MSIM 441/541 & ECE 406/506
Computer Graphics & Visualization

Programming Assignment Two: Transportation Simulation

Thomas J Laverghetta
Tlave002@odu.edu
757-620-7607
Virtual
MSIM411
11/25/20

Introduction

The transportation simulation project is designed to allow players/users to drive a simulated car through a virtual world that includes roadways with markings and an intersection with a traffic light and surveillance camera on each corner. This document will go over the design, implementation, and results for this simulation so to illustrate use and how to build upon what has been done.

The report will have three sections: program design, implementation and results, and conclusion. The program design will discuss the Doxygen documentation generated for this simulation. Implementation and result is the main section of this document, and will discuss the simulation's design, implementation, and results from implementation. Lastly, conclusion will discuss accomplishments, learning outcomes, difficulties, and future improvements.

Program Design

The design documentation for the programming assignment was made using Doxygen. Doxygen is a tool for generating documentation from annotated C++ sources (classes, enums, variables, functions, etc.), and programming languages (C, Objective-C, Java, etc.). To make documentation using your source code with Doxygen you comment your source code with specified markers and description of the code. The markers give specific meaning to the comments (such as identifying the code is a class, enum, variable, function, etc.). The markers and other documentation on Doxygen can be found on their website (https://www.doxygen.nl/manual/docblocks.html). Using these markers, I commented my code and ran Doxygen wizard tool. This generated HTML design documentation (PA_2_Laverghetta_Thomas.chm). The HTML design documentation illustrates the functions and variables used. The functions and variables will all have short briefings explaining their use within the program and any corresponding dependencies. Doxygen will also illustrate how the functions are connected and where dependencies are.

Implementation and Results

The transportation simulation has 6-implementation tasks: constructing roadway, traffic light and surveillance camera loading and placement, traffic light control system, car dynamics, third person camera, and camera viewports. With these tasks, a car driving a virtual world with four traffic lights and four surveillance cameras and four viewports into the simulation is generated.

Constructing Virtual Roadway

The virtual roadway consists of constructing virtual road with intersection area, adding yellow centerlines that do not extend into the intersection, and adding dashed white lines as lane dividers for traffic going the same direction. All virtual objects were constructed using GL_QUADS (quad).

For purposes of this simulation, the virtual world is built on (z, x)-plane (y-axis is altitude) and north (180-deg) is negative z-direction, south (0-deg) is positive z-direction, west (270-deg) is negative x-direction, and east (90-deg) is positive x-direction. All units of measurement are in meters.

The road was constructed using two quads. These quads construct two virtual roads centered at x- and z-axis with width of 20m from extending from -1000 to 1000 in both x- and z-direction, respectively.

The yellow centerline was constructed using four 2m width quads. First quad goes from 1000 to 10, where 10 is boarder with intersection, along z-axis (south-north direction). Second quad went from -10 to 1000 along z-axis, finishing south-north direction. Same operations were conducted with west-east direction. Only difference is instead of moving along z-axis, the quads move along x-axis. After construction of quads, it produced a yellow centerline to separate traffic on south-north and west-east roadways and where yellow centerlines do not enter intersection.

White dash lines were constructed by iteratively constructing quad segments of 3m with 0.2m width and 9m gaps (dimensioned gathered from U.S. Department of Transportation [1]). The quad segments were generated in the center between yellow centerline and road outer edge (where road meetings grass) for all directions (north-south and east-west). Dash lines do not go into intersection, they stop where yellow centerline stops.

For future endeavors, I would suggest trying to use glLineStipple instead of iterating with quad segments to generate dashed white lines. I tried to use glLineStipple, however, it never turned out correctly for me.

Object File Parser

Before continuing to virtual object placements (traffic lights, cameras, and car), I will discuss the application interface (API) used to parse the object files (OBJ).

OBJ are parsed and loaded using ObjModel. ObjModel was provided by Dr. Shen to parse objects. Where the OBJ parser is ObjModel's ReadFile method. To parse the OBJ and its associated object material template library file (MTL), ObjModel class uses the following data structures:

- Material: Structure for storing material information read from OBJ and MTL. Declared in Material.h
- Vertex: A dedicated structure for storing the position of 3D vertex. Declared in Vertex.h.
- Normal: A dedicated structure for storing the 3D normal vector. Declared in Normal.h.
- Face: Define indices of vertices, normals, and textures for a face. Declared in Face.h
- TextureCoord: Structure for storing 2D texture coordinates. Declared in Texture.h
- BoundingBox: Define the bounds along x, y, z-axes (i.e., AABB). Declared in BoundingBox.h
- Vector3: Structure for storing a 3D vector. Declared in Vector3D.h

More information can be found in auto generated Doxygen documentation (PA_2_Laverghetta_Thomas.chm).

ObjModel uses the data-structures above through C++ STL vector and/or map data-containers. ObjModel uses vectors to hold mesh information (i.e., vertices, normals, textureCoords, faces) and for storing names to materials and identifiers to materials and meshes. ObjModel uses maps to associate names with materials, textures, and meshes. I will now discuss functionality and behavior of vectors and maps so to better understand their use.

Vectors are dynamic arrays that allocate more space when needed. These are used instead of standard dynamic arrays and linked lists because of their ease of use. A common method to add data to vectors is push_back(data). This will add data to the back of vector.

Maps are associative containers that stores data using key value (maps value pairs key->data). Maps are generally used to associate sorted, unique key values with associated data. ObjModel uses maps to map material names (key) with materials (data), to map material names to face indices, and material names to textures.

If maps were not used, I would recommend using unordered maps instead of maps. Since ObjModel does not use order to map keys, an unordered map can be used instead. Ordered maps (maps) are only advantageous when key must be ordered. Ordered maps use self-balancing binary search trees (BST), and has such, search times are O(log(n)), insertions times are O(log(n)) + rebalance, and deletion times O(log(n)) + rebalance [2]. On the other hand, unorder maps do not sort keys allowing it to use hash tables resulting in search times of O(1) (average case) or O(n) (worst case), insertion times of O(1) (average case) or O(n) (worst case), and deletion times of O(1) (average case) or O(n) (worst case) [2].

Object Parser's Mesh Drawler

ObjModel's DrawMaterials method uses OpenGL primitives GL_POINTS and GL_POLYGON to draw mesh geometry. Where the three display modes supported are point, filled, and wireframe. This can be set using ObjModel SetDisplayMode method.

Object Parser's Center and Bounding Box Calculator

The ObjModel calculates the center and bounding box using GetCenter and GetBoundingBox methods. GetCenter locates the center of the bounding box. GetBoundingBox returns the bounding box limits that constrains the object placement in world-space. They are not used within current project, however, might be useful in the future for ensuring objects are constrained within boundaries.

Traffic Lights and Surveillance Cameras – Loading and Placement

Each corner of the intersection (north-south, south-west, south-east, north-west) has a traffic light pole and a surveillance camera. The traffic light and surveillance camera are 3D objects found in the program's model folder. The traffic lights and surveillance cameras are placed into the simulation by first loading their associated OBJ and MLT using two ObjModel instances. The traffic light uses a derived ObjModel instance called TrafficLight (this will be discussed further in the next section).

Once objects have been loaded into the system, transforms can be conducted to move the objects to desired locations every frame. Traffic light's origin is located at the base of the traffic light, therefore, traffic light transforms are scaling it from inches to meters (traffic light is defined in inches instead of meters), rotating around y-axis (pointing up) so to point lights towards oncoming traffic for that corner, then translating traffic lights to desired location (one of the four corners). Surveillance camera transforms are rotating cameras around y-axis so to face inside the intersection then translate cameras to be frontally adjacent to traffic light. In OpenGL, the transforms are called in reverse order (translate then scale) because OpenGL uses matrix post-multiplication.

Traffic Light Control System (Light Controls)

Each traffic light contains three signals: green, yellow, and red, and each signal has two modes: on and off. The ObjModel class developed by Dr. Shen supports only static models (the attributes (e.g., materials and normal) of which do not change dynamically during runtime). However, the traffic lights change their signal colors dynamically. To achieve dynamic behavior, the normal way is to find the geometry of the signals and change their material properties accordingly (i.e., change the material from Material A to Material B). This project will utilize a different and simpler approach based on some prior knowledge of the traffic light model in which the signals are the only geometry that use their materials. For example, the material _Yellow_ is used only by the yellow signal. Thus, to represent the two states (on and off) of the yellow signal, we can just change the properties of the material _Yellow_ instead of making two materials _Yellow_On_ and _Yellow_Off_inside in the model file.

Before setting up states, a data structure must be created to hold material information and traffic light state information. This data structure will be TrafficLight class. TrafficLight class is child of ObjModel, therefore, giving it all the functionality of ObjModel, particularly, access to materials and OBJ parser. More information can be found in design documentation.

Configuring traffic light states. There are 6-states (materials) in TrafficLight class: red-on, red-off, yellow-on, yellow-off, green-on, green-off. The traffic light state values are assigned within setMaterial method (method within TrafficLight class) using traffic light material map, name of light materials, and changing diffuse reflection array (Kd) values. Starting with on states (red-on, yellow-on, and green-on). The on states are assigned using the default values in material map associated with name of traffic “light” material (“_Red_”, “_Yellow_”, & “_Green_”). This is done because the default material values are all lights on. Next, the off states (red-off, yellow-off, green-off). The off states are assigned by copying the on states then changing corresponding Kd to dim light value. I dimmed traffic lights to 25%. This resulted in lights being visible while in off state not on. Figure 1 illustrates the method.

```
void TrafficLight::setMaterials()
{
    // In this function, you are supposed to assign values to the variables redOn, redOff,
    // yellowOn, yellowOff, greenOn, greenOff.

    // assigning red light on and off
    redOn = materials["_Red_"];
    redOff = redOn;
    redOff.Kd[0]= 0.25;             // for off state, set red to dim (25% of on)

    // assigning yellow light on and off
    yellowOn = materials["_Yellow_"];
    yellowOff = yellowOn;
    yellowOff.Kd[0] = 0.25;         // off state, set yellow light to 25% of on (dim)
    yellowOff.Kd[1] = 0.25;

    // assigning green light on and off
    greenOn = materials["_Green_"];
    greenOff = greenOn;
    greenOff.Kd[1] = 0.25;          // off state, set green light to 25% of on (dim)

    // initializes lights to off state
    materials["_Red_"] = redOff;
    materials["_Yellow_"] = yellowOff;
    materials["_Green_"] = greenOff;
}
```

Figure 1. Initializing Traffic Light States

To change the value dynamically during simulation, I created a setSignal method in TrafficLight class. setSignal method will receive has input a signal and change state based on signal. A signal is a enum comprising of {Green, Red, Yellow}. Once received, setSignal will determine the signal (green, red, or yellow signals) and set object material to that state. Figure 2 illustrates method.

```
void TrafficLight::setSignal(Signal signal)
{
    // You are supposed to assign the materials used in the ObjModel class based on
    // values of the input signal.

    switch (signal) {
    case Signal::Green:
        materials["_Red_"] = redOff;
        materials["_Yellow_"] = yellowOff;
        materials["_Green_"] = greenOn;
        break;
    case Signal::Yellow:
        materials["_Red_"] = redOff;
        materials["_Yellow_"] = yellowOn;
        materials["_Green_"] = greenOff;
        break;
    default: // red
        materials["_Red_"] = redOn;
        materials["_Yellow_"] = yellowOff;
        materials["_Green_"] = greenOff;
        break;
    }
}
```

Figure 2. Setting Materials to Current Traffic Light Signal

setSignal by drawScene method in main.cpp. This will draw current scene. This will not change the signal state.

To change the state of signals, a timed update method in main.cpp is used. The update method updates every 20ms through a link with timer, where the timer is registered with glutTimerFunc to be called every 20ms. Every time update is called, it will increase a counter. The counter will be used to determine if enough time as elapsed to change current to state to next state based on state-machine (Figure 3). Then any updates will be displayed via draw callback method. Draw callback will call TrafficLight class's setSignal method based on state change determined in update method.

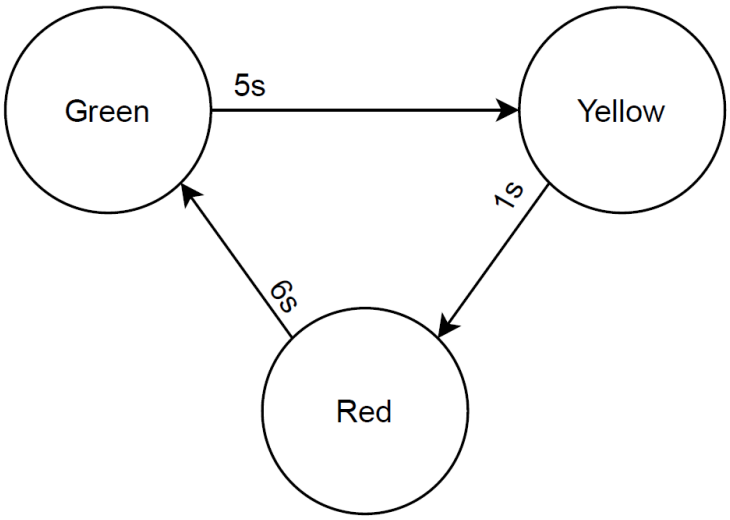

Figure 3. Traffic Light State-Machine

Car Dynamics

After preparing the terrain and traffic lights, the car was included a into the simulation. Before implementing the car model into the simulation, I loaded it into Maya to determine the object's local frame. Figure 4 shows the object's coordinate frame is in the front of the car. With this knowledge, I know to move the car forward relative to its frame, I would have to move in the z-direction.

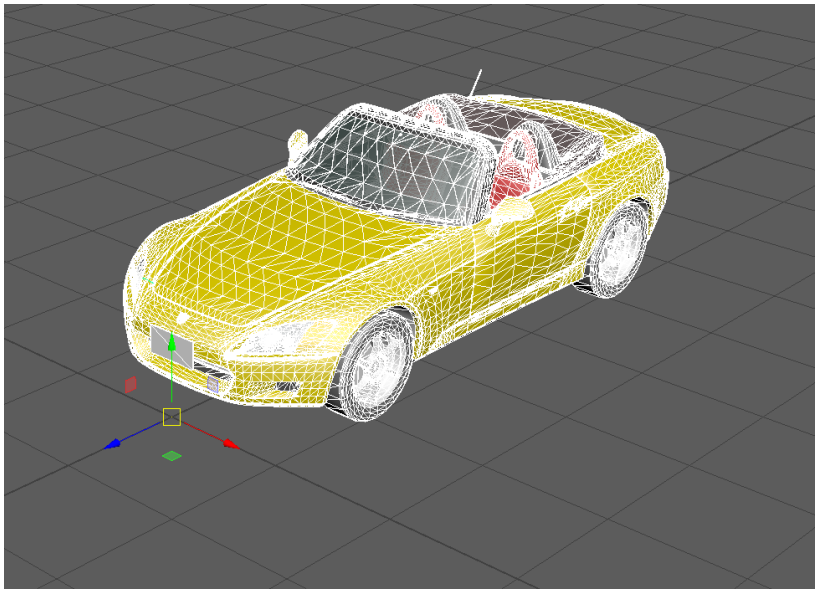

Figure 4. Honda S2000 Coordinate Frame

Based on knowledge of model frame, vehicle dynamics and controls are implemented. The car direction variable representing the car's forward direction in the world frame was implemented. At each simulation (virtual) timestep, a displacement (corresponding to the car's local speed) is added along the

forward direction of the current car's world position given reference to world car speed and delta time (update time). To help make these calculations, I used Dr. Shen's computeRotatedVector to compute world car speed given car's local speed and heading (direction).

The car controls to change car's direction and speed were implemented using keyboard commands. The following are the keyboard commands and their corresponding action:

- Up: accelerate
- Down: decelerate; will shift to reverse gear once car's speed is zero.
- Left: turn left
- Right: turn right
- R: reset the car to its initial position, orientation, and speed (0)
- B: break (and the car stops immediately)
- Escape: exit the program.

The commands and actions can be found in specialKey method in main.cpp.

Third Person Camera

To allow the player to drive the car, a third person camera view was implemented in display method in main.cpp. The third person camera is placed along the z-direction in the car's local frame pointing towards front of the vehicle. This was done using gluLookAt; where eye-point is car position vector plus world camera offset vector, reference point is car position vector with y-element has a positive offset to increase player field of view, and direction of the up vector is y-direction. Note, world camera offset vector is calculated using Dr. Shen's computeRotatedVector given local camera off set (0, -2, -6) and car's heading. This calculation is done every time the car changes heading (implemented in specialKey method). Figure 5 shows player's view.

As seen in Figure 5, In addition to third person camera, text indicating heading (north (N), south (S), east (E), west (W), NW, NE, etc.) and speed in miles per hour (mph) was added.

Figure 5. Third Person View

Camara Viewports

Three viewports were implemented into the simulation. The first two cameras are used to simulate two south-west and south-east surveillance cameras and the third camera provides a bird's-eye view of the intersection. To generate viewports, I used glViewPort, gluPerspective, and gluLookAt along with winHeight (weight of window), winWidth (width of window), sHeight, and sWidth. sHeight and sWidth are dimensions of small viewport and are calculated using window height and width, respectively. Figure 6 shows the code used to generate the viewports. Figure 7 shows the code running in simulation.

```
// Setup viewport, projection, and camera for the South-East camera and draw the scene again.
glViewport(winWidth - 3 * sWidth - 45, winHeight - sHeight - 15, sWidth, sHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(30, (float)winWidth / (winHeight - sHeight - 15), 1, 1000);
gluLookAt(10, 3, 10, -3, 3, -10, 0, 1, 0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// diplsay the camera name
glColor3f(1, 1, 0);
glWindowPos2f(winWidth - 3 * sWidth - 45, winHeight - sHeight - 15);
printLargeString("South-East Camera");
drawScene();

// Setup the viewport, projection, camera for the top view and draw the scene again.
glViewport(winWidth - 2 * sWidth - 30, winHeight - sHeight - 15, sWidth, sHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(30, (float)winWidth / (winHeight - sHeight - 15), 1, 1000);
gluLookAt(0, 50, 0, 0, 0, 0, 0, 0, -1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
drawScene();

// Setup viewport, projection, camera for the South-West camera and draw the scene again.
glViewport(winWidth - sWidth - 15, winHeight - sHeight - 15, sWidth, sHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(30, (float)winWidth / (winHeight - sHeight - 15), 1, 1000);
gluLookAt(-10, 3, 10, 10, 3, -3, 0, 1, 0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// display the camera name
glColor3f(1, 1, 0);
glWindowPos2f(winWidth - sWidth - 15, winHeight - sHeight - 15);
printLargeString("South-West Camera");
drawScene();
```

Figure 6. Viewport Generation Code

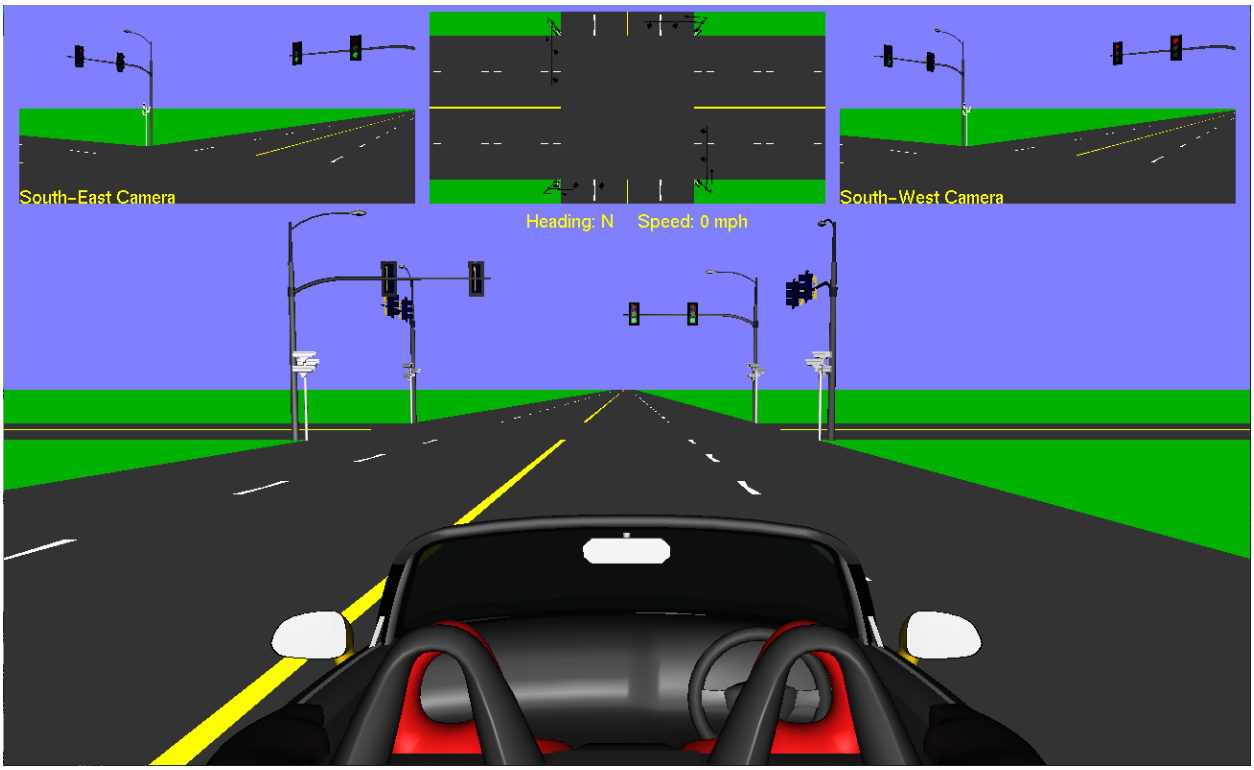

Figure 7. Viewports in Simulation

Running Simulation

Please see attached video (Traffic Simulation 2020-11-26 05-31-20.mp4) to watch simulation running. The video will watch me drive the car forward past the intersection, stop immediately (pressed b), reverse, and stop again.

Conclusion and Discussion

Accomplishments

I was able to create a basic driving simulator using OpenGL's primitive functionality and Dr. Shen's ObjModel API.

Learning Outcomes

I learned how to use different coordinate systems (world coordinates vs car's coordinates), using OpenGL's cameras and viewport functions, and transforms.

Difficulties

I had no major difficulties with the simulation; the simulation was straight-forward to construct. Dr. Shen's lectures and slides helped with the ease of implementation as it did not take me long to find what I was looking for when stuck.

Future Improvements

In the future, I would suggest adding more objects to the environment such as trees and/or buildings. Currently, the simulation seems bland. Also, I would consider going through and optimizing the program. Currently, the program is very slow. For example, the timed update function which is set to 20ms takes more than a second on average to run.