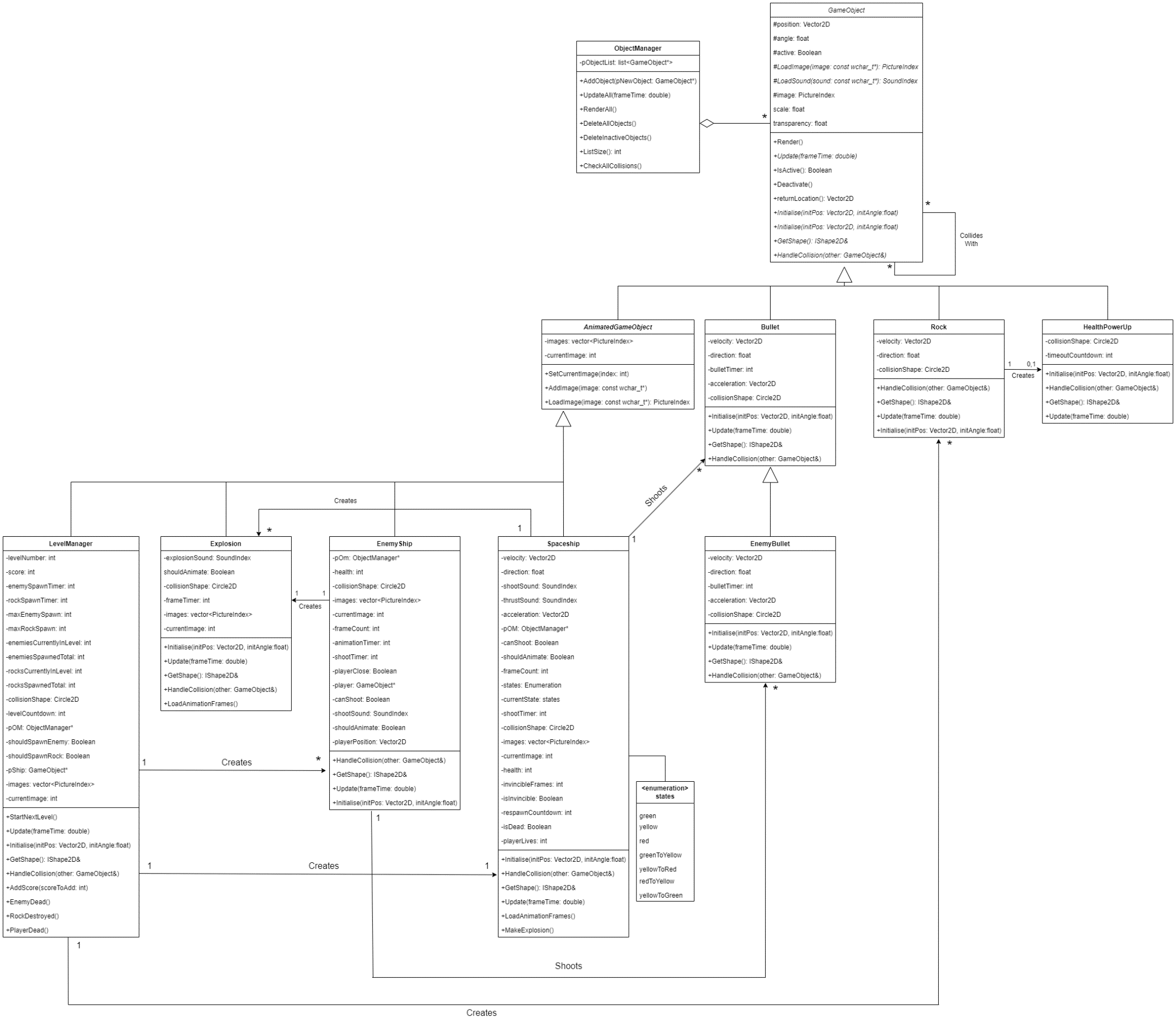
KF6017 Software Architecture for Games System Discussion – w20003739

UML Class Diagram - [KF6017 Class Diagram.png](https://livenorthumbriaac-my.sharepoint.com/:i:/g/personal/w20003739_northumbria_ac_uk/EfQ28CciGBxKsMtjNdB8gEABp_jnxUkliHcjsp5qibuAIA?e=loeQFc)



Discussion of System Architecture – Word Count: 1016

The game developed for this assignment took on the form of a 2D multi-directional shooter akin to games such as Asteroids (Atari, 1982) and Robotron: 2084 (Vid Kidz, 1982). The game tasks the player with controlling a spaceship around the game level, using the arrow keys to fire projectiles in four directions relative to their angle in order to destroy a series of continuously spawning asteroids and enemy ships – which will attempt to destroy the player with an assault of lasers. The player’s goal in a level is to defeat all of the spawning enemies and asteroids, at which point they will progress to the next level. The assets used in the game include a series of images and soundbites provided alongside the base engine for this project as well as a number of self-made images drawn using Pixilart (Pixilart LLC, 2023).

The objects and systems of the game operate on a hybrid flat and multi-level inheritance hierarchy, with all objects active within the game ultimately inheriting from the abstract ‘GameObject’ class. This abstract class provides a framework of functions for all game objects to utilise such as Update – a function that contains all code for an object that is required to be ran every frame of the game – and Render – a function that handles the drawing of all visual elements of a game object on the screen, which only needs to be declared once in this superclass as all game object types will be rendered the same way. This class then featured an abstract child class that various game objects inherited from – AnimatedGameObject – which built upon the functionality of the superclass by also providing the ability to load and swap between multiple images on-the-fly, therefore creating keyframe animated objects. This meant that objects that required this functionality were able to feature smooth, easily adjustable animation, whereas objects that did not were able to disregard this child and continue to inherit from the abstract superclass.

The use of this single abstract superclass that all other game objects inherit from is vital for the concept of polymorphism within the game, as all objects – no matter their true object type – can be handled equally within a single list in the ObjectManager class, allowing for functionality such as UpdateAll and RenderAll within the ObjectManager class, meaning that the primary GameCode class does not need reference to any of the game-specific object types and only requires a pointer to the ObjectManager, allowing for significantly lower coupling and also significantly more efficient code as this class could at any point update all GameObjects as well as rapidly delete inactive game objects without the worry of pointer-based memory leaks. Using this abstract superclass inheritance for all game objects did lead to a slight issue in some of these objects however. In order for a game object to be able to be instantiated in the game, it was required to implement an override for all abstract classes within the superclass, but some of the child game objects did not require these functions (for instance, the LevelManager did not require use of the HandleCollisions function, but a form of it was still required to be implemented). This did not cause any significant problems, as if an object did not require a given abstract function, it would just leave its implementation of this function entirely blank, but this redundancy was still far from ideal.

This low coupling was also aided with the introduction of the LevelManager class which handled the creation of the player ship as well as the obstacles in the game, which were previously handled directly in the GameCode class. This level manager was in charge of handling the creation of the player ship object at the beginning of the game, as well as incrementally creating instances of the asteroid and enemy objects when required – in addition to the handling of various elements of text output on the screen and keeping track of the player characters life status. This is poor cohesion, as some of these tasks should not have been carried out by the LevelManager – which should only be concerned with any details and tasks directly related to the management and changing of the in game levels. To remedy this in the future, the LevelManager should be split into a more dedicated LevelManager and a GameManager, which would monitor the tasks that should not have been required by the LevelManager.

There is a clear issue with cohesion in multiple game classes overall. In game development – and any object-oriented program – it is preferable that each function and object should not perform too many separate actions or tasks to improve reusability and generalisability of the code (Eder *et al.,* 1994). It can be seen that multiple objects under the abstract GameObject feature functions that perform more than one task and therefore feature poor cohesion. A potentially significant solution to this problem would be to switch from the current hybrid inheritance system to a component system for the various objects in the game, similar to the approach used in Unreal Engine (Epic Games, 2023). This approach – where a game object is made up of a series of specially designed, decoupled components, would massively avoid some of the cohesion issues that appeared during the development of this game.

There is a noticeable limitation with the generalisation of certain aspects of the game engine code. Classes such as the LevelManager and the ObjectManager should be made abstract and entirely non-specific to the game being built, with more concrete, game-specific versions of these Classes being built and inherited from on a game-by-game basis instead. Whilst the decoupling of all GameObject code from the GameCode class – as well as the use of upcasting all specific game objects to their generic GameObject forms for use with polymorphism in the ObjectManager – has allowed for a fairly generalised engine, making more of the remaining non-game-specific classes abstract would allow for an even more significantly generalisable game engine that could be extremely quickly used to implement any variety of other games without worry of needing to edit the engine code to be compatible.

Reference List

1. Atari, Inc. (1979) *Asteroids* [Video Game]. Atari, Inc.
2. Eder, J., Kappel, G. and Schrefl, M. (1994) *Coupling and cohesion in object-oriented systems* (pp. 264-272). Technical Report, University of Klagenfurt. Available at: <https://www.researchgate.net/profile/Johann-Eder/publication/2341329_Coupling_and_Cohesion_in_Object-Oriented_Systems/links/00b49526a89655aaf5000000/Coupling-and-Cohesion-in-Object-Oriented-Systems.pdf>
3. Epic Games. (2023) *Unreal Engine 5* [Computer Program]. Available at: <https://www.unrealengine.com/en-US/download>
4. Pixilart LLC. (2023) *Pixilart*. Available at: <https://www.pixilart.com/>
5. Vid Kids. (1982) *Robotron: 2084* [Video Game]. Williams Electronics.