Oriented Bounding Box Test Program

Johnathan Smith
December 14, 2008

CSU Chico
Department of Computer Science
Introduction

The Oriented Bounding Box Test Program allows users to see and test the implementation of an oriented bounding box volume used in 3D object collision tests. This program allows the user to supply various 3D objects, load them into the program, orient and move them, then see colliding volumes and polygons in real-time.

Problem Statement

The main purpose of this program was to learn how to implement oriented bounding boxes and test the collisions between them as well as learn how to detect collisions between triangles. Because of this, all of the algorithms used in this book were developed by others and merely stitched together to provide a test-bed for their use. Creating oriented bounding boxes was achieved using the covariance method for triangles described by Stefan Gottschalk in his PhD dissertation at University of North Carolina at Chapel Hill. Creation of the convex hull used for oriented bounding box create was accomplished using the Quick Hull algorithm as implemented by Tim Lambert at the University of New South Wales. The separating axis test for bounding boxes, as described by Christer Ericson in Real-Time Collision Detection, was used for checking for the intersection of bounding boxes. Checking for the intersection of triangles was accomplished using the triangle-triangle intersection test described in Faster Triangle-Triangle Intersection Tests by Olivier Devillers and Philippe Guigue. Checking for the intersection of a ray and an oriented bounding box, which is used to select objects on the screen, is accomplished using the algorithm described in Geometric Tools for Computer Graphics by Philip J. Schneider and David H. Eberly. Additional sources are used for the implementation of other functions. These sources are listed in the References section and sited in code where used.

Design

Most of this program is implemented in an Object Oriented fashion with some of the functionality implemented using classless procedural code. The classless procedural code encompasses the windowing and input handling as well as the functions for performing collision tests and doing various mathematical functions. The classes are divided into four main categories: UI, math, collision and managers.

UI

This category is comprised of the main.cpp file and the Camera class. Since all of the user interaction is done within main, it will be considered a class from here on out. The commands sent to main by the user are sent to the CollisionPane instance which manipulates everything seen on the screen or to the camera class. The main class is also responsible for add new copies of OBBObs to the CollisionPane by requesting them through the ModelManager. The Camera class controls the current camera position and orientation.
The main.cpp file contains all of the functions and variables needed to set up OpenGL and GLUT and handle all needed user input. The main function is responsible for creating a menu that contains a list of models the user can select from and add to the scene. When the user selects a model from the menu, a copy of that object is requested and then added to the scene with a call to pane.add(). User inputs are sent to the Camera if the user is currently attempting to move the camera and sent to pane if the user is attempting to manipulate the scene or objects therein.

**Camera**

This class represents the camera in the scene and contains two Vector3d objects for the camera position and the camera up vector. This class provides functions for freely rotating and moving the camera as well as a function for calling the gluLookAt function with the camera’s data.

**Math**

The math classes consist of the Vector3f, Vector3d, Vector4f, Matrix33f and Matrix44f classes. The Vector3f and Vector4f classes are used extensively in the collision classes for representing points and directions. The Vector3d class is used by the Camera class for its position and orientation. The Matrix33f class is used to perform various matrix operations in the for collision tests. The Matrix44f class is used to store rotations that manipulate the orientation of
objects on the screen. The NewMath file provides some predefined values as well as some math related helper functions.

Vector3f and Vector3d
The Vector3f and Vector3d classes represent three element mathematical vectors that contain basic functions used in vector arithmetic. These functions include: dot product, cross product, vector addition and subtraction, scalar/vector multiplication, length and distance functions, normalization and finding the angle between two vectors. These classes will be used both as traditional vectors as well as points throughout the program. The Vector3f class stores its elements as 32-bit floats whereas the Vector3d class stores them as 64-bit doubles.
The Vector4f class represents a four element mathematical vector and contains basic functions used in vector arithmetic. These functions include: dot product, cross product, vector addition and subtraction, scalar/vector multiplication, length and distance functions, normalization and finding the angle between two vectors. This class is used as either a point or a traditional vector depending on whether the fourth component is a one or zero respectively. This class is used over instead of the Vector3* classes for multiplications with 4x4 matrices when doing rotations. This class stores its elements as 32-bit floats.
The Matrix33f and Matrix44f classes represent 3x3 and 4x4 floating point matrix respectively. They provide functions for very basic matrix arithmetic including: matrix addition and subtraction, scalar/matrix multiplication, vector/matrix multiplication, matrix/matrix multiplication and matrix transposition. The Matrix44f class provides functions for creating rotation matrixes around each of the coordinate axes which can be used to transform Vector4f objects with a matrix/vector multiplication.

### Matrix33f

- **#mEntries**: float

```cpp
Matrix33f

<<create>>-Matrix33f()
<<create>>-Matrix33f(r1: Vector3f, r2: Vector3f, r3: Vector3f)
<<create>>-Matrix33f(other: Matrix33f)
<<CopOperator>>+=(rhs: Matrix33f): Matrix33f
+get(i: int, j: int): float
+set(i: int, j: int, value: float): void
<<CopOperator>>+==(rhs: Matrix33f): bool
+toIdentity(): void
+transpose(): Matrix33f
+transposeOf(): Matrix33f
<<CopFriend>>+transpose(mat: Matrix33f): Matrix33f
+setRow(i: int, row: Vector3f): void
+getRow(i: int): Vector3f
+setColumn(i: int, column: Vector3f): void
+getColumn(i: int): Vector3f
+det(): float
<<CopOperator>>+==(rhs: Matrix33f): Matrix33f
<<CopOperator>>+=+=(rhs: Matrix33f): Matrix33f
<<CopOperator>>+=+=(rhs: Matrix33f): Matrix33f
<<CopOperator>>+=+=(rhs: Matrix33f): Matrix33f
<<CopOperator>>+=+=(): Matrix33f
<<CopOperator>>+==(rhs: Vector3f): Matrix33f
<<CopOperator>>+=+=(rhs: Vector3f): Matrix33f
<<CopOperator>>+=+=(rhs: Vector3f): Vector3f
<<CopFriend>>+*(rhs: Vector3f, rhs: Matrix33f): Vector3f
<<CopOperator>>+*(rhs: float): Matrix33f
<<CopOperator>>+*(rhs: float): Matrix33f
```

### Matrix44f

- **#mEntries**: float

```cpp
Matrix44f

<<create>>-Matrix44f()
<<create>>-Matrix44f(other: Matrix44f)
<<CopOperator>>+=(rhs: Matrix44f): Matrix44f
+get(i: int, j: int): float
+set(i: int, j: int, value: float): void
<<CopOperator>>+==(rhs: Matrix44f): bool
<<CopOperator>>+=+=(rhs: Matrix44f): bool
+toIdentity(): void
+transpose(): Matrix44f
<<CopFriend>>+transpose(mat: Matrix44f): Matrix44f
+rotateX(theta: float): Matrix44f
+rotateY(theta: float): Matrix44f
+rotateZ(theta: float): Matrix44f
<<CopOperator>>+=+=(): Matrix44f
<<CopOperator>>+=+=(rhs: Matrix44f): Matrix44f
<<CopOperator>>+=+=(rhs: Matrix44f): Matrix44f
<<CopOperator>>+=+=(rhs: Matrix44f): Matrix44f
<<CopOperator>>+=+=(): Matrix44f
<<CopOperator>>+=+=(rhs: Vector4f): Matrix44f
<<CopOperator>>+=+=(rhs: Vector4f): Vector4f
<<CopFriend>>+*(rhs: Vector4f, rhs: Matrix44f): Vector4f
<<CopOperator>>+*(rhs: float): Matrix44f
<<CopOperator>>+*(rhs: float): Matrix44f
```

### Matrix33f and Matrix44f

The Matrix33f and Matrix44f classes represent 3x3 and 4x4 floating point matrix respectively. They provide functions for very basic matrix arithmetic including: matrix addition and subtraction, scalar/matrix multiplication, vector/matrix multiplication, matrix/matrix multiplication and matrix transposition. The Matrix44f class provides functions for creating rotation matrixes around each of the coordinate axes which can be used to transform Vector4f objects with a matrix/vector multiplication.
NewMath

This NewMath .h and .cpp files contain various math constants and helper functions. The Is* functions allow a way for the user to do comparisons on floats and doubles allowing for a certain amount of error. The EPSILON_* defines provide the values for these errors. The RotateVectorAbout functions rotate a given vector around a given axis by a given theta number of radians. The Jacobi and SymSchur2 functions are used to decompose a symmetric matrix into eigenvalues and eigenvectors.

Collision

This category is comprised of all the objects that are actually involved in doing the collision detection. The CollisionPane class stores all of the current 3D objects currently seen on the screen as is also responsible for manipulating said objects. The WavefrontObj stores the geometry of the model for a 3D object and the OBBObj class associates the WavefrontObj with an OBB struct. OBBObj is also responsible for positioning orienting the model geometry. Both the OBB and Triangle3f class are used by the CollisionPane to do intersection tests. The ConvexHull class is used to store the resulting convex hull created from the QuickHull makeHull function. The QuickHull class uses the Edge3f, HalfSpace and TriangleSpaceAndPoints classes when creating convex hulls. The CollisionPane class uses the BTree class for checking if duplicate triangles are being inserted into the list of colliding triangles to be drawn. The ConvexHull class uses the BTree class to store its points so that duplicate points inserted into the hull can be detected quickly.
WavefrontObj

This class is used to load and store 3D geometry information from Wavefront .obj files. This class provides functions to retrieve the vertices and triangles of the object as well as for drawing the object. This class only handles geometry defined in terms of triangles and any n-sided polygons where n > 3 or splines within the file will not be loaded. This class also loads the associated .mtl material file for the specified object if it exists. The supported fields in the .mtl file that will be loaded include: ambient, diffuse and specular color values; specularity coefficient and texture file. The texture is loaded by the TextureManager class (if the texture file exists).

OBBObj

The class encapsulates both a WavefrontObj and its corresponding OBB and ConvexHull. The class also contains an orientation matrix and translation vector that is used to move the WavefrontObj object and its bounding box around the screen. This is done so that there will be only one shared instance of each of the loaded .obj objects. When vertices or triangles are retrieved from the WavefrontObj (through the OBBObj functions) they are transformed by the matrix and vector before they are sent to the caller. This class is also responsible for drawing the OBB (if desired) and calling the draw function of the WavefrontObj. The color of the OBB can be changed to reflect whether it is colliding or not.
CollisionPane

This class stores the instances of all OBBObs objects currently being displayed in the scene. Whenever an object is transformed this class checks for collisions between all of the objects in the scene, keeping track of which OBBs and triangles that are colliding. When the draw function of this class is called, it will then draw all of these colliding features as well as calling the draw functions for all of the OBBObs objects in the scene. This class also provides functions for selecting an OBBObj and transforming it.
### BitMap

The BitMap class represents a set of bits that can be set to zero or one. The constructor allows the map to be set to any size. This class is used by the CollisionPane class to keep track of which triangles in a model have already been added to a list of colliding triangles. This class was originally written for a CSCI 340 project.

```cpp
BitMap

- bts: DWORD
  - numOfBits: size_t
  - numOfDWords: size_t

<<create>>-BitMap(numBits: size_t)
<<destroy>>-BitMap()
+ set(): void
+ set(bitAddress: size_t): void
+ reset(): void
+ reset(bitAddress: size_t): void
+ flip(): void
+ flip(bitAddress: size_t): void
+ size(): size_t
+ test(bitAddress: size_t): bool

<<CppNamespace>>

Collision

+ PointInTriangle(p: Vector3f, t: Triangle3f): bool
+ OBBOBBIntersect(a: OBB, b: OBB): bool
+ TriangleTriangleIntersect(t1: Triangle3f, t2: Triangle): bool
+ CircularPermutation(inout p: Vector3f, inout q: Vector3f, inout r: Vector3f, dot1: float, dot2: float, dot3: float): bool
+ FourVectorDot(a: Vector3f, b: Vector3f, c: Vector3f, d: Vector3f): float
+ CreateOBB(points: Vector3f*, numPoints: int, out box: OBB): bool
+ CreateOBB(hull: ConvexHull, out box: OBB): void
+ CovarianceOfTriangles(tris: Triangle3f*, numTris: int, out result: Matrix33f): void
```

### Collision

The Collision namespace provides functions for performing collision tests and creating oriented bound boxes. The CircularPermutation and FourVectorDot functions are used by the TriangleTriangleIntersect function, which determines whether two triangles are intersecting. The two CreateOBB functions create oriented bounding boxes. The first function creates the OBB using a furthest distance between two points method whereas the second uses the covariance of triangles method. The CovarianceOfTriangles function is used by the second CreateOBB function to create a covariance matrix based on an array of triangles.
Triangle3f

The Triangle3f class is used to store the three points that make up a triangle. These points are represented using three Vecto3f objects. The Triangle3f class is used in the triangle intersection test as well as for creating and oriented bounding box.

OBB <<CppStruct>>

This struct represents the oriented bounding box (OBB) for a WavefrontObj object and consists of four Vector3f objects and an array of three floats. One the Vector3f objects stores the location of the center of the OBB while the other three will hold the orientation of the OBB. The array of three floats holds the corresponding half widths along each of the boxes orientations vectors.

ConvexHull

The ConvexHull class is responsible for holding the data for a convex hull of an object. It contains a vector of points that represent the vertices of the hull as well as a BTree of the same points so that duplicate points can be found quickly. When a triangle is added to the hull, the BTree is used to see whether the points in the triangle already exist in the hull and if not, they are added to the points vector. The hull is assumed to be built a triangle at a time. The mNormals vector stores a list of normals for each point so that lighting calculations on the convex hull can be done per vertex. These normals are not normalized so when the
calculated normal of each triangle is added to a point’s normal, the triangle’s area will be reflected in the normal.

QuickHull

The QuickHull class contains static functions that are used to construct a convex hull of a set of 3d points based on the quick hull algorithm. The code for these functions was created from the Java implementation of the quick hull algorithm written by Tim Lambert at the University of New South Wales.

HalfSpace

The HalfSpace class is used to represent a mathematical half space. It is composed of a normal vector to a plane and a determinant value. A HalfSpace object can be constructed from either two points or three points. When two points are used, the plane created has a normal equal to the cross product of the vector between the two points and the z-axis that goes through the two points. When three points are used, the plane is equal to the plane in which the triangle represented by the three points lies.

TriangleSpaceAndPoints

This class is used to associate a triangle and a set of points with a half space. The triangle mT lies in the plane defined by mHalf and the points in mPoints are associated points that lie in the positive half space of this plane. When an attempt to associate a point is made with a call to addPoint, the point will only be added if mHalf contains the point. The variable mExtreme keeps track of which point is furthest in front of the plane defined in mHalf. This class is used by the QuickHull makeHull function.

Edge3f

The Edge3f class is used to represent the edge between two points in a triangle. This class is used by the QuickHull class to keep track of which edges need to be used to construct new triangles.
The BTree class implements a B tree that holds records consisting of vector/integer pairs. Duplicates are not allowed in the tree and when a duplicate is found, the Record that was attempted to be insert will be set equal to the Record found in the tree. The order of the tree can be specified in the construct but defaults to five. This class was originally written for CSCI 311.

The BNode class implements a node in a B tree. It contains an array of Records as well as an array of pointers to child nodes. This class was originally written for CSCI 311.

The Record class is used to store a Vector3f/unsigned int pair that serve as entries in a B tree. The class provides overloaded comparison operators that are used to sort the Records in the tree based on the corresponding comparison operators for the Vector3f class.

**Managers**
There are two manager classes that are responsible for keeping track of some of the shared resources used in this program. The ModelManager class is responsible for loading and storing WavefrontObjs and creating OBBs for them. New 3D objects can be created on the scene by requesting copies from the ModelManager. The TextureManager is responsible for loading image files used for textures and storing the OpenGL texture index values associated with each image.

<table>
<thead>
<tr>
<th>ModelManager</th>
<th>TextureManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mOBBSByFile: map&lt;string, OBBObj&gt;* &gt;</td>
<td></td>
</tr>
<tr>
<td>-mObjects: vector&lt;WavefrontObj&gt;* &gt;</td>
<td></td>
</tr>
<tr>
<td>-mOBBSByName: map&lt;string, OBBObj&gt;* &gt;</td>
<td></td>
</tr>
<tr>
<td>-mInstance: ModelManager</td>
<td></td>
</tr>
<tr>
<td>+instance(): ModelManager</td>
<td></td>
</tr>
<tr>
<td>+refresh(): void</td>
<td></td>
</tr>
<tr>
<td>+getModelNames(names: string): int</td>
<td></td>
</tr>
<tr>
<td>+getModelByName(name: string): OBBObj</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;create&gt;&gt;-ModelManager()</td>
<td></td>
</tr>
<tr>
<td>+mTextures: map&lt;string, GLuint&gt; *</td>
<td></td>
</tr>
<tr>
<td>-mInstance: TextureManager</td>
<td></td>
</tr>
<tr>
<td>+instance(): TextureManager</td>
<td></td>
</tr>
<tr>
<td>+add(filename: char): GLuint</td>
<td></td>
</tr>
<tr>
<td>+get(filename: char): GLuint</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;create&gt;&gt;-TextureManager()</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;destroy&gt;&gt;-TextureManager()</td>
<td></td>
</tr>
</tbody>
</table>

**ModelManager**

This singleton class is responsible for loading and managing all of the WavefrontObj objects used in this program. On startup (and also with a call to the classes refresh function) this class will scan the program directory for .obj files and attempt to create instances of all of them. After a .obj file is loaded, an OBB for the WavefrontObj is calculated and both are put into an OBBObj object. Copies of the OBBObj object can then be requested by the other classes of this program.

**TextureManager**

This singleton class will be responsible for loading and managing 2D textures used in the program. The class uses the GDI+ Bitmap class to load in image files so the formats BMP, ICON, GIF, JPEG, Exif, PNG, TIFF, WMF, and EMF will be supported. Functions allow for the addition of new textures (by passing the filename) and the retrieval of GLuint index values for textures that are already loaded. Each image loaded is bound to an OpenGL texture and the GLuint associated with that texture is mapped to by the name of the texture file. This ensures that files are only loaded once. The class is only used by the WavefrontObj class.

**Classes and Relations**
Primary Sequences
At program startup the main function creates an instance of the ModelManager class by calling its instance function. When the ModelManager is created, it calls its refresh function which iterates through the files in the program directory and loads all files of the type .obj. For each file, a WavefrontObj object is created.

When the WavefrontObj is created, its import function is called with the filename and the geometry is read in from the file. The import function then calls the loadMaterial function which attempts to load the corresponding mtl file for the .obj. If the file is found in the program directory, the material data (ambient, diffuse and specular colors) are loaded. If a texture file is provided in the mtl, a call to TextureManager::get() (with the texture filename as a parameter) is made in an attempt to get an index for the texture. If MAX_INT is returned a call to TextureManager::add() is made in an attempt to load the texture. If the texture was found and loaded successfully, a valid texture index value is returned. The loaded texture can now be used by binding the texture associated with this index during OpenGL drawing routines.
After the WavefrontObj object is created, an OBBObj object is then created using the WavefrontObj object as a parameter. The OBBObj gets all the points from the WavefrontObj and uses them to create a ConvexHull. This ConvexHull is used to create an OBB. The ModelManager then stores the OBBObj in two maps. One map maps the name of the file to the OBBObj while the other map maps the name of the .obj object to the OBBObj. This .obj object name is contained within the .obj file.

If subsequent calls to the ModelManager::refresh() function are made, only the .obj files that have not already been loaded will be loaded. This ensures that no redundant loads are performed and provides a way for the user to add new .obj files to the program directory and load them without restarting the program.

After the models have been loaded, the main function then calls createMenu () which requests the array of names of the objects loaded through the function ModelManager::getModelNames(). It uses the array to build a popup menu that the user can use to select objects he/she wishes to create in the scene. There will be also refresh option in the popup menu to allow the user to do a manual refresh. Finally, CollisionPane and Camera instances are created.
When the user clicks on a .obj menu item in the popup menu, this signals the program that an OBBObj should be added to the scene. The menuCallback function is called and the selected name of the item is passed to the ModelManager in a call to ModelManager::getModelByName(). The ModelManager then creates a copy of the OBBObj with the passed name and returns it. The main function then passes this OBBObj to the CollisionPane in a CollisionPane::add() call. When the OBBObj is added, the CollisionPane calls CollisionPane::checkCollisions() to check to see if the addition of the object has caused any collisions. A description of the function is described later.

After objects have been added to the scene, the user can select these objects by left-clicking on them. When the user left-clicks, a call is made to the CollisionPane::select() function with the camera's current position and a picking vector representing where the user clicked. If the picking vector intersects an
object in the scene, that object becomes the selected object. If the vector intersects more than one object, the one closest to the camera is selected. Once an object on the scene is selected, the user is now free to modify the orientation and position of that object. The user can use the keyboard to move the object in the x, y and z directions as well as rotate the object around the x, y and z axes. If the user moves the object, a call to moveX(), moveY() or moveZ() of CollisionPane is made with the desired displacement. A Vector3f object is then created with that displacement in one of its x y or z components and that Vector3f is then passed to the OBBObj::addTranslation() of the selected scene object. This vector is added to the OBBObjs translation vector.

If the user chooses to rotate the object using the keyboard, the call rotateX(), rotateY() or rotateZ() of CollisionPane is made with the theta of the desired rotation provided. A rotation matrix is then created by creating a new Matrix44f object and calling its rotate function with theta as the value. This matrix is then passed to the OBBObj::addRotation() function of the selected OBBObj and it is concatenated onto that OBBObjs current rotation matrix. With either a call to the translation functions or a call to the rotation functions, a call to CollisionPane::checkCollisions() is always made to ensure collisions cause by these transformations are detected.

With an object selected, the user can also choose to have the OBB of the object draw or delete the object from the scene. If the user signifies that the OBB should be drawn (using the keyboard) then a call to CollisionPane::drawSelectedOBB() is made which then calls the OBBObj::setDrawOBB() function of the selected object. If the user chooses to delete the object, a call to CollisionPane::remove() is called and the selected OBBObj is deleted (from memory) and the selected is set to NULL. The user can also choose to draw all of the OBBs of all of the objects in the scene. This is done by calling the CollisionPane::drawAllOBB() function which loops through all of the objects in the scene and calls their setDrawOBB() function. When the scene is to be redrawn, a call is made to CollisionPane::draw() which loops through all of the OBBObj objects in the scene and calls their OBBObj::draw() function. The OBBObj::draw() function makes a call to glTranslate with the translation vector as a parameter. The orientation (rotation) matrix is then concatenated onto the current OpenGL matrix. A call is then made to WavefrontObj::drawObject() to draw the actual 3D object. The OBBObj::draw() function will also draw the OBB if the bool to draw it is set to true. The color of the OBB will depend on where it is colliding or not.
Every time a call to CollisionPane::checkCollisions() is made, the above sequence of events occurs. First all OBBObjs are looped through and checked to see if they are colliding with any other of the OBBObjs in the scene. This is done by doing a pair wise comparison of all the OBBs of all the OBBObjs in the scene. Every time a collision is detected, the colliding OBBObj’s setBoxColliding() function is called to change the color of the displayed OBB to show that is colliding. Also, for each colliding OBB, the triangles of each of the WavefrontObjs are obtained. A pair wise triangle-triangle intersection test is then done on one triangle from each of the objects. If the triangles are found to be colliding, they are added to a vector of triangles within the CollisionPane class. These triangles are drawn highlighted during a CollisionPane::draw() function call to show the user which triangles are colliding on all of the objects.

### Algorithms

#### OBB Creation

For the creation of the oriented bounding boxes, two algorithms were implemented but only one is used. The first algorithm is described in *Real-Time Collision Detection* by Christer Ericson. It finds a loose fitting OBB by first finding the two points furthest apart in the 3D object and using the vector between them as the first orientation vector. The plane perpendicular to this vector is then found and all of the points are projected onto this plane. The two projected points furthest apart are then found and the vector between them is used as the second orientation vector. The projected points are then projected onto the line perpendicular to this second orientation vector, and again the two furthest points are found. The vector between these two points is found and set as the third orientation vector, and the center is found by backtracking and
adding half widths (found from the distance between the points used for the orientation vectors) multiplied by orientation vectors to one of the final points found. This algorithm was the first one to be implemented and is not used in the program. It has a time complexity of $O(n^2)$.

The second algorithm is based on the Covariance Matrix of the Triangles of a model as described by Stefan Gottschalk in his PhD dissertation from the University of North Carolina at Chapel Hill. This algorithm involves computing a covariance matrix across all of the faces of the convex hull of the object. The eigenvalues and their corresponding eigenvectors are then found using the Jacobi method for finding eigenvalues and eigenvectors for symmetric matrices. These orthonormal eigenvectors are then used as the orientation vectors for the oriented bounding box. The points of the convex hull are then projected on each of these vectors to find the extent of the hull along each of the vectors. The distance between the min and max points along each vector is then halved and used as the half-width along that vector. This half width is then added to the min point to get the center point along that vector. The center points along each vector are then added together to get the center point of the box. The time complexity of this method is $O(n)$.

**Convex Hull Creation**

The quick hull algorithm is used to create the convex hulls used by the oriented bounding box creation algorithm. The code from Tim Lamberts Java implementation of the algorithm was used as a basis for writing the C++ version in this program. This algorithm works by first finding the two points that have the least and greatest x values. These two points are then used to create a half space with the z-axis. The point furthest from the plane associated with this half space is found and two new triangles with their respective half spaces are created with this point and the two original points. These two triangles have normals that point in the opposite directions. The rest of the points are then associated with at most one of these half spaces. The point is only associated if it is in or in front of the plane for this half space. For each triangle/half space the point furthest in front of it is found and set as the extreme. All triangles/half spaces are that contain this point are then removed from the list of triangles/half spaces and the unique edges of the triangles are kept. The edge is unique if it is not shared by two of the deleted triangles. Then for each edge a new triangle/half space is created with the points of this edge and the extreme point. This process is repeated until no triangle/half space possesses an extreme point.

**OBB Intersection Test**

The intersection test for two OBBs is also covered in Real-Time Collision Detection and the code found in the book is used in a slightly altered form. The algorithm works by attempting to find a separation axis between the two OBBs. The radii and center points of the two OBBs are projected one at a time onto 15 different axes. If the distance between the center points is greater than the sum of the projected radii, then the axis serves as a separating axis and the two OBBs are not intersecting. The 15 different axes correspond to the three orientation vectors of both OBBs as well as the nine possible cross products between them.
**Triangle-Triangle Intersection Test**

The algorithm for testing the intersection of two triangles was taken from *Faster Triangle-Triangle Intersection Tests* by Olivier Devillers and Philippe Guigue. This test is used to check whether the triangles that make up the model of one object intersect the triangles of another. This test will only be used when the oriented bounding boxes of each of the objects intersect. This algorithm first checks to see if the points of either triangle lie entirely on one side of the plane defined by the other triangle. If the points do all lie on one side, then the triangles are not intersecting. If the triangles lie on the same plane, then a coplanar triangle intersect test is used to determine whether they are intersecting. If the points of one triangle straddle another, then the points of both triangles are subjected to circular permutations which bring the correct points into the correct positions for a triangle intersect test to occur. This test consists of two four vector determinant calculations that must both be less than or equal to zero for the two triangles to be intersecting. The descriptions of the permutations needed are described by Devillers and Guigue.

**Closing Remarks**

I working alone for the entirety of the project and all of the code was written by me except where stated otherwise. The purpose of this assignment was for me to learn how the algorithms for creating oriented bounding boxes and testing their collisions are implemented as well as testing the collisions between triangles are implemented. I not only learned how to implement these algorithms but I also took it upon myself to implement the quick hull algorithm so that my oriented bounding box create was much more efficient. Although I relied heavily on the descriptions and code of others, I made sure that I fully understood every aspect of how these algorithms work with the exception of the Jacobian matrix decomposition. This program was probably the hardest thing I have ever had to write and I feel that it has given me a lot of valuable experience in programming math intensive algorithms in C++.
  
  Web site: http://www.cse.unsw.edu.au/~lambert/java/3d/source/