University of York

Department of Computer Science

SEPR - Assessment 1

Architecture

Team Craig

Thomas Burroughs

Huw Christianson

Joseph Frankish

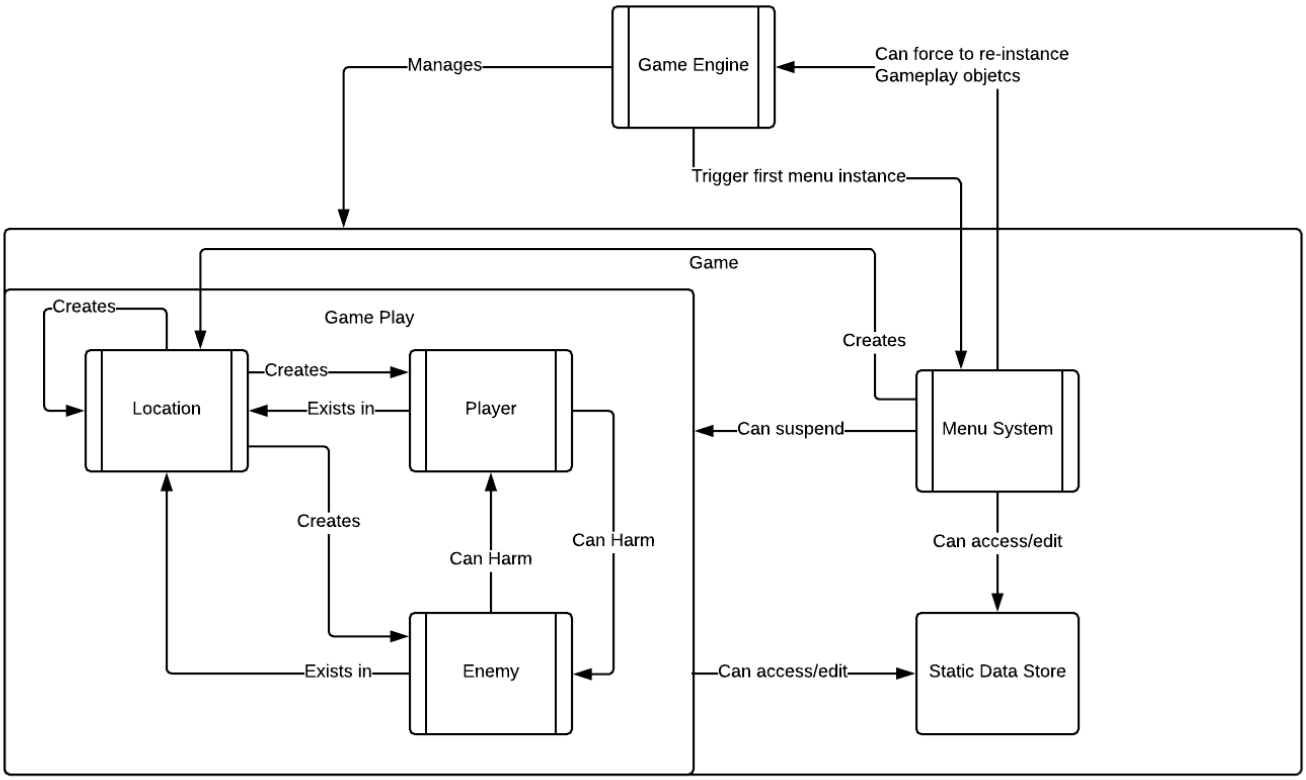
Isaac Lowe

Beatrix Vincze

Suleman Zaki

Architecture

This document covers an abstract view as to how the architecture of the deliverable product could be structured in order to demonstrate to the client how the development of the final deliverable will progress throughout the implementation section of development. This document will be abstracted from the finer details of the eventual implementation of the product, however will aim to retain enough detail to act as a guide for non-technical parties inspecting the implementation of the final deliverable product.



Representation of Architecture

The above architectural diagram has been made in an online diagram design program called, lucid chart, using a subset of the UML specification. The elements used are as follows:
Active class (Box with bars): This is being used to mark entities that have agency. They will have the ability to affect other entities.

Passive class (Box without bars): This is being used to mark entities that do not have agency. They will only have the ability to return or edit their own members and will be used for data storage.

Action Arrow (Arrow): This is being used to show how one entity can affect another. Each arrow is labelled describing the relation.

Grouping (Box): This is used to group entities to make the diagram easier to read. They do not have any impact on the underlying architecture being shown.

Explanation and Justification of Architecture

Program Layer

At Layer 0, the program layer, we can see the game engine is the outer layer of our involvement with the deliverable product, although we must perform testing whilst deciding upon implementation to ensure that the end product will run in the deliverable environment as per requirement P1. This layer shows the game engine, this being the program that will run in order to produce the elements of the game to the end user, through the handling and implementation of lower level system functions such as graphics and audio. The reason we have developed the program layer in this abstract manner is to avoid unnecessarily overcomplicating the final product by developing game engine code when all of the material that we need is readily available to us under permissive licensing terms. Using premade library code to accomplish the generation of the game environment is excellent, as it will simplify a large amount of the overhead on developers working on lower-level game systems. This will allow more work-hours to be spent productively adding to gameplay systems.

Game Layer

Below this, at Layer 1, is the Game Layer, showing a representation of the elements and processes that will run within the confines of the game engine. This high-level overview shows the partitioning and interactions between the runnable elements of the deliverable program. The most important thing here is the menu system, which is invoked by the activation of the game engine and has relationships with multiple other important elements. These include the running of the main portion of the game, and abstract systems for modifying configuration of lower level systems represented at the program layer, and general abstract game data held in the static file store. We have used a generic abstract static file store, as the likelihood is that when the implementation stage is reached we will be implementing a storage engine based on common programming practices already supported and commonly used with our game engine and any other potential library code, so as to minimise programming overheads through utilising the systems made available to us through the game engine. Not splitting this down into further detail (as this is likely going to be a system of a high degree of complexity when actually implemented), will allow us to conduct further research into the capabilities of our software development environments and what the team is already familiar with. We have chosen to do this as the static data file store will be important for the development of essentially all other components of the final deliverable, ensuring the accessibility and easy implementation of abstract structures by all team members. This will be vital to the rest of the software architecture. Likewise, the menu system will be implemented using the base systems made available within the game engine, this will likely be less vital to team understanding than the static data store, as the menu is a self-contained unit with interactions primarily focused around invoking other architectural elements (as can be seen from the direction of the lines on the UML chart).

Gameplay Layer

Finally, the lowest layer of abstraction represented on our chart is the gameplay layer. This layer represents the elements of objects or groups of objects as can be seen by the user of the finished deliverable. The relations between objects in this layer can be understood to be events within the context of gameplay that will of direct influence on the end user. This is the layer with most complex set of interactions between different elements, but also the most easily recognisable mappings between UML objects and end product classes. Within the game we expect the set of interactions between different types of objects to be pre-planned before implementation and suitable to be represented using interactions between class instances. Programming location instances to be created by other location instances is an important part of ensuring that the game can support multiple different locations (Req F4).

Having the character as a standalone class is useful here, as although it will not allow for the character model and traits to be changes in an uninterrupted manner during regular gameplay, it significantly simplifies the process of implementing multiple player characters (Req F5), whenever the character is changed through in-game menus only one re-instance will be required in order to make the change.