Code Architecture

As a basis for this project, we used the Unity game engine. Unity has a free version which is the one we used. The Unity system architecture is rather unique at the moment, as it uses a component-based architecture. Most other game engines, such as Unreal, use object-oriented and inheritance based approaches. A component based architecture means that each game object has a number of components that each carry out a specific function. Many of these are provided by the game engine, such as mesh renderers, particle systems, and navigation agents. These can be combined to create quite complex objects, although one limitation is that objects can have only one of certain types of components, such as particle systems. There is also a hierarchical structuring to game objects themselves, which allows for easy and flexible object construction, similar to referencing sub-components in a more traditional object-oriented design. In scripting, Unity has a base class called MonoBehaviour, from which all scripts must derive to interface directly with the game engine. It is possible to create other classes and your own class structure that does not derive from MonoBehaviour, however, Unity will not be able to apply the various component-based engine aspects (such as calling renderer components, physics components, etc.), nor will component searching and other utilities already provided by Unity be available.

Our architecture made use of the component modeling already available in Unity. The advantages of this are rapid and correct prototyping, while disadvantages include lack of understanding about how the different components communicate. We also created typically monolithic scripts, with a one-to-one correspondence between a holistic object (the Hellmet enemy, for example) and its controlling script. This diverges somewhat from the true strength of component based architectures since it is possible to add multiple scripts to the same game object. With our system, we tended to have many references to sub-objects within a given object like the Hellmet, which allowed for control of these sub-objects, but isn't consistent with the ideal of component-based architecture.

One thing we found useful was to derive a new class from the MonoBehaviour class called

GameBehaviour; most of our objects were then derived from this class, which gave us the ability to customize our base class to allow for easy resetting of game objects. This could probably have been done in an easier fashion by using “resettable” components, and attaching them to objects we wish to save. This is the beauty of components, especially considering how reusable they are across projects.

Apart from these issues and considerations, code generally was built to interface with the various components already provided by Unity. Most code served to glue the various components together in order to implement game-specific behaviour. Some scripts used more advanced code concepts, such as a finite-state machine for the Hellmet, but otherwise the code is straightforward. In sum, Unity proved a pretty easy way to build a game, especially given the time constraints.