Manchester Metropolitan University

Department of Computing and Mathematics

ASSIGNMENT COVER SHEET

Unit title:	6G4Z1011 Programming
Assignment set by:	Dr David McLean
Assignment ID:	1Cwk100
Assignment title:	Siege Game
Assignment weighting:	100%
Type: (Group/Individual)	**Individual**
Hand-in deadline:	Block 2 assessment week (see Moodle – assessment area)
Hand-in format and mechanism:	Electronic submission (zip file of code) via Moodle
Support:	*During lab, tutor office hours, bookable support slots with support tutor. Taught lab exercises through Programming 1 and Programming 2 incrementally build up to cover all the concepts and techniques for this assignment*

Learning outcomes being assessed:

- Apply computational thinking and fundamental programming concepts to solve problems
- Design and implement well-structured solutions to problems of varying complexity using appropriate methods, including Object Oriented techniques.
- Adopt a reasoned approach to identify and rectify software defects in simple programs.

Note: it is your responsibility to make sure that your work is complete and available for marking by the deadline. Make sure that you have followed the submission instructions carefully, and your work is submitted in the correct format, using the correct hand-in mechanism (e.g. Moodle upload). If submitting via Moodle, you are advised to check your work after upload, to make sure it has uploaded properly. Do not alter your work after the deadline. You should make at least one full backup copy of your work.

Penalties for late hand-in: see Regulations for Undergraduate Programmes of Study (https://www.mmu.ac.uk/academic/casqe/regulations/assessment/docs/ug-regs.pdf). The timeliness of submissions is strictly monitored and enforced.

All coursework has a late submission window of **5 working days**, but any work submitted within the late window will be capped at 40%, unless you have an agreed extension. Work submitted after the 5-day window will be capped at zero, unless you have an agreed extension.

Please note that individual tutors are unable to grant extensions to coursework. Extensions can only be granted on the basis of a PLP, or approved Exceptional Factors (see below).

It is your responsibility to check that you have submitted the correct working zip file of your solution.

Exceptional Factors affecting your performance: see Regulations for Undergraduate Programmes of Study (https://www.mmu.ac.uk/academic/casqe/regulations/assessment/docs/ug-regs.pdf). For advice relating to exceptional factors, please see the following website: https://www2.mmu.ac.uk/student-case-management/guidance-for-students/exceptional-factors/ or visit a Student Hub for more information.

Plagiarism: Plagiarism is the unacknowledged representation of another person's work, or use of their ideas, as one's own. Manchester Metropolitan University takes care to detect plagiarism, employs plagiarism detection software, and imposes severe penalties, as outlined in the Student Handbook (http://www.mmu.ac.uk/academic/casqe/regulations/docs/policies_regulations.pdf and Regulations for Undergraduate Programmes (http://www.mmu.ac.uk/academic/casqe/regulations/assessment.php). Bad referencing or submitting the wrong assignment may still be treated as plagiarism. If in doubt, seek advice from your tutor.

As part of a plagiarism check, you may be asked to attend a meeting with the Unit Leader, or another member of the unit delivery team, where you will be asked to explain your work (e.g. explain the code in a programming assignment). If you are called to one of these meetings, it is very important that you attend.

Assessment Criteria:	Indicated in the attached assignment specification.
Formative Feedback:	You can seek formative **verbal feedback** by booking 1 to 1 slots (see moodle) or within you weekly lab session.
Summative Feedback Format:	Summative feedback will be provided through moodle, with individual and group feedback.

Please read ALL the specification before starting and before hand-in

Sections

1. Employability Statement

This assignment will help develop your problem solving and programming skills on a relatively large and complex application involving ***multiple classes*** and objects. All the techniques used are industrially relevant and a completed application can be used as evidence of your abilities and skills for your CV and future placement and career interviews. It would be wise to produce a digital artefact (e.g. screen recording or web enabled executable) which can be referenced from your CV or covering letter.

2. Game Application

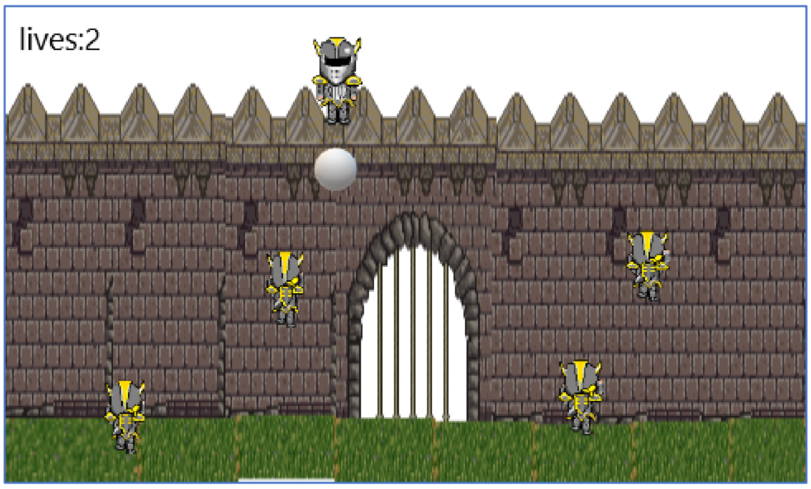
You must design and Implement in **Processing (Java)**, your own original 2-D Siege game, where the player character can move horizontally across the top of the screen (arrow cluster key controls), dropping objects on to attackers. Attackers will be moving up the screen slowly (and perhaps left and right movement as well). Your code MUST contain a minimum of 2 classes, one for the player character and one for the attackers, though more classes should be added as necessary. The minimum to pass this assignment requires that attackers must simply be avoided (no dropped objects). For higher marks it is required that objects should be dropped and, a further class of attacker objects to be avoided should be added. All the techniques you need to solve this assessment have already been covered in the lecture notes and/or lab exercises through Programming 1 & Programming 2.

Example

4 attackers, scale the wall

Defender(top of wall) drops cannon balls on attackers

An extra class of arrows travelling up the screen could be added



There are no marks available for the quality of the graphics used, just for the animation sequences and techniques, so leave any images (gif jpg etc) until later in your application development.
Some example scenarios:

- Player defends a castle wall from above avoiding arrows and attackers and dropping rocks on to attackers who are slowly climbing the wall.
- Player is a tank defending the top of the screen, can fire at attackers, attackers are vehicles moving up the screen.
- Player is a sailor on a large boat, dropping anchors on to attacking pirates, climbing the side of

the boat
- Player is aboard a hot air balloon, dropping sandbags on to attacking birds

Optional additions (will increase your grade range – see Marking Scheme criteria, section 5)
- Game levels
- Player lives
- Scoring
- Multiple animated sequences (must differ in sequencing) – e.g. spinning, explosion, walk left, walk right etc.

3. Hand-in

Submit to Moodle a **ZIP** file containing your solution **directory**, and all the associated files including:

- code file(s) which will run without errors – code with errors should be commented out
- any image files, etc.

The zip file should consist of your name and student number e.g. DavidMcLean99700733.zip

Please note that the submission inbox on Moodle will not accept submissions larger than 100MB. Your zip file is unlikely to be this large unless you have used very large image files. Please check in good time that your zip-compressed work will fit within the size limit.

4. Plagiarism

This assessment must be a product of your own efforts. We will use an **AUTOMATED PLAGIARISM CHECKER** to compare your submissions against other student submissions and any similar online examples. This system is impervious to changes of variable/procedure names, moving blocks of code around etc. Last year some students were found to have high degrees of similarity in their submissions and had to undergo formal plagiarism hearings in some cases resulting in expulsion (see Student Handbook). Do not copy blocks of code or allow blocks of code from your own work to be copied by others.

5. Summative Marking Scheme:

All features listed must be present to achieve each range of marks. For example, to get over 70%, all features from the previous bands (40-50),(50-60), (60-70) must be in your code, including some features in the 70-80 band.

Base Mark	**Features Required** (starting at top, tick off each criteria working down the list)
40%	**All** of the following to Pass (40%): ☐ Player character near top of the screen with **arrow key** controlled left & right movement ☐ Minimum 2 classes : Player, Attacker ☐ At least one attacker object that moves up the screen ☐ Simple working game (comment out code that causes errors) in Processing ☐ Something clearly happens when a player collides with an attacker
40-50%	**All** of the above, and some of the following: ☐ Player must be restricted to being on the screen ☐ At least 3 attacker objects that move up the screen (and possibly left,right) ☐ Working Collision *function* method(s)

50-60%	**All** of the above, and some of the following: ☐ Splash or game over screen (draw does different things at different times) ☐ An ArrayList (or array) of attacker objects ☐ Animated sequence of images for the attacker objects (appears to climb, fly, etc)
60-70%	**All** of the above, and some of the following: ☐ Player can drop objects (attacker removed from game on collision) ☐ A 2nd type (class) of attacker objects that must be avoided by the player (collision involves loss of life or game end) ☐ Class-inheritance for different attacker types (perhaps other classes) ☐ File handling – high score(s) saved and read from file
70-80%	**All** of the above, and some of the following ☐ Array of PImages for animation sequence ☐ Exhibits some polymorphism with the array/arrayList of attackers
80%+	**All** of the above, and some of the following: ☐ collision animation sequence (e.g. explosion) ☐ Complex attacker movements (e.g. sometimes follow the player) ☐ Polymorphism for most (or all) game entities ☐ Use of an Interface or abstract class ☐ Refactored, maintainable code

The following table will be completed during marking, where highlighted concepts increased your grade and red concepts decreased your grade within the range indicated in the table above.				
Onscreen: Score lives	Splash screen, Game Over Screen	Game levels	Animated sequence of images:	Classes: well structured Appropriate number
Meaningful Variable names: Mostly All	Constructor(s)	Suitable Methods	Multiple animation sequences	Public/private
Meaningful method names, Parameter names	constants - appropriate use	Function(s)	Switch case – game mode	
Well-structured • Mostly • All Indentation Intuitive order	Well factored – procedures/parameters names No duplication Easily read	Comments where necessary Largely self-documented code	Enum set – game modes	Concise efficient code

6. Hints & Help, Getting Started

In Programming 1 we had the Defenderz lab (week 5). Start by attempting the Defenderz lab exercise (read accompanying teaching material "Developing Multiple Classes and Objects") here you had 2 or 3 classes that can be modified to get started with this assignment. Remember the week prior to this with teaching material entitled "Using a Class" & "Writing a Class."

Start simple, add one thing at a time and test it – check it works properly.

Remember that when a collision occurs (e.g. when an obstacle hits the player) then we need to get rid of the obstacle (set the instance equal to **null**). This must be handled from code OUTSIDE the class (from the draw event or a procedure called from the draw event). A collision function should be called that can RETURN a value to the calling code, so the calling code knows to remove it (see notes on classes with arrays).

There are many sources of images (Paint 3D) can be used to produce your own or edit existing images. There are free online images for background and animated image sequences (https://opengameart.org/), these may need to be credited. There is a tool on Moodle which will split a sprite strip into a set number of images. You can also crop from a sprite sheet.

Re-read your lecture notes – we've covered similar ideas within the lectures and labs (specifically the material on Multiple Classes).

Use top-down design (as taught throughout the first term lectures) to help you implement your various methods.

Class design: carefully consider all the members within your Class(es) do they have sensible names (obvious what they are intended to do) are they all necessary, would more members simplify your code? Does each method perform only one task? Is all the information it needs passed as parameters?