Kyle Alexander - 100082709
Graphics 2 Report

Collision Detection and Response

The collision detection in the application is done in 3 stages and a single iteration is performed once for each building model loaded into the application. First there is an initial test between 2 spheres, the vehicle's sphere and the building's sphere. If the 2 spheres are determined to be colliding the then the algorithm moves on to performing separating axis theorem tests on the 2 object orientated octrees and their children. The final stage occurs if and only if there is a collision detected between a leaf of each octree and consists of using the Möller–Trumbore ray-triangle intersection algorithm to calculate whether any of the vehicles vertices defined in the vehicles leaf node have passed through the any of the buildings triangles also defined in its own octree's leaf node.

The collision in the application is done between the car and each building. Instead of checking if the current position of the car has collided and responding the algorithm calculates the cars position after the desired move and performs the collision detection with this. Upon detection of a collision the desired move is not performed otherwise it is. This has a drawback in that upon movement with a very high speed or very low frame rate the vehicle can move straight through objects but given that each situation is unlikely these negatives were outweighed by the fact that it would not be necessary to calculate how far the objects are inside each other for collision response.

Sphere – Sphere Collision Detection

This test uses spheres that have the their centre at their respective objects x, y, z position and the radius is obtained by calling getLargestExtent() on the model's bounding box. The test then compares the square of the distance between the centre points of each object and if this is less than the square of the sum of the radii then the detection algorithm proceeds to stage 2.

```
glm::vec4 buildCenter = glm::vec4(town.buildings[i]->theBBox.centrePoint.x, town.buildings[i]->theBBox.centrePoint.y, town.buildings[i]->t
float buildRad = town.buildings[i]->theBBox.getLargestExtent();
if ((carCenter.x - buildCenter.x) * (carCenter.x - buildCenter.x) +
    (carCenter.y - buildCenter.y) * (carCenter.y - buildCenter.y) +
    (carCenter.z - buildCenter.z) * (carCenter.z - buildCenter.z) < (radius + buildRad) * (radius + buildRad)) {
    Octree* tree = town.buildings[i]->octree;
    if (checkCollision(&car.model, car.model.octree, town.buildings[i], town.buildings[i]->octree, &carModelMatrix, &newcarmat)) {
```

Code Snippet 1 Sphere - Sphere Intersection test

SAT and Recurring Through Octrees

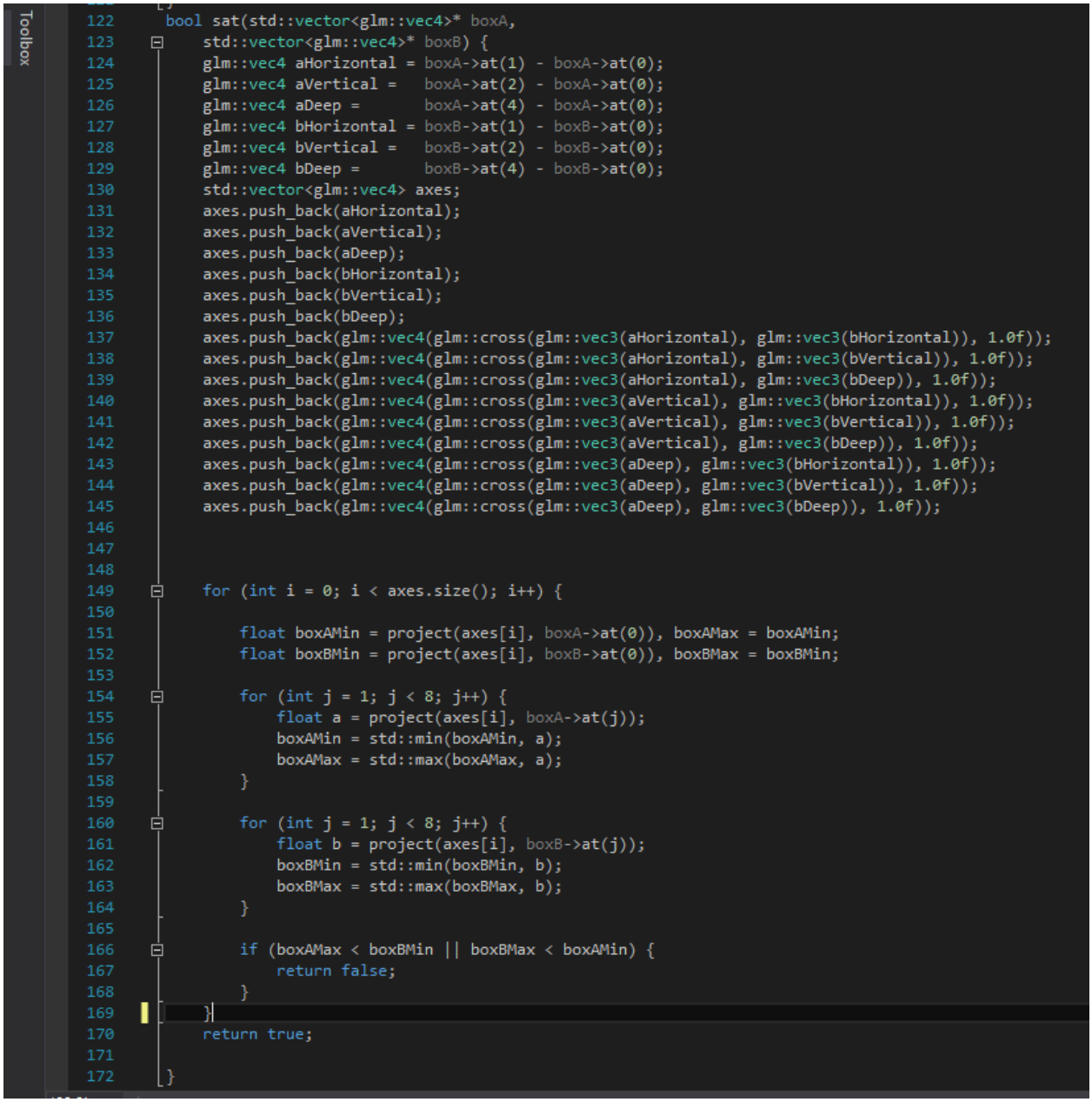
Upon detection of sphere collision as detailed above the algorithm now runs the checkCollision method and passes through each object's ThreeDModel as well as the octree that can be found in the ThreeDModel. The octree is passed separately to allow recursive calling on children of octrees. 2 model matrices for passed through as well, one transforms the vehicles model space octree coordinates to world space using its old position and the second does the same but using its position after its movement which is calculated earlier (Code Snippet 2).

```
float newcarx = car.x + speed*carModelMatrix[2][0] * updatetime;
//car.posY += ModelMatrix[2][1];
float newcarz = car.z + speed*carModelMatrix[2][2] * updatetime;
float newcarr = car.r - car.ra * speed *  0.5 * updatetime;


glm::mat4 newcarmat = glm::translate(glm::mat4(1.0), glm::vec3(newcarx, car.y, newcarz));
newcarmat = glm::rotate(newcarmat, newcarr, glm::vec3(0, 1, 0));
newcarmat = glm::scale(newcarmat, glm::vec3(0.1, 0.1, 0.1));
```

Code Snippet 2 Calculation of new car matrix for collision detection

The first thing done in checkCollision is to build a list of 8 vertices for each octree by using the octrees minx, minY, minZ, maxX, maxY and maxZ variables to build the 8 corners of the bounding box, for the sake of clarity the arrays containing these points will be referred as the octree vertex arrays. The vehicles 8 points are then each pre multiplied by the model matrix for the new cars position and these are stored in a new array to be used for the SAT (Separating Axis Theorem) test. The octree vertex arrays are passed through to a method which uses the separating axis theorem to check collision between the two current octrees which the points are built from.



```
bool sat(std::vector<glm::vec4>* boxA,
    std::vector<glm::vec4>* boxB) {
    glm::vec4 aHorizontal = boxA->at(1) - boxA->at(0);
    glm::vec4 aVertical =   boxA->at(2) - boxA->at(0);
    glm::vec4 aDeep =       boxA->at(4) - boxA->at(0);
    glm::vec4 bHorizontal = boxB->at(1) - boxB->at(0);
    glm::vec4 bVertical =   boxB->at(2) - boxB->at(0);
    glm::vec4 bDeep =       boxB->at(4) - boxB->at(0);
    std::vector<glm::vec4> axes;
    axes.push_back(aHorizontal);
    axes.push_back(aVertical);
    axes.push_back(aDeep);
    axes.push_back(bHorizontal);
    axes.push_back(bVertical);
    axes.push_back(bDeep);
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aHorizontal), glm::vec3(bHorizontal)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aHorizontal), glm::vec3(bVertical)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aHorizontal), glm::vec3(bDeep)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aVertical), glm::vec3(bHorizontal)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aVertical), glm::vec3(bVertical)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aVertical), glm::vec3(bDeep)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aDeep), glm::vec3(bHorizontal)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aDeep), glm::vec3(bVertical)), 1.0f));
    axes.push_back(glm::vec4(glm::cross(glm::vec3(aDeep), glm::vec3(bDeep)), 1.0f));



    for (int i = 0; i < axes.size(); i++) {

        float boxAMin = project(axes[i], boxA->at(0)), boxAMax = boxAMin;
        float boxBMin = project(axes[i], boxB->at(0)), boxBMax = boxBMin;

        for (int j = 1; j < 8; j++) {
            float a = project(axes[i], boxA->at(j));
            boxAMin = std::min(boxAMin, a);
            boxAMax = std::max(boxAMax, a);
        }

        for (int j = 1; j < 8; j++) {
            float b = project(axes[i], boxB->at(j));
            boxBMin = std::min(boxBMin, b);
            boxBMax = std::max(boxBMax, b);
        }

        if (boxAMax < boxBMin || boxBMax < boxAMin) {
            return false;
        }
    }
    return true;

}
```

Code Snippet 3 Separating Axis Theorem Collision Detection

The SAT method (Code Snippet 3) works by calculating 15 axes which the 16 points of the arrays are projected onto. The axes which are calculated comprise of 3 edges from each octree bounding box and can be any 3 edges as long as they are all joined at a common vertex. The remaining 9 axes are calculated by performing the cross product with one axis from the first octree and one from the second and is done for every possible combination giving 3^2 more axes.

Each point from the octree vertex arrays are projected onto each axis and for each axis the min and max result of the projection is stored for each array. Finally a simple test is performed to see of the min and max values of each array overlap. If for any one of the axes there is no overlap then the test can fail fast as it is known there is no collision.

The SAT method is used to calculate whether the 2 octrees collide and if it fails then there is no collision so the checkCollision method returns false. If there is a collision then the result depends on the octrees children. The algorithm uses a nested for loop to iterate through every combination of children from the first octree and children of the second octree (A max of 8 children in each tree means there can be 64 tests) and as long as both children are not null a recursive call to checkCollision is made with the 2 children as the new octree parameters. Three Flags are used to keep track of what checkCollision should return if the SAT test is successful and these are

- carHasChildren - a Boolean indicating whether the current car octree has any children. Is set to false to start with and changed to true if a child is found to be not null in the for loop.
- buildHasChildren – the same as carHasChildren but keeps track of the buildings octree and whether it has children
- collision – a Boolean which starts as false and is set to true if at least one recursive call to checkCollision with a pair of children returns true.

Based on the state of these Booleans there is a list of possible outcomes

- If neither of the octrees have children the next stage of the collision detection takes place as detailed in the next section (Per Triangle Collision). This is because in this situation the current octrees are both leaf node and no further octree refinement can take place.
- If at least one of the octrees has children then the result of the collision variable is returned. This works because if there is a collision between the current octrees the whether there is a collision between the models is dependent on the current octrees children. If there is at least one child present but no collisions have been found then the octrees are colliding in a space where the child that would collide has not been constructed as none of the models vertices are in its space so it is impossible for the models to collide.

Per Triangle Collision

If collision between two octree leaf nodes has been established then the result of the collision must be determined using the triangles of the model. For this part of the collision detection an implementation of Möller–Trumbore ray-triangle intersection algorithm is used. This algorithm is called multiple times to test each combination of the vehicles vertices and the buildings triangles. Each call takes a ray and a triangle and in this implementation returns whether the ray passes through the triangle but the algorithm can be easily modified to return the location of the collision in terms of both the triangle and the ray.

```
198   //Moller and Trombore Ray Triangle intersection modified to glm vectors
199   int rayIntersectsTriangle(glm::vec3 p, glm::vec3 d,
200       glm::vec3 v0, glm::vec3 v1, glm::vec3 v2) {
201
202
203
204       glm::vec3 e1 = v1 - v0;
205       glm::vec3 e2 = v2 - v0;
206
207
208       glm::vec3 h = glm::cross(d, e2);
209
210       float a = glm::dot(e1, h);
211
212       if (a > -0.00001 && a < 0.00001)
213           return(false);
214
215       float f = 1 / a;
216
217       glm::vec3 s = p - v0;
218       float u = f*glm::dot(s, h);
219
220       if (u < 0.0 || u > 1.0)
221           return(false);
222
223       glm::vec3 q = glm::cross(s, e1);
224       float v = f*glm::dot(d, q);
225
226
227       if (v < 0.0 || u + v > 1.0)
228           return(false);
229
230       float t = f * glm::dot(e2, q);
231
232       if (t > 0.00001 && t < 1)
233           return(true);
234
235       else return (false);
236
237   }
238
```

Code Snippet 4 Ray triangle intersection method

The algorithm first builds edges that share a common vertex and builds a scalar triple product from these edges and the direction of the ray. If this value is zero then the ray direction is parallel to the triangle plane so there is no collision, due to floating point maths the comparison is a check very close to zero and not exactly zero.

The intersection algorithm uses the fact that scalar triple products are equal to the determinant of the matrix built from 3 vectors used to calculate the product. After equating the ray

$$\overrightarrow{P} = \vec{O} + t\vec{D}$$

And the triangle (represented in barycentric coordinates)

$$\vec{P} = (1 - u - v)\overrightarrow{V_0} + u\overrightarrow{V_1} + v\overrightarrow{V_2}$$

It is possible to re arrange to the formula to

$$\begin{pmatrix} -D.x & (V_1 - V_0).x & (V_2 - V_0).x \\ -D.y & (V_1 - V_0).y & (V_2 - V_0).y \\ -D.z & (V_1 - V_0).z & (V_2 - V_0).z \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} (O - V_0).x \\ (O - V_0).y \\ (O - V_0).z \end{pmatrix}$$

From this equation Cramer's rule can be used to solve t, u and v. Given that the ray is built from a line segment that is (the vertex's position after movement - the vertex's position after movement) a successful collision only occurs if

$$u \geq 0 \;\&\&\; v \geq 0 \;\&\&\; u + v \leq 1 \;\&\&\; 0 \leq t \leq 1$$

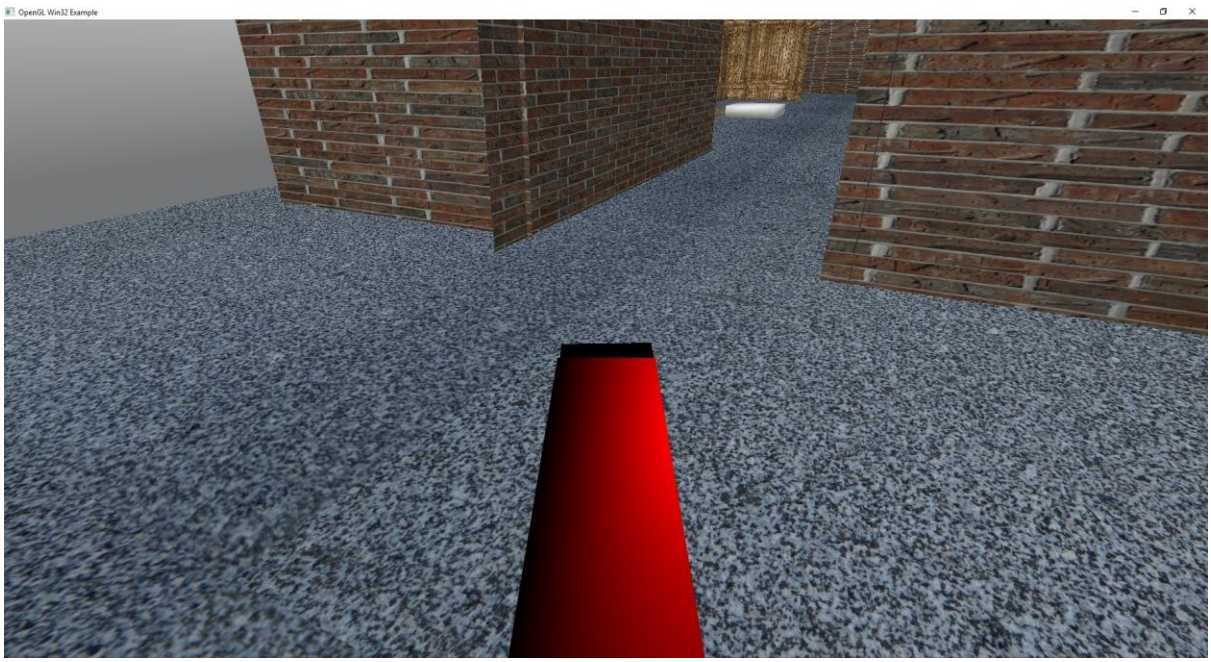
Upon a successful collision the checkCollision method returns true.

Lighting

Lighting in the application is done using the Phong reflection model and there are two states the lighting can be in. The first is termed day mode and consists of a single light source positioned high above the world with larger ambience and diffuse values. This mode uses the basicTransformations shaders that were in the object loading project.

```
//Day
float Light_Ambient[4] = { 0.5f, 0.5f, 0.5f, 1.0f };
float Light_Diffuse[4] = { 0.5f, 0.5f, 0.5f, 1.0f };
float Light_Specular[4] = { 0.5f,0.5f,0.5f,1.0f };
float LightPos[3] = { 0.0f, 200.0f, 0.0f };
```

Code Snippet 5 Day mode lighting values and position



Code Snippet 6 Day mode Result

The second mode is night mode and differs in that there are 2 light sources and they are mobile. The light values are also different with much lower ambience and a higher diffuse lighting values. Also as the 2 light sources are directional and have a cut off angle a forward vector must be defined.

```
float nightLight_Diffuse[4] = { 0.8f, 0.8f, 0.6f, 1.0f };
float nightLight_Ambient[4] = { 0.2f, 0.2f, 0.2f, 1.0f };
float nightLight_Specular[4] = { 0.0f,0.0f,0.0f,0.0f };
```

Code Snippet 7 Night mode light values.

```
if (night == true) {
    glm::vec3 light1 = car.getPos() - carForward * 3.0f - carRight * 2.0f;
    glm::vec3 light2 = car.getPos() - carForward * 3.0f + carRight * 2.0f;
    glm::vec3 lightFor = -carForward;
```

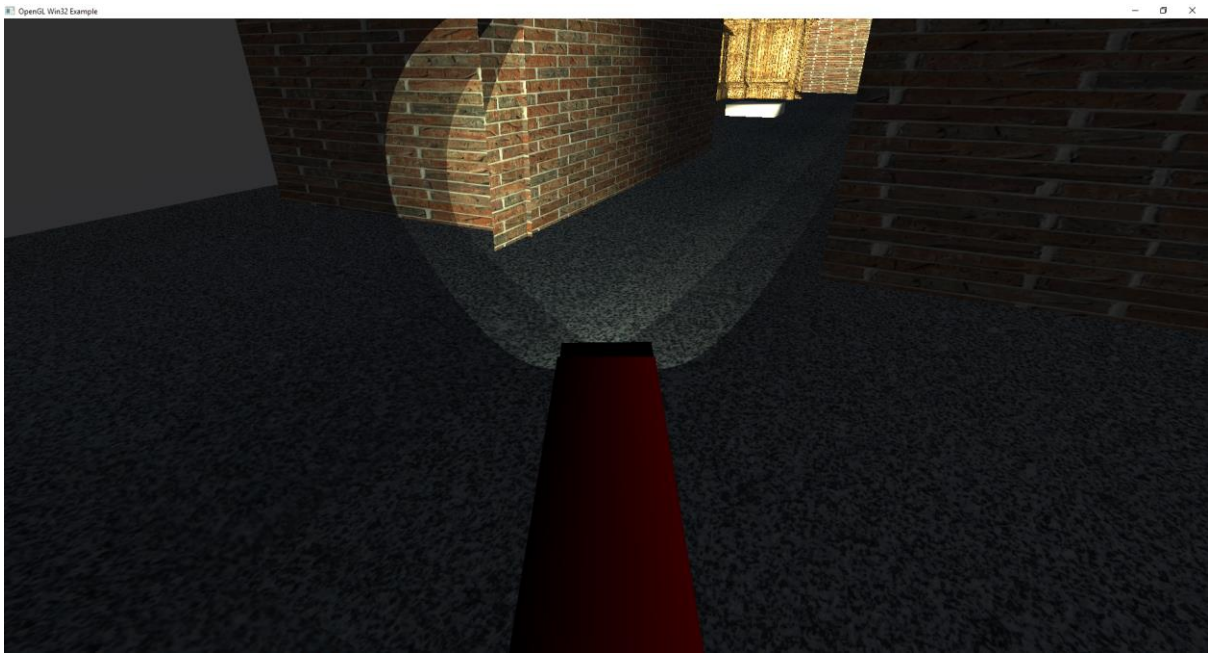
Code Snippet 8 Night mode light position and direction

Due to differing number of light sources a separate shader is used for night mode. Using a second shader also allows the use of calculations to introduce a cut-off angle to what fragments receive the diffuse light. By only applying diffuse lighting if the dot product between the light forward vector and the vector created by subtracting the fragments position from the light position is above 0.9 we get a cut-off angle of about 25 degrees.

```
NdotL = dot(f, -L1);
if(NdotL > 0.9){
        NdotL = max(dot(n, L1),0.0);
        color += (light_diffuse * material_diffuse * NdotL);
}

NdotL = dot(f, -L2);
if(NdotL > 0.9){
        NdotL = max(dot(n, L2),0.0);
        color += (light_diffuse * material_diffuse * NdotL);
}
```

Code Snippet 9 Shader code for only applying lighting within 25 degrees



Code Snippet 10 Night Mode Result

SkyBox and Cube Mapping

A Sky box is implemented using opengl's cube mapping. A cube map texture is generated and bound and then the png files for each image are loaded onto to the cubes faces. The cube is then rendered at the cameras position with no rotation and the depthmask turned off so that everything rendered after the skybox is rendered in front of it. This gives the effect of a skybox always in the background.

The skybox needs its own shader as a cubemap is passed through as a samplerCube instead of a sampler2D and its texture coordinates our queried by a 3d coordinate which in this case is simply the fragment position in world space.

Movement

Movement is done via a speed variable which is incremented and/or decremented based on whether certain keys are pressed. Each frame the cars x and z are modified by the current speed as to simulate an acceleration based movement system. A similar system is used for the cars turning with a car.ra variable keeping track of rotation acceleration. To maintain the forward vector after rotation the x and z coordinates are multiplied by the top 3x3 section of the model matrix that keeps track of rotations. The vector representing where the negative z is currently pointing is pulled out of the matrix and the movement incorporates this by multiplying by carforward.x and carforward.z respectively (Code Snippet 2)

```
glm::mat4 carModelMatrix = glm::translate(glm::mat4(1.0), glm::vec3(car.x, car.y, car.z));
carModelMatrix = glm::rotate(carModelMatrix, car.r, glm::vec3(0, 1, 0));

glm::vec3 carForward = glm::vec3(carModelMatrix * glm::vec4(0,0,1,0));
glm::vec3 carRight = glm::vec3(carModelMatrix * glm::vec4(1, 0, 0, 0));
carModelMatrix = glm::scale(carModelMatrix, glm::vec3(0.1, 0.1, 0.1));
glm::mat4 camModelMatrix = glm::translate(glm::mat4(1.0), glm::vec3(camx, camy, camz));
camModelMatrix = glm::rotate(camModelMatrix, car.r, glm::vec3(0, 1, 0));
```

Code Snippet 11 creation of car model matrix and extraction of post rotation forward and right vectors

Camera Positions

The application contains four possible camera positions.

- A 3rd person camera placed behind the car that focuses just above the car
- A 1st person camera placed inside the car that focuses in front of the car
- A 3rd person top down camera that is placed directly above the car. It is aimed at the car and matches the cars rotation
- A 3rd person camera that is placed independently of the cars position and can be moved with WASD. This camera is always aimed at the cars position.

Building viewing matrices for these views is made very easy by the glm lookat function and all that is required is where the camera is placed and the position of what it will focus on. Given extra time a feature I was planning to add was using glScissor and glViewport to display the top down view in the bottom corner of the screen like a mini map.

```
glm::mat4 viewingMatrix;
if (viewingMode == 1) {
    // 1st
    viewingMatrix = glm::lookAt(glm::vec3(car.x - carForward.x*2, car.y+1, car.z - carForward.z*2),
        glm::vec3(car.x - carForward.x * 5, car.y+1, car.z - 5 * carForward.z), glm::vec3(0, 1, 0));
}
else if (viewingMode == 0) {
    // 3rd
    viewingMatrix = glm::lookAt(car.getPos() + carForward * 10.0f - carRight * (ThirdCamOffset) + glm::vec3(0, 8, 0),
        glm::vec3(car.x, car.y + 3, car.z), glm::vec3(0, 1, 0));
}
else if (viewingMode == 2) {
    //top
    viewingMatrix = glm::lookAt(glm::vec3(car.x, car.y + 100, car.z),
        glm::vec3(car.x, car.y, car.z), glm::vec3(-carForward.x, 0, -carForward.z));
}
else if (viewingMode == 3) {
    viewingMatrix = glm::lookAt(glm::vec3(camx, camy, camz), glm::vec3(car.x, car.y, car.z), glm::vec3(0, 1, 0));
}
```

Code Snippet 12 Building of view matrix based on current camera setting

Frame Independent Updating

Using the built in windows functions QueryPerformanceCounter and QueryPerformanceFrequency the game loop keeps track of the time between each loop iteration and uses this to find the frames per second. Given that the application was built with a machine that has a 60 fps cap and constants like what the speed and rotation acceleration are changed by per frame have their values set to work correctly at 60 fps. The target fps was set to 60 and a ratio is made by dividing the target fps by the current fps and all movement code is multiplied by this ratio to make movement dependent on the time between frames (Code Snippet 2 and Code Snippet 13).

```
LARGE_INTEGER perfValue;
LARGE_INTEGER oldValue;
LARGE_INTEGER perfFreq;

while(!done)                                    // Loop That Runs While done=FALSE
{
    oldValue = perfValue;
    QueryPerformanceCounter(&perfValue);
    QueryPerformanceFrequency(&perfFreq);
    fps = perfFreq.QuadPart / (perfValue.QuadPart - oldValue.QuadPart);
    updatetime = TARGET_FPS / fps;
    //std::cout << "FPS: " << fps << endl;

    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))   // Is There A Message Waiting?
    {
        if (msg.message==WM_QUIT)               // Have We Received A Quit Message?
        {
            done=true;                          // If So done=TRUE
        }
        else                                    // If Not, Deal With Window Messages
        {
            TranslateMessage(&msg);             // Translate The Message
            DispatchMessage(&msg);              // Dispatch The Message
        }
    }
    else                                        // If There Are No Messages
    {
        if(keys[VK_ESCAPE])
            done = true;

        processKeys();

        display();                  // Draw The Scene
        update();                   // update variables
        SwapBuffers(hDC);               // Swap Buffers (Double Buffering)
    }
}


// Shutdown
```

Code Snippet 13 Game loop with timing code