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 design 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 Graphical User Interface (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. The usage of MVC ensures orthogonality between these program layers. Many games these days are available on multiple platforms. This necessitates flexibility in terms of porting over a game to another platform. The game would require specific libraries; most commonly the display libraries. A game could be ported easily if there is loose coupling between program layers. For example, a game developed initially on a PC would have a different presentation layer requirements compared to mobile game. If the separation of layers is successfully achieved, a considerable less amount of effort will be required to port the game. A game in which the port is almost indistinguishable from the original is know as an Arcade Perfect Port [tvtropes].

The Agile methodology to project management is an alternative to traditional project management. The focus of Agile is on developing a product that is potentially shippable. This requires a iterative and incremental approach to product development. It facilitates communication between the development team in which the project requirements and design are continuously revisited. This allows for flexibility as problems in the product development life-cycle can be solved sooner, particularly if that problem relates to a change in requirements [agile].

This methodology is well suited to the development of software. The focus is on having software that can be shipped to users sooner, so that early feedback can be given. Overall, the usage of Agile helps developers put together the right product in as little time as possible.

4. DESIGN OVERVIEW

The chosen design is based off the MVC GUI model.the main 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 following section presents the conceptual model of the game in terms of the MVC architecture.

5.1 Controller

The controller module of the design is based on management cycles or ticks. These cycles occur within the main program loop. Management tasks are divided between different managers, each of which have specific management tasks which introduces single responsibility classes. These managers perform their tasks based on user input. Information regarding this user input, as well as other temporary main loop data are accessed by the managers. Each manager only has access to data pertinent to it's functions. These functions include create, read, update and delete (CRUD) operations.

The usage of managers also exploits the principle polymorphism. The fundamental task of a manager relates to an inherited property of the objects which the manager is managing. For example, all objects with the inherited property "Movable" are accessible by the Move Manager. An object may have many inherited properties. As such, the management of that object is taken care of by the relevant managers, purely because that object inherits certain properties. A new manager may be added based on the inherited properties of objects, or as per the abstraction requirements.

This type of approach in the controller module ensures that the CRUD operations are dependant upon the characteristics of game objects. This also means that the controller is able to meet the processing requirements of the model in a modular way. If there is a problem with a particular aspect of the program, one simply needs to search in the relevant manager. If a new manager is added to the controller module, it has impact on the other managers. This ensures that if new game functionality is added, the other functionality is not affected. Fundamentally, controller modules i.e. the manager, are agnostic to other management processes.

Another component of the controller module includes the user input. Due to the nature of the game, a user does not input information via the User Interface (UI), as is the case with the Model View Presenter (MVP) architecture. Rather, the information is received from the user via the keyboard. This is another reason why the MVC model is used in the game design. User input can go directly to the controller module. Thus, managers can act on user commands without involving the view module.

5.2 Model

All game entities (models) exist within the model module. Each model inherits properties from certain Abstract Based Classes. As such, a model may inherit multiple properties as far as the level of model abstraction requires. All game models inherit the base properties "Deletable" and "Drawable". This is so that all models can be deleted or removed from the game world as required. Also, all game entities are drawn at some stage in the program cycle. An example of a game model which inherits multiple properties is a tank. It inherits the properties "Movable", "Collidable" and "Trackable". Each of these properties gives an indication of the functionality of the tank model. Also, the managers (move, collision and tracking) that have access to the tank model data are evident.

In line with MVC and the separation of layers principles, the models do not perform any decision making. Most of the functionality of the models include setter functions, or "tell, don't ask" functions.

5.3 View

The view module contains all the functions required for drawing game entities within the game window (UI). In line with passive MVC, the data from the model is not pushed to the view. Rather, the view polls for changes in data. Only information required for drawing is retrieved.

This methodology further emphasises the point of separating the layers of the program. If a view is required on a different platform, then the existing view can be replaced without any impact on the model or controller modules. The controller module is also loosely

coupled to the view. Overall, the separation of the view makes the game potentially portable. Interface of the view classes can remain mostly unchanged while the implementation can be changed accordingly.

6. CLASS RESPONSIBILITIES

6.1 Managers

6.2 Usage of SFML

As per the game requirement, SFML 2.1 is used. Attempts have been made to decouple SFML from the main components of the design. No SFML is included in the model module. It has been included in the view and controller modules. However, classes have been design in such a way that these modules are only linked to SFML via interfaces. The only instances where SFML is used is for capturing user input and drawing sprites in the game window.

A Keyboard class, which makes use of the SFML keyboard functionality, is included in the controller. Due to the fact that the controller uses this class via an interface, the controller is ignorant of the library used in the keyboard class. The same principle applies to the Display class, except with respect to its use in the view module.

One of the key functions in any game is collision detection. SFML does include functionality for collision detection. However, this was not used. The reason for this is not only as an attempt to decouple SFML. Rather, the replacement of existing collision detection functionality is for the purpose of code reuse.

Certain functions of collision detection are used

Discuss each class responsibility in terms of the program layers and CRUD operations. Define these layers.

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 entitiy'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 perfoms 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.

6.3 Abstract Based Classes

7. DYNAMIC BEHAVIOUR

Life cycle of game entities. Where they exists are 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.

7.1 Memory Management

All game entities are manipulated via smart pointers. Objects within a manager are controlled via weak pointers. This is so that when a 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.

8. TESTING

Gameplay testing revealed that the program met the requirements for the basic functionality. One noticable 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 unnessary 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?