Sean Burns
GAM-415
Final Task Document

Architecture Task

For this task I explored the following patterns used in Snapshot's architecture: callbacks, singleton pattern, resource acquisition is initialization (RAII), and object pooling. The definition of each other's patterns, their pros and cons, and an example of how they are used in Snapshot's architecture will be explained.

A callback is when a function pointer is in the parameter of another function, meaning that this function will be called within this other function. A pro for callbacks is that it avoids duplicate or unnecessary code keeping better organization. A very basic example of this pro would be calling some calculate function when trying to print an integer to a console. This would remove the need to create a new variable to store the result of the function to then be printed. Another pro for callbacks is flexibility, for example, you may have a math function that may call back to an operational function to be used within the math function. Therefore, giving this function more uses and more flexibility. A con of callbacks is the ability to hinder performance. For example, if a function has a loop and calls back to another function that also has a loop within this loop the function would become n^2, meaning it could hinder performances. Understanding the function and the function you are calling back completely will overcome this con. Snapshot uses a lot of callbacks in the math class named PolyDefinition. This class can be found in RAEMath.h. In this file, there are multiple functions that call another function to complete the task. This helps keep organization and readability when this file is referenced.

The singleton design pattern is at its most basic level a single instance of a class. This means a singleton is used when you only intend to have one instance of data. A pro of using the singleton design pattern is to have functionality that applies to a global set of data that you want to be reused. An example of this is having a singleton be used for a random number generator. A con of the singleton design pattern is it can be easy to create additional instances of a singleton class if not used properly, this defeats the purpose of using a singleton and reducing resources. Snapshot uses the singleton design pattern in the Singleton.h file which keeps a template instance of whatever is being used with the singleton. For example, in the Object.cpp file a singleton is used to retrieve the instance of a camera being used in an equation to get a game object's x parallel position.

Resource acquisition is initialization or RAII, is a technique in which a resource life cycle is bound to the lifetime of an object. For example, this means that if heap memory is allocated in the constructor of the class, this memory would be destroyed in the deconstructor of the class. Smart pointers are a tool that can be used to complete this technique without forgetting to manually delete data. A pro of using this technique is to avoid memory leaks which leads to

better performance. This avoids memory leaks as all the information is deleted and none is left over after the destructor is called. A con of using this technique is that it becomes useless. This means that what it was created to do doesn't happen. This could happen by forgetting to unlock while using locks, or an exception is thrown and the program never reaches the point where the object is out of scope and the destructor is called. Snapshot uses this technique in the form of a class in the files HeapTrackerWin.h and HeapTrackerWin.cpp. This class tracks all the data allocations and then destroys this data if out of scope which prevents memory leaks.

Object pooling is at a basic level a pool of data that is stored to be used later in the program. This means that instead of re-creating objects or re-destroying objects they are stored to be reused. This creates a pro of being very efficient and optimized. This optimization comes from not having a need to constantly use resources to re-create and destroy objects. A con of object pooling is that it is bad for memory usage, as you have a pool of memory waiting to be used and taking up space. Snapshot uses this technique in their RAII class which collects data that becomes out of scope. In this class, there is a pool in which data that becomes out of scope gets put into. This pool then collects until it is called to empty and destroy the data within it.

Multiplayer Task

For this task, I implemented a multiplayer mechanic allowing a second controller to be in charge of the camera mechanic. This means that player 1 would be in control of the character, with player 2 being in control of the camera.

This task consisted of doing work in the "PlayState.cpp", "Player.h", and "Player.cpp" files. Firstly, in the "Player.h" file I had to add a second controller class object to the player. Along with this I had to copy the functions associated with the first controller and duplicate them to go with a second controller, these functions included setting the support for a second controller and the second controller's speed. Next, I needed to set the controller support as true, and set the controller speed for the second controller when loading the player. This was done in the "PlayState.cpp" file. The second controller and on/off bool which was declared in the player header then needed to be defined in the cpp. This was done by setting the bool to false as default as well as setting the controller id to 1 for the second controller. The id was set to 1 as the first controller’s id was set to 0. The last work that needed to be done was to duplicate the code which makes events happen when a button is pressed, specifically the portion of code for the camera controls that needed to be duplicated. Then one of these duplicates was placed inside an if statement checking if controller 2 was connected and the bool was set to true for supporting this controller. This code within the if statement was edited to check if controller 2's button was pressed, meanwhile the other duplicate code was placed in an else statement below this if statement. This meant if there was no second controller connected or the support for it was turned off the else statement would be conducted resulting in the controls being for controller 1.

To conclude this task I also updated the controls menu adding an option to turn on or off the second controller, meaning you can still play single player if two controllers are connected. This work was done in the ControlsMenu.cpp file in which I added a new controls option for "usb controller 2". Then I edited the "ToggleControllerEnabled()" function to work on both controllers 1 and 2, making both menu options functional.