CSC 122: Gravity Simulator

Advanced User Interface

1 Introduction

In this lab, you will create a gravity simulator. The simulator will respond to user touch events in order to create planets of varying size and exude limited control over their motion. You will accomplish this by designing a GUI layout, handling touch events initiated by the user, and rendering the planets. A custom GUI element will be provided to aid the touch event handling and animation rendering.

2 Objective

The purpose of this lab is to present the student with an advanced user interface experience including complete control of the GUI layout and interactive functionality. The student will learn how to handle touch events and render simple animations on an Android system.

3 Activity

This lab will be fairly freeform in execution due to the large control you have over the implementation. A theory section is provided to aid in your design of the project solution. It is recommended that you look over the theory section at least once before beginning to code. Details for the implementation can be found in their own section after Theory.

3.1 Theory

This section provides background information for various parts of the lab. Refer to this section as needed during your implementation.

3.1.1 Kinematics

First, it should be noted that while certain formulae are provided, you are in no way obligated to use them. However, there are certain restrictions upon your simulator: the gravitational force must be attractive and proportional to mass, and collisions must be modeled in some way, either as a bounce or merge. You may add any functionality you wish so long as these restrictions are not violated. For example, you could multiply the masses upon merging instead of summing. The

`Planetoid` class is provided with partial implementations of some of the required physics and functions.

Kinematics is a branch of mechanical physics that describes motion. In this case, the kinematics is mainly constrained to that induced by gravity. The gravitational force between two objects is given by

$$F = G\frac{m_1 m_2}{r^2}$$

where G is the gravitational constant, m_1 and m_2 are the masses of both objects, and r is the distance between them. F is a vector directed from one object's center of mass to the other's. For the purposes of this lab, G may be freely defined as any positive value.

Eventually, two objects attracted by gravity are going to hit eachother. One of two things (provided the objects do not break) will occur: an elastic or inelastic collision. An inelastic collision happens when the two objects stick together, and can be modelled by the following formula:

$$v_f = \frac{m_1 v_{1i} + m_2 v_{2i}}{m_1 + m_2}$$

where subscripts of 1 and 2 denote one object or the other, the v_is are the initial velocities pre-collision, and v_f is the velocity of the merged object. Each dimension of motion (x or y) can be solved for separately. Elastic collisions, however, are more complicated to adequately model, especially in the presence of 2 or 3 dimensional objects. Basically, the momenta are split along the point of collision with the momenta perpendicular to the contact surface exchanged between objects and the momenta tangent to the surface unaffected. The velocities perpendicular to the surface of contact between the objects follow the rules for a one-dimensional elastic collision:

$$v_{1f} = \frac{m_1 - m_2}{m_1 + m_2} v_{1i} + \frac{2m_2}{m_1 + m_2} v_{2i}$$

and

$$v_{2f} = \frac{2m_1}{m_1 + m_2} v_{1i} + \frac{m_2 - m_1}{m_1 + m_2} v_{2i}$$

These formulae may also be applied independently for each dimension for point-like particles. Once the particles gain form, however, the results do not appear right.

3.1.2 Touch Events

Touch events on Android can be handled in many different ways. The primary class used in their handling is the `MotionEvent` class. A brief description of it will be given here. A `MotionEvent` object holds data concerning a single instant of a user's touch, and may hold information for multiple touches. We will call each individual touch a pointer. Two pointers would indicate that the user has two fingers touching the screen. Each `MotionEvent` also comes with an `ACTION_MASK` to describe its state. The chief actions that we are interested in are `ACTION_DOWN`, `ACTION_MOVE`, and `ACTION_UP`. In order, they describe the beginning, middle, and end of a pointer, though an `ACTION_CANCEL` may interrupt. Though each pointer has its own lifecycle, only one `ACTION_DOWN` and `ACTION_UP` are generated for concurrent pointers. Therefore, after one pointer goes down, others may come and go without affecting the action. The number of pointers, however, can be directly queried with a call to

`MotionEvent.getPointerCount()`, and each pointer's position may be individually accessed. In order to track pointer velocities, a `VelocityTracker` object can be used to aggregate multiple `MotionEvents` and estimate pointer velocities individually.

3.1.3 Rendering Animation

We will be employing the `Canvas` class to render the `Planetoids`' motions. The functional aspects of the `Canvas` class are provided for you in the `GravCanvas` class. A `Canvas` is relatively easy to operate; whatever is drawn last at a given pixel is visible. You should not need to actually implement any of the animation. However, you may wish to change the visuals. A background image as well as images for selected and unselected planets are provided. You are free to replace these with your own (preferably in .png format with transparent backgrounds). For example, in order to replace the background, import the desired image into the `/res/drawable` folder in Eclipse. After accomplishing that, change the assignment of `background` in the `GravThread` constructor to `BitmapFactory.decodeResource(res, R.drawable.<your-image-filename>)`. The filenames you use must be lowercase. Another example of changing the graphics is provided in the `GravThread.draw()` function. Take care that your image files are not too large; Android applications do not have much available memory.

3.2 Implementation

This section will provide a general guide to getting your app running. Pieces of code that require your attention are prefaced with `STUDENT:`. Some of these may provide important information not mentioned here.

3.2.1 Research

If you have not encountered enumerated types or the `final` keyword before, look up their usage in Java. In addition, review their usage in the code provided, specifically inside of the class `GravCanvas` and its inner class `GravThread`.

3.2.2 Design the layout

You may use the XML Layout editor in Eclipse to aid you in designing the GUI. The layout must provide an ample amount of space for a `GravCanvas`, which extends `SurfaceView`. In addition, it must provide buttons or some other device to control pausing/playing of the simulator and locking/unlocking of the screen for touch events.

3.2.3 Complete the Activity

A skeleton activity (`GravitySimActivity`) is provided to hold references to the `GravCanvas`, its `GravThread`, and buttons in the layout described above. This activity will handle button clicks and their results as they pertain to the `GravCanvas`. Button labels should be kept up to date with their current function if any are multipurpose ("Play" or "Pause"). Override

`onDestroy()` such that it contains a call to `GravThread.setRunning(false)`.

3.2.4 Implement Kinematics

Decide which physical laws your program will simulate and give an initial value to `GravThread.GRAVITATIONAL_CONST`. Inside of class `Planetoid`, add and modify methods for the planet to react to stimuli according to your physical laws. Minimum recommended methods include force application and movement for given durations and set methods for momentum and area.

3.2.5 Model Gravitational Interaction

Tasks contained in this section should only occur if animation is set to play. Inside of the function `GravThread.gravitate()`, apply forces based upon the gravitational field each planet exerts upon the others. These forces should be applied for a certain duration equal to the time elapsed since the last time `gravitate()` was called. After all the forces have been applied to a planet, move it based upon one or more of the formulas given for motion under constant acceleration. If any planets collide (if they would overlap eachother when drawn), either apply an elastic (they bounce off eachother) or inelastic (they merge into one) collision to each. You are in control of which of the two types of collision takes effect. If a planet would move off of the screen, you may either let it continue or make it bounce off the edge at an appropriate angle (alter the direction of its velocity while retaining the magnitude). Planets that continue off the edge of the screen may very well return if the gravitational force is strong enough.

3.2.6 Handle Touch Events

Tasks contained in this section should only occur if the screen is unlocked. In `GravCanvas.onTouchEvent()`, handle touch events initiated by the user. You will need to interact with `GravCanvas.mThread` in order to affect the animation. If one of the motion pointers is positioned over empty space when it begins, create a `Planetoid` at that location and associate it with the pointer. Otherwise, associate the existing `Planetoid` at that location with the pointer. In addition, the `Planetoid` should be *selected* such that it obeys the following constraints: its position is the same as its associated motion pointer, its image is different from unselected `Planetoid`s, and its size and mass slowly increase over time. Note, however, that the size and mass should only increase if the `Planetoid` was newly generated, i.e. its associated motion pointer started over empty space. When a motion pointer ends, the `Planetoid` is deselected and continues along the trajectory set by the motion pointer's last estimated velocity. The speed of a `Planetoid` released this way should be inversely proportional to its mass. Flinging of existing `Planetoid`s may optionally be allowed on locked screens (but not `Planetoid` generation). Finally, take note that the selected image will be automatically displayed as long as a planetoid is selected with `Planetoid.select()`. A `HashMap` is a suitable choice for handling pointer-`Planetoid` associations.

3.2.7 Debug

Try running the app, performing both positive and negative tests. For example, do any `Planetoid`s stay selected after a motion event ends or simultaneous finger movements end? In addition, does it perform to your desire? You may wish to tweak the strength of flinging or your gravitational constant if the planets move too slowly or quickly for your liking. Experiment until you find a balance that you enjoy.

3.2.8 Optional Refinements

If you would like to save your progress (at least for as long as the app is running either on the screen or in the background), then you should do the following in your activity. Override `onPause()` and `onResume()`. Inside of `onPause()`, call `GravThread.quit()` with a `static Bundle`. Inside of `onResume()`, call `GravThread.restart()` with the same `Bundle`. `GravThread.restart()` should not be called if it is the first time for the activity to start. Look inside these functions to discover how your buttons should be labeled.

4 Conclusion

Upon completion of this lab, you should be capable of designing Android applications that can handle touch events. In addition, you should have gained a greater understanding of how to render images and animations.

5 Deliverables

Deliverables include a project export from Eclipse as well as the following requirements unless directly contradicted elsewhere. When running, the simulator should create, move, and grow planets according to user touch events only if the screen is unlocked. Planets should only move under their own influence when the simulator is playing. Pausing should not prevent touch events, and upon resuming behavior should be as though the pause never occurred.