Computer Graphics Ex6

Overview

In this exercise you will extend an awesome OpenGL 3D racing game. The implementation of this exercise will use the locomotive 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. For windows, to run the jar, place the jar in some directory together with the provided 'Textures' folder, Then simply click on the jar to run it.

For MAC users – place the jar in the some directory together with the provided 'Textures' folder, then from that directory run **java -XstartOnFirstThread -jar Game-macos.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.
      vi. 'v' (V) - switches the view point
   b. The locomotive 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 locomotive can only move forward, right and left (it cannot go backwards).
   d. The track is filled with wooden boxes. As the locomotive 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.
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 direction the locomotive is moving in and is placed behind the locomotive.
   b. The locomotive position should be fixed relative to the camera. This means - as the locomotive moves, so does the camera moves along with it. Note, this doesn't mean that the camera and locomotive share the same location in space.
   c. Spotlights should be fixed relative to the camera and locomotive. This means - as the locomotive moves, so does the light sources. The spotlights are only available during night mode and they should be positioned exactly at the locomotive front lights' position.
3. **Track (Already implemented – see appendix for more 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 must support shading for the locomotive and for 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 a single directional light source.
   e. In night mode –
      i. place two spotlight sources at the location of the locomotive front lights. The spotlights should always be directed parallel to the locomotive forward direction. This means that the spotlight should be part of the car.
      ii. In addition, you should set a global ambient light in the scene that will simulate moonlight.

5. **Textures (bonus – up to 5 points)**:
   a. The supplied JAR uses textures to render the grass, asphalt, and wooden boxes that are on the track. The textures are provided along with the source files.
   b. If you wish to support textures, first investigate the 'Texture' class. You need to create and use instances of this class in certain parts of the code – we left hints in the TODOs.

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 are 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.

Viewer.java

This class implements the rendering of the game. The render method is used to start drawing the game. The game is rendered 30 times per second, so each 1/30 seconds the render method is invoked and the scene is redrawn. As the user interacts with the game, the game state changes (locomotive position and orientation) and hence the scene is rendered differently. For example, when the locomotive accelerates, then the locomotive changes its position and rendering the scene again will move the locomotive 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 locomotive 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 locomotive and camera. When this method is invoked, it returns the locomotive 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 ($t2$), is the distance that the locomotive moved, between the interval $\Delta t = t_2 - t_1$, in each direction.
2. getCarRotation - this function returns the locomotive 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.

Box.java

For the Box class, you will use your code from HW5. To add shading support, you must change your code to including setting normal vectors for the box's vertices (see TODOs).

Locomotive.java

For the locomotive package, you will use your code from HW5. To add shading support, you must set material properties for each part of your locomotive. Particularly – the chassis, the roof, the wheels and the rims. This means adding calls to 'setMaterial' methods in the included Materials class.

Materials.java

This class is used to set the material properties for the different parts of the locomotive while rendering. This class includes several methods that need to be implemented, and two methods that have been implemented by us. We have also included several color vectors for you to use. Use the implemented methods and the included color vectors to plan your implementation for the remaining methods.

Lighting

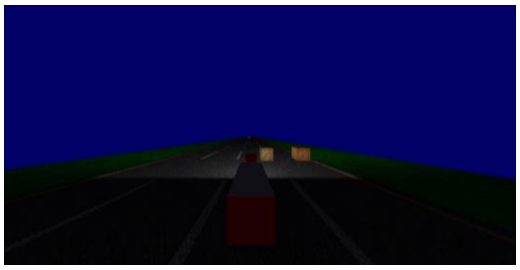
There are two lighting modes we want to support.

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

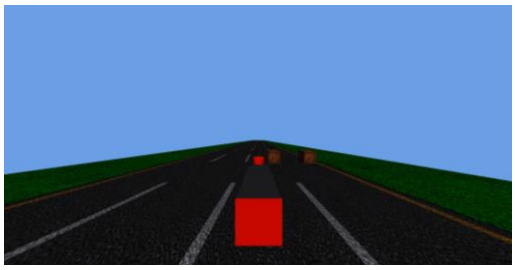
Night mode in which the scene is render at night. This means there are no light sources except for the locomotive's front lights, and a global ambient light should be set in the scene to simulate the moon. Here, the spotlights should always be directed parallel to the locomotive's forward direction.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

Note: you should set two different background colors for the scene (depending on the lighting mode – day or night).

Recommendation: as we said in class, the light source position and direction is affected by the model-view matrix. So in order to simulate the car lights, you should define the locomotive light properties right before you render the locomotive. This way both the light-sources that you defined and the locomotive will undergo go the same rotation transformation, which could affect the orientation of the car-lights. By doing this, you don't actually have to consider the location of the light sources for the locomotive's lights in the scene, but simply in the local coordinate system of the locomotive.



Night



Day

The Interdisciplinary Center: Computer Graphics Course 2021
Instructor: Prof. Ariel Shamir
Submission due: 18th of June 2021 (17:00)

World/Scene 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. We will now specify the unit of measure in our world coordinate system. These measures were used in our implementation and are given to you for guidance.

General:

- The unit of measure we use is **meter**.
- The track is spanned along the -z direction.
- The start of the track is at $z = 0$.

Projection plane (for the purposes of the projection transformation):

- In Third-Person view: The projection plane should be near (a small distance, you can choose a specific value by checking it achieves the desired result). The camera and the projection frustum depth is at least half of a Track Segment's length. This means that the projection frustum covers almost half a track segment. The field of view in the y-direction should be between 100-120 degrees.
- In Birdseye view: The projection plane should be above the track (similarly choose a specific value that achieves the desired result) and the projection frustum depth should be large enough so that the track and the locomotive are rendered and not clipped. The field of view in the y-direction should be around 60 degrees.

The locomotive initial location:

- The locomotive is originally centered on the middle of the track (x=0). The car wheels touch the asphalt and the back end of the locomotive is at z=-2.
- The locomotive's dimensions are uniformly scaled by 3.0, compared with the measures used in the locomotive local coordinate system.

***Note : In the locomotive class we used depth to describe distances in the z axis, from this point on we will use the terms 'length' and 'depth' interchangeably.**

***Note : In the local coordinate system of the locomotive, the locomotive points in the +z direction. You need to apply a transformation that orients the locomotive in the -z direction.**

The camera initial location:

- In Third-Person view: The camera is located **2.0** meters away from the back end of the locomotive, it is 3 meters above the ground and is looking in the -z direction.
- In Birdseye view: the camera is 50 meters above the ground, it is looking down in the -y direction and its up vector is the -z direction. The camera is 22 meters away from the front of the locomotive in the -z direction (i.e, if the locomotive front body ends at z=a then the camera is located at z=a-22).

The track segment:

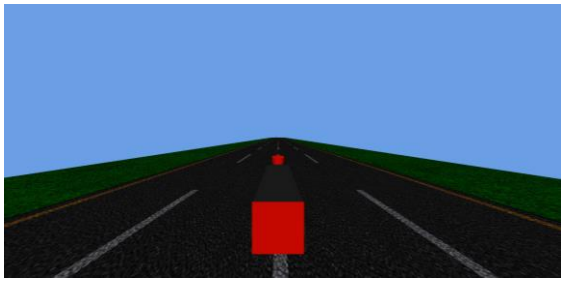
Each track segment is comprised of blocks of asphalt and grass (we specify the dimensions for the track segment, and for each block of asphalt and grass that comprise the track):

- Each Track Segment should be 500 meters length:
- Asphalt block width is 20 meters.
- Asphalt block length is 10 meters.
- Grass block width is 10 meters.
- Grass block length is 10 meters.
- Wooden box length is 1.5 meters.

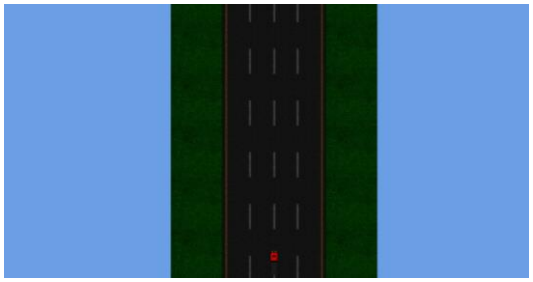
Note: we don't mind if you choose to setup the scene differently (as long as you satisfy the requirements).

View and Projection

- The camera needs to be setup (the viewing transformation) such that the locomotive and the camera are fixed to each other.
- You should define your viewing transformations with gluPerspective.
    - Make sure your viewing volume depth is at most the length of a track segment (depth).
    - You should support two different perspectives : Birdseye and third person view.



Third Person



Birdseye

Textures (bonus – 5 points)

To implement textures in the code, you will use the supplied Textures.java class. This class is used to create new textures, and bind textures for rendering – read the supplied code and make sure you understand it. You are expected to use textures to decorate the race track – the asphalt, the grass, and the boxes. The textures you need to use are provided in the 'Textures' folder.

To assist you with implementing textures, review the 'OpenGL_Textures_Self_Reading.ppt' file that is posted in the recitation section on Moodle.

The Interdisciplinary Center: Computer Graphics Course 2021
Instructor: Prof. Ariel Shamir
Submission due: 18th of June 2021 (17:00)

Submission

- Submissions are in pairs.
- Zip file should include :
    - Your project source folder only, including files you didn't change. Make sure to not include the lib folder.
- 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).

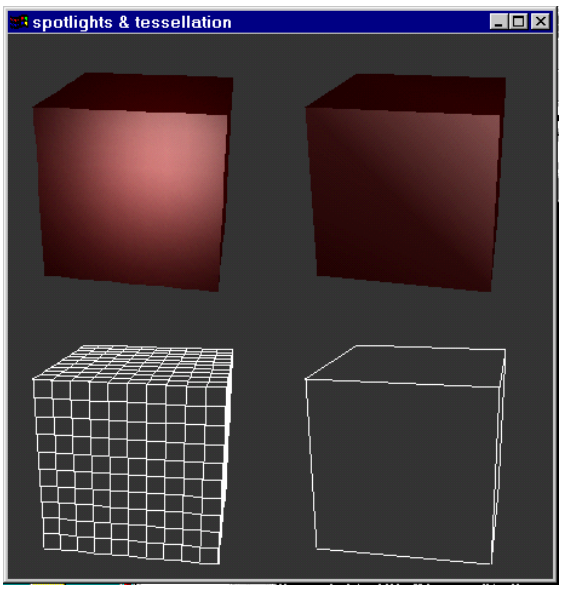
Final remarks:

1. The above pseudo code is not complete - there are some details left out. Make sure to test yourself.

2. We don't need different objects for $s_1, s_2, s_3, \ldots, s_n$. 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.

Appendix B – Track Segment Rendering, Lighting Limitations

Recall our discussion in recitation 10 about lighting limitations : OpenGL uses Gouraud shading which means reflectance calculations only take places at the vertices, and the colors for the interior of any primitive are interpolated by the colors of the vertices. This means that the more vertices we have, the more reflectance calculations take place, and the result is more realistic. For example :



In this example, we model the same cube in two different ways – one is by setting one vertex for each cube vertex, the other is by diving each face of the cube into many squares, and by so setting many additional vertices. Although we model the same object we have major shading differences.

In the illustration on the right, reflectance calculations are only carried out in the cube's vertices. In the cube's face that is visible to us only one of three vertices is actually well lit. As every point on the interior of the face will be colored by an interpolation on the vertices, this causes colors on the interior of the face to be biased towards a darker tone.

To accommodate this, when rendering the track segment, we did not model each segment of asphalt/grass using a single primitive, but divided it into multiple primitives to achieve more realistic lighting results.