RobotRace 2015/15
2IV60 Computer Graphics Assignment

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

November 3, 2015

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: Students have to prepare 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 for the assignment; a class `Vector` is provided, with a number of basic operations; and a minimal working example. The class `RobotRace` describes the overall scene, and 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 interface provided; students can focus on refining methods in the template project, such as `setView()` and `drawScene()`. These are in the file `RobotRace.java`.

Preparation

Install NetBeans and the Java Development Kit, 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 Oase (http://oase.tue.nl, 2IV60 Computer Graphics) or from the course page (http://www.win.tue.nl/˜vanwijk/2IV60), and extract its contents to a local folder.
- Start NetBeans and open the project RobotRace.

Finally, read this document, to get an overview and to be aware of features asked for following rounds.

Submission

Results of the assignment have to be submitted in two rounds. The strict deadlines for submission are:

- Round 1: Monday November 30, 2015 (after 3 weeks);
- Round 2: Monday January, 11, 2016 (after 7 weeks).

Students are recommended to work on the assignments throughout the course, and not to wait for the last days before the deadlines. Results must be submitted via Peach (http://peach.win.tue.nl), under the course Computer Graphics. For each round, students submit the java source files `Camera.java`, `Material.java`, `RaceTrack.java`, `Robot.java`, `RobotRace.java`, and `Terrain.java`. For the second round, additional texture files must be submitted. Please upload them separately, so not packed in one archive file. One submission per pair of students suffices.
The instructors:

- check (via Peach) if the work submitted is original work;
- check if the required features have been implemented;
- judge the code on understandability and explanations provided via comments.

Concerning comments, the following guidelines apply. About half of the program should consist of comments. New methods must be provided with a description of their effect and of all parameters; all variables must be described, as well as all loops. Furthermore, non-trivial steps must be described, possibly including derivations. Enable the human reader of the program to understand quickly what is done and why.

Results should not produce OpenGL errors, caught through glGetError at the end of each rendering cycle in the given framework for RobotRace.

For the first round maximally 100 points can be scored; for the second round 200 points. Per part the number of points is indicated.

1 Round 1: Robots

In all assignments we assume a world space coordinate system with meters as units, and with the Z-axis pointing up.

1.1 Axis frame - 20 points

Elaborate the method `RobotRace.drawAxisFrame()` 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 three axes. Draw the frame if `gs.showAxes` is true. Use `glutSolidCube()`, `glutSolidCone()`, and `glutSolidSphere()` for drawing the primitives, and standard OpenGL transformation functions to position and scale them. Call `RobotRace.drawAxisFrame` from `RobotRace.drawScene`.

1.2 Viewing - 20 points

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 a spherical coordinates approach. 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`, and the width of the scene to be displayed is given by `gs.vWidth`. Specifically, consider a line through the center point C, perpendicular on the view direction and parallel with the XY-plane. This line is shown horizontally on the screen. The projection has to be set such that the visible part of this line has length `gs.vWidth`.

- Derive the position of the eye point E from these parameters;
- Implement the viewing transform using `gluLookat()` and `gluPerspective()` according to this specification. The last function requires values for `znear` and `zfar`. Here for instance 0.1*`gs.vDist` and 10.0*`gs.vDist` can be used.

1.3 Robot - 40 points

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, 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 on the origin, with the length axis along the Z axis, and looks in the direction of the positive $Y-$axis. Use meters as units, 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 on easily manipulate (position, vary in shape, draw, move) the robot. Extend the given class `Robot` with a set of variables for these parameters;
- Implement the method `Robot.draw()` and call it to show an instance of your robot. If `gs.showStick` is true, a schematic stick-figure has to be shown, else the robot with all shapes attached.

A practical remark: Do not use the variable names `track`, `brick`, `head`, and `torso`, as this will give rise to clashes of names of textures to be used later.

1.4 Shading - 20 points

Add shading and lighting:

- Enable shading, ambient light and one light source. Use a light source at infinity. The direction of the light is such that light comes more or less from the direction of the camera. The direction of the light is shifted by 10 degrees to the left and upwards with regard to the view direction.
- Define four sets of material properties for diffuse and specular reflection in the enum `Material`, to give (parts of) the robot a gold-like, silver-like, wood-like, and an orange plastic look. Show four robots side by side, such that they are easy to compare.

2 Round 2: Robots on the move

2.1 Race track - 30 points

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 given class `RaceTrack` defines two methods for this:

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

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

- We work with a test track. Implement `getPoint()` and `getTangent()` for

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

Given this centerline, we define the track as follows. It 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()`. 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, pointing outward.

- Let the robots move over the track. If `gs.tAnim=0`, the robots are positioned at the start line, after that they move 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 $t \in [0, 1]$.

2.2 Moving camera - 20 points

Now that we have a race, we want to follow it in a natural and exciting way. The parameter `gs.camMode` gives various options:

0: Default camera, gives an overview and enables the user to view it from different sides. This camera model is similar to the one defined in assignment 1.2, in the sense that a natural center is chosen and that the user can view the scene from different directions, using a spherical coordinates approach.

1: Helicopter mode. The robots are followed by a helicopter that flies above the robots, such that they are shown centered and from above;

2: Motor cycle mode. The (leading) robots are shown from the side, as if the camera man is located at a motor that drives along with them;

3: First person mode. Suppose each robot has a camera mounted on top, give the view from the perspective of the last robot in the race;

4: Auto mode. Switch automatically between the previous modes (or more), to get a lively and entertaining animation.

The task is now:

- Adjust the viewing model to handle these options.

For camera modes 1 and 2, a (group of) robots has to be selected to focus on. You are free to make a choice here. The most simple (and most boring) option is to focus on just one particular robot, it is more interesting to change the focus periodically over all robots. Aim for an interesting result, and explain your choice in the comments in your code. The modes can be implemented in `Camera.setHelicopterMode()`, `Camera.setMotorCycleMode()`, `Camera.setFirstPersonMode()`, and `Camera.setAutoMode()`.

2.3 Walking robots - 30 points

So far, the robots were allowed to slide over the track as rigid objects, next we make this more interesting:

- Let the robots walk or run over the track with separately moving limbs. Try to make the motion look natural, without slip.

Feel free to use separate ankle, elbow, knee, and wrist joints; let your robots walk and jump. And how about *To Di World* (when a robot has won the race in a new world record) or walking Gangnam Style?

2.4 Spline tracks - 30 points

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 for parameter value t.

- Implement the function

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

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

- Define and show different tracks, defined by cubic segments. Dependent on the value of `gs.trackNr`, different tracks are used. If `gs.trackNr` is 0, the test track of assignment 2.1 is used, for higher values tracks defined by cubic segments are used: (1) a track in the form of a letter O; (2) the outline of the letter L; (3) the outline of the letter C; or (4) a track you defined yourself is shown. The race tracks with their control points are initialized in the constructor of `RobotRace`. Define the tracks such that all corners are smooth. Make sure that the tracks fit in an area $x, y \in [-20, 20]$.

2.5 Textured tracks - 30 points

Use texture mapping to decorate the track and the robots.

- 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 side of the track, pretending the mass below the track is constructed from large bricks, and map it to the sides. Again, minimize distortion, to pretend that the same bricks are used throughout.

Put the two textures (`track.jpg, brick.jpg`) in the same folder as `RobotRace.java`. Read the comments in the given template (see ”Textures”) for information on how to load and bind these textures.

2.6 Textured robots - 10 points

Apply also texture to the heads and torsos of the robots, using textures `head.jpg` and `torso.jpg`. Note that basic GLUT-objects, such as cubes, do not provide texture coordinates. One solution to fix this is to draw an additional quad for for instance the face of a robot; another solution is to draw cubes yourself via a new method. Put the two textures (`head.jpg, torso.jpg`) in the same folder as RobotRace.java.

2.7 Terrain - 30 points

Next, we add a terrain. The terrain is given as a height field $height : [-20, 20]^2 \to [-1, 1]$. Such a height field can be defined in many ways. Here we give a simple height field, defined as the sum of two cosine waves:

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

but there are many other options, like using a Bézier-surface, measured data, and procedural methods to produce fractal landscapes.

- Implement the method `Terrain.heightAt()` that computes the height at each location (x, y).
- Show the given height field in the scene by implementing `Terrain.draw()`. Approximate the height field by a mesh of triangles or quads. Make sure that normals are calculated and set properly: unit-length, pointing outward.
- Use 1D texturing for the height field to give the effect that areas below zero are blue (water), from 0 to 0.5 meter yellow (sand), and from 0.5 meter and higher green (grass).
- Add a transparent, grey polygon at the $z = 0$ level, to simulate a water surface.

2.8 Extra features - 20 points

The scene is still somewhat empty, here are some more challenges. Successful implementation of two of these is awarded with 20 points.

- Add trees to 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;
- Add a digital clock, either in the scene or on screen, which shows the current time;
- Add a PictureInPicture option: Show a separate view from a different camera in a small viewport on top of the standard view;
- Add 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;
- 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 m above the terrain, while the bricks defined in the texture are still horizontal.

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 described by:

```
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 more detail, see RobotRace.java.