COMP2006, 2021/22, Coursework Requirements: GUI Framework

Contents

Overview

This coursework has two parts – an initial 'easier' part to get you working on this earlier, and the final part which is more complex and lets you innovate more. Your part 2 coursework can be completely different and separate to your part 1 – you do not need to build on the part 1 submission to make part 2, although doing so may save some work.

Part 1 aims to introduce you to a C++ framework for building GUI programs. You should be able to get full marks on that part if you work through the demos and spend time doing the coursework. **Part 2** aims to give you the freedom to do more with the framework and potentially create something impressive. The aim is that your total mark for part 1 and part 2 should be representative of your ability with C++ (or the effort you put into the coursework).

The coursework framework is a simplified cut-down framework covering just the essentials and aims to simplify the process of software development for you. It may seem large to you but in fact is very small compared with the sizes and complexities of class libraries you would usually be working with in C++. You will need to understand some C++ and OO concepts to be able to complete the coursework, but will (deliberately) not need to understand many complex features initially, so that the coursework can be released earlier in the semester.

There are two demo tutorials (labelled Demo A and Demo B) which you should work through to help you to understand the basics. Please complete these to understand how to do part 1. There are also some examples in the coursework code provides, to illustrate how you can do some things. Your submissions must be different enough from the demo tutorials and supplied examples that it is clear to the markers that you actually understand the concepts rather than just copying them.

Part 1 will be marked via a demo in lab times (via MS teams screen-sharing) . It is designed to be fast to mark this way and you will know your mark immediately. (You should be able to check the marking criteria in advance and know what mark to expect anyway – hopefully full marks for each of you.) A zip file of your project (including all source, project and resource files) MUST be uploaded to moodle before the submission deadline in order for your mark to remain valid **– failing to upload your project will count as a failed submission and result in a mark of 0**. As long as you submit by the deadline and demo your work by the deadline, you may demo your work before submission or after submission, whichever is easier for you - but you do need to both demo it AND submit the files. Failing to demo your project will also result in a mark of 0. Except when extenuating circumstances apply, no late submissions are permitted (because of the need to demo your work), however early marking and submissions are encouraged – since you should be able to get full marks anyway.

Part 2 is more advanced and is designed to be harder to complete. Part 2 is designed to be hard in some parts! You may not be able to do all parts. Please set yourself a time limit of 30 hours after completing coursework part 1 (i.e. 30 hours for a 30% coursework), *do what you can and then stop* (making sure it compiles and runs etc). DO NOT SPEND TOO LONG ON CW part 2!

Part 2 requires a submission to moodle by a deadline and is then marked in specified demo times (via MS teams screen-sharing). Submission of documentation and code (and any extra demo videos – see the criteria) will be via moodle. This allows more time for marking while ensuring that everyone has the same amount of time to complete the coursework (this part may need more thought and/or time).

You may complete and demo the coursework on Windows, Linux or Mac. As long as you can submit your code and do us a live demo via MS teams we don't mind what operating system you use.

Getting started:

1. **Download the zip file of the coursework framework. Unzip it and open it in Visual Studio** – on your own computer or the windows virtual desktop. Compile and run the initial framework to ensure that it compiles and runs with no problems on your computer. Read the getting started document if you are stuck. You can also use the lab PCs or the windows virtual desktop.

   (Mac,Linux) Optionally: If you prefer then you can follow the instructions provided on moodle to build on Mac or Linux. We are assuming that, if you wish to do this, it is because you are the expert on Mac or Linux so will be able to resolve any issues. This should ensure that those who are more comfortable on a different operating system are able to use it, but please be aware that we may not be able to support issues on these platforms.

   Whatever you run this on, you will need to be able to screen share on teams to demonstrate your work for marking as well as to submit your source code files to moodle. (It should be possible for you to demo on any platform you wish and/or demo on the windows virtual desktop if necessary by copying files back into the visual studio project.)

2. **Do the coursework tutorial demo A** and ensure that you understand about the basic features of the BaseEngine class and drawing the background, including the tile manager and handling input.

3. **Do the coursework tutorial B** and ensure that you understand about the basic features of simple displayable objects, moving them around and displaying them.

4. **Skim through the Frequently Asked Questions document to ensure that you know what is in it.** This document has developed over the years to address questions which I often get asked. It will also give you clues to things you may not know of may not think of.

5. **Start on the requirements for part 1**, using what you learned from demo tutorials A and B. Coursework part 1 should not be too bad using the walkthroughs for demo A and demo B and the samples. Try changing which object type is created in the mainfunction.cpp file (change which line is commented out to change which demo to run) and look at the provided demos.
   You may want to consider the 'SimpleDemo' demo, for things like string displaying, and potentially either the BouncingBall or Maze demos if you can't work out things like the TileManager. Work on only the simple demos initially! You will not need the more advanced demos until you get to part 2.

   You can do something completely different for part 1 and 2, so just put something together for part 1 marking, and don't worry about part 2 initially.

6. **Once you have had part 1 marked, look at the part 2 requirements and decide on how you will develop a program to be able to do as many of these as you wish, considering what you learned from part 1.** You should think about which requirements you will attempt at the start rather than trying to consider this later because some programs will make them more complex to achieve than others. Note: in some cases you may find it impractical to integrate some requirements into your main program. In that case, don't forget that you could use a fancy 'intro' or 'ending' screen (state) to demo things that may not fit into your main program (e.g. TileManager in past years has not always fitted for some students, and has been used on an intro screen). Think outside the box and

remember that your aim is to get the best mark, not be exact in matching some existing program idea or to make the best game ever.

General requirements (apply to both parts):

You MUST meet the following general requirements for both part A and part B:

1. **All of the code that you submit must be either your own work or copied from the supplied demos.** Where you use code from demos it will not get your marks unless you have modified it enough to show your own understanding.
   You may not use work from anyone else.
   You may NOT work with other people to develop your coursework.
   You may NOT have someone else write any or all of the code in your submission.
   You will be required to submit a simple documentation sheet making clear that you understand this.
2. **Create a program which runs without crashing or errors and meets the functional requirements** listed below and on the following page. Look at the Hall of Fame page to see some examples of previous work and decide on a basic idea for your program. You don't need to implement that much for part 1, but part 2 will involve finishing it.
3. Avoid compilation warnings in your code. Usually these mean that there is a potential issue with your program. Try to avoid compilation warnings as any warnings may make us investigate whether they mean that there are C++ issues with your program.
4. **Your program features which are assessed must be different from all of the demos and from exercises A and B.** i.e. do not copy demo code or the lab exercises code too closely. You will not get marks for copying code, but would get some marks for your own modifications which show your understanding of C++ if you did *similar* things to the demos. In other words, change the code at least enough to show that you understand what it is doing and haven't just blindly copied it.
5. **You must not alter existing files in the framework** without prior agreement with from the module convenor (which would require a good reason) – the aim of the coursework is to work with an existing framework not to improve it (even though there are many possible improvements).
   Note: subclassing existing classes and changing behaviour in that way is permitted, and is encouraged, as you don't need to change the existing files to do that.
   I tried to make the classes flexible enough to allow behaviour to be modifiable by subclasses relatively easily.
6. The aim of this coursework is the investigate your ability to understand existing C++ code and to extend it using sub-classing, rather than to use some alternative libraries:
   - **Your program MUST use the supplied framework in the way in which it is intended**. i.e. the way in which demo tutorials A and B use it to draw the background and moving objects is the way that you should use it.
   - **You must not alter the supplied framework files.** See comments above for point 5.
   - **You should not use any external libraries other than standard C++ class libraries**, and libraries which are already used by the framework code, unless a requirement specifically says you may do so. There any many useful libraries which you could use to improve your programs, but the point of this exercise is to show that you can understand and use the library I supply.
7. **There are 10 functional requirements in part 1**, worth 1% of the module mark each, for a total of 10% of the module mark for this part of the coursework.
   When you have completed the coursework part 1, demo your work to a lab helper in a lab (ask for

part 1 marking) and have it marked. (The marker will upload your mark to our system and tell you which marks you got and didn't get.) **Also submit a zip of your source code to moodle:**

- **Clean your program** (use the 'Clean Solution' option under the build menu – to delete the temporary files.
- **Delete the hidden .vs folder in the root of your project directory**. This can become huge but is not needed, and will make your zip file unnecessarily big.
- **Zip up the rest of your project directory** and submit it to 'part 1 coursework submission' on Moodle.
- **Your submission should include all of the supplied files** (including demo files) as well as your new files in the zip file you submit.
- **Also include your completed documentation sheet(s) in your submission – as a separate file outside of the zip.**
- If you worked on Mac or Linux please include your make files etc that you used, ideally in a format which we can use to build and test the code ourselves, along with summary instructions of how we can build and run the program ourselves if necessary.

It should be possible when you have done this for someone to download the zip file, unzip it, do a 'rebuild all' to rebuild it, and run it, with no other changes. (Or for us to build/run it on Linux/Mac if appropriate – in which case please provide simple instructions.)

Importantly, it should also be possible for us to check your documentation file to see whether the mark matched what you expected or where any issues that you knew about were.

8. **There are 30 marks available for part 2**, worth 1% of the module mark for each mark, for a total of 30% of the module mark for that part of the coursework. These are usually a lot harder than the part 1 requirements.
Start with the easy requirements! Limit your time on part 2 to a maximum of 30 hours!
Read the requirements carefully!
When you have completed part 2 upload the project file to the CW2 submission on moodle, as you did for CW1 (i.e. clean it, delete the .vs directory and zip the folder).
A demo will be arranged where everyone will give a short live demo showing their program running and answering questions.
You will need to upload a documentation file before the demo explaining which features you achieved.
For some marks you will also need to provide a justification/explanation in the documentation file, and potentially even submit a video of the running program, so that moderation can determine whether this really is one of the best courseworks in the year (in terms of complexity of task or of impact).

Some clarifications (based on questions in previous years):

The assessment aim is to **test your ability to understand C++ code and to write it** – not to copy paste. Here are some examples of things which are and are not allowed as far as marking is concerned:

- If you take a demo and just add some changes then you are assessed only on your changes.
  i.e. if maze demo already did something (e.g. background drawing, pause facility, etc) then it doesn't count for a mark for you because you did not write it yourself.
  Similarly, it does not count as you creating a new subclass because you didn't.
  Please start from a new class and add your own code to it.
- If instead you write your own code, and take from the demos the *understanding* of how they do it – possibly even copying *some small parts* of the code, *but showing your own understanding of it when you do*, then that is OK. (This is partly why we ask you to demo it – so you can answer questions if necessary.) You should ensure that you have enough understanding of each feature that you can explain it when we ask you about it in the demo.
- As another example, if you copy whole functions, such as a draw function, from a demo, that is also a case of something being the same as maze demo. If you do so and make a slight change to make it a different colour then it's basically the same apart from that slight change, so it is not different enough. Don't do that please as you won't get the marks.
- However, you CAN copy *small parts* into your own classes, showing your understanding of them. In which case you should be able to explain fully how they work if asked during the marking – that is an important reason that we ask you to demo your work, so that we can test your understanding.

General marking criteria for both part 1 (CW1) and part 2 (CW2):

You will lose marks if any of the following apply. In each case we will apply a percentage mark penalty to your mark.

- Your program crashes on exit or has a memory leak. (Lose 10% of your mark.)
- Your program crashes at least once during its operation. (Lose 20% of your mark.)
- Your program crashes multiple times. (Lose 30% of your mark.)
- Your program crashes frequently. (Lose 40% of your mark.)
- Your program has some odd/unexpected behaviour/errors. (Lose at least 10% of your mark.)
- Your program has a lot of unexpected behaviour/errors. (Lose at least 20% of your mark.)
- Your program crashes on exit or has a clear memory leak. (Lose at least 10% of your mark.)

The mark sheet that will be used for marking for parts 1 and 2 has entries for the above and your marker will annotate it according to their experience of your demo of your software.

Coursework Part 1, Functional Requirements

Functional requirements: (1 mark each, marker will tick off all that you have done on their mark sheet).

Note: the markers will use the **literal text below** (**both the bold and unbolded text**) to determine whether a requirement was met or not. I have tried to split them into bullet points this year to try to make it harder to accidentally miss one of the criteria. Please read the criteria carefully to ensure that you don't miss anything.

Please check all requirements, because you may find an early one more difficult than a later one. You should be able to do all of the part 1 requirements though with enough time and thought.

Please be prepared to explain in the demo how you implemented any of these features, as this is an important way for us to understand that you did the coding yourself.

1. **Create an appropriate sub-class of BaseEngine with an appropriate background which is different from the demos.**
    a. Create an appropriate new sub-class of the base engine.
    b. Give it an appropriate background on the screen
    c. Ensure that the background is different to the demos
    d. Name your class using your capitalised username followed by the text Engine. e.g. if your username was psxabc then your class would be called PsxabcEngine.
    e. You MUST create a new class for this – you must not just change/rename one of the existing demo classes. The demo classes should all exist in your submitted zip file so that we can see you created a new one.

2. **Show your ability to use the drawing functions:**
    a. Draw your background ensuring that you use at **least one** of the **shape drawing** functions to draw on the background
    b. and that you draw at **least one image to the background**,
    c. which is different from the demos and shows your understanding.

    A blank background will not get this mark – even if you change the colour – you MUST use one of the shape drawing functions (i.e. a drawBackground…() function other than drawBackgroundPixel()) and at least one image, to show your understanding of these.

3. **Draw some text on the background.**
    a. Draw some text onto the background (not foreground)
    b. and ensure that the text is not destroyed removed when moving objects move over it – i.e. it reappears when the moving object has moved, and it is visible in the background of the moving object, in any gaps or around the moving object.
    c. Ensure that moving objects can/do move **over** at least part of this text, so that the object appears in front of the text, and demonstrate that it is redrawn correctly after the object has moved.

4. **Have some changing text, refreshing/redrawing appropriately which is drawn to the foreground (not background), in front of moving objects.**
    a. This text may change over time (e.g. showing the current time, or a counter) or could show a score which changes, for example.

b. It could also be drawn to the foreground as a part of an object (e.g. a moving label) if you wish, but does not need to move around with objects if you don't want it to.
c. When the text changes, the old text should be appropriately removed from the screen.
d. Be prepared to explain how this works to the marker if asked. This shows your understanding of drawing text to the foreground.
e. The text has to be drawn such that **moving objects would move under it** rather than on top of it though. i.e. **not to the background**, and basically it means it'll be drawn after at least some of the objects.
f. For marking we will check the code where it is drawn if there is any doubt. E.g. whether the function which draws it is called before or after drawing objects. (Look at the different functions to see which to use – the point of this mark is to see whether you realised the difference between drawing changing text to foreground rather than background.

5. **Provide a user controlled moving object which is a sub-class of DisplayableObject and different to the demos:**
    a. Have a moving object
    b. that the user can move around,
    c. using either the keyboard OR the mouse (or both)
    d. and is different to the demos.

    Note: you could have an indirect subclass of DisplayableObject if you wish (i.e. a subclass of a subclass of DisplayableObject). The aim of this requirement is for you to show that you understand how to use EITHER the keyboard or mouse input functions (or both) as well as to show that you can create a moving object.

6. **Ensure that both keyboard and mouse input are handled in some way and do something.** *At least* one of these input methods should influence one or more moving objects, from requirement 5. The starting point is probably to look at whether you used mouse or keyboard for the moving object above and just use the other one here. E.g. if you user-controlled object is mouse controlled then make it so that something happens when a key is pressed (e.g. a counter is incremented, which appears on the screen – see requirement 4). Or if your object is keyboard controlled, you need to handle mouse movement or button pressing. Both mouse and keyboard could influence the moving object if you prefer, or one could change something else.
    a. You handle **both** keyboard input AND mouse input and they both do something
    b. **Something(s) should visibly change for both** – e.g. some position of something or visible value of something or displayed image, etc

7. **Provide an automated moving object which is a sub-class of DisplayableObject and different from the one in requirement 5.**
    a. Provide a second moving object (separate to the user-controlled one, with a **different class**)
    b. whose movement is not directly controlled by the player,
    c. which moves around,
    d. which acts differently to the objects in the samples/demos/code that I provided,
    e. and which looks different to the objects in the demos and to your object in requirement 5.

    i.e. show that you understand something about how to make an object move on its own, without user input (changing its x and y coordinates and redrawing it appropriately). Be prepared to explain how this works to the marker if asked. Your object must have some behaviour that is different to the demos (i.e. a copy-paste of the demo code is not sufficient) and you must be able to explain how it works and justify that it is different from the demos in some way. Your object must also have a different appearance to the object in requirement 5, and look different to the demos/samples as well.

8. **Include collision detection between a user-controlled (req. 5) and an automated (req. 7) moving object, so that they interact with each other.**
    a. You need to check for a collision between the two objects.
    b. Something should happen when they collide, and something should visibly change – e.g. something moves, direction of travel changes, or something is shown.
    c. Collision detection should be at least as good as rectangle-rectangle interaction and should work properly.

   This means that at least two of your objects should react to each other when they collide.
   Hint: look at UtilCollisionDetection.h rather than doing the maths for rectangle or circle intersection yourself – and see MazeDemoObject.cpp for an example of using these functions.
   Assessment of whether you achieved this or not will be on the basis that the intersection of two objects is correctly assessed and something happens in reaction to this (e.g. objects move, change direction, something else changes (e.g. a score) etc).
   Rectangle-rectangle or circle-circle interaction is fine for meeting this requirement.

9. **Create your own subclass of TileManager**.
    a. Create a subclass of the tile manager class which has different behaviour (at least a little) to the demos, is drawn to the background, and is visible to the user when the program is run.
    b. Name your class using your username followed by the text TileManager. e.g. if your username was psxabc then your class would be called PsxabcTileManager.
    c. Be prepared to explain the difference(s) from the demo versions to the marker and be prepared to explain how this works to the marker if asked: your understanding is important.
    d. Use a different tile size and number of rows and columns to the demos (e.g. 13x13 tiles and 6 rows and 9 columns if you can't think of some numbers yourself).

    Hint: Look at the existing demos, including the bouncing ball demo. Display the tile manager on the background, so that the tiles are visible. It must be different from the demos but can still be simple. Your TileManager must not be a copy of an existing one, or just an implementation of the demo tutorial example without change. i.e. you must show some understanding of how to do this, not just blindly repeat the steps of demo tutorial A.

10. **Have at least one moving object interact correctly with the tile manager, changing a tile:**
    a. Ensure that at least one of your moving objects visibly changes specific tiles *when it passes over them* – using the position checking appropriately.
    b. The tile must be changed and redrawn correctly so that the user sees the change.

    Consider the bouncing ball demo if you can't work out how to do this from the information in demos A and B, but you need to have at least some difference in behaviour from that.
    Assessment will check that you correctly detected the specific tile that the moving object was over and handled it appropriately.

Coursework part 2 functional requirements

Again, key parts of requirements have been highlighted with letters to try to ensure that you don't miss something – you need to meet ALL of the lettered sub-requirements to get the mark. It is all or nothing.

Note: 11 marks are labelled advanced. Without these you can get 29/40 on the coursework overall, which is the equivalent of a first class mark. 28/40 is 70% (first class) on the coursework.

The following requirements apply, and have a variety of marks available:

~~Handling of program states (total max 3 marks, 1 is advanced)~~

1. **Add states to your program (max 3 marks).** This means that your program correctly changes its behaviour depending upon what state it is in. Each stage should have a correctly drawn background which is different to the demos, and are **NOT** trivial (e.g. a blank background). This could use an image, a tile manager or be drawn using the fundamental drawing functions on the base engine. You can get a variety of marks for this:

   1 mark: You provide at least:
   a. a start-up state, a pause state and a running state,
   b. which differ in some way in **both** the appearance and behaviour.

   2 marks: As for one mark, but ALSO provide:
   c. at least five states
   d. including at least one win or lose state as well as those mentioned above. (Note: if you are not doing a game, then have a state which allows a reset, e.g. 'load new document' in a text editor.)
   e. There must be significant differences between the states in behaviour and/or appearance. The program must be able to correctly go back from the win/lost (or reset) state to the starting state to start a new game/document, correctly initialising any data.

   3 marks (advanced): As for the 2 marks but you must also:
   f. implement the state model (design pattern) using subtype polymorphism rather than if/switch statements in each function. Note: You may still need some if statements etc to enter the states but the current state object is used to determine how to act in each function.

   Look up the 'State Pattern' to see what this means and think about it. This is an advanced mark so we will not explain how to do this beyond the following: there will be a basic state base class and a subclass for each of the different states. BaseEngine is NOT your state class. Your BaseEngine sub-class will need to know which state object is currently valid and the different methods will call virtual methods on this object. The different behaviour will therefore occur due to having a different object being used for each state rather than having a switch in each of the methods. If you have if or switch statements specifying what to do in different states then you won't have done in properly so you won't get this mark. You should NOT have more than the one sub-class of BaseEngine if you do it correctly. If you had to create multiple sub-classes of BaseEngine then your implementation is wrong (and there will be other issues since you may have more than one window as well).

~~Input/output features (total max 3 marks, 1 is advanced)~~

2. **Save and load some non-trivial data (max 3 marks).** You can use either C++ classes or the C functions to do this – your choice will not affect your mark.
   You can get a variety of marks for this:

1 mark: most basic saving and loading, as follows:

a. save AND load at least one value to/from a file, e.g. a high score to a file (e.g. a single number to a file)

2 marks: completed the saving/loading above, and also **load some kind of more complex, structured data**, e.g. map data for levels, or formatted text documents where the formatting will be interpreted. The main criteria are:

b. It must be multiple read/write operations, of multiple values with some kind of structure to the file, rather than just a string of 3 numbers, for example.
c. Something must visibly change depending upon what is loaded (e.g. set tiles according to data read and/or set positions of moving objects according to the data).

3 marks (advanced): completed the above but also save/load all of the non-trivial state of the program to a file.

d. A user should be able to save where they are (e.g. the current document that they are working on for a text editor, or the state of a game for a game – saving **all** relevant variables for **all** objects)
e. and they should be able to **reload this state later**, with everything acting correctly and continuing from where it was.
f. Note: this means saving/reloading the positions/states of all moving objects as well as anything like changeable tiles, etc.

Note that point e means that you will need to provide some way to reload from the state as well as save it, but doesn't say when this is. e.g. it could have an option to load when the program starts (e.g. choosing between loading a saved game or starting a new game) or in response to some command from the user (e.g. pressing S for save and L for load).

This is meant to be non-trivial and may need some thought and debugging to make it work properly.

Displayable object features (total max 7 marks, 2 are advanced)

~~3.~~ **Use appropriate sub-classing with automated objects (max 1 mark):** To get this mark you must be using:

a. multiple displayable objects
b. from at least *three* different displayable object classes,
c. with different appearances and behaviour from each other
d. and you should have *an intermediate class* of your own between the framework class and at least two of your end classes,
e. the intermediate class should add some non-trivial behaviour.

i.e. you are showing that you can create a subclass which adds some behaviour, and some other subclasses of that class which add more behaviour.

Example: a complex example can be found in ExampleDragableObjects.h: DragableObject and DragableImageObject are both DragableBaseClassObjects, which is a DisplayableObject, so DragableBaseClassObjects is an intermediate class which adds some non-trivial behaviour.

~~4.~~ **Create and destroy displayable objects during operation of a state (max 2 mark):** for example, add an object which appears and moves for a while for the player to chase if a certain event happens, a bomb that can be dropped by a player, or a bullet that can be fired. Note that something like pressing a key to drop a bomb which then blows up later, while displaying a countdown on the bomb, and then changes the tiles in a tile manager would meet a number of requirements (i.e. marks) in one feature, so please also consider the other requirements at the same time.

1 mark: Objects appear to be created or destroyed over time, which means you:

a. dynamically add one or more displayable objects to the program temporarily after a state has started (i.e. you add to the object list *or* make some object visible).
b. and ensure that the object cannot interact with anything else before it is added or after it is 'destroyed'. E.g. if you make the object invisible then it is not collision detecting with anything while it is not visible.
c. You must not just re-create the entire object array contents to do this, although you could add to the end of it. (Please see the methods of DisplayableObjectContainer.)
d. This requirement is not met by the re-creation of objects when you change states. The idea behind this requirement is that moving objects appear and disappear while using the program within a single state.

For the 1 mark criterion you can hide/show an object rather than actually creating/deleting them. Remember if you do though that you may also need to ensure that their own and other DoUpdate() functions ignore the objects for things like collision detection or movement. e.g. just because you don't draw an enemy on the screen, the collision detection may still check for it if you don't stop it doing so. If this is unclear when marking, we may ask you to show us in your code how you avoid this problem.

2 marks: as for 1 mark but also correctly create and destroy displayable objects during operation**.** This means meeting the requirements above but also:

e. when making the object appear during the running of one state, actually create the object and add it to the DisplayableObjectContainer (rather than just making it visible),
f. when making the object disappear, within the running of the state, remove it from the object list in the DisplayableObjectContainer.
g. notify the program when you change the contents of the DisplayableObjectContainer. If you change the object array you need to tell anything using it that you did so, hence why this is in as a separate requirement – it can be trickier to get right. Please see the drawableObjectsChanged() method and investigate how it works to get this right.
h. correctly delete the object at the time that it is removed (not at some later point, such as when the state changes). Making sure that you destroy/delete the object at the right time is important for this – it is trying to get you to understand how to get objects to destroy themselves safely without causing issues with using the 'this' pointer after the object was destroyed. This is an important thing to understand in C++.

5. **Complex intelligence on an automated moving object (max 4 marks):** this is harder criterion to assess since there are so many different ways you could do this. As a minimum this criterion would involve something more than moving randomly or homing in on a player, but you should consider the marking criteria below to work out what sort of mark you will get.

1 mark: this could involve something like 'if player is on the same column then home in, otherwise move randomly', or 'keep the same direction until I bump into something then change direction towards player and repeat'.
Simple equations and decisions would get this mark rather than a higher mark.
Note that this has to be more than a single constant behaviour – e.g. more than just moving randomly or homing in on a target behaviour: the behaviour has to change somehow according to the situation.

An example of this is where your logic can be formulated as **"if <condition> then do behaviour type 1, else do behaviour type 2"** where behaviour types 1 and 2 are things which themselves involve a decision.
e.g.:

- "if player is within 200 pixels, go towards player, else move randomly"
- "if in this area of the screen, move using these rules, else use these rules…"

2 marks: this means a good implementation of the intelligence of a moving object.
There should be some level of calculation or multiple-decision-making element involved.
This needs to be more complex than the one mark criteria (i.e. it could not be expressed in the way that the one-mark criteria could), and could involve some more complex calculations or a series of decisions.
E.g.:

- calculating how to cut off a player, or intercept them, assuming that they maintain the same speed (predict where they will be and plan a path to get there)
- Showing some apparent intelligence which is not obvious to the markers how to implement, and not just random.

Note that the important thing for marking here is the skill you **show in your C++ ability** to write algorithms by implementing this – so something trivial will not count.

3 marks (advanced, see * below): this means a **good implementation** of some **more complex algorithm** for the intelligence of a moving object, e.g.:

- use a shortest path algorithm to find the shortest way through a maze to get to a player,
- tracking a player's decisions and predicting a player's path (in some non-trivial way, such as understanding what player decisions have been so far and what they may imply for later behaviour) and moving towards that rather than the player itself
- showing some apparent intelligence, so that the markers are impressed at how intelligent the object(s) appear to be.

  Note that the important thing for marking here is the skill you show in your C++ ability by implementing this – so something trivial will not count.

4 marks (advanced, see * below): this is an even more advanced version of the 3 mark criteria. This means:

- An **exceptional** implementation
- where the marker **is impressed with the complexity of the problem** you are solving,
- and the elegance of the C++ code that you are using to implement it.

Note that this talks about the complexity of the problem, rather than an overly-complex algorithm. This is deliberately not easy to get and is designed to allow the most capable students to excel. For the most capable students, doing things like this will not take as long as many of us may think. For the rest of us, it is probably too time-consuming to be worth the time to do it.
Please don't be upset if you don't get this mark.
One key criterion to consider for the 4 mark criteria is:

- "does this show you to be one of the best C++ programmers in the year?"

**** For 3 or 4 marks you need to include in your submitted documentation a clear description of what you did and how you did it, including any screen shots or diagrams, and also upload a video of up to three minutes (preferably shorter) demonstrating it working.***
Important: the person who marked your demo will not be the moderator, so it must be possible to work out from the video and documentation whether you deserve 2, 3 or 4 marks. If the marker agrees that your implementation is worth at least 2 marks then on the day of marking you will be told that you got two marks and the other 2 are the subject of moderation.

~~Collision detection (2 marks, 1 is advanced)~~

~~6.~~ **Non-trivial pixel-perfect collision detection between objects (max 2 marks):** this means to implement collision detection beyond the collision detection that I gave you examples of. These marks are for implementing some really complex collision detection, such as complex outline interactions (e.g. someone did bitmap-bitmap interaction in the past, checking for coloured pixels interacting, and someone else split complex shapes into triangles which could be collision detected and checked every triangle interaction) then you get this mark. Using the supplied collision detection class is not sufficient to get either mark. Collision detection for rectangles and/or ovals is not sufficient.

1 mark: improved collision detection which will work on more complex shapes than the supplied collision detection methods, such as polygons, and which works well in your program. Particularly that should be shown to work with convex shapes, but may not with concave shapes. E.g.:

- Putting a more convex polygon around the shape and accurately detecting collisions, or having collision detections between a range of polygonal shapes (triangles, squares, pentagons, etc), which goes beyond the square-square or circle-circle collision detection. As implied, this is likely to be most appropriate if your shapes are already polygons.

2 marks (advanced): pixel-perfect collision detection on an **arbitrarily complex irregular shape**.
Importantly, this should include dealing with concave shapes, so that one could be within a concavity of another but not count as a collision. (Think of a small circle being able to move inside a C shape, and as long as it doesn't touch the edges of the C then it is not a collision.)
Note that this is hard to get, so if your implementation is not solving a really complex task and/or was trivial to code then you probably will not get this mark.
Your method should work with concavities in object outline and your demo should show this.
One examples would be pixel-perfect checking by comparing bitmaps (if a coloured pixel in one image is in the same place as a coloured pixel in another image).
Although it may not seem like it, the easiest way to get a good implementation of this requirement is to use images and to check every pixel in the overlap area to see whether the relevant position in both images is coloured. (If either is not coloured then there is no collision at that point.) If there is a collision for ANY pixel in the overlap area then the objects have collided.
In particular, for 2 marks:

- Your approach should work even if the shapes of the objects colliding are modified (potentially with some minor changes to account for shape types or colours). i.e. it should use the object itself, not some hard-coded representation of the object, for instance of coordinates of the edges.
- Your method should work with concavities in objects outlines and your demo should show this. (Note: checking for pixel collisions in images will do this automatically.)

Animations, foreground scrolling and zooming (Max total 7 marks, 1 is advanced)

~~7.~~ **Implement a scrolling background by manipulating the way that the background image is drawn (1 mark).** This requirement basically requires you to look and understand the StarfieldDemo example, which manipulates the position at which the background is drawn.
You do not need to have a constantly scrolling background – it could scroll in response to some activity, such as the player moving for example.
You may want to implement this for a startup screen, high score screen or ending screen if it does not really fit with your main screen display, or to think about how it could be incorporated into your program.

Using a background surface which is bigger than the displayable screen could give the illusion of having a view onto a larger background image, and moving around it.
The main requirements are:

a. Setup an appropriate background image to use (which may be bigger than the window).
b. Modify copyAllBackgroundBuffer() appropriately to work with your background and ensure that it is redrawn as needed (redrawDisplay()).
c. Demonstrate an apparent smooth scrolling of the background in the demo.
d. Be able to answer questions about how you did it, showing your understanding.

The aim is for you to work out for yourself by looking at the example code, so we will NOT explain to you how the demo code works – please don't ask.

~~8.~~ **Have an animated or changing background by utilising multiple images (1 mark).** Having multiple drawing surfaces is an important concept which is widely used, and is easy to do using pointers. This requirement basically asks you to consider and understand the FlashingDemo sample. You need to:

a. Have at least five drawing surfaces, set up with at least slightly different contents
b. switch the appropriate surface in to be used as the background (noting that m_pBackgroundSurface is accessible to subclasses and can be changed, deliberately for this purpose).
c. Make it appear as if the background has been animated (at least slightly) using this process. E.g. students in the past made torches flicker, things change shape, etc. The changes should look smooth rather than the (deliberate) big changes used in the demo.
d. Be able to answer questions about how you did it, showing your understanding.

The aim is for you to work out for yourself by looking at the example code, so we will NOT explain to you how the code works – please don't ask.

~~9.~~ **Correctly implement scrolling and zooming of the foreground, allowing the user to scroll around using keys and/or mouse (max 2 marks).** Hint: look at the FilterPoints class to see how this can be done relatively easily, and consider the ScrollingAndZooming demo. You do NOT need to scroll or zoom the background (although you could integrate this with requirements 7 and 8 if you wished to do so).
One aim for these marks is for you to work out for yourself how to do this, or to work out from looking at the code which uses FilterPoints, so we will NOT explain to you how the existing code works – please don't ask.

1 mark: you wrote C++ code which works to:

a. allow a user to scroll the screen
b. using keys or mouse.
c. The user must be able to move the apparent view on the foreground objects up, down, left AND right (you can choose which keys/mouse operations do this).
d. Note: there is no requirement to actually use the FilterPoints class to do this for the one-mark criterion, although it will be simple to do if you already plan to do the 2 mark criteria. So, in theory you could manipulate the positions of all of the objects manually if you wanted to, as long as it works properly, if you can't understand FilterPoints.

2 marks: you achieved the above but also:

e. implement **both** scrolling and zooming of foreground objects

f. using the supplied FilterPoints class by creating a FilterPoints subclass which works to allow a user to zoom in and out using keys or mouse.
g. Your code should show your understanding of how to integrate your zooming with the framework provided – using the FilterPoints class that you create and you should be prepared to explain how it works.

~~10.~~ **Animate moving objects (max 2 marks):** To get this mark you need to show your ability to create smooth animations for moving objects. Note: there are various advanced demos which show you how you could do some elements of this, but you should not just switch between two images (as one of the demos does), it should look smooth (e.g. having enough images to be smooth, or looking at how to make it smoothly animate in another way).
This cannot just be a copy-paste of the demo code – even for one mark you need to show enough awareness of what you are doing to justify that you 'showed understanding'.

1 mark: for this you need to:
a. have at least one animated object
b. show understanding of how to animate at least one object so that its appearance changes over time, however this could be a proof of concept instead of being smooth

2 marks (advanced): you need to meet the criteria for 1 mark and:
c. all of your objects should have animated rather than fixed appearances
d. the animation should be *smooth*
e. **and** *visually impressive*. Please do note the 'impressive'. So please don't just rotate objects using facilities from a demo if that won't be impressive as you would not get the mark.
f. Be able to explain in the demo what you have done and how it works if asked.

~~11.~~ **Image rotation/manipulation using the ImagePixelMapping object (max 1 mark):** for this mark you need to show that you understand how to create a new ImagePixelMapping class which acts differently to the existing ones and does something at least slightly differently to the existing examples, and that you use it appropriately. Note: this could integrate with other requirements so please do think ahead.

1 mark: show your understanding by:
a. creating and using your own ImagePixelMapping class and object
b. draw at least one object to the screen. Your class must do something different enough to the provided ones to allow the marker to see that you understand how to use it fully.

~~Tile manager usage (Max 2 marks, 1 is advanced)~~

12. **Interesting and impressive tile manager usage (max 2 marks):** To get this mark you must be using a tile manager, and must have multiple displayable objects.

1 mark: You meet the following requirements:
a. Your tile manager must draw a number of appropriate and different *pictures* (either using images or the drawing primitives) for the different tile types, which are not just different colours.
b. You should have at least 5 different tile types (not just different sizes ovals/rectangles).
c. At least one tile must be drawn using an image.
d. You should load the image only once and keep it in a relevant object, NOT keep reloading it each frame if you want these marks.

Note: If you use multiple images then you could also potentially meet other requirements.

2 marks (advanced):
As one mark, but also:

a. The marker is impressed by what you have done, it looks great, works well and has some interesting behaviour.
b. You need a situation where moving onto some tiles would have effects on tiles elsewhere, e.g. standing on a key tile visibly unlocks a door tile.
c. This requires *visibly* changing one tile from a position where the displayable object which does so is NOT over that tile (i.e. you update a different part of the screen.)

Other features (Max total 3 marks, 1 is advanced)

~~13.~~ **Allow the user to enter text which appears on the graphical display (max 1 mark):** For example, when entering a name for a high score table, capture the letter key presses and pressing of delete key, and show the current string on the screen (implementing delete at well may be important). This needs thought but is useful to demonstrate your understanding of strings.
Entering text into the console window does not count for this.

1 mark: Do the above, including meeting the following key requirements:

- Capture the key presses for letters/characters.
- Store the key presses somewhere
- Capture delete key press and handle it appropriately
- Display the text on the screen

~~14.~~ **Show your understanding of templates, operator overloading OR smart pointers (max 1 mark).** To get this mark you should use at least one of these features in an appropriate and useful manner in your program. Using one of these is sufficient.

1 mark: for this mark you need to:

a. have created and used a template class, template function, operator overload (other than = and ==) or smart pointer
b. in a way which shows your understanding of how to use it
c. and is different to any provided examples/demos
d. and uses it in an appropriate and non-trivial manner
e. to achieve something useful, for which it is the appropriate feature to use.

Hints:

- You may want to consider whether you can use operator overloading as a part of your loading/saving, since you can use global functions to overload << and >>.
- If you use normal pointers it may be an easy change to convert them to smart pointers once you understand what these are.
- If you have something which can be applied to multiple classes, which cannot be resolved using sub-classing then consider whether a template is appropriate.

15. **Additional complexity, pre-agreed in advance with Jason (max 1 mark, advanced).** The students who find C++ easiest often have great ideas for things that they would like to do, which display good knowledge of C++, are complex and interesting but don't fit into these marking criteria. This mark is available for covering features where you show a good knowledge of C++, use some complexity, or create something that is exceptionally impressive. The mark aims to reward only the students who are the very best at C++ and is an advanced mark.

Talk to Jason in advance of the submission deadline if you think that you are doing something extremely impressive in some way which is not covered by the marking scheme as written and we can discuss it.

- Importantly this mark has to show exceptional ability in C++, and must cover some feature which is not covered by another criterion.
- It may be possible to get this mark for a topic covered by one of the other marking criteria if your implementation was truly impressive, well beyond the scope of the existing mark scheme (like an 'extra advanced' mark for any of the other criteria – agreed in advance with Jason).
  Important: Again, you need to get this mark agreed in advance, specifying clearly what you plan to do, why it is beyond the current criteria and how it demonstrates your considerable skill with C++! Get this agreed before you actually spend the time doing it, and discuss with Jason in the lab time if possible.

Overall impact/impression (Max 3 marks, all 3 are advanced)

16. **Impact/impression/WOW factor! (max 3 marks)** These marks are awarded based upon the overall impact/impression of the program and whether people would probably pay money to own it. Many people will just do the minimum and will not get these marks. These marks will take some effort to attain and are designed to ensure that the people who are most capable with C++ get higher marks. Be careful not to spend too long on this coursework if you are struggling!

    0 marks: your program works well and meets the requirements, rather than having a great impact or impression. These are meant to be advanced marks, so don't be disappointed with this 0 mark.

    1 mark (advanced): This is non-trivial to get. You made an effort to ensure that it is beyond the minimum to just tick boxes. As long as you completed part 1 of the coursework, this basically just means that you made an effort to make it look good and work well. If the marker looks at it and thinks 'looks nice' and when it is used it works properly and well, with no problems, you will get this mark.
    As a minimum this means you meet all of the following:
    - you made some effort with the graphical appearance,
    - the background is relevant and not plain or the same as any of the demos,
    - the background includes at least some use of relevant shapes (e.g. separating off a score by putting it in a box and labelling it) and/or images,
    - moving objects are not just plain circles or squares.
    - have at least three moving objects
    - at least one is user controlled
    - you accept both mouse and keyboard input in some way
    - you use both images and drawing primitives appropriately
    - you have appropriate states (at least 2)
    - have something beyond the minimum required for the other marks (*please say what*)
    - your program should work smoothly
    - everything looks good with no problems

    You need to include one or more screen shots in your documentation, along with an explanation of what you did that was beyond the minimum in order to get this mark.

2 marks (advanced): Meets all criteria for 1 mark and it is also *very impressive for the markers*. It needs to work very well, doing some complex tasks and you obviously made a significant effort on this coursework. If it's a game then it's fun and interesting to play, and has at least 2 different levels (to demonstrate your ability to do this). As a non-game (e.g. a drawing package or word processor?) it should be a complex program with different states (e.g. different page view layouts) performing a useful task and it should achieve its purpose well.
It should be impressing the markers to get this mark.
This mark is subject to moderation across markers, to avoid subjectivity.
You need to provide information in your documentation (in the section provided for this) about why this is so impressive, including some screen shots, making an argument for getting this mark.
Note: If you think that you MAY have a sellable quality game (see the criteria below) which is one of the best in the year, then please also submit the short video mentioned in the 3 mark criteria so that this can be assessed.

3 marks - sellable quality: This is supposed to be hard to get and basically means that your 'impact' of your program was even more than the 2 mark value of the impact/impression mark, which will not be easy. This basically means that both the marker and Jason went 'wow' when they saw this and thought that people would easily pay money to buy this program.
This mark is there so that we can differentiate the courseworks of the top students. In the same way that one would not always expect to be able to get 100% in an exam, not all students would be able to achieve this mark in a reasonable time, so should not be trying to do so.
Note: this mark is to assess the C++ ability, not the length of time in level design or data entry. As long as we can see where this is going, you should not spend a long time designing lots of extra levels or entering a lot of data for use in the program. You shouldn't need to be designing more than three levels as a proof of concept, if the levels you design illustrate your program well.
Your program should be working really well, smoothly, look good and make a good demonstration and should be comparable in quality to the sort of programs available on an app store.
I also note that in some cases it is possible that a program could be sellable while not implementing too many of the other features, since some app store programs are relatively simple but still sellable if presented well.
This mark is subject to moderation.
To get this mark you need to provide the same level of documentation as for the 2 marks, but also a short demo video (maximum 3 minutes, but shorter is better) of it running and the sort of sales pitch that one would see in an app store, to illustrate why it is so great and why people should buy it.