**Team Coding Experience.**

The team has varying levels of coding experience and differing levels of experience in a team programming environment. Coding errors can easily  occur within a project of this scope.

Mitigation:

**Coding Standards.**

To help streamline the process of discovering and fixing these coding error  the team will adhere to a coding standard. Using a coding standard will allow  the team to quickly identify probable locations of coding error by

communicating the purpose and justification of code without needing the  original coder to be present.

**Variable Naming Standard.**

When naming a variable the name will begin with a lower case letter that is the first letter of the base data type. Variables will then be named using a

description of the purpose of the variable to justify its purpose. Names will try  to avoid numerals to better describe the purpose. All variables will be  formatted using camel case for legibility.

Example:  int = iVariableExample;

String = sVariableExample;

**Function Naming Standard.**

When naming a functions the name will be formatted using camel case.

Function names will always start with a lower case letter. “Getter” functions  will always start with “get” then the variable name. “Setter” functions will

always start with “set” then the variable name. Any variables parsed into the  function will follow the naming standard of variables. Function names should  be descriptive to help justify its purpose.

Examples:  “Getter” Function would be named “getVariable”.

“Setter” Function would be named “setVariable”.

Game Loop function example would be “gameLoop”.

**Class Naming Standard.**

Class names will begin with an upper case letter then follow camel case

formatting. As this project designed using object orientated design, identifying  classes quickly can help locate coding errors that are causing major game- breaking problems.

Example: Entity Manager class would be named “EntityManager”.

**File Naming Standard.**

Graphic and sound files will be used in the game to create atmosphere. File  names will start with a base description of the file then continue using more  descriptions to narrow down its use. Underscores will be used instead of  spaces in file names.

Example: A background music file will be named “Music\_Background”.

A texture for a platform left corner will be named  “Texture\_Platform\_Corner\_Left”

**Commenting Standard.**

Comments will be left on every variable, function, and loop to briefly describe  its use. This will be used to justify the purpose of the code and will help when trying to locate an error. If a section of code is commented out of the program  there should be an additional comment above the section describing the

justification behind this. All comments will be initialled at the end to show who  has made alterations to the code. If changes have been made then the  comment will start with “Change” and numbered with the times its has been  altered. This will allow the team to see a timeline of changes made to the  code and help identify if a change may have created a new error in the code.

Comments will not be deleted unless discussed and agreed to be obsolete in  a meeting.

Examples:  Loop commenting - “ //This loop reads through each line of file  – MM”

Edit commenting - “//Change1 Loop was reading  backwards. Changed X and Y co-ordinates - RR”

1. Risk:

**Game Design Experience.**

Our team has varying experience in game design. The structure of a game is important to keep the program cohesive and working as intended. As this is a  team project different interpretations of how to create the program may exist. If the team is creating the code and structure using differing interpretations  this can lead to compilation errors, lost work and time.  

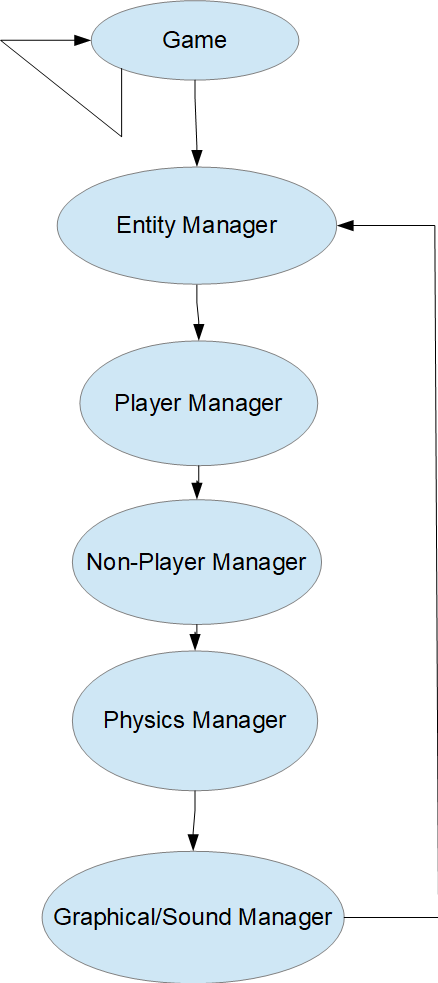
Mitigation :

To help the team work towards the same goal a basic game design structure  was provided and presented to the team in a meeting. This structure went  through the concepts of a game loop and how to manage and create a game  using object orientated design. Simple diagrams were used to describe  concepts.

**Game Loop**

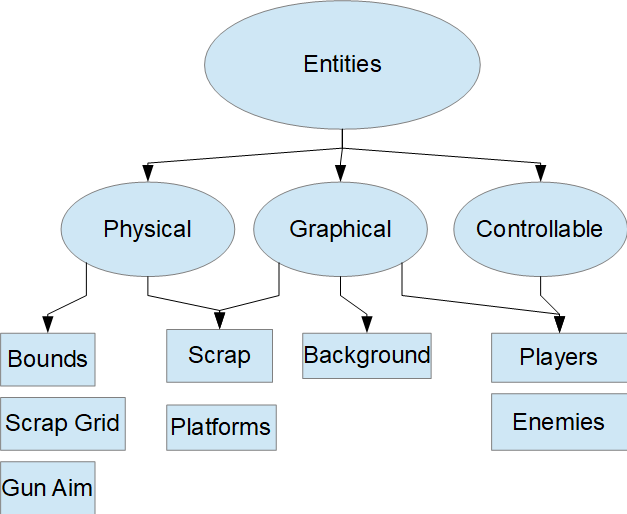
The basic game loop starts with a “singleton” object called “Game”. A

singleton is an object in a program that can only be created once because it  points back to itself. Each loop of the game will start by processing the   entities with the entity manager. This will process the players, non-players,  physics, and graphics/sounds. Once complete the loop will then run again   until game play ends.



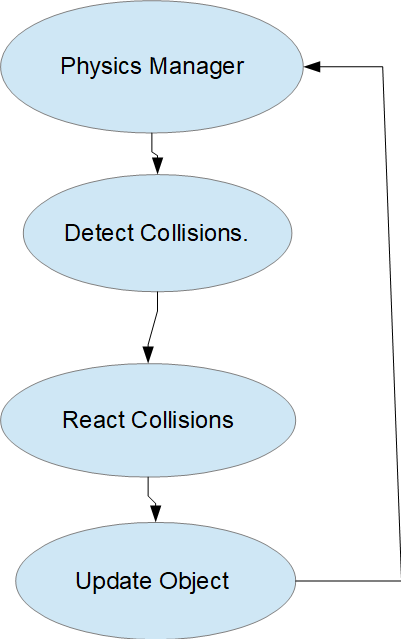
**Entities.**

Entities are objects in the game that are processed. Each entity could be   physical, graphical, and controllable. Entities could also be any combination  of the 3 attributes. This will decide what object are processed in each  manager. All physical objects will be managed in the physics manager. All  graphical objects will be managed in the graphics manager, etc.



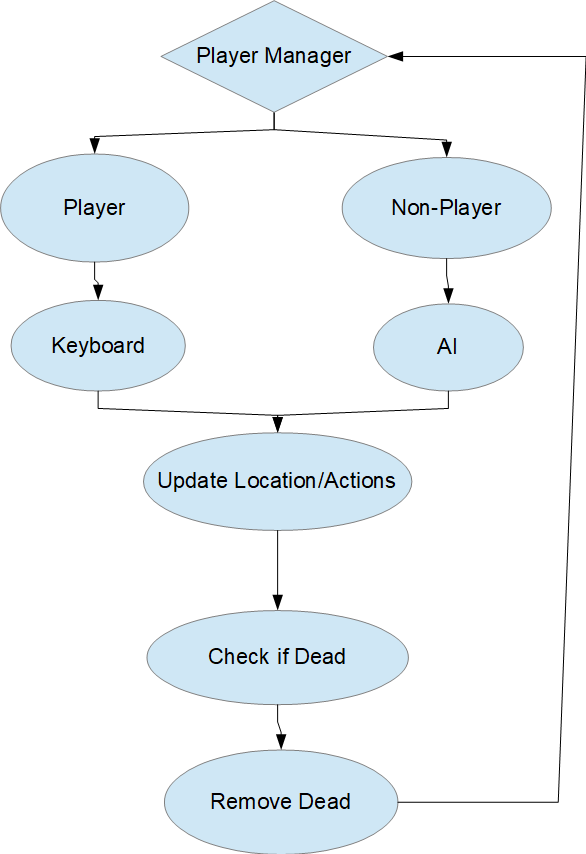
**Physics Manager.**

A physics manager will process all physical objects in the game. It will detect  if physical objects need to be altered. If objects are required to be altered  they will be altered by the physics manager and object variables will be  updated.



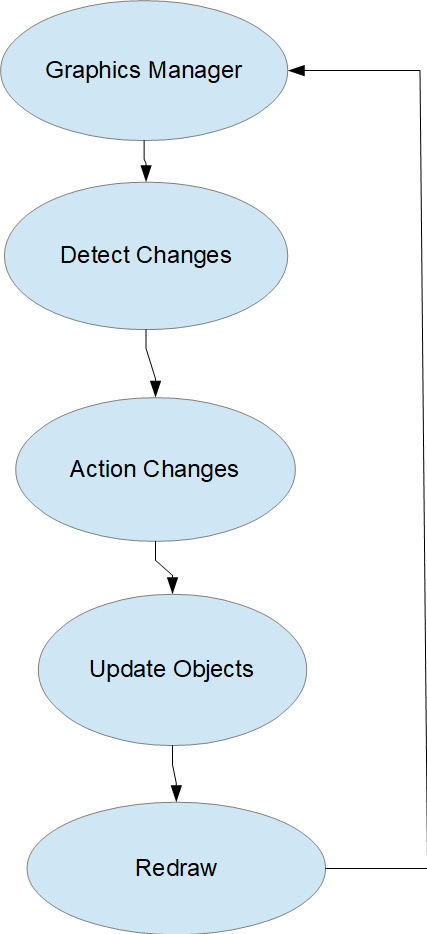
**Player Manager.**

The player manager will process all player objects. It will detect if the player is  human or AI and then process the control of the object accordingly. It will  then update locations and actions. If the player object is dead then it will  either end the game or remove the enemy.



**Graphics Manager.**

After physics and player objects are processed the graphics manager will   detect any graphical changes. It will then update any changes and redraw the  screen to show any changes.



**Sound Manager.**

The sound manager will detect any need for sound effects in game. If the is a  need for sound to be play the sound manager will play the appropriate file.  Background music will also be handled by the sound manager but due to it  being ambient it will consistently play.

