RobotRace Assignment 2017/18
2IV60 Computer Graphics

Jack van Wijk, Kasper Dinkla, Paul van der Corput, Niels Rood
Eindhoven University of Technology

December 18, 2017

Introduction

This document contains the assignments for the course 2IV60 Computer Graphics of the Faculty of Mathematics and Computer Science of Eindhoven University of Technology. The assignment is split up into a number of exercises, where in each exercise a graphic application is developed further, using the material presented in the course. Assignments should be done by pairs of students, forming pairs is left to the students. The overall theme is RobotRace: an animated view of simplified human figures, moving over a race track.

RobotRace.zip

A template project is provided to enable students to make a quick start and to focus on graphics. It provides a simple user interface, a class `Vector` with a number of basic operations, and a minimal working example. The class `RobotRace` describes the overall scene, and it contains instances of the classes `Robot`, `Camera`, `RaceTrack`, and `Terrain`. The current state of the user interface is recorded in a variable `gs` (an instance of the class `GlobalState`). The attributes of `gs` describe the current state, including aspects like what has to be shown and how. Appendix A gives an overview of `gs`. All attributes of `gs` are controlled by the provided interface; students can focus on refining methods in the template project, such as `setView()`, `drawScene()`, etc. These methods are mostly in the file `RobotRace.java`.

Preparation

Install NetBeans 8.1 or higher and the Java Development Kit 8.0 or higher, if you have not done so already:

- Go to http://www.oracle.com/technetwork/java/javase/downloads;
- Choose the NetBeans option (captioned JDK 8u... + NetBeans);
- Accept the license agreement;
- Download and run the installer that matches your platform.

Next:

- Download the file RobotRace.zip from Canvas and extract its contents to a local folder;
- Start NetBeans and open the project RobotRace.

Finally, read this whole document to get an overview and to be aware of the required features.

Required features

Axis frame

In this exercise we are going to draw a standard axis frame. The axes have length 1 meter, are aligned with the X, Y, and Z axes, and consist of a block with a cone at the end to show the direction. Show the position of the origin with a yellow sphere. Use red, green, and blue as colors for the X, Y, and Z axes, respectively.

1. First elaborate the `RobotRace.drawArrow()` method, such that it draws a single arrow.
2. Use `RobotRace.drawArrow()` to make `RobotRace.drawAxisFrame()` draw the standard axis frame; call the latter from `RobotRace.drawScene()`. Make sure that the axis frame is only drawn if `gs.showAxes` is true.

Viewing

In this exercise we extend the given methods `RobotRace.setView()` and `Camera.setDefaultMode()`. The Z axis must point upward, and the center point is given by `gs.cnt`. For the specification of the eye point we adopt spherical coordinates, as follows. Suppose V is a vector pointing from the center point to the eye point. Its direction is specified by two angles: the azimuth angle `gs.theta` (angle between V projected on XY-plane and positive X axis) and an inclination angle `gs.phi` (angle between V and XY-plane). Furthermore, the distance of the eye point to the center point `gs.cnt` is given by `gs.vDist`.

- Derive the position of the eye point E from these parameters;
- Apply the viewing transform by implementing the `setDefaultMode()` method of the `Camera` class.

Robot

Model and draw a robot. This robot consists of at least the following elements: a torso, a head, two arms and two legs. Each of these elements is displayed using one or more shapes, such as spheres, ellipsoids, blocks, and cones.

- Think about your robot. What do you want to express? Do you aim at PlayMobil style or at a figure of a science fiction movie?

- Design your robot on paper. Start with a stick-figure, where the positions of important points, such as joints, are indicated and connected by lines; next attach shapes. Make multiple sketches until you are satisfied. Use views from front and side. Use as convention that the robot stands in the origin, it is aligned parallel to the Z axis, and it looks in the direction of the positive Y axis. Use meters as units, and give your robot a human size;
- Define parameters to describe the shapes and the relative positions of the elements. Choose these parameters such that you can later easily manipulate/animate the robot. Extend the given class `Robot` with a set of variables for these parameters;
- Implement method `Robot.draw()` and call it in `RobotRace.drawScene()` to show an instance of your robot. Make sure that you model your robots in a hierarchical manner, as this will make animating the robot a lot easier in later stages; see `RobotRace.drawHierarchy()` for an example of drawing a hierarchy of simple objects.

Now let's make the robot model look like an actual robot by applying material effects on it:

- Define four sets of material properties for diffuse and specular reflection in the enum `Material`, to give (parts of) a robot a gold-like, silver-like, wood-like, or an orange plastic look.
- Modify the robot shader program such that it can render robots with these material properties, using per-fragment shading. Show four robots side-by-side, such that they are easily inspected visually.

Next we start animating the robots:

- Make the arms and legs of the robots move using the sliders provided by the user interface; you can use the variables `gs.sliderA` through `gs.sliderE` to do so. Make sure that the arms and legs move correctly.
- Let the robots walk or run, while also (individually) animating their limbs. Try to make the motion look natural, without slip.

Race tracks

Parametric Race track

Next we consider the race track. Suppose the centerline of the track is defined as a curve $P(t), t \in [0, 1]$, and that the curve has a tangent vector $V(t)$. The provided class `RaceTrack` defines two methods for this:

```
Vector getPoint(double t);
Vector getTangent(double t);
```

which should return the position and tangent for parameter $t \in [0, 1]$.

- We work with an oval-shaped test track. Implement `ParametricTrack.getPoint()` and `ParametricTrack.getTangent()` for

$$P(t) = (10\cos(2\pi t), 14\sin(2\pi t), 1).$$

Use this curve as a centerline, and define the race track as follows. The track should consist of 4 lanes, with each lane being 1.22 meters wide. The track can be assumed to be flat, in the plane $z = 1$. Furthermore, the track has sides: polygons from the boundary of the track to the plane $z = -1$, such that it looks like the track is on a solid concrete foundation.

- Draw the track in `RaceTrack.draw()` and use the methods `getPoint()` and `getTangent()` to do so. Determine the number of triangles or quads such that the track looks reasonably smooth, while not using too many elements. Make sure that normals are calculated and set properly: unit-length and pointing outward.
- Let the robots move over the track. If `gs.tAnim=0`, the robots are positioned at the start line. After this, as `gs.tAnim=0` increases, the robots advance along the track. At first, let the robots move with a constant speed; next, introduce some variation in their speed during the race, such that the race is more interesting to watch. Implement and call `RaceTrack.getLanePoint()` and `RaceTrack.getLaneTangent()` to find the position and orientation of a robot in a given lane for a given parameter $t \in [0, 1]$.

Use texture mapping to decorate the track.

- Modify the track shader program so that it allows texturing the track.
- Design a texture `track.jpg` for the track, showing the lanes, indications of distance traversed, and possibly track numbers. Use a texture that is replicated about each 10-20 meter. Map the texture on the track such that distortion along the centerline is minimal.
- Design a texture `brick.jpg` for the sides of the track, pretending that the object on which the track is placed is constructed from large bricks. Map this textures on both sides of the race track. Again, minimize distortion, to pretend that the same bricks are used throughout.
- Finally, enhance your shader program with per-fragment shading, and modulate texture colors with colors produced through shading.

Spline Race track

Define the centerline of the track using cubic Bézier segments. A track is defined by a sequence of control points $P_i, i = 0, ..., 3N$, where N is the number of segments and each sequence of points $P_{3k}, P_{3k+1}, P_{3k+2}, P_{3k+3}$ defines a segment. To obtain a closed track, $P_0 = P_{3N}$ should hold.

- Implement the method

  ```
  Vector getCubicBezierPnt(double t, Vector P0, Vector P1,
                           Vector P2, Vector P3);
  ```

  in the class `RaceTrack` to evaluate a cubic Bézier segment from parameter value t.
- Implement function

```
Vector getCubicBezierTng(double t, Vector P0, Vector P1,
                         Vector P2, Vector P3);
```

in class `RaceTrack` to evaluate the tangent of a cubic Bézier segment for parameter value t. Implement these methods within RobotRace.

- Define and show an additional track, defined by cubic segments. Depending on the value of `gs.trackNr`, different tracks are used. If `gs.trackNr` is 0, the parametric track is used; otherwise the Bézier track is shown. The race tracks are initialized in the constructor of RobotRace. Define the tracks such that all corners are smooth; make also sure that the tracks fit in an area $x, y \in [-20, 20]$.

Moving camera

Now that we have defined a race, we want to follow it in a natural and exciting way. Extend your application to provide two possibilities for placing the camera, according to the value of parameter `gs.camMode`. The two settings should be:

0: Default camera, gives an overview, and enables the user to explore the scene from different positions. This camera model is similar to the one already implemented above using spherical coordinates, in the sense that a natural center is chosen and that the user can view the scene from different directions;

1: First-person mode. Suppose each robot has a camera mounted on the top of his head, pointing ahead. Your application should provide at least the view from the perspective of the last robot in the race. Thus, the task is to adjust the viewing model to handle first-person camera mode, which should be implemented in `Camera.setFirstPersonMode()`.

Terrain

Next, we extend our scene to include some landscape/terrain surrounding the race track. The terrain is modeled as a height field $height : [-20, 20] \times [-20, 20] \to [-1, 1]$. Such a height field can be defined in many ways. Here we give just a simple possibility, in which the height field is defined as the sum of two cosine waves, *i.e.*,

$$height(x, y) = 0.6 * \cos(0.3 * x + 0.2 * y) + 0.4 * \cos(x - 0.5 * y),$$

Other options could be to define the height field using a Bézier-surface, measured data, procedural/fractal methods, etc.

For this exercise, we are going to implement the height field using GLSL shaders.

- Implement a 'flat' height field in `Terrain.draw()`, which draws a plane made of multiple smaller planes (made out of quads or triangles).

- Modify the terrain shader program, such that it implements the height field by manipulating the height of all points. Have this shader also create the effect that areas below zero meters are blue (water-like appearence), from 0 to 0.5 meter yellow (sand-like), and from 0.5 meter and higher green (grass).

- Finally, draw a transparent, grey polygon at the $z = 0$ level, to simulate a water surface, which could also be animated in a GLSL shader, see below.

Finishing touch

The scene is still somewhat empty, so here we provide some additional ideas for features. Implement at least two of the features from the list below.

- Include trees in the scene, making sure they are not on the tracks. Trees must consist of at least three primitives and must have variation in size and shape;
- Include a large video screen in the scene, which shows a close-up from the race;
- Add obstacles on the track, which have to be circumvented, climbed or jumped-over by the robots;
- Allow the robots to leave their lanes and select an optimal track, while avoiding the other robots;
- Apply textures to body parts of the robots;
- Use a smooth animation when the view is changed from one camera position to another;
- Define a custom height field, mimicking a more realistic mountain landscape;
- Define the track such that it is always 1 meter above the terrain, while the bricks defined in the texture are still horizontal;
- Implement bump mapping to make the track/terrain appear more realistic;
- Make the water surface more interesting by making it wave using shaders;
- Create a skybox around the scene, and have this skybox reflect on the gold and silver robots, and on the water surface, using environment mapping;
- Have the robots cast shadows on the track and terrain;

Appendix A - Global State

The instance gs of class GlobalState contains the state variables that describe the scene to be displayed. The class GlobalState is:

```
class GlobalState {
   boolean showAxes; // Show an axis frame if true
   boolean showStick;// Show robot(s) as stick figures
   int     trackNr;  // Track to use:
                     // 0 -> test; 1 -> O; 2 -> L; 3 -> C; 4 -> custom

   float tAnim;      // Time since start of animation in seconds

   int   w;          // Width of window in pixels
   int   h;          // Height of window in pixels

   Vector  cnt;      // Center point
   float   vDist;    // Distance eye point to center point
   float   vWidth;   // Width of scene to be shown
   float   phi;      // Elevation (colatitude) angle in radians
   float   theta;    // Azimuth angle in radians

   int   camMode;  // In race mode: 0 -> overview,
                   //               1 -> tracking helicopter
                   //               2 -> view from the side on leader,
                   //               3 -> viewpoint of the last robot,
                   //               4 -> autoswitch
}
```

The values of the parameters can be set with the given user interface:

- The camera viewpoint angles, phi and theta, are changed interactively by holding the left mouse button and dragging;
- The camera view width, vWidth, is changed interactively by holding the right mouse button and dragging upwards or downwards;
- The center point cnt can be moved up and down by pressing the ’q’ and ’z’ keys, forwards and backwards with the ’w’ and ’s’ keys, and left and right with the ’a’ and ’d’ keys;
- Other settings are changed via the menus at the top of the screen.

The main elements of the class RobotRace are:

```
class RobotRace extends Base {

   private final Robot[]     robots;
   private final Camera      camera;
   private final RaceTrack[] racetracks;
   private final Terrain     terrain;

   initialize();       // Initialize (OpenGL) settings
   setView();          // Initialize viewing transformation
   drawScene();        // Draw the entire scene
}
```

For full detail, see RobotRace.java.