Coursework commentary 2018–2019

CO1112 Creative computing I: image, sound and motion

Coursework assignment 2

Introduction

This coursework assignment explored the design and programming of interaction and motion in *Processing* sketches.

The overall performance on this coursework was very good, with 64 per cent of submissions being graded as *Excellent* or *Very Good* (that is, grades A and B), and a further 27 per cent graded *Good* (grade C). 5 per cent were graded as *Acceptable,* and only 4 per cent received a *Fail* grade. Those who failed generally did so because they had not completed every part of the coursework assignment.

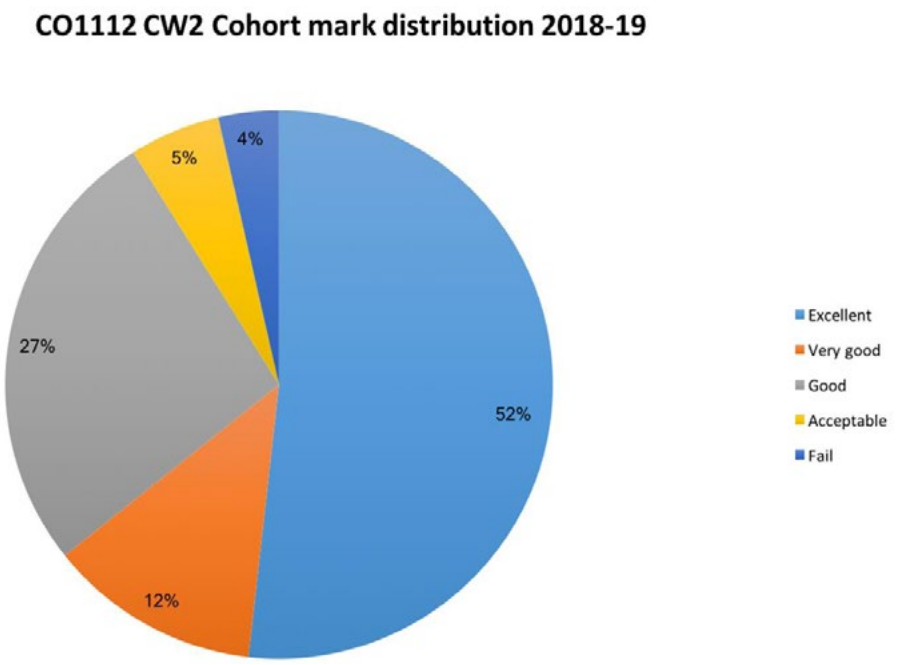
General remarks

In terms of the quality of the submitted code, although many students submitted very neat, well-designed and well-commented code, in some cases there was room for improvement in coding standards. In addition to a lack of useful comments, there were many examples of code with inconsistent indenting and spacing. People new to programming may regard the neatness of their code as quite inconsequential, but as you get more experienced with writing code – and, in particular, with *reading* code (possibly written by other people) – you realise that having consistent standards of code presentation greatly helps the reader in understanding how the code is structured. In *Processing* in particular, there is little excuse for producing poorly formatted code, because there is an *Auto Format* function in the *Edit* menu, which will tidy up your code for you.

Some students also had very long lines of code. These can be hard to read, especially when they require use of horizontal scrolling to read the whole line (this can be a particular problem when reading the code on laptop screens). In *Processing* line breaks can be used to run long statements over multiple lines. Do bear this in mind when writing your programs.

A small number of students were still using *Processing* 2.2.1. This is an old version of the system and using it is now strongly discouraged. All students should be using *Processing* 3 or higher.

See 2018–2019 cohort mark distribution for CO1112 CW2 below:



Comments on specific questions

Part A

Most students completed this part very well. There were essentially four conditions that needed to be checked to fully implement the collision detection, which could be summarised as follows:

```
if ((ballBottom > barTop) && (ballTop < barBottom) &&
   ((ballLeft < holeLeft) || (ballRight > holeRight))) {
   collision = true;
}
```

Each of the variables in the code shown above would need to be defined in the code, or else replaced with appropriate expressions. Note that it is much better to use variables with meaningful names in your code, rather than using hard-coded numbers. For example, the variable `ballLeft` in the code above represents the x coordinate of the left-hand edge of the ball. Most students represented this as "`ballX-(ballDiameter/2)`", which is also fairly clear. However, some students used "`ballX-15`" instead, but reading this code doesn't immediately tell you what the number 15 represents. Using variables instead of hard-coded numbers also makes the code more robust should you decide to change the values of the variables later on (e.g. if you decided to change the value of `ballDiameter` then code that referred to this variable would still work, whereas the "`ballX-15`" would no longer work correctly).

The most common mistake was to not check the condition "`ballTop < barBottom`", which meant that a collision could still be (incorrectly) detected even after the ball had dropped below the bar. It was surprising that most students who made this mistake still confidently asserted that their code was working correctly, indicating that their testing procedures were not very thorough.

A few students also lost marks by not including the paragraph about this part in their written PDF submission.

Part B

Many students provided good answers to this part, including some nice refactoring of the code to adopt an object-oriented design.

For those who did attempt the OOP approach, the most common weakness was that some students defined two separate classes for the two bars. This meant there was a great deal of repeated code in the two classes, and this is not a good approach to the design. Instead, a generic Bar class should be defined, which stores the position variables for a given bar. The code should then make two instances of that class, one for each of the required bars, and initialise each one with the correct position information.

Another weakness was that many students – even those who had defined a generic Bar class – were still doing the collision detection in the main code. A better design choice would be to move the collision method into the Bar class, and to pass in the ball position as a parameter to this method. In the `draw()` method, you could then just call the collision method for each bar, without having to duplicate the collision code.

Another questionable design decision seen in some submissions was the definition of a Hole class independent from the Bar class to represent the hole in the bar. Given that the hole is a feature of a bar, and its position (certainly its y co-ordinate) depends on the position of the bar, it makes sense that the position of the hole is stored as part of the Bar class rather than being defined independently.

A problem seen in a few submissions was that the Bar objects were being created within the `draw()` method. This means that new Bar objects were getting created every time `draw()` gets called (which is many times per second). As it happens, the Java garbage collection will destroy the old ones so you don't run out of memory, but this is still doing a lot of unnecessary extra work. It is nearly always the case that objects should be created in the setup() method rather than in `draw()`.

Some students went a bit too far with their OOP design, creating classes in cases where they were not needed. For example, some students created a class specifically to print instructions on the screen. The class did not contain any variables and just had a single method to print the instructions, so there would never be a case where you would want/need to create two instances of this class. In this case, a better design might have been to have an overall Game class, and just define a method within that class to print the instructions.

Some of the best solutions not only defined a set of appropriate classes, but also split their code into different files, with each class being defined in its own source file. This makes it easier to navigate between different parts of the code, and also makes it easier if you want to reuse one of your classes in another sketch.

Part C

There were some impressive submissions for this part of the coursework, many involving interesting extensions of the game with graphics, animation and sound. The best submissions included a clear description of the ideas and motivation behind their extensions, and the approach they took to implementing them. On the other hand, some weaker submissions failed to provide proper justification for their decisions (e.g. saying that images were added "to appeal to users" is a rather weak justification – a better discussion would explain why the particular images that you used were selected, and why you thought they would appeal to users).

Some students did not fully report everything they did (either in the written description requested for this part, or in on-screen instructions), leaving it in the hands of the marker to discover some of their game's functionality by studying their submitted code. The markers might not always spot all features of the game in this way, so be sure to fully explain what you have implemented in your submissions.

Some submissions suffered from obvious problems and bugs when they were run, and yet the student failed to discuss these in their written submission for this part. If some part of your code is not working correctly, you can gain marks by discussing the problem in your submission, including any ways you tried to solve it (even if unsuccessful).

Some students who added sound to their games did so using the Minim sound library. Please avoid using this library and use the official *Processing* Sound library instead. This is described in the 2019 edition of Volume 2 of the subject guide.

Many students did a good job at clearly stating the sources of any assets that they used (e.g. providing URLs for the images and sound files they had found on the web), but some did not. Remember that it is just as important to acknowledge the source of software assets that you use as it is to cite and reference written texts that you use in your essays.

Part D

Many students submitted good or excellent responses to this part, including a thoughtful discussion of what they had attempted to achieve in Part C, of what they perceived to be the best aspects of it, and which parts they were not so happy with.

The most common weakness for this part was simply not providing enough discussion. The coursework suggested around 500 words for this part, but some responses were much shorter.