3D Modeling of JQH Arena using OpenGL

Vanna Bushong
Missouri State University

December 9, 2010
CSC 625 (Folder: Bushong10)

Abstract

3D modeling of structures has become a common technique for visualizing a location without physically being there. One of the interesting applications of this technology is the modeling of sports arenas and stadiums, resulting in a virtual environment fans may view before purchasing tickets. The following report describes the development of an application for generating and viewing a 3D model of JQH Arena at Missouri State University. A laser scanner was used to collect data points for the model, which were stored in text files and displayed as coordinates in a 3D world using OpenGL and the GLUT (OpenGL Utility Toolkit) library. Various techniques were implemented for enhancing the look of the virtual arena, including OpenGL primitive drawing and texture mapping. The resulting application - the JQH Arena Viewer - allows users to select a section of the arena on the lower level and view the model from that section, as if they were virtually seated there.

Keywords: 3D modeling, laser scanners, sports arenas, OpenGL, Missouri State University

INTRODUCTION

Project Background: In recent years, the technology available for scanning the dimensions of buildings and other structures has been put to use in a growing number of forms. Bui et al. [3] have done research on laser scanning applications for construction and maintenance, and Unver et. al. [7] demonstrated scanning of automotive designs. Laser scanners are used to collect data on buildings, machinery, areas of land, or other environments by shooting a laser and calculating the distance between the scanner and the laser dot on the subject. The data points are used to build a model of the subject in three dimensions.

3D modeling of sports arenas has become a popular trend, as consumers want to see the view from their seats before spending money on tickets. Many professional sports venues already have models, such as Yankee Stadium, Daytona International Speedway, and Cowboys Stadium, to name a few. Ballena Technologies Inc. in Alameda, California, offers a market-leading product for sports venue visualization called Seats3D [2]. The product is used by a growing number of clients, including teams in the National Basketball Association, Major League Baseball, National Hockey League, National Football League, Major League Soccer, and more. IOMEDIA in New York, New York, has developed a "Virtual Venue" for the New York Yankees which offers additional options for viewing the entire stadium from different angles and at different times of day for an idea of where sunlight will hit the seats [6, 4]. These seat viewing applications are powerful marketing tools that give fans a realistic look at where their seats may be.

JQH Arena in Springfield, Missouri, is home of the Missouri State Bears and Lady Bears basketball teams. The 11,000 seat arena was completed in 2008 and also hosts concerts, Professional Bull Riders (PBR), and other special events. The arena currently does not have a 3D seat viewing application, so it was an ideal location for conducting this research project. The initial goals for December 2010 were:

- Collect 3D coordinates of JQH Arena using a laser scanner.
- Create a 3D model of the arena by building a mesh from the data points.
- Develop a user interface that allows users to view the model from each seating section on the lower level.
- Allow for panning of the camera while viewing a section by clicking and dragging the mouse over the model.

Program Design: Before developing the program, the first step of the project was to collect data from the arena using a laser scanner. The data points were separated and sorted based on the horizontal angle of the scanner at the position of each point. This process is described in greater detail in the "Data Collection" section.

Once the text files were prepared, the next step was designing the program to display the data. The following outlines the logical flow of the program, with each step and the algorithms involved being described in subsequent sections of this report:

- Run initialization and startup functions
- Load pixel maps
- Load data points for the model
- Draw 2D seat map
- Check for mouse input
    - On click, determine which section was clicked
    - Position camera appropriately
    - Draw model
    - Draw HUD
    - Check for mouse input on the model
        - On click and drag, pan camera
        - If back button clicked, draw 2D seat map

Results: The resulting program - named the "JQH Arena Viewer" - nearly meets all of the goals established for December, with a few modifications. Upon startup, the program displays a user interface that allows for selection of a seating area on the lower level of the arena, as shown in Figure 1.

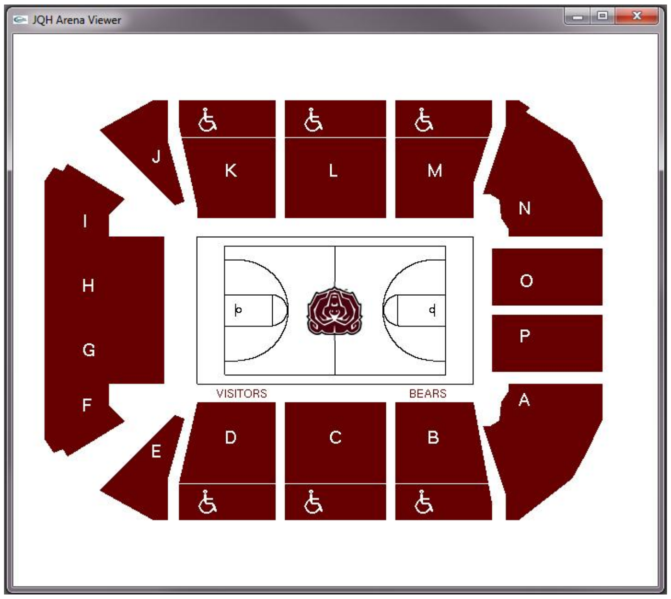


Figure 1. JQH Arena Viewer user interface.

When the user clicks a section, the camera is positioned appropriately, and the model and HUD are drawn, as shown in Figure 2. The user can click and drag the mouse over the model to pan the camera, or click the "Back to Seat Map" button to return to the main screen.

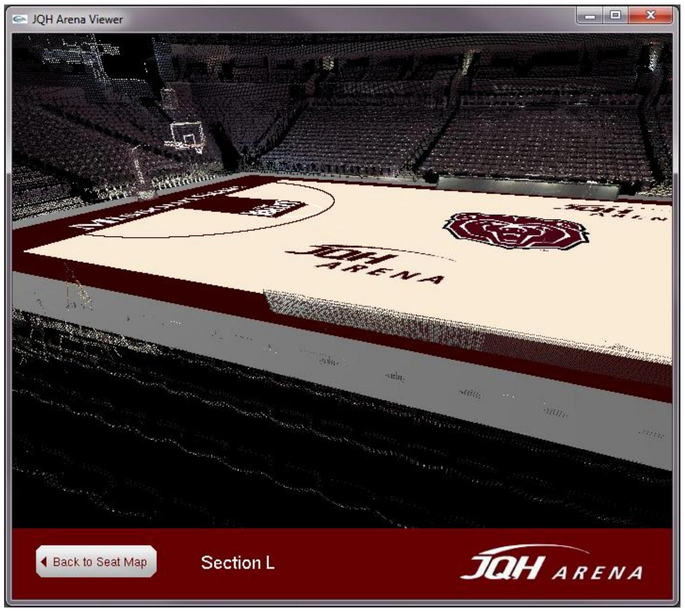


Figure 2. A view from Section L.

The main difference between the results and the original goals is that currently there is no mesh structure built from the data points. Instead, the floor and court have been drawn using polygons and texture mapping. The rest of the arena (seats, ceiling, etc.) is simply composed of a bunch of dots. Although it was not the original intent, this solution is a suitable workaround until further research can be completed.

DATA COLLECTION AND ANALYSIS

The Scanning Process: Laser scanning of JQH Arena was performed on September 30, 2010, with the assistance of Dr. Kerry Slattery, Assistant Professor of Technology and Construction Management, of Missouri State University. Dr. Slattery's scanner - a Leica ScanStation2 - was initially placed in the center of the basketball court and connected to his computer. The computer runs Leica Cyclone, a software program to control the scanner, and Leica Cloudworx to analyze the data collected.

Two reflective targets were placed on tripods on opposite sides of the court. The scanner emitted a laser beam to detect both targets and establish them as fixed points among the x, y, and z coordinates with the scanner itself being the origin. The targets were not moved throughout the process so that each scan could align with the others at those coordinates. Figure 3 shows the setup of the scanner and targets.

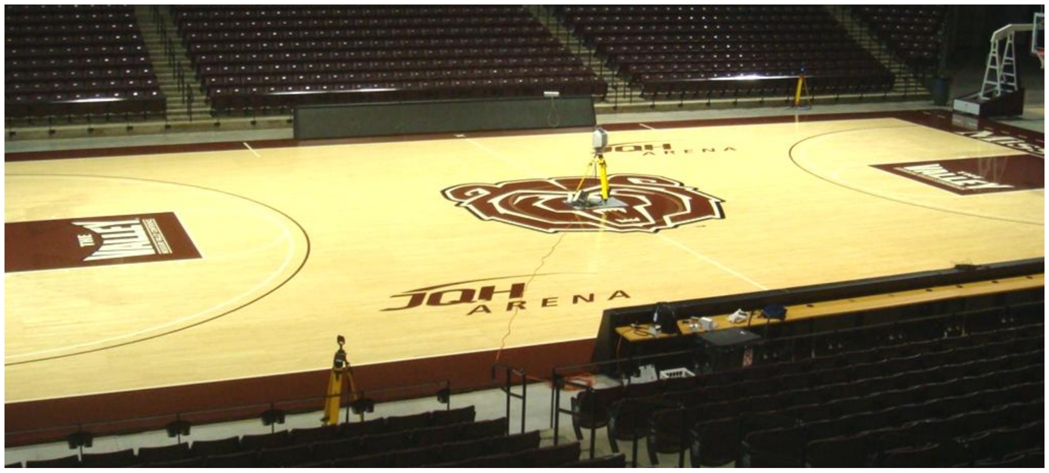


Figure 3. Setup of scanning equipment. The yellow tripods hold the scanner at center court and the targets on each side.

The scanner first took photographs of the lower half of the stadium, then the upper half and ceiling on the second rotation. Then a scan was completed with a resolution of 1/4 degree vertical and horizontal, which resulted in 540 vertical and 1440 horizontal points. A second scan was performed at a higher resolution of 3/16 degree, which resulted in 720x1920 points.

The photographing and scanning process was repeated for three more sections of the arena. For these scans, an interval of 1/8 degree was used and the scanner was set to rotate 184 degrees facing the court to ensure accurate data for a full 180 degree view from that position. The resulting data was 1080x1440 points. For these scans, the scanner was first set on the south side of the arena at the top of the lower section, facing the half court line. Then it was moved to the east side at the center of the upper level. The final scan was taken from the north side, at the center of the upper level.

The resulting data from the scans was stored in five separate text files. The number on the first line of each file gives the number of points. Each line that follows contains the x, y, and z coordinates, intensity, and r, g, b, values for every data point. Figure 4 shows a partial view of one of the files.

```
690136d
87.487203 -53.348134 -1.208786 -562 78 69 86
-103.886965 -65.682997 -5.188378 -607 46 40 44
-103.475860 -66.057858 -5.187277 -803 48 43 43
-95.561230 -57.541088 -5.201594 -876 67 66 64
-124.384064 -80.162798 -0.387476 -886 78 68 69
-126.216917 -79.793645 -4.338934 -1088 34 28 34
```

Figure 4. Sample lines from a data file produced by a scan.

The Sorting Process: One of the initial challenges of this project was preparing the data for use in the program. The data files created by the scanner were not in any sorted order, making it difficult to understand where the points were located and impossible to build a mesh. To break up the data into manageable pieces and sort it for better usability, it is helpful to understand the orientation of the world in which the scanner collects data.

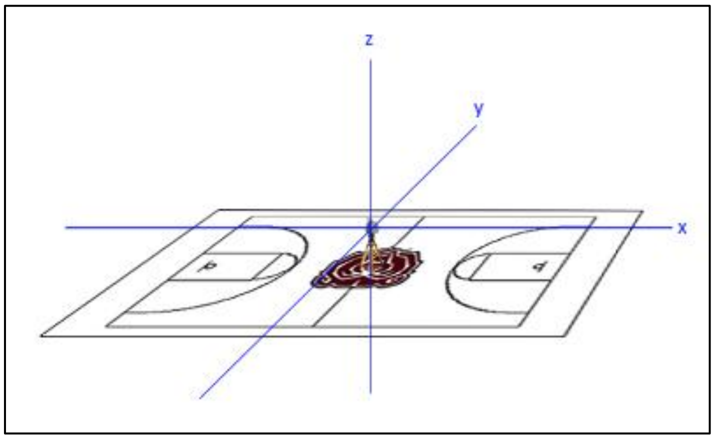


Figure 5. Orientation of the scanner.

As shown in Figure 5, at the origin of the scanner, the x and y axes run parallel to the surface of the court, while the z axis serves as the height. As the scanner rotates, its horizontal angle is constantly changing. At each interval, every 1/4 degree for example, the laser moves up at an increasing vertical angle, collecting data points. By calculating the horizontal angle of every point, it was possible to separate the data from each scan into four different files, based on which quadrant it lies in. See Figure 6 for an illustration.

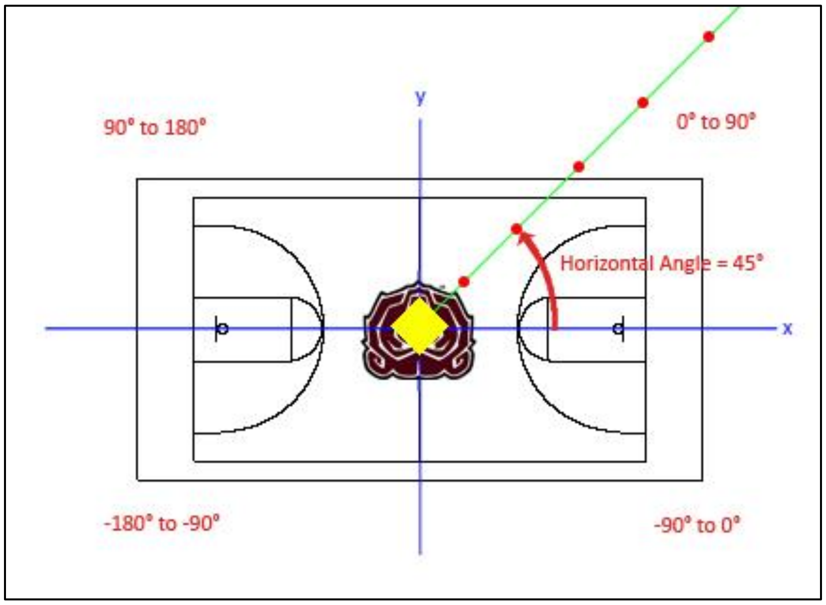


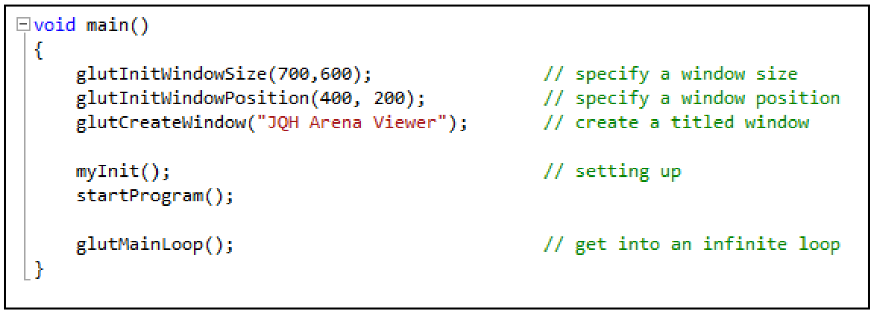
Figure 6. The four quadrants of the court; not to scale. The green line represents the laser at a horizontal angle of 45 degrees, and the red dots represent points along this line. By calculating the horizontal angle of every point, it was possible to split the data into four files, one for each quadrant.

Horizontal angles were calculated by importing the data to Microsoft Excel and applying the arctangent function, then converting from radians to degrees. The data was then filtered and separated according to quadrant. At this point, the column of intensity values was removed, as this data is not currently needed for the program, leaving a file with only x, y, z, and r, g, b values. The data files in this state are the ones currently being used in the program.

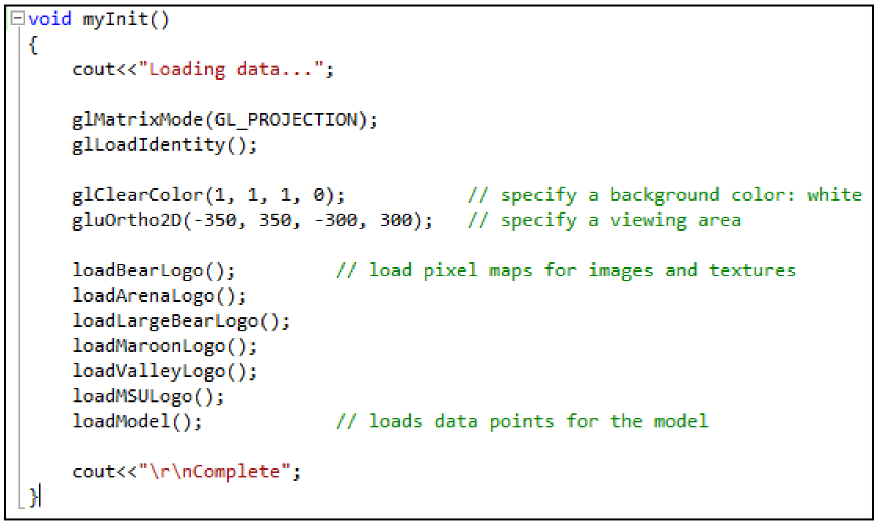
INITIALIZATION AND STARTUP FUNCTIONS

The JQH Arena Viewer was written in C++ using OpenGL, which is a powerful, industry standard 2D and 3D graphics application programming interface (API) [5]. Also, the OpenGL Utility Toolkit (GLUT) library provided many of the necessary drawing functions. GLUT is a set of support libraries that provide basic functions for input, windows, and menus, which OpenGL does not support on its own [1].

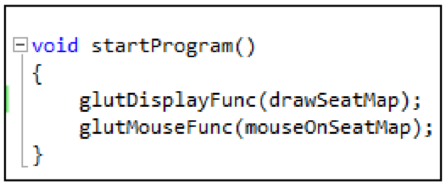
void main(): The program begins in the main function, which creates a window and sets its size and position on the screen. The function calls myInit() for initialization, startProgram() to begin drawing, and glutMainLoop(), which sends the programs into an infinite loop for continuous drawing to the screen.


```
void main()
{
    glutInitWindowSize(700,600);                // specify a window size
    glutInitWindowPosition(400, 200);           // specify a window position
    glutCreateWindow("JQH Arena Viewer");       // create a titled window

    myInit();                                   // setting up
    startProgram();

    glutMainLoop();                             // get into an infinite loop
}
```

void myInit(): The initialization function calls all of the routines for loading images and the 3D coordinate data. One of the challenges of this program was that the view must be able to switch back and forth from 2D to 3D. Since the 2D seat map is the first thing to be drawn, projection mode and gluOrtho2D are initially used to set up the screen.

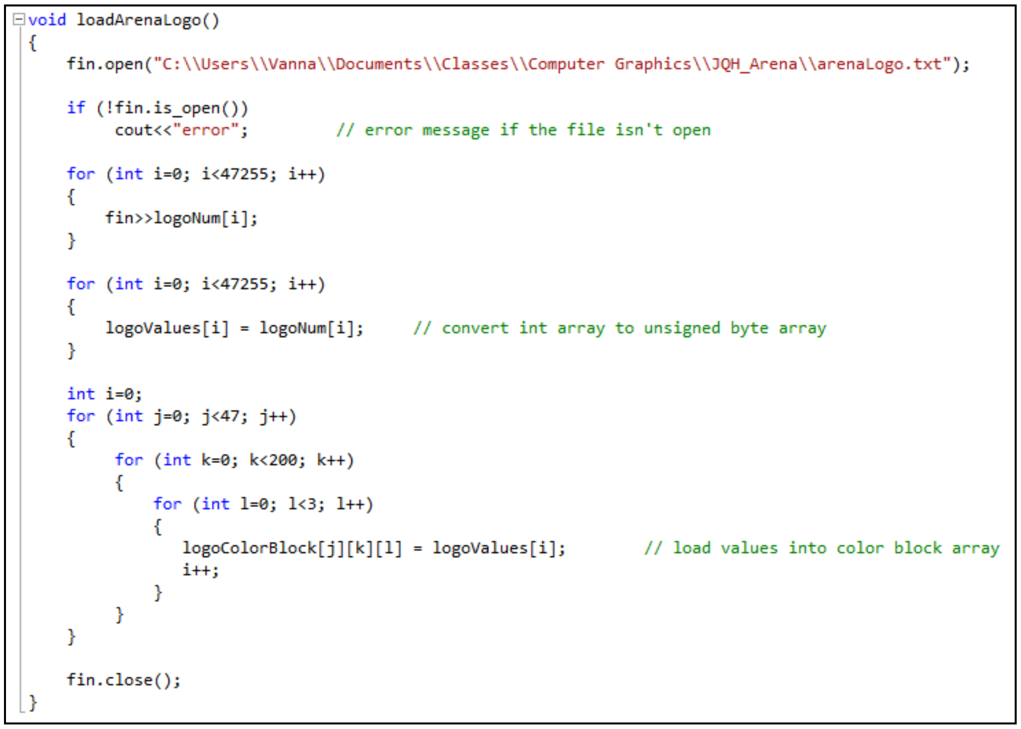

```
void myInit()
{
    cout<<"Loading data...";

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glClearColor(1, 1, 1, 0);           // specify a background color: white
    gluOrtho2D(-350, 350, -300, 300);   // specify a viewing area

    loadBearLogo();         // load pixel maps for images and textures
    loadArenaLogo();
    loadLargeBearLogo();
    loadMaroonLogo();
    loadValleyLogo();
    loadMSULogo();
    loadModel();            // loads data points for the model

    cout<<"\r\nComplete";
}
```

void startProgram(): This short routine registers a callback for displaying the 2D seat map and checks for a mouse click on a seating section using glutMouseFunc.


```
void startProgram()
{
    glutDisplayFunc(drawSeatMap);
    glutMouseFunc(mouseOnSeatMap);
}
```

LOADING DATA

Pixel Maps: A pixel map is an array of unsigned bytes containing the rgb values of each pixel in an image. For this project, six different images were saved as text files with their color data stored as integers using the GNU Image Manipulation Program (GIMP). The following example, loadArenaLogo(), opens the arenaLogo.txt file and reads the integers into an array (logoNum[i]). These values are assigned to an array of unsigned bytes (logoValues[i]), the necessary data type for a pixel map. The final for loop stores every three values (r,g,b) into a row, column position of the logoColorBlock array. The number of rows and columns of the pixel map corresponds to the height and width of the image, in this case 47x200.

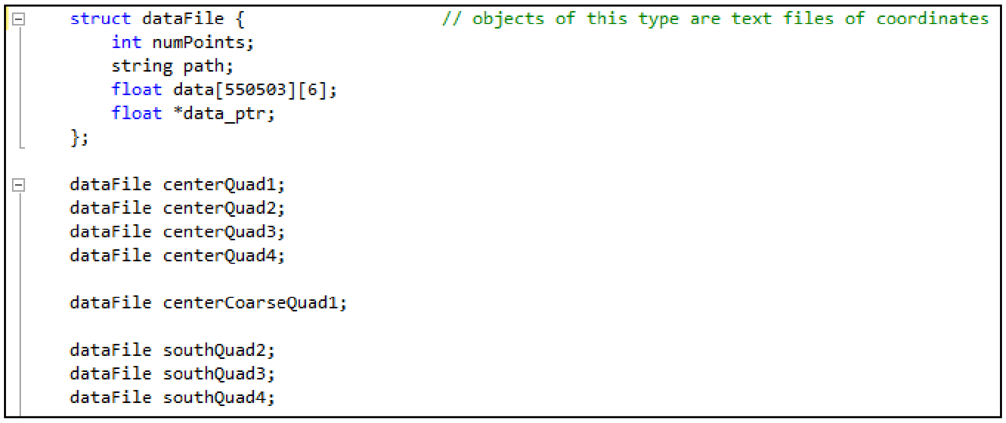


```
void loadArenaLogo()
{
    fin.open("C:\\Users\\Vanna\\Documents\\Classes\\Computer Graphics\\JQH_Arena\\arenaLogo.txt");

    if (!fin.is_open())
         cout<<"error";         // error message if the file isn't open

    for (int i=0; i<47255; i++)
    {
        fin>>logoNum[i];
    }

    for (int i=0; i<47255; i++)
    {
        logoValues[i] = logoNum[i];     // convert int array to unsigned byte array
    }

    int i=0;
    for (int j=0; j<47; j++)
    {
         for (int k=0; k<200; k++)
         {
             for (int l=0; l<3; l++)
             {
                logoColorBlock[j][k][l] = logoValues[i];        // load values into color block array
                i++;
             }
         }
    }

    fin.close();
}
```

Once the pixel maps are stored, they may be drawn directly to the screen in a 2D view or bound as textures in a 3D rendering, as explained in following sections.

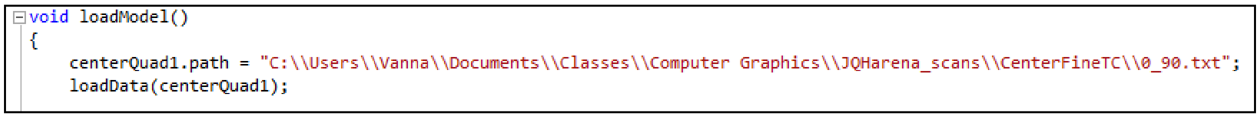
3D Coordinates: Since the coordinates of the arena are stored in several different text files, it was inefficient to write separate routines for loading each of them. To solve this problem, a structure called "dataFile" was declared so each data file could be treated as an object with its own members. The member array of floats called "data" is set to a size of 550,503 rows, since this is the largest number of points in a single file. The array has 6 columns for the x, y, z, and r, g, b values.

Objects of type dataFile were declared for each of the files used in the program. For example, centerQuad1 contains data from the scan taken at center court that lies in quadrant 1.



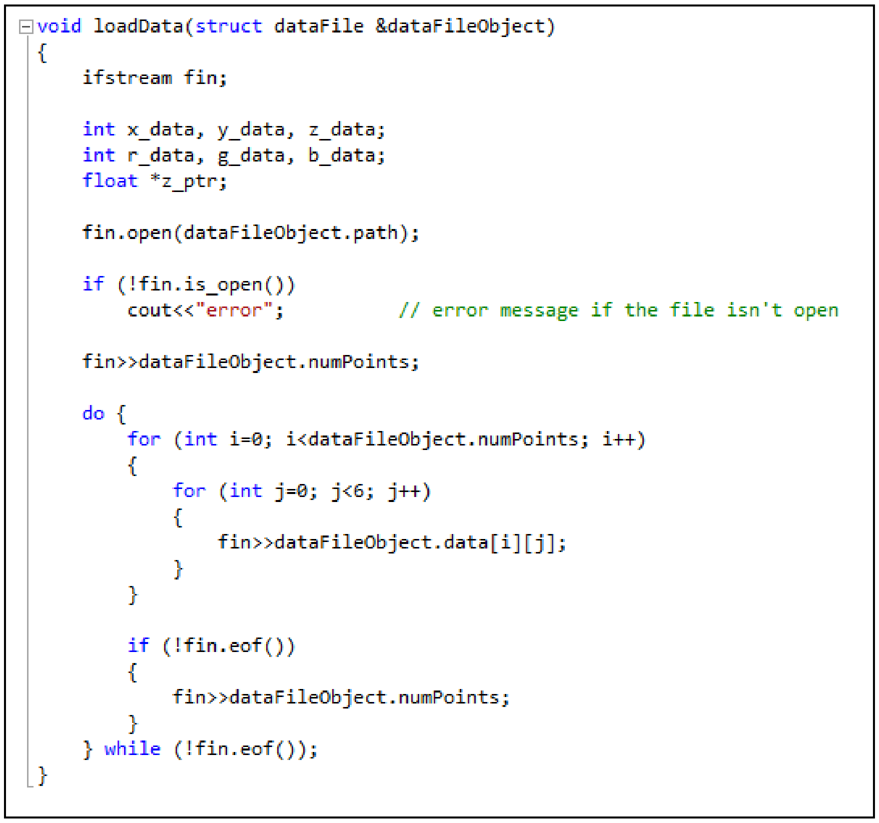
```
struct dataFile {                          // objects of this type are text files of coordinates
    int numPoints;
    string path;
    float data[550503][6];
    float *data_ptr;
};

dataFile centerQuad1;
dataFile centerQuad2;
dataFile centerQuad3;
dataFile centerQuad4;

dataFile centerCoarseQuad1;

dataFile southQuad2;
dataFile southQuad3;
dataFile southQuad4;
```

The loadModel() function sets the path for each dataFile object and makes a call to the loadData function, passing in the object as a parameter. The following code snippet shows an example of this for the centerQuad1 object.



```
void loadModel()
{
    centerQuad1.path = "C:\\Users\\Vanna\\Documents\\Classes\\Computer Graphics\\JQHarena_scans\\CenterFineTC\\0_90.txt";
    loadData(centerQuad1);
```

The loadData function opens the path of the dataFile object, reads the number of points from the first line, then stores the x, y, z, and r, g, b values for each point into dataFileObject.data[i][j]. This array will be used later in the program for drawing the points.



```
void loadData(struct dataFile &dataFileObject)
{
    ifstream fin;

    int x_data, y_data, z_data;
    int r_data, g_data, b_data;
    float *z_ptr;

    fin.open(dataFileObject.path);

    if (!fin.is_open())
        cout<<"error";          // error message if the file isn't open

    fin>>dataFileObject.numPoints;

    do {
        for (int i=0; i<dataFileObject.numPoints; i++)
        {
            for (int j=0; j<6; j++)
            {
                fin>>dataFileObject.data[i][j];
            }
        }

        if (!fin.eof())
        {
            fin>>dataFileObject.numPoints;
        }
    } while (!fin.eof());
}
```

DRAWING THE 2D SEAT MAP

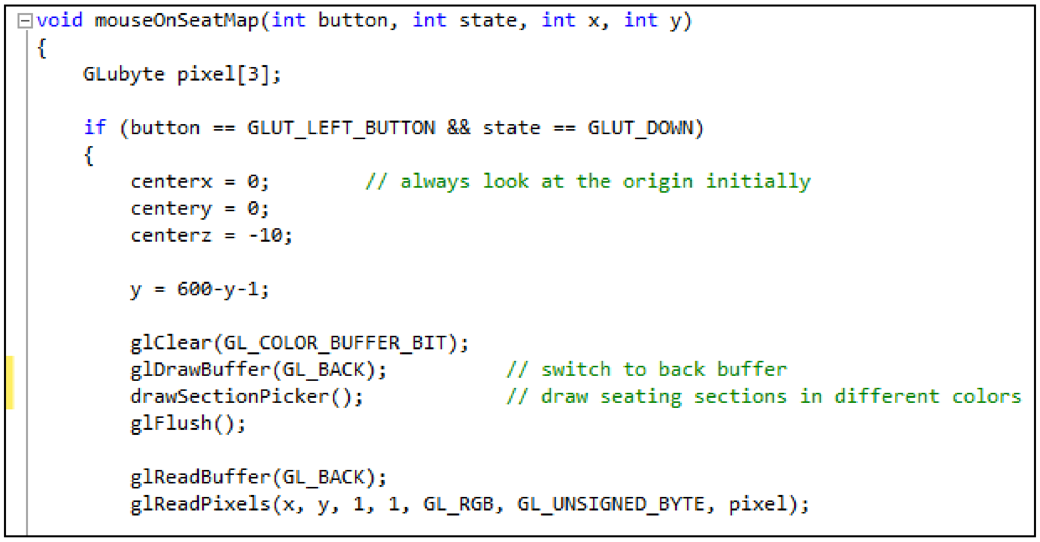
After the data has been loaded, the startProgram() function registers a display callback to drawSeatMap(). This is a fairly straightforward routine, consisting of various subroutines for drawing each object on the map.

- The court and seating sections are drawn using OpenGL functions for points, lines, and polygons.
- The text "VISITORS" and "BEARS" are displayed using the glutBitmapCharacter function, which outputs a character string.
- The handicapped icons are bitmaps, drawn using the glBitmap function.
- The bear logo is drawn using glDrawPixels, which reads from the pixel map array created during initialization.

At this point, everything in the buffer needs to be displayed, so glFlush() is called at the end of the routine.

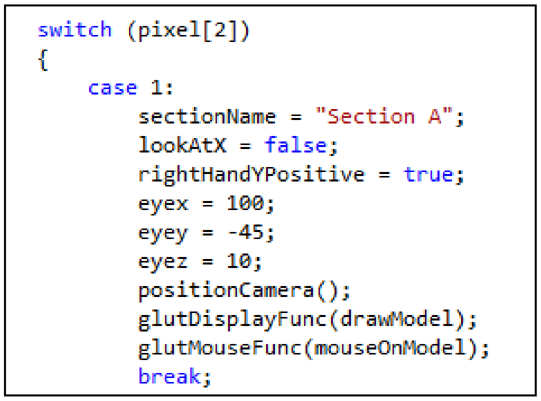
MOUSE INPUT ON SEAT MAP

After the seat map has been drawn, the program continuously checks for mouse input with the glutMouseFunc(mouseOnSeatMap) callback. An initial challenge was figuring out how to know if the coordinates of the user's mouse click were within the coordinates of a seating section. Since the sections are not perfect rectangles, it would not work to assign a maximum and minimum value for x and y, and check to see if the mouse click fell within those coordinates. To solve this problem, the mouseOnSeatMap function switches to the back buffer and calls drawSectionPicker(). This routine draws all of the polygons forming the seating sections in different colors. When glReadPixels is called, the rgb values of the pixel that was clicked are stored into an array of unsigned bytes called "pixel."



```
void mouseOnSeatMap(int button, int state, int x, int y)
{
    GLubyte pixel[3];

    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        centerx = 0;        // always look at the origin initially
        centery = 0;
        centerz = -10;

        y = 600-y-1;

        glClear(GL_COLOR_BUFFER_BIT);
        glDrawBuffer(GL_BACK);          // switch to back buffer
        drawSectionPicker();            // draw seating sections in different colors
        glFlush();

        glReadBuffer(GL_BACK);
        glReadPixels(x, y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, pixel);
```

Each section was assigned a red and green value of 0, but the blue value was incremented by 1 for each section. So section A's values were (0, 0, 1), sections B's were (0, 0, 2), and so forth. Knowing this, a switch statement was written to check for the third value in the pixel array, and to handle the events for the section corresponding to that color value. The following code shows what happens when the blue value is a 1, meaning section A was clicked. The initial camera position is set, and callbacks are used for drawing the model and checking for mouse input. These functions will be described in detail in the remaining sections.



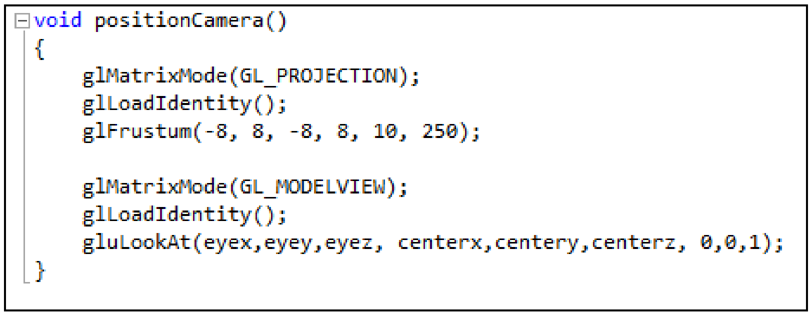
```
switch (pixel[2])
{
    case 1:
        sectionName = "Section A";
        lookAtX = false;
        rightHandYPositive = true;
        eyex = 100;
        eyey = -45;
        eyez = 10;
        positionCamera();
        glutDisplayFunc(drawModel);
        glutMouseFunc(mouseOnModel);
        break;
```

POSITIONING THE CAMERA

Each time a section on the 2D seat map is clicked, the camera position must be set accordingly. This position must also be updated when the user clicks and drags the model to simulate looking around the arena. The positionCamera() function is called to handle these events.

Whenever positionCamera() is called, the scene needs to be drawn in 3D. Therefore, the function sets up a viewing volume in projection mode using glFrustum. It was difficult to find the best settings for viewing this model, but after plenty of experimentation with the numbers, the parameters in the code below give the best results. The left, right, bottom, and top coordinates of the near plane are defined by the first four parameters of glFrustum: -8, 8, -8, 8. The last two parameters, 10 and 250, represent the distances to the near and far planes, respectively.

After switching to modelview mode, the gluLookAt function sets the location of the camera, the coordinate it is looking at, and the up vector. In the code below, eyex, eyey, and eyez are variables for the camera's location, while centerx, centery, and centerz represent the coordinate it is pointing towards. The last three parameters specify which vector is "up" - x, y, or z. Since the z axis represents the height of the model, z will always be set to 1.

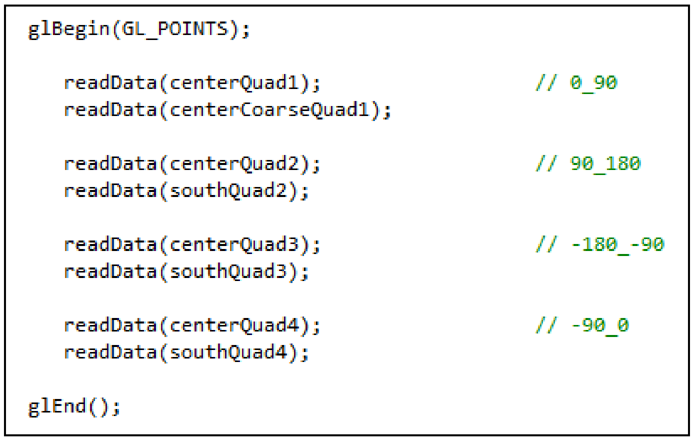

```
void positionCamera()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-8, 8, -8, 8, 10, 250);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(eyex,eyey,eyez, centerx,centery,centerz, 0,0,1);
}
```

The camera position is not perfect at this point, but it does provide a reasonable viewing area for the arena. As it is now, the viewer appears to be about halfway up each section of seats, but too high on the z axis to actually be sitting down. This is a problem that will be addressed in future work on this project.

DRAWING THE MODEL

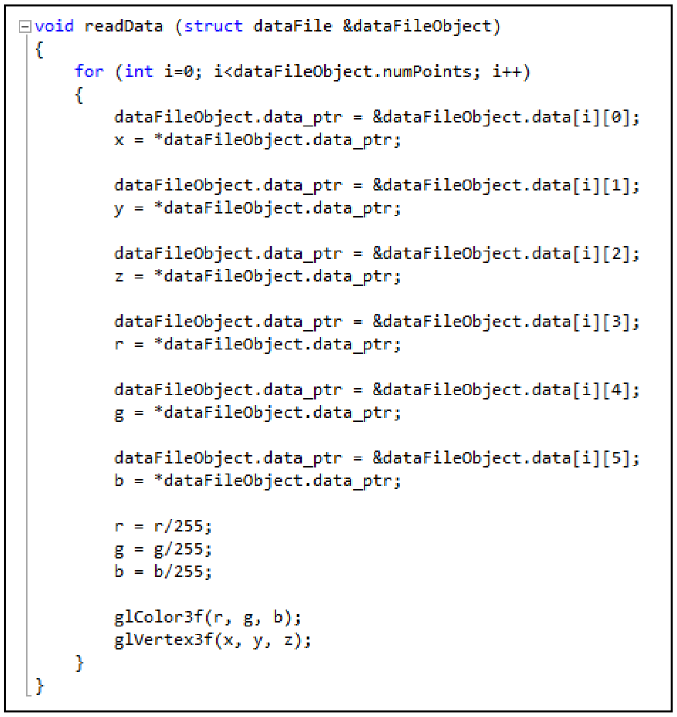
The drawModel() function is where the coordinate data is actually drawn to the screen. At this point in the research, there are no successful algorithms for building a perfect mesh, so each coordinate is simply drawn as a point. To make the scene more realistic, the floor and basketball court have been drawn using points, polygons, and texture mapping. The drawModel() function initially sets up the scene for 3D rendering by enabling depth testing and setting the background color to black. After drawing the data points and additional polygons, drawHud() is called and glFlush() displays the buffer contents on the screen.

Displaying Coordinate Data: Coordinates of the model are displayed by reading the data from each dataFile object. In order to display a better picture, it was necessary to have as many dots as possible without having so many that performance was noticeably hurt. The current solution is to have two scans for each quadrant. The centerQuad scans - used in each quadrant - were taken from the center of the court with an interval of 3/16 degree. The southQuad scans are used in quadrants 2, 3, and 4, but did not return the best quality data in quadrant 1, for reasons not completely known. Data from centerCoarseQuad1 is used in its place, which is part of a scan taken from center court at an interval of 1/4 degree.



```
glBegin(GL_POINTS);

   readData(centerQuad1);                    // 0_90
   readData(centerCoarseQuad1);

   readData(centerQuad2);                    // 90_180
   readData(southQuad2);

   readData(centerQuad3);                    // -180_-90
   readData(southQuad3);

   readData(centerQuad4);                    // -90_0
   readData(southQuad4);

glEnd();
```

The readData function takes a dataFile object as its parameter, loops through the object's data array, and sets a vertex for each point by reading the x, y, z, and r, g, b values.



```
void readData (struct dataFile &dataFileObject)
{
    for (int i=0; i<dataFileObject.numPoints; i++)
    {
        dataFileObject.data_ptr = &dataFileObject.data[i][0];
        x = *dataFileObject.data_ptr;

        dataFileObject.data_ptr = &dataFileObject.data[i][1];
        y = *dataFileObject.data_ptr;

        dataFileObject.data_ptr = &dataFileObject.data[i][2];
        z = *dataFileObject.data_ptr;

        dataFileObject.data_ptr = &dataFileObject.data[i][3];
        r = *dataFileObject.data_ptr;

        dataFileObject.data_ptr = &dataFileObject.data[i][4];
        g = *dataFileObject.data_ptr;

        dataFileObject.data_ptr = &dataFileObject.data[i][5];
        b = *dataFileObject.data_ptr;

        r = r/255;
        g = g/255;
        b = b/255;

        glColor3f(r, g, b);
        glVertex3f(x, y, z);
    }
}
```

Texture Mapping: To improve the overall look of the model, the floor and basketball court were drawn using OpenGL functions for points and polygons. Four different textures were applied to display the images on the surface of the court: the bear logo, JQH Arena logo, Missouri Valley Conference logo, and the words Missouri State written across the baseline.

To map textures in OpenGL, an array of unsigned integers specifies the number of texture objects, in this case four. The glGenTextures() function sets up four numbers as unique identifiers for each texture. In the code below, glBindTexture() indicates that textureObjects[0] will be a 2D texture. The parameters for this object are subsequently listed, followed by glTexImage2D(), which specifies that this texture comes from the array of unsigned bytes "largeBearColorBlock." The texture binding process is repeated for each of the other three textures, but not shown below.

```
glPixelStorei (GL_UNPACK_ALIGNMENT, 1);
GLuint textureObjects[4];

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glGenTextures(4, textureObjects);

glBindTexture(GL_TEXTURE_2D, textureObjects[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 240, 210, 0, GL_RGB, GL_UNSIGNED_BYTE, largeBearColorBlock);
```

The coordinates of a texture object are then mapped to the coordinates of a polygon using the glTexCoord2f() function as shown below.

```
glEnable(GL_TEXTURE_2D);                                // large bear logo
glBindTexture(GL_TEXTURE_2D, textureObjects[0]);
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(8, 8, -4);        // lower left
   glTexCoord2f(0.0, 1.0); glVertex3f(8, -8, -4);       // upper left
   glTexCoord2f(1.0, 1.0); glVertex3f(-8, -8, -4);      // upper right
   glTexCoord2f(1.0, 0.0); glVertex3f(-8, 8, -4);       // lower right
glEnd();
glDisable(GL_TEXTURE_2D);
```

DRAWING THE HUD

After the 3D model has been drawn, the drawModel() function calls drawHud() to create a 2D heads-up display at the bottom of the screen, as shown in Figure 7. At this point, the program must switch back to drawing in 2D, so the projection and modelview matrices are set to identity, glOrtho is used to set up a parallel projection, and depth testing is disabled.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-350, 350, -300, 300, -300, 300);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glDisable(GL_DEPTH_TEST);
```

The HUD is then drawn using polygons and bitmap characters, and the JQH Arena logo is a pixel map displayed via glDrawPixels().

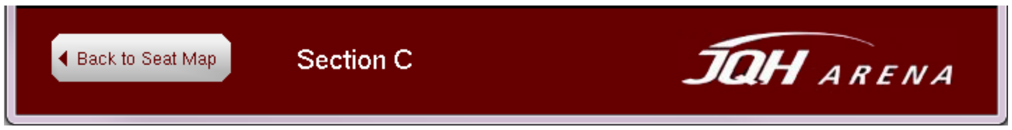


Figure 7. The HUD as displayed at Section C.

One of the future goals for this project is to make the HUD more elaborate with additional features. There will eventually be a "Go to Section" button, which pulls up a menu allowing users to select another section to view without having to return to the seat map. The possibility exists that a user might someday type in his or her section, row, and seat number, and the view will update to that exact location.

MOUSE INPUT ON MODEL

The mouseOnModel() function converts the screen coordinates of the mouse to world coordinates, then checks for two different scenarios. In one case, the user clicks the mouse anywhere over the model (not the HUD), triggering a callback to glutMotionFunc () for panning the camera. In the other situation, the user clicks the "Back to Seat Map" button on the HUD.

Panning the Camera: The ability to pan the camera gives the user a more realistic view of the arena. The panning event is triggered when the user clicks anywhere within the coordinates of the screen where the model is drawn, excluding the HUD. The x and y coordinates at the point of the mouse click are saved as previous_x and previous_y, which are used in the pan() function.

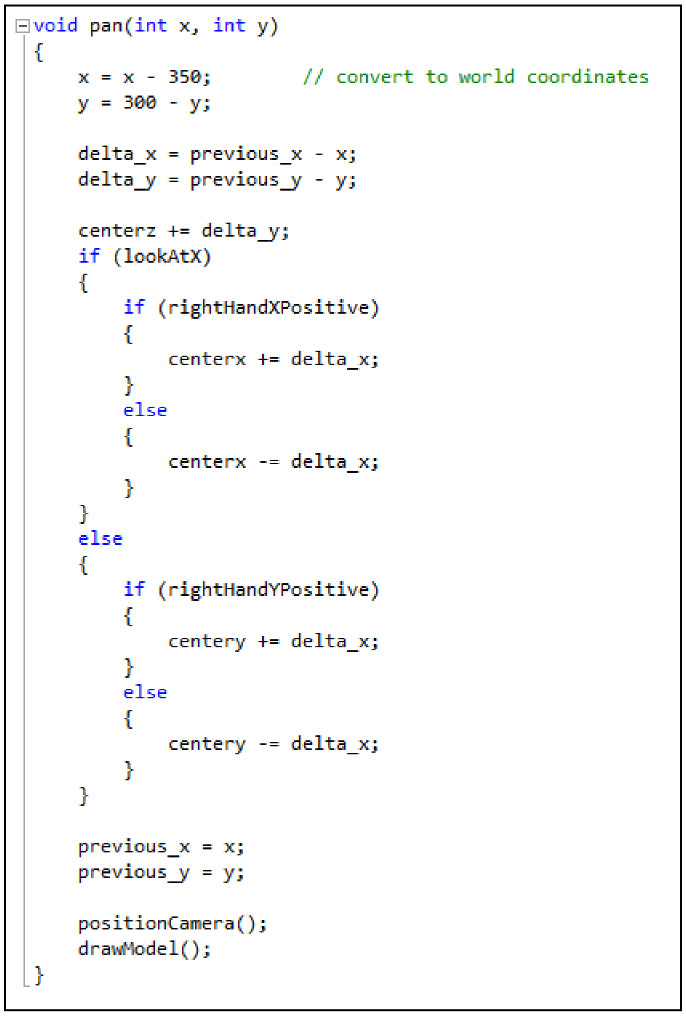
```
if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN && x>-350 && x<350 && y<300 && y>-225) // move camera
{
    previous_x = x;
    previous_y = y;

    glutMotionFunc(pan);
}
```

The glutMotionFunc() is a callback that occurs when the user moves the mouse while holding down the button. Once the pan() function is called, x and y are converted to world coordinates, and the change in position of x and y from their previous positions is calculated and stored in delta_x and delta_y.

The change in y coordinates of the mouse (up and down on the screen) directly correlates to a change in z coordinates (height) of the model. Therefore, the z position the camera looks at - stored in the variable centerz - is incremented by delta_y.

The change in the mouse's x coordinates is not quite as straightforward. For example, if the viewer sits in section C, he is facing the x axis, with the positive side of the axis to his right. In this case, centerx could be incremented by delta_x with no problems. However, if the viewer is sitting in section L, he is still facing the x axis, but the positive side of the axis is to his left. If centerx was incremented here, the model would move in an opposite direction of the mouse. In this case, centerx must be decremented by delta_x. The same scenarios exist if the viewer faces the y axis.

To solve this problem, boolean variables for lookAtX, rightHandXPositive, and rightHandYPositive are updated when each section is clicked. These variables are used in the logic making the camera pan in the expected direction. The code for the pan() function follows.



```
void pan(int x, int y)
{
    x = x - 350;        // convert to world coordinates
    y = 300 - y;

    delta_x = previous_x - x;
    delta_y = previous_y - y;

    centerz += delta_y;
    if (lookAtX)
    {
        if (rightHandXPositive)
        {
            centerx += delta_x;
        }
        else
        {
            centerx -= delta_x;
        }
    }
    else
    {
        if (rightHandYPositive)
        {
            centery += delta_x;
        }
        else
        {
            centery -= delta_x;
        }
    }

    previous_x = x;
    previous_y = y;

    positionCamera();
    drawModel();
}
```

At the end of the function, previous_x and previous_y are updated, the camera is repositioned, and the model is redrawn.

Returning to Seat Map: The second scenario of the mouseOnModel() function is a click on the "Back to Seat Map" button. If the mouse is clicked within the coordinates of the button, the appropriate functions are called to reestablish a 2D view. The background color is set to white once again, and depth testing is disabled. The seat map is drawn, and the program restarted.

```
if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN &&         // click back button
    x>-325 && x<-197 && y<-243 && y>-277)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-350, 350, -300, 300);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClearColor(1, 1, 1, 0);
    glDisable(GL_DEPTH_TEST);

    drawSeatMap();
    startProgram();
}
```

CONCLUSION

The results of this research project show a great deal of potential for an application that could realistically be used by Missouri State University and its fans. Overall, the JQH Arena viewer has already met the following objectives:

- Displays 3D coordinates of JQH Arena from data collected by a laser scanner.
- Has a user interface that allows users to view the model from each seating section on the lower level.
- Allows for panning of the camera while viewing a section by clicking and dragging the mouse over the model.

The next step of the project is to correct and improve the things that could have been done better. This phase should solve some immediate problems and enhance the usability and maintainability of the application. On the to-do list:

- Build a mesh using the data points. This will be possible with more time and preprocessing of the data files. The data points need to be sorted by horizontal angle and vertical angle, with an equal number of points in each column.
- Write a single routine for loading pixel maps. Right now, each pixel map has its own loading function, which is inefficient and difficult for reuse.
- Implement lighting. The arena was only partially lit during the scanning process, which explains the darkness of the points, especially in the upper sections. Some experimentation with OpenGL lighting functions may help this situation.
- Improve camera positioning. This will require more experimentation with glFrustum() and gluLookAt() to get a more accurate picture.
- Research ways to improve performance.
- Improve the UI by adding the JQH Arena logo and text giving directions on how to use the program.

At this point, work on the project should continue with some long term goals in mind. With more time and research, these goals could very well be reached:

- Enhance functionality of the HUD. Allow users to input their seat numbers and move directly to that location.
- Convert the program to a web application. Once completed, a web application would be a practical way for fans to use the program.

ACKNOWLEDGEMENTS

Special thanks to Dr. Kerry Slattery for use of the scanning equipment and advice on data sorting, and to Dr. Yang Wang for instruction in OpenGL and overall helpfulness in making this project a reality.

REFERENCES

1. Astle, Dave & Hawkins, Kevin. (2004). "Beginning OpenGL Game Programming", Premier Press.
2. Ballena Technologies Inc. (2005-2006). Ballena Technologies Inc. Retrieved Oct. 11, 2010, from http://www.seats3d.com/index2.html.
3. Bui, Triet, Ravani, Bahram, Lasky, Ty A., & Yen, Kin S. (2008). "Applications of 3D Laser Scanning for Construction and Maintenance", Advanced Highway Maintenance and Construction Technology UC Davis Caltrans
4. New York Yankees Virtual Venue. (2008). IOMEDIA. Retrieved Oct. 11, 2010, from http://yankees.io-media.com/.
5. OpenGL Overview. (1997-2010). OpenGL. Retrieved Oct. 11, 2010, from http://www.opengl.org/about/overview/.
6. The Sports & Entertainment Studio. (1997-2010). IOMEDIA. Retrieved Oct. 11, 2010, from http://www.io-media.com/#/251/sports/who-we-are.
7. Unver, Ertu, Atkinson, Paul, & Tancock, Dave. (2006). "Applying 3D Scanning and Modeling in Transport Design Education", Computer-Aided Design & Applications, Vol. 3 (1-4) 41-48.