DH2323 DGI16
Lab3
Hierarchical Transformations

In Lab 2 of the animation track, you investigated the transformation of primitives in 3D environments. In this lab, we will use that knowledge to animate a hierarchical tank model in real-time using OpenGL and hierachical transformations. You will define a Bounding Volume Hierarchy (BVH) for the tank, a data structure that will enable more efficient intersection tests to take place. This will form a basis for doing intersection tests between projectiles and the tank in order to calculate whether they hit it. You will apply particle effects for successful hits using the particle system code from Lab 2.

Finally, you will apply a discrete level-of-detail (LOD) scheme for the tank so that meshes with fewer primitives are used when the tank is further away from the viewer and specific details cannot be seen.

Note that in order to complete this lab you should have first completed Lab 2 of the animation track. Once you check that everything is building properly, you should integrate your math library code from Lab 2 into the relevant source files in this lab.

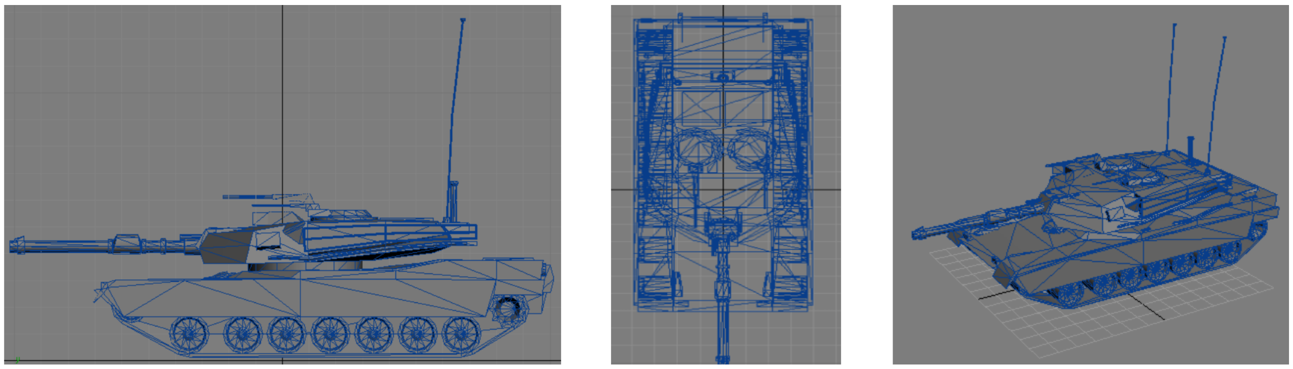


Figure 1: Wireframe view of the tank model from a number of different view angles. Tank model by *knoxville@list.ru*.

Tasks:

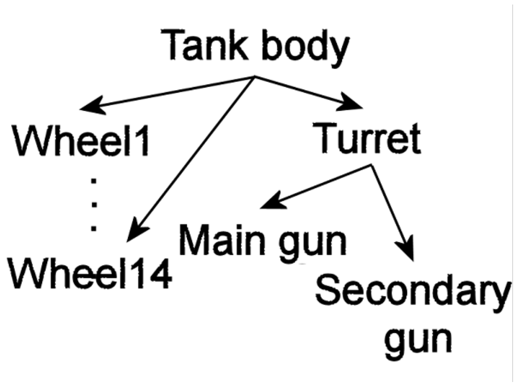
1. Assemble, display and enable the interactive animation of the various parts of the tank model (Section 1).

2. Define a bounding volume hierarchy for the tank (BVH) and use it to perform some simple collision checks with projectiles in the environment (Section 2).

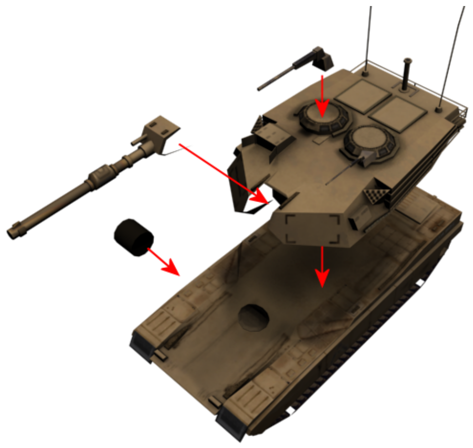
3. Implement a basic discrete level-of-detail (LOD) scheme for the thank based on its distance to the camera, so that simplified tank geometry is displayed in cases where the viewer cannot see any noticeable differences with respect to the more detailed version (Section 3).

1 Display and Animate the Tank Hierarchy

The task in this section is based on a tank model and accompanying code, found in *tankLab.zip*. The tank is composed of a free, low polyon mesh and accompanying 2D texture. It is divided into 5 main objects – the body, the turret, the main gun, the secondary gun, and the wheel as shown in Figure 2.



(a) Hierachical structure.



(b) Tank model.

Figure 2: The hierachical structure of the tank.

1.1 Background

In this lab, we will use *display lists* to draw the various components of the tank. Here, we give a brief introduction to display lists in OpenGL.

1.1.1 Display Lists

OpenGL commands are normally executed in immediate mode. A display list is a group of OpenGL commands that have been stored for later execution. Display lists are used to:

- Cache commands to redraw the same geometry multiple times.
- Manage state variables.
- Encapsulate mode changes.
- Store any series of OpenGL commands which will be repeatedly executed.

1.1.2 Naming a Display List

Each display list must be identified by a unique integer identifier e.g. 1, 2, 3... A series of unused identifiers can be found using *GLuint glGenLists(GLsizei range)*. where

- range is the number of consecutive identifiers desired
- The return value is the first of the consecutive identifiers
- Each of the lists associated with the identifiers is marked as empty and used
- A return value of zero indicates the requested number of identifiers are not available

```
//Allocating three display lists
GLuint listOne, listTwo, listThree;
listOne = glGenLists( 3 );
listTwo = listOne + 1;
listThree = listTwo + 1;
```

1.1.3 Creating a Display List

Display lists are created between *glNewList* / *glEndList* pairs. Memory and the values of any necessary variables are allocated for OpenGL commands which appear in a list. The definition of a new display list beings with *void glNewList(GLuint list, GLenum mode)*. The end of the display list is signalled through the use of *void glEndList(void)*.

```
//Creating display lists
floorList = glGenLists( 3 );
cubeList = floorList + 1;
ballList = cubeList + 1;

glNewList( floorList, GL_COMPILE );
   //GL_COMPILE: save the commands for later execution, do not execute them ↩
       immediately.
    drawFloor( 0.0, -3.0, 0.0, 8.0 );
glEndList();

glNewList( cubeList, GL_COMPILE );
    drawcube( 0.0, 0.0, 0.0, 1.0 );
glEndList();

glNewList( ballList, GL_COMPILE );
    drawRedBall( 0.0, 0.0, 0.0, 1.0 );
glEndList();
```

1.1.4 Executing a Display List

Once a display list has been created, it is executed using *void glCallList(GLuint list)*, where *list* specifies the list to be executed. Executing the list typically will result in the previously defined transformations and primitive drawing operations being executed i.e. this is when objects will be drawn to the screen.

```
//Executing a display list
glCallList( floorList );

glPushMatrix();
    glRotatef( 5.0f, 1.0f, 0.0f, 0.0f );
    glCallList( cubeList );
glPopMatrix();

glPushMatrix();
    glTranslatef( 3.0f, 0.0f, 0.0f );
    glRotatef( 5.0f, 0.0f, 1.0f, 0.0f );
    glCallList( cubeList );
glPopMatrix();

glPushMatrix();
    glTranslatef( -3.0f, 0.0f, 0.0f );
    glRotatef( 5.0f, 0.0f, 0.0f, 1.0f );
    glCallList( ballList );
glPopMatrix();

glFlush(); // Flush all drawing commands
SwapBuffers(); // swap buffers
```

1.2 Tasks

Build and run the *tankLab* program. You will notice that only the tank body is drawn, as shown in Figure 3 .

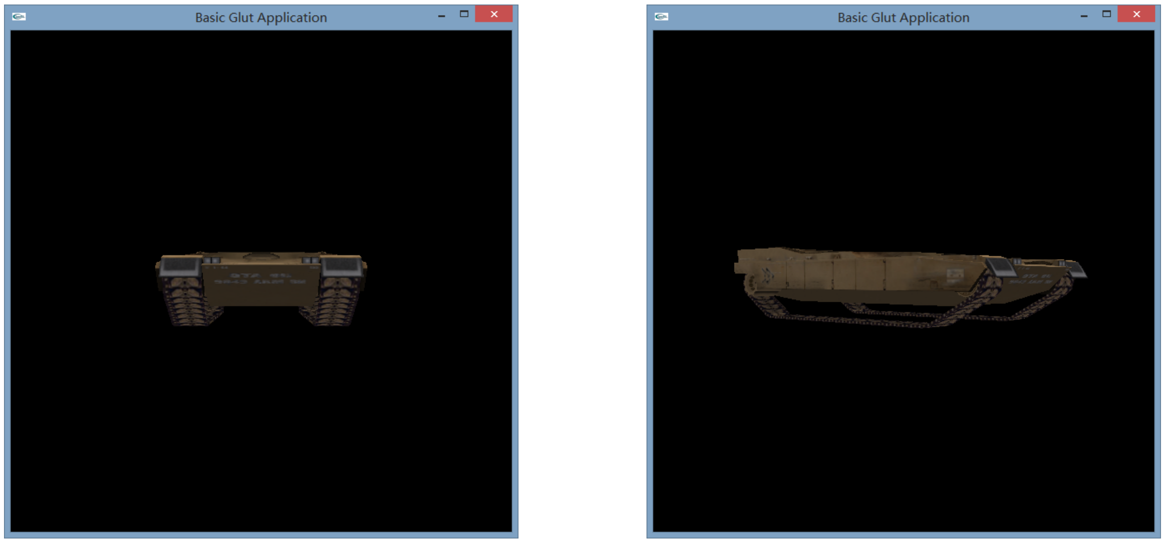


Figure 3: The initial tankLab code will draw only the *tankBody* object.

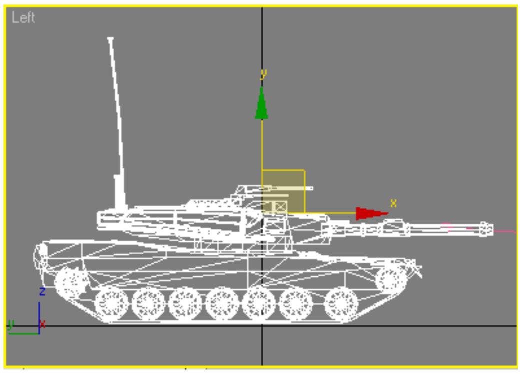
Looking at the code in the *main.cpp* file, you will see that the tank is loaded in as separate meshes in the *load_tank_objs()* function. Each part can then be displayed in the *draw_tank* function by calling *DrawOBJ(tankPart->m_iMeshID)*; where *tankPart* is the part of the tank that you want to draw i.e. *tankBody*, *tankTurret*, *tankMainGun*, *tankSecondaryGun* or *tankWheel*. For example, *DrawOBJ(tankTurret->m_iMeshID)* will draw the turret.

```
void draw_tank(float x, float y, float z)
{
    glPushMatrix();   //save the current transformation matrix (i.e. current position and orientation) (1)
    glTranslatef(x,y,z);

    glScalef(0.1,0.1,0.1);      //reduce the size of the tank on screen
    DrawOBJ(tankBody->m_iMeshID);

    //Use your own draw code here to draw the rest of the tank
    //Here is the code for each individual part
    //Each part is placed with respect to the origin
    //NOTE: you will need to add in glPushMatrix/glTranslatef/glRotatef/glPopMatrix commands as necessary
    //DrawOBJ(tankTurret->m_iMeshID);
    //DrawOBJ(tankMainGun->m_iMeshID);
    //DrawOBJ(tankSecondaryGun->m_iMeshID);
    //DrawOBJ(tankWheel->m_iMeshID);

    glPopMatrix(); //load the previously saved (1) transformation matrix (i.e. current position and orientation)
}
```

1. Assemble the rest of the tank, drawing the other parts with correct translations so that they are rendered properly. Do this inside the *draw_tank(float x, float y, float z)* function. The meshes

have already been loaded int for you in the *load_ tank_ objs()* function. You will need to draw a lot of wheels (see the side view of the tank in Figure 4a as a guide of where to place them).

- Note: you will need to add in *glPushMatrix/glTranslatef/glRotatef/glPopMatrix* commands as necessary to place each part in a suitable position. This may be more tricky than it initially seems: You will need to properly position the glPushMatrix() and glPopMatrix() commands in order to ensure that the tank is in a properly defined hierarchy (see Figure 2) or it will not animate properly in the later tasks. Also, not all of the tank meshes have had their local space origins set to sensible positions. You will need to do this by defining appropriate transforms.



(a) The side-view of the tank.

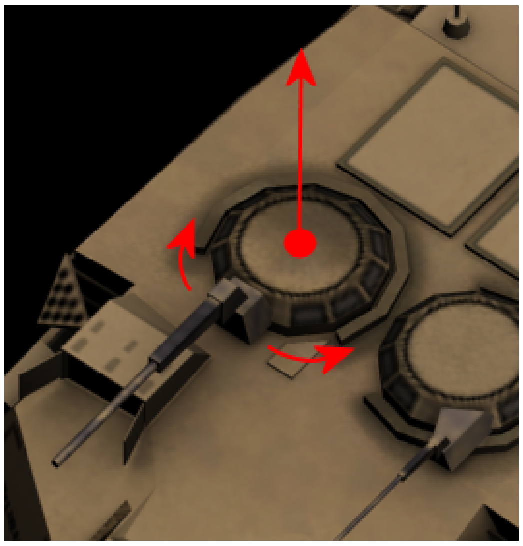

(b) Rotate the secondary gun.

Figure 4: Illustration of the secondary gun in a side view (a) and its axis of rotation (0,1,0) (b).

2. Allow the user to rotate the turret, main gun and secondary guns, and wheels on the tank by pressing various keys. This will involve defining appropriate axes of rotation for each tank part. For example, the secondary gun should rotate around the axis shown in Figure 4b i.e. (0,1,0). You could use two keys per object in order to allow the user to rotate both clockwise and anti-clockwise, respectively, around the axis of rotation.

3. Add each component of the tank into its own display list. To do this, define the display list ID as a global variable, e.g. *int tankBodyID*; for the body of the tank In *load_ tank_ objs()*, after all the objects and textures have been loaded in, generate your new display list with *tankBodyID = glGenLists(1)*; And then compile the tank body into this display list:

```
glNewList(tb,GL_COMPILE);
         DrawOBJ(tankBody->m_iMeshID);
glEndList();
```

   You can now draw the body of the tank by calling *glCallList(tankBodyID)*; from the part of your code that draws the tank.

4. Each mesh object contains an array of faces, vertices, normals and texture coordinates:

```
m_aFaces[];
m_aVertexArray[];
m_aNormalArray[];
m_aTexCoordArray[];
```

To link these together, each face contains a set of vertex, normal and texture coordinate indices:

```
m_aVertexIndicies[3];   //3 of these per face
m_aNormalIndicies[3];   //3 of these per face (one per vertex)
m_aTexCoordIndicies[3]; //3 of these per face (one per vertex)
```

Each vertex has an (x, y, z) coordinates, each normal has (x, y, z) coordinates and each texture coordinates has (u, v) coordinates e.g. *m_ aVertexIndicies[0].x*, *m_ aTexCoordArray[0].u*

Let's say this is your mesh

```
ObjMesh *pMesh;
```

To get the number of faces in the mesh:

```
pMesh->m_iNumberOfFaces;
```

To get a specific face i:

```
ObjFace *pf = &pMesh->m_aFaces[i];
```

To get the index (k) of vertex j in face pf:

```
k = pf->m_aVertexIndicies[j];
```

To get the actual vertex data based on the index:

```
pMesh->m_aVertexArray[k];
```

Example:

pf->m_ aVertexIndicies[0] is the index of vertex 0 in the face (there are a total of 3 vertices per face). Let's say the index turns out to be 4, then: *pMesh->m_ aVertexArray[4].x* is the x coordinate of vertex 4 in the vertex array. The above result is the same as writing:

```
pMesh->m_aVertexArray[pf->m_aVertexIndicies[0]].x
```

To get a normal index for face j:

```
pf->m_aNormalIndicies[j];
```

To get a specific normal k:

```
pMesh->m_aNormalArray[k];
```

To get a texture coord index for face j:

```
pf->m_aTexCoordIndicies[j];
```

To get a specific texture coordinate k:

```
pMesh->m_aTexCoordArray[k];
```

The pseudo-code for drawing an object given that mesh description is:

```
for_each_polygon_in_mesh
{
   for_each_vertex_in_the_poly
   {
      use_the_vertex_index_to_find_vertex_in_the_meshs_array
   }
}
```

Implement your own *DrawObj(...)* function that, given a mesh object of type *ObjMesh* which has been loaded in earlier (using the *LoadOBJ()* command), will draw the mesh onto the screen using OpenGL. To draw a single triangle of the mesh, you will need to use command similar to:

```
glBegin(GL_TRIANGLES);
    glTexCoord2f(u, v);
    glNormal3f(nx, ny, nz);
    glVertex3f(vx, vy, vz);
glEnd();
```

where the values $u, v, nx, ny, nz, vx, vy, vz$ have been taken from the mesh for this specific face using the information on the previous page.

That all sounds complicated, but once you figure it out, the code should only be around 6 or 8 lines.

- Hint: if you get desperate, you can always look at the equivalent function in the *DrawObj()* command in *ObjLoader.cpp*, around line 1127.

2 Bounding Volume Hierarchy (BVH)

In this section, you will calculate and store a hierarchy of bounding spheres for the tank and use it to perform simple collision detection between the tank and a projectile.

2.1 Background

Recall that a bounding sphere is represented by a *centre position* and a *radius*, and can be calculated as follows:

```
find the average vertex position vp_avg, over all the vertices
for (each vertex v in the mesh)
{
    calculate the distance d between vp_avg and v
    if (d is the largest encountered so far)
    set the sphere radius to d
}
```

In order to do the calculations, you will need to access the vertex information of each object. Given a *pmesh* object, such as that you previously used for the tank (here called *tankBody*), you can access the vertices with code similar to the following:

```
tankBody->m_iNumberOfVertices; //number of vertices in the tank
float x,y,z;
//iterate through all the vertices in the tank
for(int i=0; i < tankBody->m_iNumberOfVertices; i++)
{
    x = tankBody->m_aVertexArray[i].x;
    y = tankBody->m_aVertexArray[i].y;
    z = tankBody->m_aVertexArray[i].z;
}
```

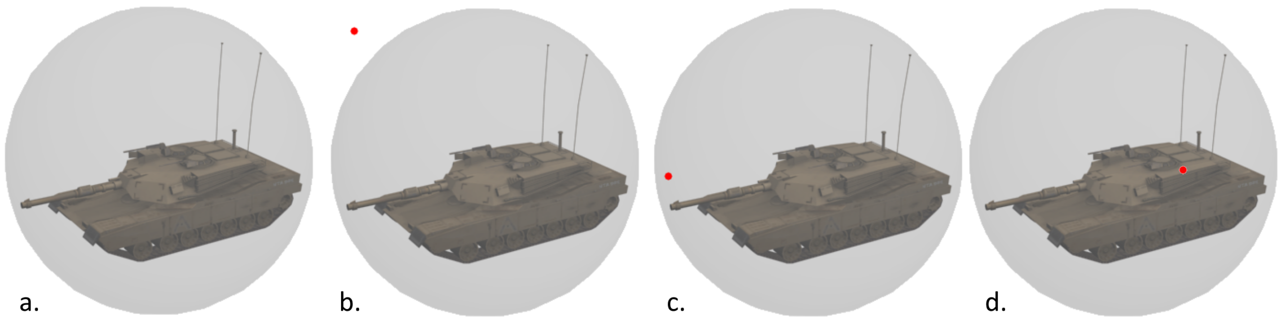


Figure 5: A bounding volume for the tank, in this case a bounding sphere (a), and sample intersection tests with respect to a point (b-d).

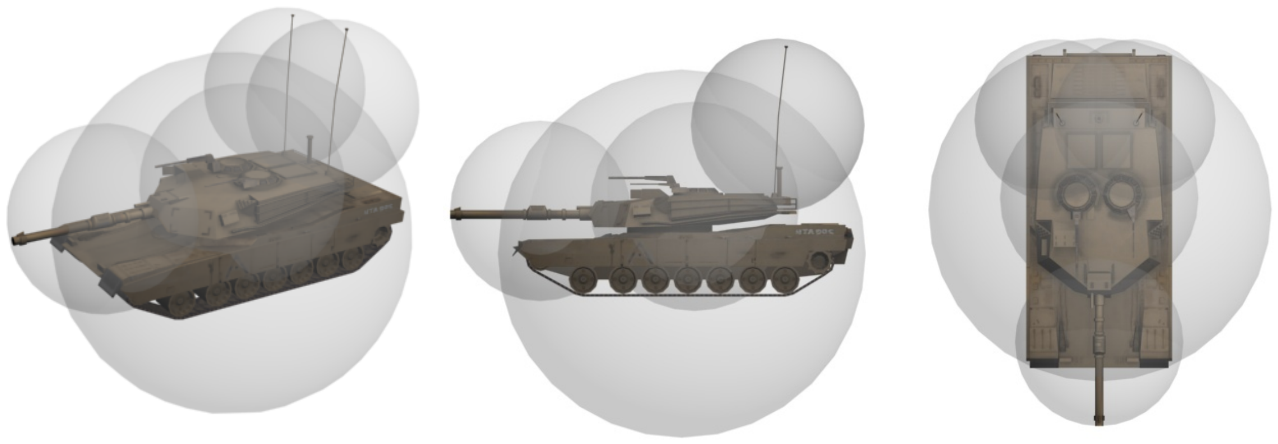


Figure 6: Illustration of the second level of a bounding volume hierarchy for the tank. Each sub-object in the tank is enclosed by its own bounding sphere.

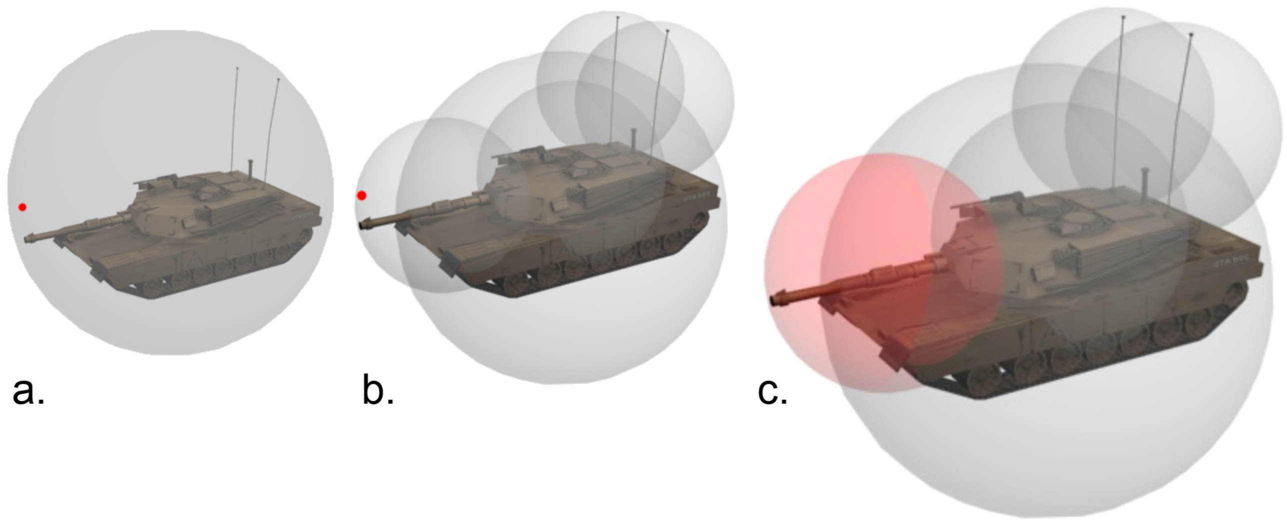


Figure 7: Example intersection tests between the bounding volume hierarchy of the tank and an example point (a-c). The point occurs inside the main bounding sphere (a) and is therefore processed at the next level of the hierarchy (b). There it is found to be potentially colliding with one of the sub volumes in the hierarchy: the turret sub-object (c). Further checks must therefore be conducted between the point and all triangles in the turret sub-object. The point is finally rejected in this case since it is found not to be colliding with the turret sub-object.

2.2 Tasks

1. Using the information above, you should create a new *BoundingSphere* class which stores the bounding sphere details for a single mesh. The class will contain a function to initialise the bounding sphere given an input mesh, in addition to the other relevant details (sphere center, radius, etc).

2. Use the bounding sphere class to create and store a bounding sphere for each of the sub-objects and for the whole tank object, respectively.

3. Create a function to test if a given (x, y, z) position is *potentially* penetrating (a) the tank bounding sphere and, if so, (b) its subparts. The function should first test the position against the bounding sphere encapsulating the whole tank (from task 1). If the position is inside this sphere, then continue to test it with each of the bounding spheres of the sub-objects (from task 2). Print out the details of any bounding spheres that the position is penetrating.

4. Extend the previous question to display the point (representing a projectile) on the screen and allow the user to move it around interactively using the keyboard. Highlight collisions by using the *gluSphere* function to draw the bounding spheres that the point is penetrating. Enabling transparency of the spheres, as seen in Lab 2, is also a possibility.

5. Do the same as question 3 above, except create a function to test a line, rather than a position, with the bounding sphere hierarchy.

    - Hint: You will need to find the closest point on the line to each sphere centre and test this to see if it is inside or outside of each sphere's radius.

6. Integrate the particle system code from Lab 2 into the tank project and define a suitable particle system to simulate the generation of sparks. Generate sparks when the projectile is potentially colliding with the tank centered around the position of the projectile.

- Note: You do not have to find out if the projectile is penetrating into the actual mesh of the tank.

3 Discrete Level Of Detail

This section builds upon the *tankLab.zip* from the previous sections. Additionally, you will need the *tankLOD.zip* file. The program contains some new .obj files for a new, lower resolution version of the tank. Extract the low details .obj files to the same directory as those of your original tank model (*tankobjs*).

3.1 Background

You can define a new low detail level version of the tank and use the obj loading library to load it in. Do not change any of the previous code for loading the original tank – you will need to use that model again, so just add new lines to load the new LOD version. Assemble (it should be similar to the last model with a few tweaks) and display the new low level-of-detail version of the tank on the screen beside the original version and compare them. The lower level-of-detail tank does not look as detailed of course, and contains fewer polygons than the more detailed version.

Your overall object and function definitions should look similar to those below:

```
ObjMesh* tankBody = NULL;
ObjMesh* tankTurret = NULL;
ObjMesh* tankMainGun = NULL;
ObjMesh* tankSecondaryGun = NULL;
ObjMesh* tankWheel = NULL;

ObjMesh* tankBodyLow = NULL;
ObjMesh* tankTurretLow = NULL;
ObjMesh* tankMainGunLow = NULL;
ObjMesh* tankSecondaryGunLow = NULL;
ObjMesh* tankWheelLow = NULL;

void load_tank_objs(void);
void load_low_detail_tank_objs(void);
```

In your main() function, code can be added in order to load in all of the new low detail tank objects:

```
  //load the tank meshes and texture
  load_tank_objs();
  load_low_detail_tank_objs();
```

Here is an example of the low detail version of the main body of the tank.

As for the high detail version of the tank, you should define a drawing function for the low detail version, with a function such as:

```
//draw low detail version of the tank model
void draw_tank_low(float x, float y, float z)
```

Note that the low polygon versions of the various tank objects do not have any texture coordinates associated with them, so textures will not display properly on them. Because of this, you may achieve

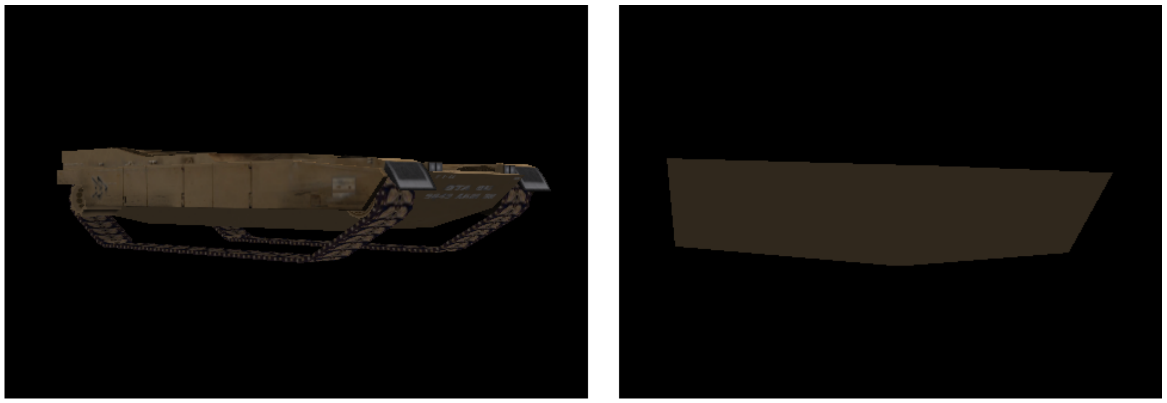


Figure 8: Two details levels representing the high detail *tankbody* mesh (left) and the low detail *tankbodyLow* mesh (right).

a better visual effect for the low detail version of the tank by using a single, well-chosen colour for each of the tank objects. This can be done by disabling lighting and texturing and using the glColor3ub function. In the case of the body of the tank shown in Figure 8 above, the following code was used:

```
glDisable(GL_TEXTURE_2D);  //disable lighting
glDisable(GL_LIGHTING);    //disable texturing
glColor3ub(49,41,30);      //select RGB colour (49,41,30)
DrawOBJ(tankBodyLow->m_iMeshID);
```

3.2 Tasks

1. Define a new low detail level version of the tank as described above, using the obj loading library to load it in. Determine and set suitable colors of each of the mesh parts. Use transformations to position them in order to compose the tank.

2. Based on task 1 above, start to move the lower level of detail tank back further along the –Z axis (further into the screen). Keep doing this until the differences between it and the higher resolution tank appear to be fairly minimal i.e. until you have difficulty seeing a difference between both versions. Make a note of this distance along the –Z axis: we will refer to it as the threshold distance, *d*.

   - Hint: You may need to extend the *far plane* of the camera in order to be able to move the tank back far enough. To do this, see the *gluPerspective* command.

3. Implement a discrete LOD technique based on viewing distance so that, as the viewer presses a key to move the tank further away from the object, a level of detail switch takes place beyond distance *d.* above. Similarly, the high detail version should be displayed when the tank is moved closer to the viewer, inside the threshold distance, d.