

# SOFTWARE PROJECT TECHNICAL REPORT

**Daniel G. Holmes 551240, Jonathan D. Gerrand 349361**

*School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa*

**Abstract:** Abstract

**Key words:** SFML 2.1, game development, MVP

## 1. INTRODUCTION

Problem understanding - Modular code - Orthogonal  
- Object-oriented decomposition - Separation of layers (MVP, MVC) - Illustrate design cycle of software  
- Iterative design process - Unit testing approach (regression testing)

## 2. REQUIREMENTS AND CONSTRAINTS

The game is required to be based on the 2 player tank battle Atari game "Combat". Certain basic functionality is required to be met. Each player should be able to control their tanks from the computer keyboard. Tanks should have the ability to fire missiles and lay mines, both of which destroy either players tanks. Movement of tanks and firing of tanks should be done both horizontally and vertically. Game play should take place in a maze consisting of barriers which block tank movement. Once either of the tanks have been destroyed, the game should end.

The game software is required to be implemented in ANSI/ISO C++. The usage of the SFML 2.1 library is also required. The software development life-cycle requires the usage of unit tests using the google-test framework. Unit tests are required at the end of each iteration of the game.

The game is constrained to run on the Windows platform. No other libraries that are built on SFML may be used. With respect to game graphics, no OpenGL may be used. Also, the maximum resolution of the game window should not exceed 1600x900.

It is assumed that the computer running the game will meet the minimum system requirements for Windows 7.

## 3. BACKGROUND

Games are classic examples of highly interactive GUI applications. Over the years, many different GUI architectures have emerged. One such architecture is Model View Controller (MVC). The main idea of MVC is a separation of the domain objects and GUI elements. Domain objects model real world perceptions, whereas the GUI elements are what the user sees

on the screen. Simply put, the model objects have no knowledge of the UI. The controller role within this architecture is to process user input. This either directly or indirectly would modify the model, or the domain objects. Once data changes to the model are complete, information from the model is pushed to the presenter and the users view is updated accordingly [martin-fowler].

One of the benefits of the MVC architecture is the fact that there is a distinct separation of program layers. These include presenter, logic and data layers. Many games these days are available on multiple platforms.

The data layer contains information only. There is no decision making in this layer. Logic layer manipulates, processes or changes information from the data layer. This data can also include information that is input from the user. The presenter is responsible for what is displayed to the user. No data manipulation takes place here.

Games these days are capable of being run on many different platforms. This may required changes to the display layer. If a layered approach is used, the game can be ported over to another platform without changing the cope interface. An example of this is Rayman.

## 4. DESIGN OVERVIEW

The chosen design is based off the MVC GUI model. Figure 1 shows the adapted version of MVC know as passive MVC used as a high level design for the game. A typical MVC design would include the model actually updating the view. With passive MVC, the display polls the model for changes. The polling methodology is suited to be used in the way the main program loop executes, where the display is updated at the end of the loop.

Another modification inherit in the design is the connection between the controller and the data. Typically, MVC would have the controller simply updating the data. Figure 1 shows information moves in both directions between the controller and data. Model data is not only updated, but it is also retrieved by the controller to be used in logical operations. Another reason for this particular modification of MVC

is due to the use of managers. During a particular management cycle, information from the model is retrieved, processed and then the model is updated accordingly. Only once all the data has been processed by the managers does the display get updated.

## 5. CONCEPTUAL MODEL

The model of the program is management based. Each manager has specific responsibilities. The managers exploit the principle of polymorphism. This is to emphasize modularity and single responsibility classes. It also allows for maintainability. New managers can be added or if there is a problem you know where to look for it. This way, the interfaces of the classes can stay the same. Managers also run functions that follow the tell, don't ask principle. Managers have access to information only pertinent to what they are doing. They can be agnostic to everything else that is going on. Another principle that was followed combines single responsibility classes with polymorphism. The task of a manager relates to an inherited property of the objects that a manager manages. Such as a move manager handles all movable entities.

All game entities are manipulated via smart pointers. Objects within a manager are controlled via weak pointers. This is so that when an object gets deleted anywhere within the code, it will be deleted everywhere else. This helps with making sure that the code can be modular without having to worry about pointers that are still existing on the heap. Objects can stay in the same place and only their pointers can get passed around.

The model attempts to decouple SFML from the main program as much as possible. This is so that a new graphics library can be used. The interface of the display class can remain the same and the implementation can be changed to suit the new graphics library. An attempt to decouple SFML is also evident in the Keyboard and GeometryEngine classes.

## 6. CLASS RESPONSIBILITIES

Discuss each class responsibility in terms of the program layers and CRUD operations. Define these layers. Certain classes were given very specific responsibilities because we wanted to decouple SFML. Implemented generic substitutes. These included Display, Keyboard and GeometryEngine classes.

Game: responsible for running the main program loop. Managers: responsibly for manipulating and changing the data of the game entities. Draw manager: Slightly different from the other managers as it prepares the data for the display. It is the only manager that forms part of the display. Display: SFML display of the game entities sprites based on the data received from the draw manager. Geometry Engine:

responsible for all the logic and algorithms relating to the geometry of the entities. This includes collision, range and line of sight detections. Much of the functionality of this class replaces the SFML collision detection. Game object classes: These include tank, missile, mine, turret and barrier. Their responsibilities include holding of the data for the respective game entity. These classes get told what data needs to be changed within them. Orientation: Data class that holds more specific data regarding the game entity's 2D orientation. Game data classes: These include game management and state data, as well as action data. Action data is changed based on the user input to the game. These objects are passed around between the managers. Each manager only has access to the data that is required to perform its management tasks. Keyboard: Acts as an interface between SFML and the game class. If another library was to be used, then the implementation of the keyboard class could be changed.

### 6.1 Abstract Based Classes

## 7. DYNAMIC BEHAVIOUR

Life cycle of game entities. Where they exist and how they are deleted. Data containers; updated and passed around between managers, each with access to only what they need.

Draw manager is a connection between the logic layer and presentation layer. Draw manager is somewhat part of the presentation layer as it does not modify the data or do CRUD operations. It only extracts the data and gives what is needed to the display class. Tells what needs to be drawn and where.

## 8. TESTING

Gameplay testing revealed that the program met the requirements for the basic functionality. One noticeable problem was the drop in framerate when there are a lot of sprites being drawn on the screen.

## 9. CRITICAL ANALYSIS

Tie back to what was observed in testing. The slow gameplay can be attributed to the order  $n^2$  calculations in collision detection. In this function, the bounding box of each entity is compared to the bounding box of every other entity. This results in many unnecessary calculations.

Structs were used extensively, it would have been better if these were separated into classes. Have to always reference our structs file, would be better to reference the individual class.

Talk about things that were not observed from testing, including: violated dry principle in certain places.

Would be better to use templates.

There are several refused bequests, such as with a turret which is a movable, but cannot move forward or backward.

Talk about monolithic classes.

### 9.1 MAINTAINABILITY

## 10. RECOMMENDATIONS

Don't recommend changing the interface. Suggest improvements on the existing design, not on how to change the design.

To reduce the number of calculations for collision detections, there can be certain checks put in place to see if the entities are within a certain threshold of each other. The entities are not within the range of each other, then the collision detection algorithm does not need to be run.

Perhaps recommends a better design methodology, not too specific though.

Create more specific abstract based classes.

## 11. CONCLUSION

- MVP, MVC (martin fowler) - Logic layer definition
- Games (Rayman) - Textbooks?