10531 Video Game Technologies Lab. 3 - 06/04/2016

Collision Detection

Supervisor: Abel Gomes *Scribe: G. Amador*

In this laboratory, students will learn how to perform collision detection in the JMonkeyEngine 3.0 (jME3), either using student implemented collision detection or the engine bindings to JBullet `http://jbullet.advel.cz/`, a port to the Java programming language of the Bullet `http://bulletphysics.org/wordpress/` physics library.

1 Specific Learning Goals

Finalized this worksheet, students should know and be able to:

1. Associate to an object in the scene graph either a bounding sphere or an axis aligned bounding box (AABB).
2. Understand how broad phase collision detection is made, specifically, AABB with AABB, bounding sphere with bounding sphere, and bounding sphere with AABB.
3. Enable and disable the display of bounding volumes, implemented by the students in jM3.
4. Detect collision among convex shapes using the GilbertJohnsonKeerthi (GJK) distance algorithm, implemented by the students in jM3.
5. Use JBullet within an jME3 application in order perform collision detection among elements of the scene graph.

2 Collision Detection Overview

Without proper collision detection in game objects will overlap, resulting in hard to solve visual artifices, as illustrated in Fig. 1.

In this laboratory we will overview one of the two stages of collision detection, namely, broad phase. Suppose we have N objects in a given game scene. Then there are $(N \times (N-1)/2$ objects to compare for collision, i.e., for each in game object compare if it collides with all other objects in the game scene without retesting an already tested pair. To make this test even worst, objects in 3D games are comprised of regular polygons, most often triangles. Assuming objects of M triangles we would have $(N \times (N-1)/2 * (M \times (M-1)/2$ tests. For example if we have 100 objects each with roughly 10 triangles, we have 222750 possible collisions to check per game frame. In order to reduce largely the number of tests broad phase collision detection is performed first, i.e., a crude but fast method of eliminating large numbers of collision checks. Its purpose is primarily to determine if objects do not collide so that they need not be considered in narrow phase collision detection, and return a list of pairs of objects that potentially collide.

Broad phase collision detection is performed by enveloping complex game objects into simpler geometries (bounding spheres, bounding ellipsoids, bounding boxes, etc) and testing these geometries for collision instead of the enveloped geometries. This, speeds up testing and reduces the amount of tests to perform. However, some games solely use broad phase collision detection due to hardware limitations and depending on the enveloping geometry chosen two bounding volumes night overlap but the enveloped geometries might not in fact collide.

In this laboratory students will learn how to use two of such geometries, namely, axis aligned bounding volumes (AABB) and bounding spheres.

Figure 1: Result of poor collision detection, possibly just broad phase, in the game Mercenaries 2.

3 Programming Exercises

In order to solve each of the following exercises first open the projects `jME3Project4a`, `jME3Project4b`, and `jME3Project4c` available at the course web page `http://www.di.ubi.pt/~agomes/tjv/`. To open a project file:

1. Open jME3.
2. In jME3 choose `File > Open Project` from the main menu, and select the directory where the project folders were downloaded and select the projects `jME3Project4a`, `jME3Project4b`, and `jME3Project4c`.
3. In jME3 choose `Window > Action Items` from the main menu, to navigate between tasks in each exercise.

Before proceeding to the following exercises students should run the projects `jME3Project4a`, `jME3Project4b`, and `jME3Project4c` and uncomment each of the lines in the `public static void main(String[] args)` method ending with a `// TODO: UNCOMMENT ...` statement. These are some of the basic default features of jME3 projects, that the student may require to activate deactivate in the course game project. Note that the default `flyCam` camera is activated in either of the projects, and can be translated with the `WASDZQ` keys and rotated with the `UP DOWN LEFT RIGHT` keys or with the mouse.

REMARK 1:

The following exercises are to be completed by the student in the lab class. Unsolved exercises during any lab class are recommended to be completed at home.

REMARK 2:

Each exercise is a single step of a project. Each exercise starts with a `// BEGIN OF EXERCISE` comment, and finalizes with an `// END OF EXERCISE`. Fig. 1 illustrates in `Action Items` (red ellipse) the tasks or `TODO`s to perform in the third exercise (dark square). To navigate in the source code to any of the `Action Items` simply double click in any of them with the left mouse button.

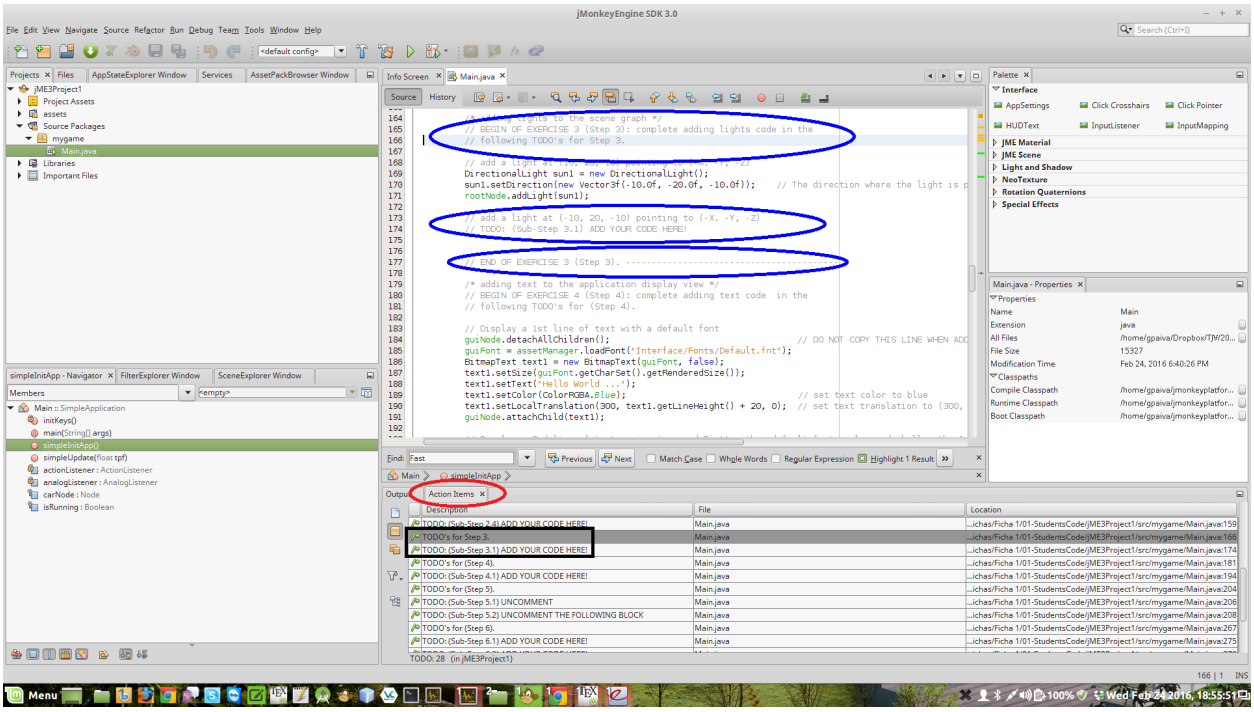

Figure 2: Exercise 3 tasks and termination comment.

Exercise 1 — adding models to the scene graph

This exercise (Step 1) consists in finishing the process of adding the robot model to the scene. Students must finish parsing the file, setting up the parsed data (vertices, normals, and texture coordinates), and finally add the robot into the scene graph. Students should use the model.data file provided in the assets model folder of `jME3Project4a`.

Terminated this exercise, the output., after some translation backwards of the camera, should be the one shown in Fig. 3.

Exercise 2 — adding a second axis aligned bounding box (AABB) to the scene graph

This exercise (Step 2) consists in finishing adding a second axis-aligned bounding box (AABB2) to the scene graph. AABB2 should be a red box with `min2` and `max2` values already initialized in `jME3Project4a` source code.

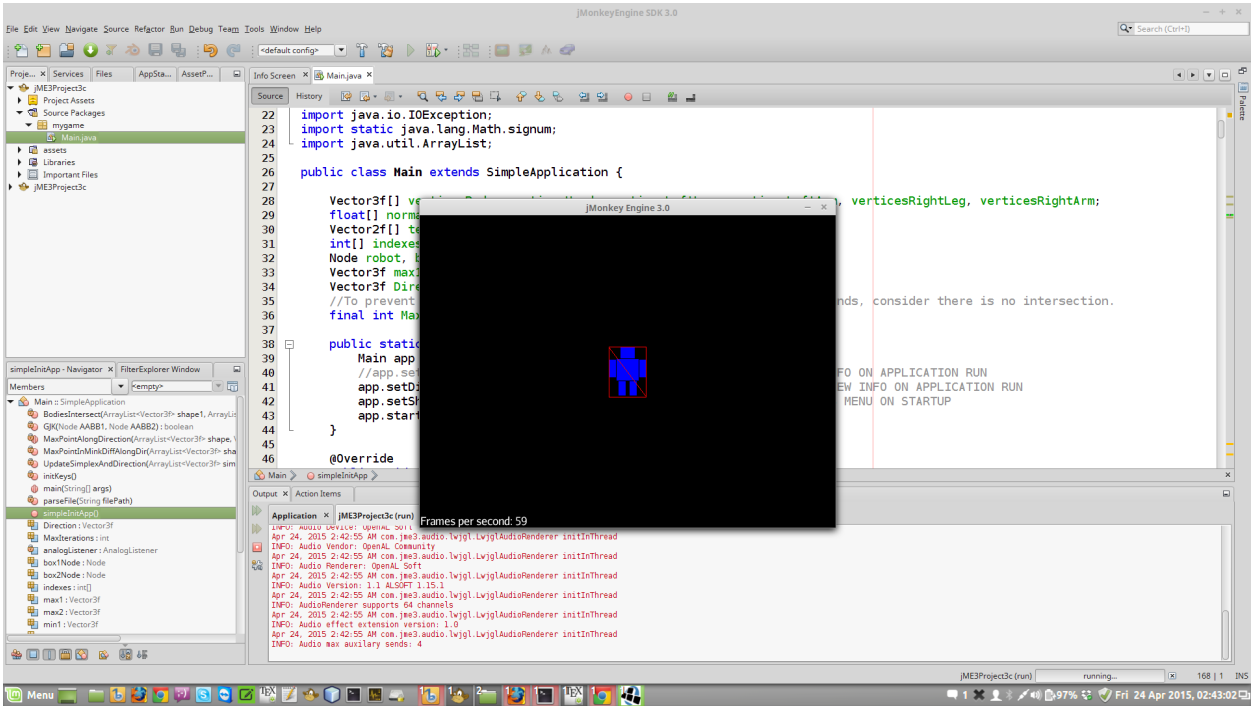

Figure 3: Exercise 1 output.

Finalized this exercise the output., after some translation backwards of the camera, should be equal to the one illustrated in Fig. 4.

Exercise 3 — register event-sensitive keys

This exercise (Step 3) is the first thing to program event-sensitive keys. More specifically, it consists in registering such keys with the `inputManager`. This is usually called reactive programming (or event-based programming) so that any key may be associated to a single event-sensitive callback (i.e., a function). In the present exercise, we intend to program the up, down, and right keys, so that they to be must be mapped or added to the `inputManager`. For that purpose, please first study the code associated to other keys in the project code. In particular, carefully read any comments in Step 3 to understand what each line of code does.

Exercise 4 — reactive programming of the keyboard

This exercise (Step 4) consists in programming keys of the keyboard. This is usually called reactive programming (or event-based programming) so that any key may be associated to a single event-sensitive callback (i.e., a function). After registering keys as we did in the previous exercise, it remains to program the keys themselves to react to events by triggering the execution of some callback. A callback is a function that describes what (i.e., the behavior) will be done after clicking in a key More specifically, the robot and its bounding box (AABB1) should mode up, down, left, right, respectively when the keys TGFH are pressed. Students should carefully read any of the comments regarding Step 4 to understand what each line of code does.

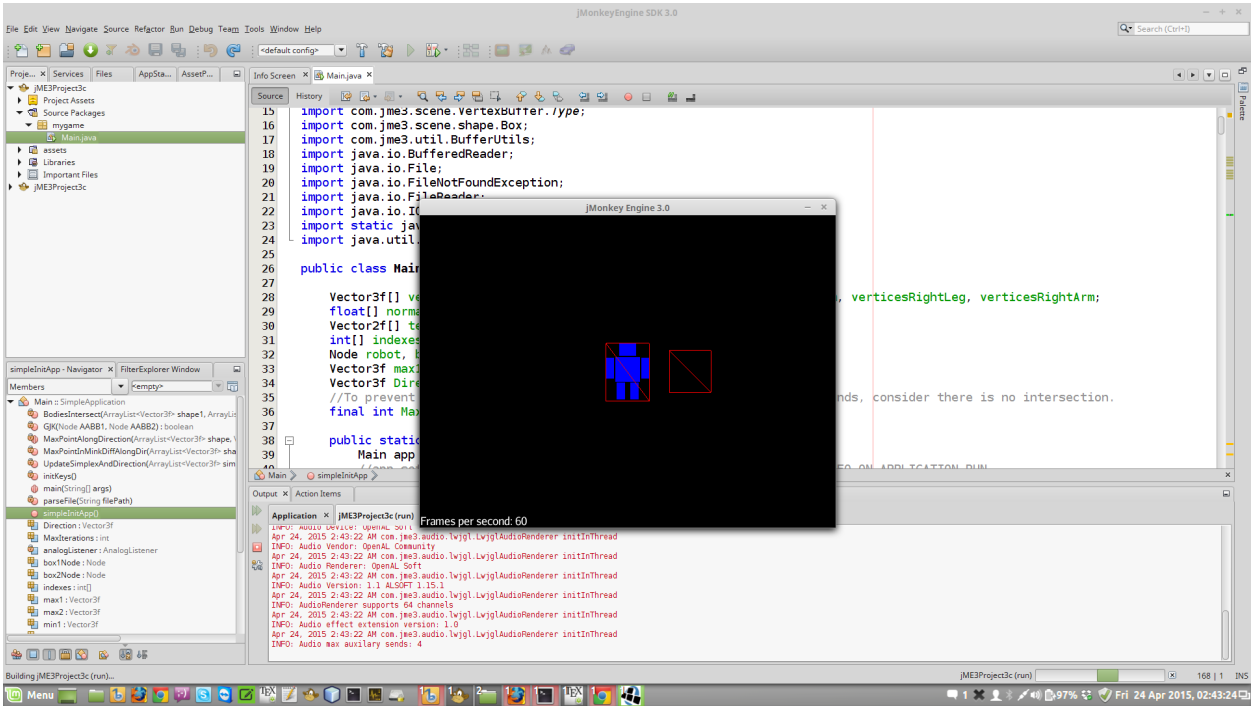

Figure 4: Exercise 2 output.

Exercise 5 — finish implement the GJK algorithm

This exercise (Step 5) consists in finishing the GJK algorithm [4]. Students are advised to consult the information available at [4], [5], and [6]. Finalized this exercise the robot should detect a collision among AABB1 and AABB2. When a collision occurs the AABB1 and the robot will not overlap AABB2 and will remain in their prior position. In particular, carefully read any comments in Step 5 to understand what each line of code does.

Exercise 6 — adding an AABB and an bounding sphere to the scene graph

This exercise (Step 6) consists in adding to the scene graph a second AABB and second bounding sphere, based on the provided AABB and bounding sphere examples. Translate the mouse backwards to look down and see the bounding volumes in the scene graph plus the car enveloped by a AABB and a sphere. Students should carefully read any of the comments regarding Step 6 to understand what each line of code does.

Finalized this exercise the output., after some rotation and translation backwards of the camera, should be equal to the one illustrated in Fig. 5.

Exercise 7 — activate AABB with AABB collision test

This exercise (Step 7) consists in finishing implementing AABB with AABB collision test, method `collideAABBwithAABB`, and uncomment a few lines of code as indicated in the step `TODO's` in the source code. All the required variables in the method are already initialized students must solely implement AABB with AABB collision detection as explained at the link `http://www.miguelcasillas.com/?p=30`. If properly implemented the first AABB and the second AABB should not overlap when a collision is detected. Also, notice the amount of wasted space inside the first AABB which is not occupied by the car model. In particular, carefully read any comments in Step 7 to understand what each line of code does.

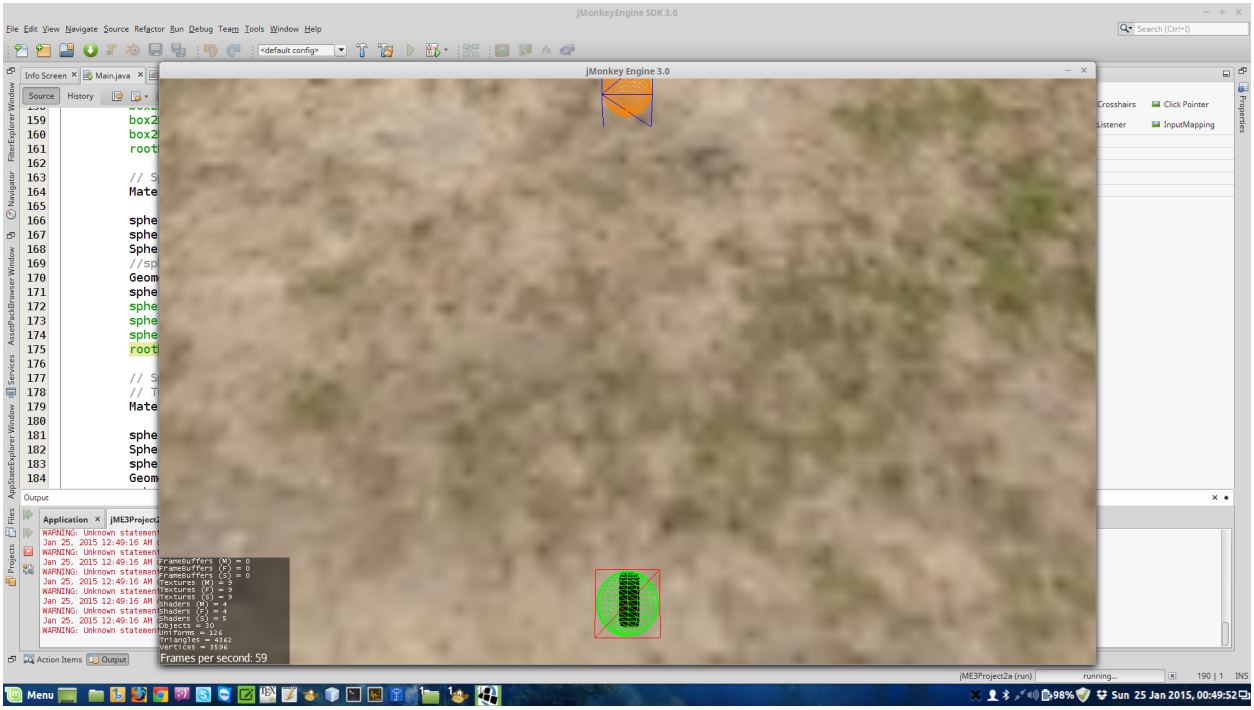

Figure 5: Exercise 6 output.

Finalized this exercise the output., after some rotation and translation backwards of the camera, should be equal to the one illustrated in Fig. 6.

Exercise 8 — activate bounding sphere with bounding sphere collision test

This exercise (Step 8) consists in finishing implementing bounding sphere with bounding sphere collision test, method `collideSphereWithSphere`, and uncomment a few lines of code as indicated in the step `TODO's` in the source code. All the required variables in the method are already initialized students must solely implement bounding sphere with bounding sphere collision detection as explained at the link `http://www.miguelcasillas.com/?p=9`. If properly implemented the first bounding and the second bounding sphere should not overlap when a collision is detected. Also, notice the amount of wasted space inside the first bounding sphere which is not occupied by the car model. Students should carefully read any of the comments regarding Step 8 to understand what each line of code does.

Finalized this exercise the output., after some rotation and translation backwards of the camera, should be equal to the one illustrated in Fig. 7.

Exercise 9 — activate AABB with bounding sphere collision test

This exercise (Step 9) consists in finishing implementing bounding sphere with AABB collision test, method `collideSphereWithAABB`, and uncomment a few lines of code as indicated in the step `TODO's` in the source code. All the required variables in the method are already initialized students must solely implement bounding sphere with AABB with AABB collision detection adapting the test to see if a point is inside a sphere as explained at the link `http://www.miguelcasillas.com/?p=38`. If properly implemented the first bounding and the second bounding sphere should not overlap when a collision is detected. Also, notice the amount of wasted space inside the first bounding sphere which is not occupied by the car model. In particular, carefully read any comments in Step 9 to understand what each line of code does.

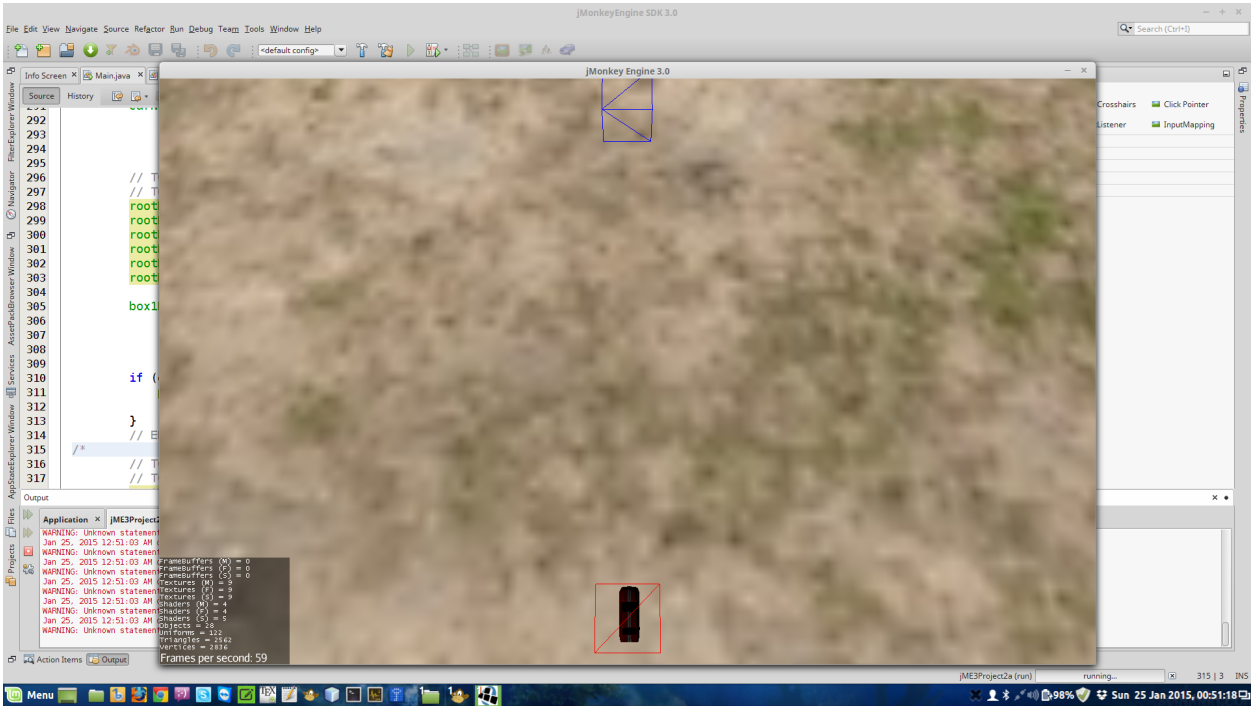

Figure 6: Exercise 7 output.

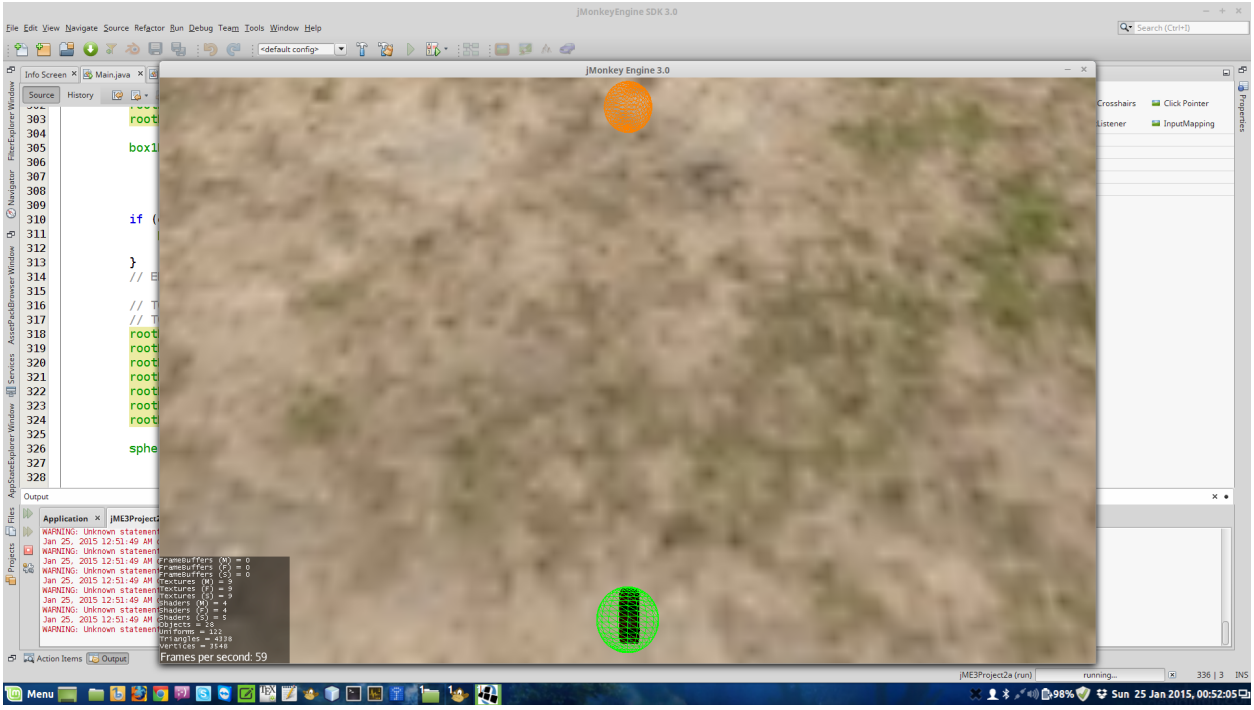

Figure 7: Exercise 8 output.

Finalized this exercise the output., after some rotation and translation backwards of the camera, should be equal to the one illustrated in Fig. 8.

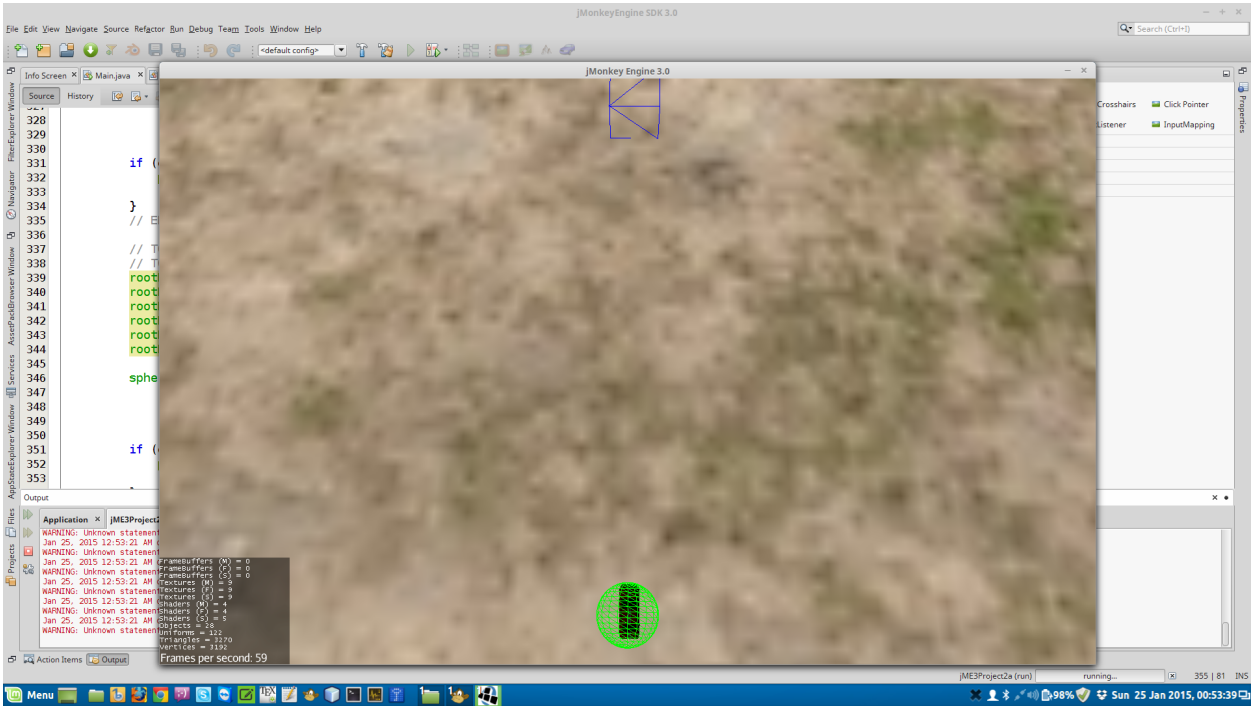

Figure 8: Exercise 9 output.

Exercise 10 — setup a skybox without adding it to the scene graph

This exercise (Step 10) consists in finishing the creation of walls of the sky box [2]. Examples of the north and down walls plus all walls textures are provided, available in the `assets/Textures/Skybox` project folder. **Students must not add each wall geometry to the scene graph root node!** Contrary, to prior laboratory sheets this will be done in a further step. Students should carefully read any of the comments regarding Step 10 to understand what each line of code does.

Exercise 11 — setup the car and city nodes without adding them to the scene graph

This exercise (Step 11) consists in finishing setting up models to latter add to the scene. The student must load a `city` model, available in the `assets/Models/city` project folder. Instructions on what affine transformations (translation, rotation, scale) should be performed on the city node are indicated in the preceding comments. Students have an example of the car model but notice that the city node does not require latter manipulation (statical obstacle). Consequently, it is not a global variable. **Students must not add the city or the car models to the scene graph root node!** Contrary, to scene graph laboratory this will be done in a further step. In particular, carefully read any comments in Step 11 to understand what each line of code does.

Exercise 12 — set up collision detection and add nodes to the scene graph

This exercise (Step 12) consists in adding walls and models to the scene graph and to the physics space. Examples for the car, north and down walls are provided. The scene graph is used to visualize models, while the physical space is a copy of the game world managed by JBullet to deal with physics, e.g., collision detection. After concluding this step, collisions among JBullet physical space registered objects should be detected and a player controlled car, that is bellow the fly camera, should collide with the cite models and the sky box walls. Students should carefully read any of the comments regarding Step 12 to understand what each line of code does.

Finalized this exercise the output., after some rotation and translation backwards of the camera, should be equal to the one illustrated in Figs. 9 and 10.

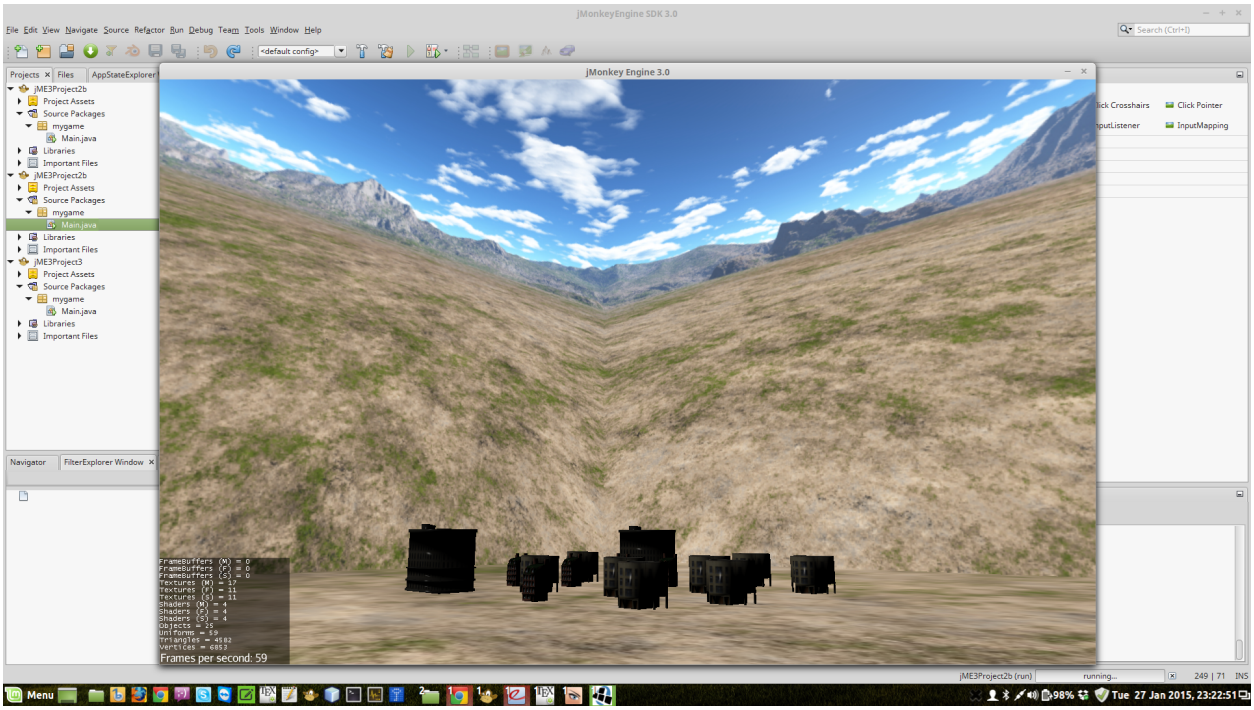

Figure 9: Exercise 12 looking forward from far away output.

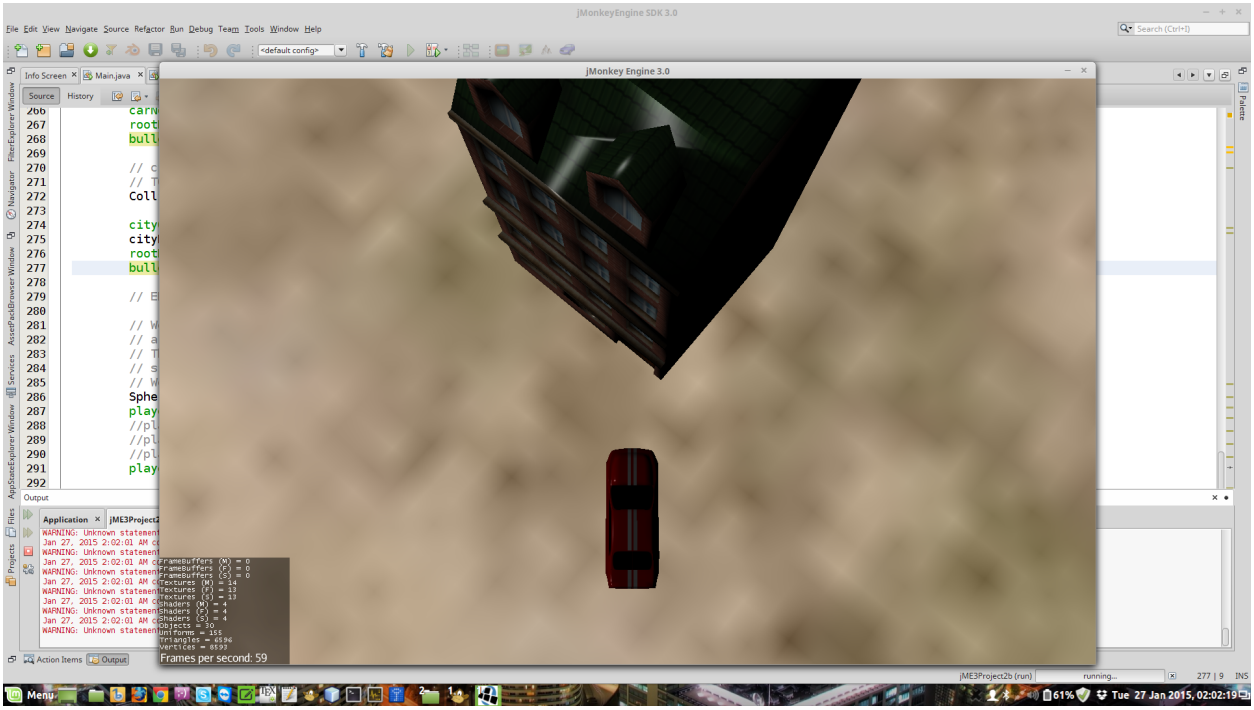

Figure 10: Exercise 12 looking down output.

References

[1] jMonkeyEngine Tutorials and Documentation `http://hub.jmonkeyengine.org/wiki/doku.php/jme3#installation`, last access on 04/04/2016.

[2] Wikipedia SkyBox (Video Games) `https://en.wikipedia.org/wiki/Skybox_%28video_games%29`, last access on 04/04/2016.

[3] Miguel Casillas collision detection tutorial `http://www.miguelcasillas.com/?mcportfolio=collision-detection-c`, last access on 04/04/2016.

[4] Wikipedia Gilbert-Johnson-Keerthi distance algorithm `https://en.wikipedia.org/wiki/Gilbert%E2%80%93Johnson%E2%80%93Keerthi_distance_algorithm`, last access on 04/04/2016.

[5] dyn4j physics engine documentation on GJK `http://www.dyn4j.org/?s=GJK`, last access on 04/04/2016.

[6] Entropy interactive documentation on GJK `http://entropyinteractive.com/2011/04/gjk-algorithm/`, last access on 04/04/2016.