AABBs are simple to test for intersection, there is still the issue of resizing the box during runtime if required. If the actual object is rotating in the environment, the AABB will have to be recomputed so that its faces form the tightest rectangular volume containing the object. This can become a problem in applications with large numbers of rotating objects. To remove the problem of dynamic resizing, oriented bounding boxes (OBBs) can be used. OBBs are bounding boxes that can be arbitrarily rotated to match the alignment of the object. They also provide a tighter fit than AABBs for a majority of objects. However, the computation for OBB intersection is more complex than AABBs. The OBB intersection test must take into account all possible orientations of both boxes. This creates fifteen possible distance tests to be performed instead of the three for AABBs. However, it should be noted that either test could return a false value when any one of the axes being tested is found to be separating the two boxes. In fact, most OBB non-collisions are found in the first 6 tests along the

principal axes (Ericson, 2005). This makes the average test require less computation, but any true collision will require all comparisons to be performed.

Other Bounding Volumes

There are a variety of other bounding volumes that are used for intersection testing such as cylinders, capsules, cones, frustums, and general convex polyhedra. Discrete orientation polytopes (DOPs) are a particular class of bounding volume formed by the intersection of half-spaces (Ericson, 2005). The type of DOP is determined by how many half-spaces are intersected to create the volume. For instance, 8-DOPs are 8-sided volumes that provide a tighter fit than boxes and are usually the next best thing to testing the actual geometry.

Bounding Volume Hierarchies

Bounding volumes may also be grouped into hierarchies to provide a very tight fitting intersection testing while still using simple intersection tests. Generally, these hierarchies are arranged in a tree structure with each layer being more fine-grained than the previous. Any type of bounding volume can be used, but less computation is always better, so spheres and boxes are the most desirable.

Polygonal Meshes

If the application requires completely accurate intersection testing, then the individual polygons of an object can be tested against each other to find if objects collide. This method provides no false collisions, but requires a large amount of computation. To

increase efficiency, polygons may be grouped and stored in a bounding volume hierarchy made up of previously discussed objects to quickly determine non-collisions, but the polygonal meshes of the specified objects must be compared to find true intersections.

Raycasting

Raycasting is a different type of collision detection that involves finding the point of intersection between a ray and the closest object in the scene (Foley, et al., 1997).

Raycasting is often used for high-quality rendering of 3-D images, but it can be used for simpler applications such as simulating sonar or other distance-measuring devices. When used for rendering, the closest object is located and its color is calculated with an illumination equation that uses the color information for the object and relevant light sources. When used for distance measuring, only the point of intersection is needed, so less computation is required.

SPATIAL PARTITIONING TREES

In software environments, especially ones with large amounts of static objects, spatial partitioning trees may be used to divide the scene in such a way to minimize the number of intersection tests that need to be performed. By storing a scene in a tree structure, large segments of the world can be eliminated quickly, which cuts out all the more complex tests for the objects in the eliminated segments.

BSP Trees

Binary Space Partition (BSP) trees have been used quite successfully for both 2-D and 3-D environments (Ericson, 2005). For example, Quake 3: Arena uses BSP trees for all of its worlds. BSP trees work by dividing the world with a plane to create two partitions. Then those two partitions are subdivided. This continues until the entire world is made up of convex regions that are open space or solid objects. Another method is to subdivide until each region contains only one object. When testing an object for collisions, it is compared with the plane that divides the current region. Then the object is compared with the sub-region that it is inside. If the object intersects the plane, then both sub-regions are tested. BSP trees can require a large amount of pre-computation to create the tree structure, but are very fast during runtime.

Quadtrees and Octrees

Octrees and quadtrees divide 3-D regions into eight and four equal parts, respectively. By dividing space parallel to the major axes, this creates regions that are identical to AABBs. This makes testing the regions easy and allows for the elimination of even larger regions than a BSP tree in a single iteration. Quadtrees can be set up to partition the entire space to regions that are completely filled or completely empty. This method can produce a very large tree depending on the number and shapes of the objects in the scene. Another method for creating quadtrees involves adding the objects to the tree and dividing the tree to the point where only one object exists in each region. This method results in a smaller tree and can be much faster when using objects with simple geometries.

***COLDET* – A COLLISION DETECTION LIBRARY**

Coldet is a collision detection library written by Amir Geva. It is a library written in C++ and freely available under the GNU public license. It allows for real-time collision detection between complex polygon models. *Coldet* is interesting because it provides 100 percent accurate collision detection, while still producing results very quickly. In models with 3000 to 4000 polygons, the worst case took six milliseconds, and the average case took approximately 1 millisecond (Geva, 2000).

Coldet stores objects as a collection of triangles, then creates a bounding volume hierarchy with bounding boxes. Individual triangles are placed inside of boxes to minimize the number of triangle vs triangle intersection tests. *Coldet* supports the intersection testing of spheres, boxes, rays, segments, and polygon models. It is designed for use in 3-D game environments where very accurate collision detection is required. Another nice feature of this library is the ability to set the level of desired computation. The library can return the two triangles that intersect and the exact point of intersection, or a timeout point can be set. When a timeout point is set, the computation is stopped at that point, and the most accurate result at that point is returned.

CHAPTER 3: Geometry Module Design and Implementation

The geometry module was designed to be as flexible as possible while still maintaining a simple interface for use with RoboSim and future applications. The main entry points for the module were designed to be as general as possible to allow the application freedom to produce any type of objects that it required and move them in any desired way.

MODULE DESIGN

The core of module consists of a main geometry class that uses several types of data objects to store the environment information. All the basic functionality is accessed through this main class. This allows applications to create and populate new environments, move objects, and determine distances. A number of geometric primitives are supported, but users can also specify objects with arbitrary geometries via polygonal meshes. The geometry module is based on a right-handed coordinate system where the Y-axis is vertically aligned. All rotations are specified in degrees, and a positive rotation corresponds to a counter-clockwise movement. Environments can be any size, but are limited to a default size if the application does not specify one. All movement and distance measures are specified in world units, so any conversions to different units must be handled by the application. ArrayLists are used often for storage purposes. ArrayLists are part of the standard Java library, and are used throughout RoboSim. ArrayLists are quite desirable because they are dynamic structures that easy to use and not as complex as Vectors (Sun Microsystems, 2002).

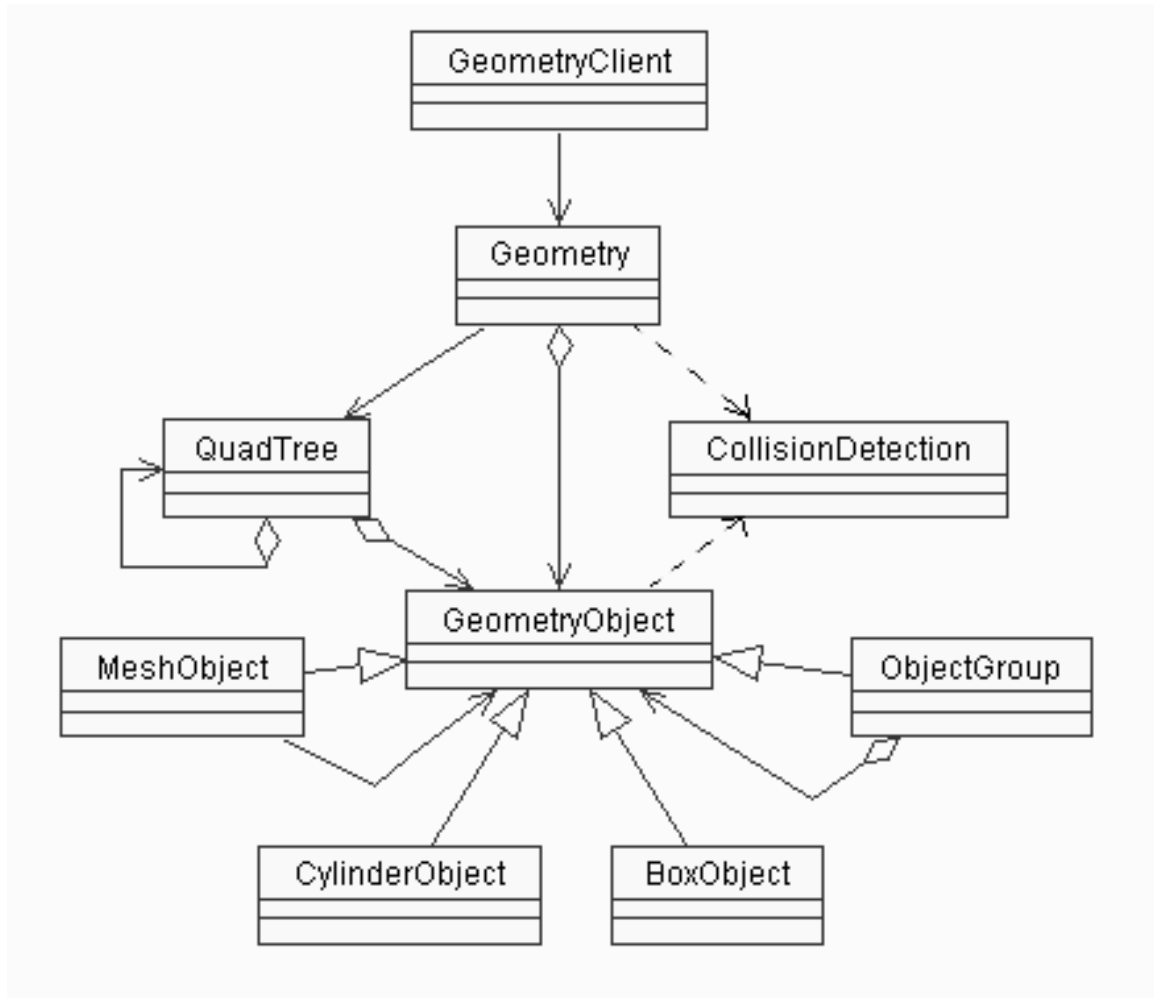


Figure 3.1. An overview of the geometry module class structure.

All of the environment objects are derived from the GeometryObject class as shown in Figure 3.1. These objects are then stored in an ArrayList or the Quadtree, depending on the object type. An object can be added to the world with or without checks for collisions with existing objects. RoboSim environments are stored in static XML files that are assumed to be correct, so RoboSim uses unchecked additions to ensure all objects are created and to reduce the time for initializing the environment. The CollisionDetection

class consists of static methods for testing geometry objects for intersection with other objects.

Geometry Functions

The functions for the geometry module are simplistic, but some are overloaded to give the user choices as to how they want to use certain features of the module. (For a diagram of all the functions in the module, please refer to Appendix A.) All the addition functions take in a GeometryObject and return a Boolean value if the object was added successfully. Objects being added to the module must be specified as moveable if the application will need to move them during run-time. Due to the static nature of the Quadtree structure, all stationary objects are placed in the Quadtree and all moveable objects are placed in an ArrayList where they can be accessed quickly. All objects that are moveable are referenced by a unique name for movement and sonar requests.

Movement functions include translations and rotations in object-relative and global coordinates. RoboSim uses object-relative coordinates to request moves. The positive X-axis is considered to be forward in a relative move; so all object-relative moves will be rotated so the X-axis lines up with the actual forward direction of the object. When using object-relative coordinates, the application doesn't have to keep track of the previous location to request moves and sonar requests.

The geometry module supports different levels of accuracy, so there are several different types of algorithms used in collision detection. The simplest and fastest algorithms use Axis-Aligned Bounding Boxes (AABBs) and the most precise test against individual polygons. Since not all applications require three dimensions, the module has simplified tests for 2-D movement. This provides increased efficiency by skipping unnecessary rejection tests.

DATA OBJECTS

The most important part of the module is being able to accurately store the geometric data for the objects in the scene. All objects are based on the base type GeometryObject, but have attributes and operations unique to a particular type of object. This simplifies collision detection, since the test intersection method can take in objects of the base type GeometryObject so the exact object type does not need to be known before calling the method.

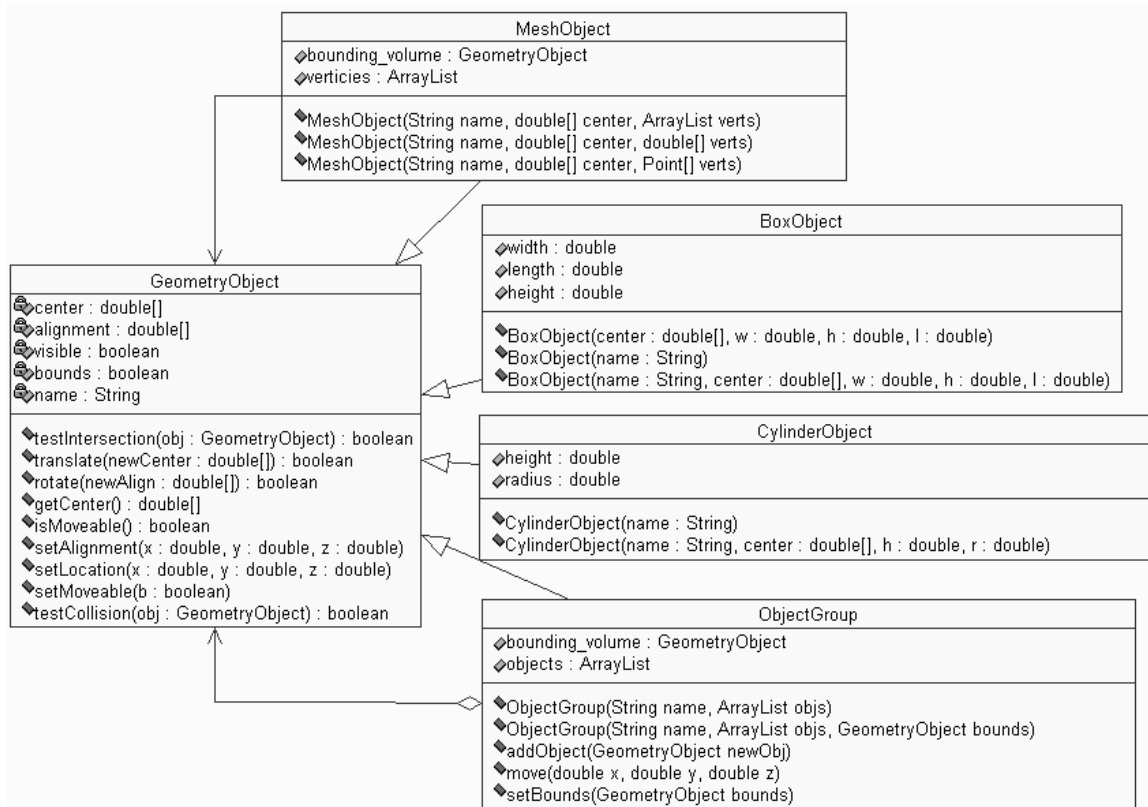


Figure 3.2. Data Objects used in the geometry module.

As Figure 3.2 shows, there are 4 types of objects supported by the module: MeshObject, CylinderObject, BoxObject, and ObjectGroup. The cylinder and box objects are both primitives that just take a few parameters to create. A center point, width, height, and length define a BoxObject. A cylinder only requires radius and height with a center point. These two objects are all that is required for current RoboSim simulations, but the module includes the two additional objects for use with future environments and different applications. The MeshObject is a polygon mesh with an accompanying bounding volume. Polygon meshes are formed from a list of vertices that are grouped into

quadrilateral polygons (quads). An application can assign a bounding volume to the mesh during creation or a default bounding box will be created for it. The class uses the maximum values of all the vertices in the polygon mesh to create the default bounding box. The last object, ObjectGroup, is used for creating complex objects or bounding volume hierarchies. ObjectGroup contains a bounding volume, which may be specified by the user. If a bounding volume is defined, it will be checked for collisions before any of the objects in the group.

COLLISION DETECTION

The CollisionDetection class was designed to be as simple as possible. The class is static with only two public methods. One method is for object-object intersection testing and the other is for ray-object intersections. These methods determine what type of object is passed in and then call the corresponding private function. This makes it very simple for testing collision of arbitrary objects, because the type of object being tested against is not required. Most of the object-object intersection tests are set up with multiple sets of rejection tests. As soon as an object fails one of the rejection tests, the method will return false to signify a non-collision. If none of the tests fail, then the objects are intersecting and true is returned. The ray-object methods are slightly different. Raycasting is concerned with the nearest point of intersection, not just if an intersection exists. The raycasting methods determine the exact points of intersection and will return the distance from the start of the ray to the nearest point of intersection with the object.

ROBOSIM INTERFACE

The geometry module connects to the RoboSim environment with the GeometryClient class that creates a TCP/IP socket connection. RoboSim sends messages over the socket to create the objects in the world, move existing objects, and request distance calculations. After receiving a message from the environment module of Robosim, the GeometryClient sends the commands to the main Geometry class and then sends any results back on the same socket. There are three message sequences that occur during the running of a RoboSim simulation.

The first sequence sends an InitBundle from the environment module to the GeometryClient. An InitBundle is a serialized message that holds a list of Geometry objects to be added to the world. The GeometryClient takes all the objects out of the InitBundle and adds them all using the add method in Geometry. In RoboSim environments, objects are always added successfully, so no results are sent back to the environment.

The second sequence consists of movement requests. The environment module specifies move requests in a MoveBundle. A move bundle is another serialized object that contains a group of moves. Each move consists of the name of the object to be moved, and the relative coordinates it wants to move to. The GeometryClient receives these move requests and attempts the moves in Geometry. The environment gets the results of the moves via a second MoveBundle that is created and sent by the GeometryClient. This second move bundle contains the moves that were allowed by the geometry module.

The third message sequence involves requesting and receiving distance measures. In RoboSim, the distance measures simulate sonar; so all requests are given in object-relative coordinates. A SonarRequestBundle consists of some number of sonar requests, which are defined by an object name, an x offset, and a z offset (the y offset is always 0, as robots are assumed to be horizontal.) The GeometryClient queries the Geometry for the distances and places them into a DistanceBundle that is sent back to the environment module.

MODULE TESTING

To ensure the proper operation of the Geometry Module, testing was performed on the module as a whole and on individual functions. Functions were tested with a small application that creates a specific environment and moves an object. Object collisions were tested for accuracy in additions and move attempts, and sonar requests were verified with known values. Once the functions were operating properly, the interface to RoboSim was tested for accuracy and data consistency.

The Quadtree was tested for both accuracy and efficiency. A testing class was set up with a number of objects that were added to the Quadtree. This testing class was also instrumented to measure the time it took to perform various operations. This testing class was set up to sequentially add a specified amount of simple objects. The test used verified additions, so each new object was tested against the existing object for intersections. With environments of 1000 objects or less, the Quadtree was up to 4 times

faster than using an ArrayList structure, but when the number of objects were increased to 10,000 objects, the Quadtree performed up to sixty-seven times more efficiently.

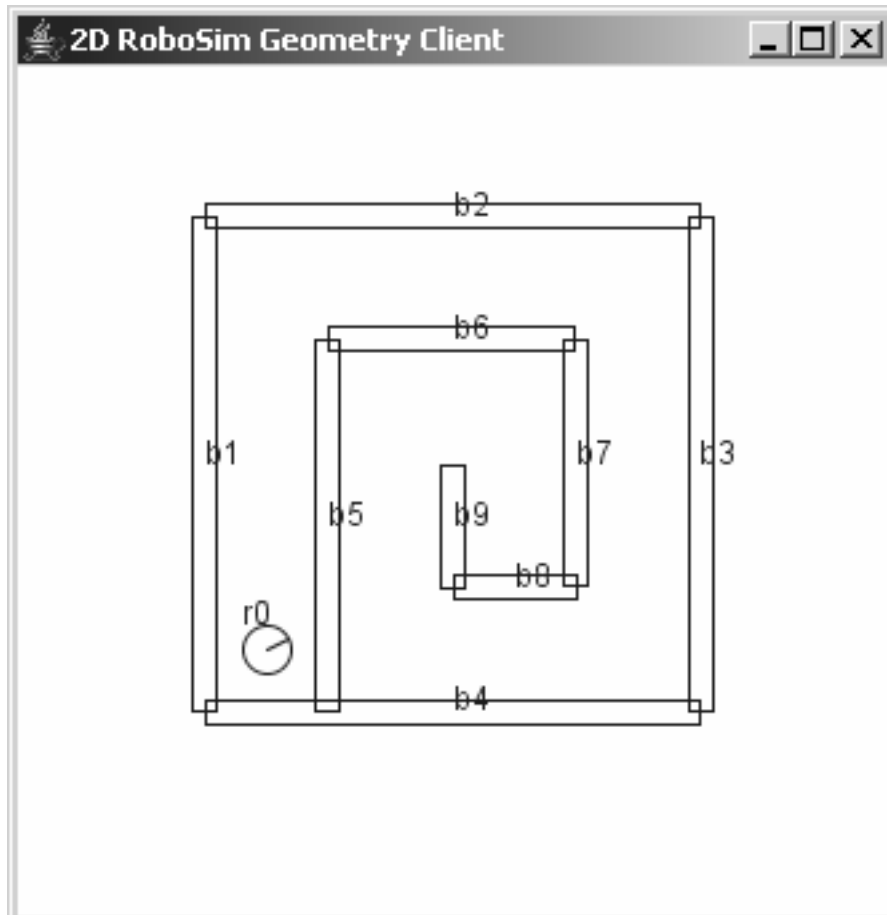


Figure 3.3. The 2-D Geometry Viewer

The geometry module has a 2-D viewer that can be used to display the world from a top-down perspective. The viewer, displayed in figure 3.3, is based on the design of the 2-D viewer included in the RoboSim module. By running both viewers in the same simulations, the state of the world was verified to be consistent throughout RoboSim. The world shown in figure 3.3 was used in the middle stages of testing to simulate

RoboSim environments. Initially, there were problems with objects not being added to the Geometry module. It was discovered that the specified objects were overlapping, so the module was not allowing them into the scene. To correct this, a new type of object addition was added that allowed for overlapping walls. The map shown in the figure was used to test if overlapping object were added correctly to the geometry module.

Distance finding was tested with and without connecting to the RoboSim environment. The initial testing was done with a simple application that created a room of a known size and requested sonar measurements in various directions. The distances returned were compared to values calculated by hand. Once the sonar was returning correct distances, it was tested with the RoboSim environment. On the first run, the values obtained from environment were much smaller than expected. This was due to a unit mismatch between the modules. RoboSim was specifying requests in radians, but the Geometry Module was expecting degrees. The proper conversions were added and the values between the environment and geometry were verified to be accurate and consistent.

Chapter 4: Conclusion

From a technical standpoint, current RoboSim applications gain almost negligible benefit from the addition of the Geometry Module, but the main purpose of the geometry module is to provide for the future benefit of RoboSim and other applications. The Geometry module meets this purpose by providing the capabilities for more complex robots and environments. The quadtree structure that partitions the environment provides an efficient way to store and test against larger and more complex geometries. This module allows for RoboSim to take a large step towards becoming a realistic environment simulator.

FUTURE WORK

While the geometry package provides a robust basis for providing collision detection for arbitrary geometries, it is essentially designed to meet the needs of RoboSim. The module design would not be as efficient for 3-D applications that had a large degree of vertical movement or environments with a very large number of moveable objects. The module is also static in the sense that it has no reactive physics. A physics model that supported inertia and bouncing objects would greatly increase the usability of the module. A realistic physics model would allow for simulations with larger timesteps and off-load some of the movement calculations from the application.

The Geometry module could also be extended to include a rendering engine if color and material attributes were added to GeometryObject. The module already supports

raycasting for range-finding purposes, so extending it to create ray-traced images would be a reasonably simple process. If desired, the module could be further extended to support light sources and textured surfaces.

BIBLIOGRAPHY

- Angel, Edward. (2003). *Interactive Computer Graphics : a Top-Down Approach with OpenGL*. (3rd ed.). Boston: Addison Wesley.
- Berg, de Mark. Kreveld, Marc Van. Overmars, Mark. Schwarzkopf, Otfried. (2000). *Computational Geometry*. (2nd ed.). Springer-Verlag.
- Eppstein, David. (2005). *Geometry in Action: Quadrees*. Retrieved Feb 20, 2005, from <http://www.ics.uci.edu/~eppstein/gina/quadtree.html>
- Foley, J.D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1997). *Computer Graphics: Principles and Practice*. (2nd ed.). Boston: Addison-Wesley.
- Gargantini, Irene. (1982). An Effective Way to Represent Quadrees. *Communications of the ACM* '82 25:12. p 905-910.
- Geva, Amir. (2000). *coldet – Free 3-D collision detection library*. Retrieved May 3, 2005, from <http://photoneffect.com/coldet/>
- Gilbert, Chris. (n.d). “Lesson #30: Collision Detection” Retrieved Feb 20, 2005, from <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=30>
- Harmon, Scott. (2004). *Co-operative Robotics Simulator – Environment Simulator*. Master’s report, Kansas State University, Manhattan, Kansas.
- Sun Microsystems. (2002). *Java 3-D 1.3 API Specification*. Retrieved Feb 20, 2005, from http://java.sun.com/products/java-media/3-D/forDevelopers/J3-D_1_3_API/j3dapi/index.html

APPENDIX A: Module Class Diagrams

Main Classes.....	26
Geometry Object Classes.....	27
Other Data Classes.....	28

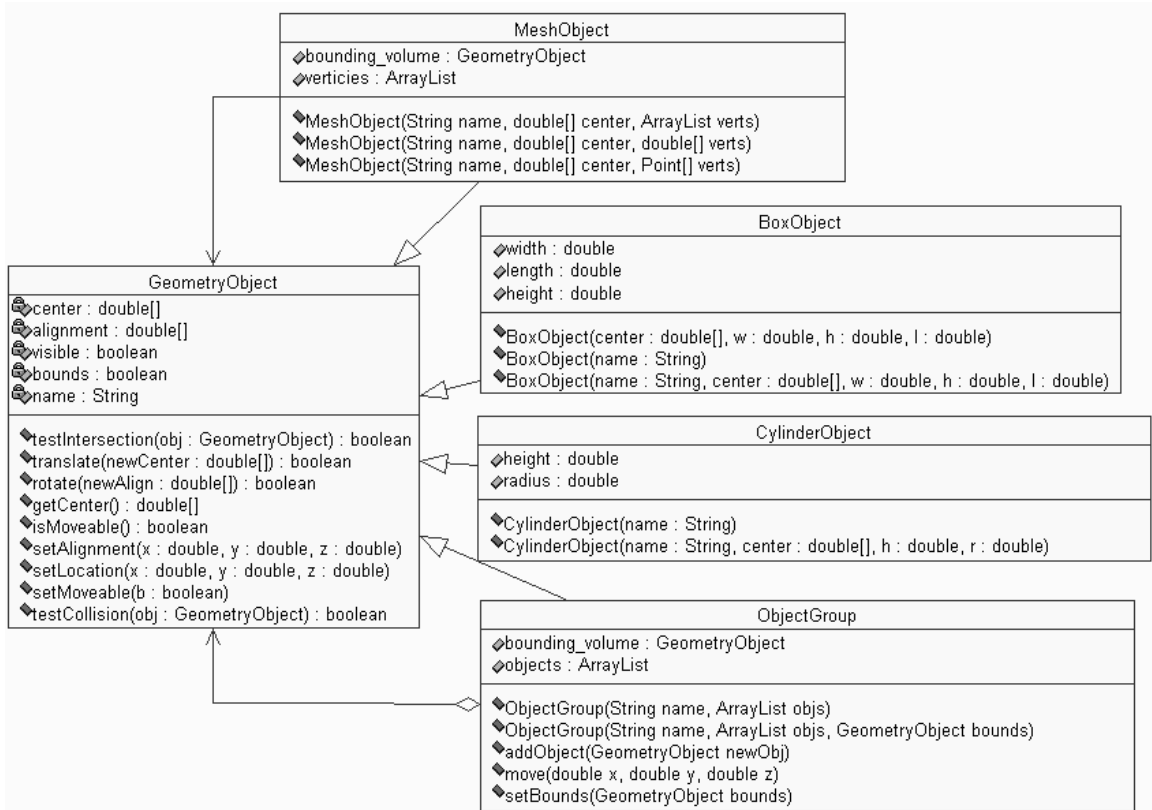
Main Classes

GeometryClient
<ul style="list-style-type: none">geo : Geometryrunning : BooleantcpSocket : SocketinputStream : ObjectInputStreamoutputStream : ObjectOutputStream
<ul style="list-style-type: none">GeometryClient(host : String, port : int)run()main(args : String[])

Geometry
<ul style="list-style-type: none">scene : QuadTreesimpleScene : ArrayList
<ul style="list-style-type: none">addObject(obj : Object) : BooleantestObject(obj : Object) : BooleansimpleMove(objName : String, move : Double[]) : BooleangetRayDistance(ray : Ray, min : double, max : double, width : double) : DoublesimpleTurn(obj : String, degrees : double) : BooleanGeometry()Geometry(w : double, h : double, l : double)

QuadTree
<ul style="list-style-type: none">bounds : GeometryObjectI, II, III, IV : QuadTreeLEAF : Booleanobjects : ArrayList
<ul style="list-style-type: none">testCollision(obj : Object) : BooleanaddObject(obj : GeometryObject) : booleandivide() : voidQuadTree(bounds : Geometry Object)QuadTree(bounds : GeometryObject, objs : ArrayList)addObjectUnchecked(obj : GeometryObject) : boolean

Geometry Objects



Other Data Objects

Move
◆x : double
◆y : double
◆z : double
◆rx : double
◆ry : double
◆rz : double
◆Move()

Ray
◆start : double[3]
◆direction : double[3]
◆RayObject()

Point
◆x : double
◆y : double
◆z : double
◆Point(double x, double y, double z)
◆Point()