Computer Graphics Ex6

Due 5.7 (23:55)

Overview:

In this exercise you will extend an awesome OpenGL 3D racing game. The implementation of this exercise will use the racing car you modeled in Ex5. We provide partially implemented code, follow the TODOs and fill in the gaps.

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.

Note: for mac users we will provide a demo video to help you compare your implementation with ours as well.

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.
        vi. 'v' (V) - switches the view point
    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 that the track should contain more boxes. For simplicity, all boxes shapes are equal.
    e. The car should avoid hitting the boxes. If it does hit one of the boxes, then the game is over and the user is notified.
2. **View:**
    a. There are two viewing options (birds-eye and third-person view).
        i. Birds-eye view: The camera is looking at the track from above.
        ii. Third-Person: the camera is looking towards the track and is placed behind the car.
    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. Spotlights should be fixed relative to the camera and car. This means - as the car moves, so does the light sources. The spotlights are only available during night-mode and they should be positioned exactly at the car front lights.

3. **Track (implemented - see appendix for further details):**
   a. The track is 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 0.2 and 0.75. The higher the value, the more boxes appear on the track.
   d. Boxes are axis-aligned, and arranged in rows. As the difficulty increases, more boxes will be arranged on the same row, and the distance between two consecutive boxes-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 in the appendix).
   f. For each track segment, we model the asphalt using more than one polygon. This is crucial for better shading results.
4. **Lighting:**
   a. You need to support shading for the F1 car and the track.
   b. Set the material properties as you like - we provide initial material properties.
   c. You need to support two modes - night/day mode.
   d. In day mode - use one directional light source.
   e. In night mode - place two spotlight sources at the location of the car front lights. The spotlights should always be directed parallel to the car forward direction. This means that the spotlight should be part of the car.
5. **Textures (up to 5 points bonus)**:
   a. You need to add objects and textures to decorate the track (buildings on the sides, billboards, etc.).
   b. Remember to provide the textures in your submission.

Implementation:

In this section, we will discuss and highlight 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 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.

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).

Shading

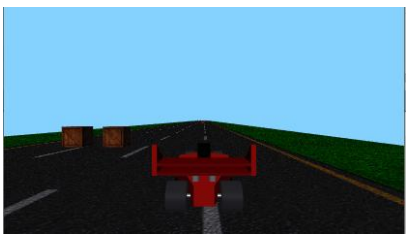
You need to set the materials properties for every primitive you render (we provide initial properties, but you can change them if you want). There are two lighting modes we want to support.

Day mode in which the scene is rendered at day-time. This means there is only one (directional) light source which is the sun.

Night mode in which the scene is render at night. This means there are no light sources except for the car front lights. Here, the spotlights should always be directed parallel to the car's forward direction. So if the car is rotated by α degrees, so will the spotlight direction.

Note: you should set two different background colors (depending on the light mode).

Night



Day

Collision detection

Checking if the car crashes into one of the boxes requires computationally exhausting operations and non-trivial implementation. There are, however, easier methods to (approximately) check if the car collides with one of the boxes. To do this, we will use the bounding spheres you implemented in HW5. The general Idea is this:

1. Get the bounding spheres for all boxes on the track (invoke gameTrack.getBoundingSpheres for this).
2. Get the bounding spheres for the car (remember the hierarchy of the spheres).
3. Check collision between the spheres of the boxes and the spheres of the car.
    a. Take into consideration the hierarchy of the car spheres. This means if the bigger sphere doesn't intersect the sphere of a box, then the car surely doesn't intersect this box.

To support collision you need to implement the method "NeeForSpeed.checkCollision()". Finally, you need to remember because we bound our car with spheres, we get less accurate results and for those of you who have implemented the hierarchical

World Coordinate System (NOT the camera coordinate system)

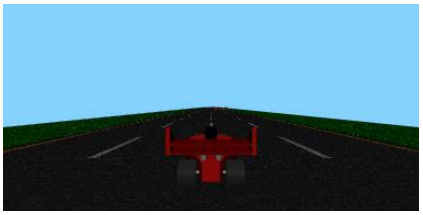
As we talked in class, when we render the whole scene, some objects need to be scaled because they were modeled in a local coordinate system. Here we stress out the unit of measure in our world coordinate system.

- The unit of measure we use is **meter**.
- The track is spanned along the -z direction.
    - The start of the track is at $z = 0$.
- The projection plane should be 2 meters away from the camera and the view angle is at least 60 degrees.
- The car and camera should be:
    - In Third-Person view: the camera is 4 meters behind the car back, it is 2 meters above the ground and is looking in the -z direction.
    - In Birdseye view: the camera is 50 meter above the ground, it is looking down in the -y direction and it's up vector is the -z direction. The camera projection on the track is 22 meters away from the car's front bumper.
- 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

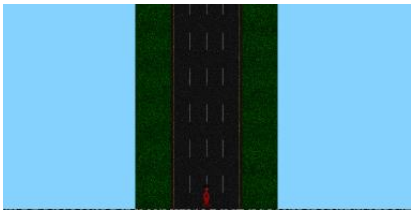
Note: we don't mind if you choose to setup the scene differently (as long as you satisfy the requirements). But, we gave you the measures so it will be easier for you to implement the game.

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). There are two view options, Birds-eye and Third-person view:



Third-person



Birds-eye

Textures Bonus (up to 5 points)

Textures should be used to decorate the racing track. The path to the texture file is relative to the project folder, so when you want to access a texture file you only need to give the relative path. For example, for images that are in the project root directory:

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

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 mapping:

```
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.

The amount of points you will receive will be based on creativity and complexity.

Submission

- Submissions are 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 may be taken off due to any JAR that fails to run!**
    - For Mac users: since mac users are having issues exporting Jar files, include screenshots proving your implementation works, and that you fulfill all requirements. This doesn't mean we will not compile and run your code.
- A short readme document where you can **briefly** discuss your implementation choices - Especially for the bonus part.
- Zip file should be submitted to Moodle.
- Name it:
    - <Ex##> <FirstName1> <FamilyName1> <ID1> <FirstName2> <FamilyName2> <ID2>

Appendix A - Rendering the scene track

Here we provide an overview of how the Track was rendered.

TrackSegment.java

This class is used to render a portion of the racing track. The track segment consists of an asphalt, grass and the wooden boxes. All boxes locations are relative to the track segment.

Track.java

This class represents the whole racing track. For efficiency purposes, we 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 is of length 500 meters. Further assume we set the view volume so that its depth is also 500 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 s_1 is the initial track segment. So, initially, the only track segment we see is s_1. 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 (because we changed the camera position). Now, the view volume will partially cover s_1 and s_2, and as time passes, the car will finish track segment s_1 (it will be behind the car), 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 the track is rendered is given below:

1. Let S_L be the track segment length.
2. Let $current \leftarrow s_1, next \leftarrow s_2$
3. If **carPosition.z > S_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 %= S_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 (S_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. We 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 we store two track segments and swap them when we want to change track.