Computer Graphics Ex6

Due 16.6 (23:55)

Overview:

In this exercise you will implement an OpenGL 3D racing game that will consist of a F1 car moving on a racing track. The implementation of this exercise will use the racing car you modeled in Ex5.

Requirements:

You are provided with an executable jar. The jar implements the final game (except for the bonus section), and your racing game should be similar. In order to run the jar, you need to copy the jar file (along with the Textures folder) into one of the native folders). For example, if you use windows, then copy Ex6.jar and Textures into 'natives/windows-amd64' and then click on Ex6.jar

1. **General:**
   a. The following keys should be enabled:
      i. 'UP' - acceleration.
      ii. 'DOWN' - breaks.
      iii. 'LEFT' - rotates the steering to the left direction.
      iv. 'RIGHT' - rotates the steering to the right direction.
      v. 'l' (L) - switches the light mode from Day mode to night mode.
   b. The F1 car should always be on the track (It shouldn't deviate from the track) - This means, if the user tries to deviate from the track, she hits an invisible wall.
   c. The F1 car can only move forward, right and left (it cannot go backwards).
   d. The track is filled with wooden boxes. As the car propagates in track, the difficulty of the game increases, this means, the track should contain more boxes. For simplicity, all boxes shapes are equal.
   e. **(Bonus)** The car should avoid hitting the boxes. If it does hit one of the boxes, then the game is over. You should prompt a message that indicates the user score (for example; show total distance (in km) and elapsed time). You should allow the user to restart the game.
2. **View:**
   a. You should place the camera in the middle of the scene.
   b. The car position should be fixed relative to the camera. This means - as the car moves, so does the camera moves along with it. Note, this doesn't mean that the camera and car share the same location in space.
   c. Light sources should be fixed relative to the camera and car. This means - as the car moves, so does the light sources.
3. **Track:**
   a. The track should be split into segments.
   b. Each **Track Segment** (see TrackSegment.java), models a portion of the track (the asphalt, grass and boxes).

    c. The track segment has a difficulty level, which dictates the way boxes are placed in the track segment. The difficulty value is between 0.05 and 0.95. The higher the value, the more boxes appear on the track.
    d. Boxes should be axis-aligned, and arranged in rows. As the difficulty increases, more boxes will be arranged on the same row, and the distance between rows decreases.
    e. Since the track have many segments - we don't want to hold an object for each segment. We only consider two segments at a time (more info later).
    f. For each track segment, we model the asphalt using more than one polygon. This is crucial for better shading results.
4. **Lighting and Textures:**
    a. You need to support shading - the shading must be applied on the F1 car and the track.
    b. Set the material properties as you like.
    c. You need to support two modes - night/day mode.
    d. In day mode - use one directional light source.
    e. In night mode - place at least two spotlight light sources above the middle of the track. Use low ambient light to model the moon-light.
    f. You need to add textures - Use the provided images (or any other textures you see fit). You need to texture map the asphalt, grass, and boxes.

Implementation:

In this section, and the following subsections, we will discuss important implementation details. You are provided with a partial code, and you need to finish it in order to get similar results (see TODOs in the partial code). You're more than welcome to implement the game by-your own, but the resulting game should be at least as impressive as our implementation.

We now begin by giving an overview on different parts of the code.

NeedForSpeed.java

This class implements the game canvas. Use the display method to start drawing the game. The game is rendered 30 times per second, so each 1/30 seconds the display method is invoked and the scene is redrawn. As the user interacts with the game, the game state changes (car position and orientation) and hence the scene is rendered differently. For example, when the car accelerates, then the car changes its position and rendering the scene again will move the car forward.

GameState.java

The game state stores most of the information needed for rendering the game. This class is already implemented. You can use it to get information on the car and camera state. There are two important methods you should be aware of:

1. getNextTranslation - This method can be used to get the next translation that should be applied on the car and camera. When this method is invoked, it returns the car movement since previous call to getNextTranslation. For example; assume you called

getNextTranslation at t_1 and then called it again at t_2. The return values at second invocation (t_2), is the distance that the car moved, between the interval $\Delta t = t_2 - t_1$, in each direction.

2. getCarRotation - this function returns the car angle of rotation about the y-axis. So, if the user is pressing the 'RIGHT' key, then the return value is a positive integer which is the angle of rotation.

IRenderable.java

This interface represents an object that can be rendered. The interface underwent minor modification, where we added a new method (the destroy() method). The destroy method should be invoked when you want to free resources used by the object. In practice you will need to free texture resources used by the object in this function.

SkewedBox.java

It is recommended to implement the wooden box (the Obstacle in the track) as a skewed box. Think how to modify the SkewedBox, so you can support textures. Note, that you will also need to define Normals for each vertex in-order to support lighting.

F1Car.java

You need to set materials properties for vertices in order to support shading. The material properties should set for each component of the car (back, center and front).

TrackSegment.java

This class should be used to render a portion of the racing track. The track segment consists of an asphalt part, grass part and the boxes. Few notes on this class:

1. You should implement a method called setDifficulty. This method, changes the difficulty of the track. In this method you should reposition the boxes in the track segment. Note that you only need to store the centroid of all boxes locations. You don't need to store an actual object for each box.

Track.java

This class will represent the whole racing track. For efficiency purposes, we will set the view volume (when you set the projection matrix), so that at every moment, at most two Track Segments are visible. This means, in order to represent the whole track, you only need two track segments.

To see this, let us assume that each track segment has length 200 meters. Further assume we set the view volume so that its depth is also 200 meters. **Recall**, OpenGL only renders objects that are within the view volume, and the depth of a view volume is the difference between the **near** and **far** values when you use gluPerspective/glFurstum.

If the track is composed of the segments $s_1, s_2, s_3, \dots, s_n$ - where segment s_i is connected with segment s_{i+1} - and initially s_1 is within the view volume. Then, as the car moves forward, so does the camera (since the car and camera moves in the same direction), and the view volume is translated as well. Now, the view volume will partially cover s_1 and s_2, and as time passes, the car will pass track segment s_1 (it will be behind it), and the segment will no longer be part of the view volume (remember, the car doesn't move backwards). Thus we can forget about s_1, and we will need to render s_2 and s_3 only. This procedure continues as time passes.

A pseudo code on how to render the track is given below:

```
1. Let $TS_L$ be the track segment length.
2. Let $current \leftarrow s_1, next \leftarrow s_2$
3. If carPosition.z > $TS_L$ then: // The car has passed through current segment
    a. Change segments by replacing: $current \leftarrow s_2$ $next \leftarrow s_3$
    b. carPosition.z %= $TS_L$ // this step is crucial - see note below*
4. Render both $current$ and $next$
```

Note*: The carPostion should always be relative the current track segment. When the car position in the z value is more than the track segment length (TS_L), this means that the car has passed the current track segment, and it is behind it. So, the current segment is the next segment. The carPosistion now should be relative to the new current track segment, and you need to set the z value back to the beginning (we do so by the % operation).

Final remarks:

1. The above pseudo code is not complete - there are some details left out. Make sure you to test yourself.
2. You don't need different objects for $s_1, s_2, s_3, \dots, s_n$. The only difference between s_i and s_{i+1} in our implementation is the difficulty of the track segments. Thus just store two track segments and swap them when you want to change track (remember to set the new difficulty for the relevant segment).

Textures

Textures should be used for the track asphalt, grass, and boxes. Image files are supplied for your convenience (inside 'Implemented/Textures' folder), but you can use any other images you choose. Files should be placed in such a location that

```
File texFile = new File("texture.jpg" /* or .png */);
```

would find them. In Eclipse this means relative to the project folder. If you have many textures, you can use a subfolder:

```
File texFile906 = new File("tex/texture906.png");
```

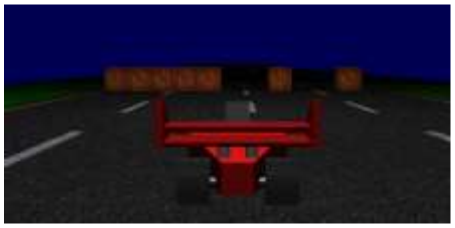
Remember, if you want lighting to be blended (modulated) with textures, you need to set the following property before you set the texture maping:

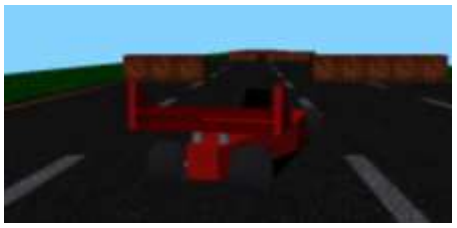
```
gl.glTexEnvi(GL2.GL_TEXTURE_ENV, GL2.GL_TEXTURE_ENV_MODE, GL2.GL_MODULATE);
```

Note: when you're done with the implementation, and you want to export an executable jar, eclipse will not place the texture files automatically in the jar. You need to add the textures in the same place as the jar file. There are other ways to overcome this issue, but copying the files to the folder will work.

Shading

You need to set the materials properties for every primitive you render. There are two lighting modes we want to support. Day mode- where the scene is rendered at day-time, which means there is only a sun light source. Night mode - where the scene is render at night, which means there are spotlights that illuminates the scene. You should add at least two spotlight sources (remember - increasing the number of light-sources comes at a computation cost). Further note that, you should set two different background colors (depending on the light mode).


Night


Day

World Coordinate System

As we talked in class, when we render the whole scene, some objects need to be scaled because they were modeled in a local coordinate system. Here we stress out the unit of measure in our world coordinate system.

- The unit of measure we use is **meter**.
- The track is spanned along the -z direction.
- The camera should be 2 meters away from the scene.
- The car should be 4 meters away from the camera.
- Each Track Segment should 500 meters length:
    - Asphalt total width is 20 meters
    - Asphalt texture width is 20 meters.
    - Asphalt texture height is 10 meters.
    - Grass width (on each side) is
    - Grass texture height and width are 10 meters.
    - Wooden box length is 3.0

Note: we don't mind if you choose to setup the scene differently (as long as you satisfy the requirements). But, we gave you the measures so it will be easier for you to implement

View and Projection

You need to setup the camera and the car so that they are fixed to each other. Further, you should define a perspective projection (use gluPerspective) such that your view volume depth is at most the track segment length (depth).

Bonus (10 pts)

Currently, the 3D game doesn't support collusion testing. This means that the car passes through the wooden boxes. You need to implement collusion testing - in such way that if the user passes through a box, she will lose the game.

Submission

- Submission is in pairs
- Zip file should include
    - All the java source files, including the files you didn't change.
        - Include the eclipse project if you can.
- 'Implemented' folder - which contains your implementation.
    - This folder should contain a runnable JAR file named "ex6.jar"
    - You should also put the textures you used inside this folder
    - This JAR should run after we place JOGL DLLs in this directory
    - Make sure the JAR doesn't depend on absolute paths – test it on another machine before submitting
    - **Points will be taken off for any JAR that fails to run!**
- A short readme document where you can **briefly** discuss your implementation choices - Specially for the bonus part.

- Zip file should be submitted to Moodle.
- Name it:
    - <Ex##> <FirstName1> <FamilyName1> <ID1> <FirstName2> <FamilyName2> <ID2>