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: This document presents the design of a basic 2D tank battles video game. The game was developing using C++ along with SFML 2.1. The design is based off an object-oriented adaptation of Model View Presenter (MVC). The Agile methodology with the usage of unit tests was used in the game development life-cycle. A discussion of the game design and class responsibilities is given with respect to MVC. Attempts were made in the design decouple SFML from the design. These attempts were successful. Gameplay testing revealed an n^2 performance issues in the collision detection algorithm. A recommendation for a early exit condition in this algorithm is given. Final testing revealed a bug in the a function used by the turret firing algorithm. Reasons for this issue include the omission of a sigle unit test. Critical analysis of the code highlighted issues with respect to monolithic classes and violations of the DRY principle. A suggestion is given to implement template functions as well as two new classes in order to address these issues.

Key words: C++, game development, MVC, polymorphism, SFML 2.1

1. INTRODUCTION

Video games are classic examples of highly interactive Graphical User Interface (GUI) applications. Many of these games undergo numerous upgrades to improve the overall game experience. Often, these games are even ported over to other platforms. This requires the game to be designed structurally in a way to facilitate such changes.

The following document presents the design process and critical evaluation of an object-oriented C++ video game using the SFML 2.1 gaming library. This game is based on the 2-player 2D tank battle Atari game "Combat". Initially the design is presented and discussed in terms of a chosen GUI architecture which is seen to provide the necessary framework for creating a solution. Following this, a discussion regarding the methodology of game development process is given. The conceptual model as well as a discussion on the high level responsibilities and memory management is presented. Gameplay testing results along with a critical analysis of the game design is given. Recommendations on improvements in game performance and code robustness are also included.

2. BACKGROUND

Games are classic examples of highly interactive Graphical User Interface (GUI) applications. Over the years, many different GUI architectures have emerged as a means to streamline the maintainability, structure, interaction and the operational efficiency of the tasks these applications perform. One such architecture is Model View Controller (MVC). The main idea of MVC is a separation of the domain objects and GUI elements found within a system. Domain objects model real world perceptions in the form of data within program memory, whereas the GUI elements are what the user sees on the screen. Simply put, the model objects have no knowledge of the User Interface (UI). As an interface between these layers, the controller's role within this architecture is to process user input on to the object model. 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 [1].

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, thus providing a means for program classes to become decoupled from one another and to emphasize specialization. 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 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 considerably less amount of effort will be required to port the game. Within gaming jargon, a game in which the port is almost indistinguishable from the original is know as an Arcade Perfect Port [2].

As another important aspect to software development, the management techniques behind successful game development can be assessed. In this light, 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 one such problem relates to a change in requirements [3].

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.

3. REQUIREMENTS AND CONSTRAINTS

The game is required to meet basic functionality in order to attain a minimal level of playability. Each player should be able to control their tanks from the computer's keyboard. Tanks should also 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 while game play should take place in a maze consisting of barriers which block the movement of a players tank. 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 GoogleTest (gtest) 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 upon 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. It is also assumed that SFML functions have been tested and work as expected.

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.

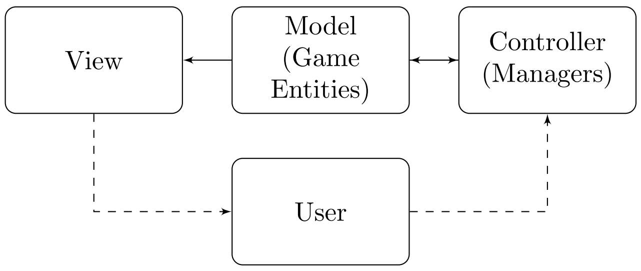

Figure 1 : Modified Passive Model View Controller GUI Architecture

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", "Collible" 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 Game Class

This class is responsible for running the main game loop and exists within main. Within this loop, the Game class receives user input, runs the managers and creates new entities. The creation of new entities depends upon user input, such as firing a missile or laying a mine. User input is received via the Keyboard class.

The game class is also responsible for getting map data. The map is read from a text file in which certain text symbols represent a game entity. Upon initial game execution, the map text file is read and the starting positions of game entities are set accordingly.

It is within the Game class that game entities are added and prepared for managers to work with. This is where the polymorphic nature of game entities is used. All managers exist within the this class. It is from here that all managers are called to manage. The Display class is also used in the Game class. However, the Display object does not exist within the Game class. Display functions are called in the Game class by reference.

6.2 Display Class

This class is responsible for drawing the UI. This includes game sprites and textures. It is also responsible for loading sprite images and textures. The object of this class exists in main along with the Game class. This ensures a limitation of the control that the Game class has over the Display object. Also, it provides a separation between the controller and view, as per MVC.

6.3 Manager Classes

Table 1 summarises the main responsibilities of each of the game managers. Although the draw manager is included in Table 1, it does not form part of the controller module. The draw manager does not manipulate data as most of the other managers do. Instead, the draw manager prepares data for drawing by extracting it from the models. It then passes this data to the view for drawing. For this reason, the draw manager forms part of the view module of the program.

6.4 Data Classes

6.4.1 Model Classes The model classes include all game entity classes. Their responsibilities are limited to holding information about a given game entity. An important aspect of these classes is that all their functions are inherited virtual functions. None of these classes have private functions. This ensures that these

Table 1 : Summary of the main responsibilities of the game managers

Manager	**Responsibility**
Move	Instructing game entities to move based on user input
Collision	Determining if game entities have and with what game entities have collided with
Tracking	Keeps track of player and turret orientation
Turret	Rotates all game turrets
Game State	Manage all game timers
Destruction	Deletes destroyed game entities
Draw	Prepares data of entities to be drawn

classes have very little flexibility in terms of what they are capable of. They exits to be not to act, but be acted upon.

6.4.2 Game data Apart from model classes, the game also includes a GameManagementData class. This class inherits from ActionData and GameStateData classes. These classes have no specific responsibilities other than to provide information for the managers. Managers will only receive either, one or both of the parent classes of GameManagementData. This is so that a manager only receives information required to perform its management tasks.

6.5 Usage of SFML

As per the game requirement, SFML 2.1 is used. The following subsection discusses the usage SMFL libraries throughout the code. Justification relating to why SFML was not used where is otherwise may have been used is also given.

6.5.1 Dependant Classes 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.

The 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.

6.5.2 Draw Manager Although the draw manager forms part of the view, it is not dependant upon any SFML libraries. Thus, the orthogonality of the code with respect to the drawing of entities is maintained. It is only the Display class that is dependant upon SFML drawing functionality.

6.5.3 Collision Detection One of the key functions in most games 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. This point is discussed further as part of the next subsection.

6.5.4 Orientation Class This class is used in order to remove SFML from the data layer. It is also used in an attempt to avoid repeating code. The implementation of this class ensures that each game entity class does not have the same code.

6.6 GeometryEngine

Collision detection is included in the GeometryEngine class. This is a class that assists several managers with geometrical related functions. Some of the functions used for collision detection are also used for determining if a tank is in direct line of fire of a given turret. These functions would need to have been coded anyway.

Thus, code is reused in two different major decision making functions. This is one of the reasons why SFML collision detection was replaced.

Another responsibility of the GeometryEngine was to handle logic for tank missile reflection. This is implemented as part of a major feature which does not form part of the basic functionality requirements.

7. MEMORY MANAGEMENT

Due to the extensive use of polymorphism, memory management forms a crucial part of the design. A game entity is created as a shared pointer and exists within the Game class object. In order to take advantage of polymorphism, each game entity pointer is added to a vector container. To ensure that correct memory management practices are adhered to, these entity pointers are cast as weak pointers.

These vectors thus contain weak pointers that are of a type that corresponds to an inherited property of game entities. For example, all collidable entities added to a vector of weak pointers of type ”Collidable”. It is from these vectors that managers can have access to relevant game entity objects.

Using weak pointers has several benefits in this case. Firstly, using weak pointers ensures that that the pointer and object get properly deleted. If the weak pointer gets deleted within a certain manager, all other pointers within other vectors will also be deleted. This ensures that the other managers do not try and work with pointer to objects that do not exist any more.

8. TESTING

8.1 Unit Tests

The Agile methodology was used for the development of the game. As such, testing occurred at multiple stages in the development life-cycle. This testing included normal gameplay testing, as well as unit tests. These tests took place at three different stages. Firstly, while doing exploratory use of SFML. Secondly, after basic functionality was implemented. Thirdly, once the final version of the game was completed.

This approach was taken to facilitate regressing testing. Using regressing testing ensures that added functionality does not break previously implemented functionality. As part of using an iterative approach, source code was revised and tested several times. Edge case testing was included for certain functions. Analysis of the unit tests not discussed in this report. Testing results are discussed in terms of the high level requirements for the game.

8.2 Final Game Functionality

All basic game functionality was implemented. In addition, two minor and two major features were implemented. The minor features include a scoreboard and game time-limit, as well as map editor functionality. One major feature includes the ability of tanks to move and fire and any direction, with missiles able to reflect off walls/barriers. The other major feature includes the existence of gun turrets which target both players. These turrets form part of the barriers and can be destroyed by either player. Players are only targeted when the tanks are within a certain range of the turrets and are within the line of fire.

8.3 Gameplay Testing

Several bugs and performance issues were highlighted during normal gameplay testing. The most noticeable issue is that a framerate drop occurs when there are many sprites in the game window.

One of the major bugs that exist in the game relate to the turrets and their firing at tanks. If the tank is behind the turret and at a certain rotation, the turret fires. After inspection of the source code, it was found that a certain function was not giving correct information. This information included the Cartesian coordinates of a game entity. The issue was not picked up in the unit testing. This is because the test cases for a rotated game entity were not included.

9. CRITICAL ANALYSIS

9.1 Performance

The performance issue mention in the the previous section is due to poor algorithm design. The collision manager checks iterates through every game entity and check it for a collision with every other game entity. The only exception is when the entities that are being compared are barriers. As a result , the collision detection algorithm is of order n^2.

9.2 Maintainability

One of the strengths of the overall game design is the modularity. The design effectively implements a modified version of MVC. Separation of the program layers has also been achieved. The design is flexible to the degree where new game entities or managers may be added with no changes to existing interface. The game is also potentially portable. Decoupling of SFML is to the degree where a different keyboard or display library could be used.

9.3 Use of Idiomatic C++

The final program made extensive use of idiomatic C++. Vector iterators were used instead of traditional integer based for loops. The auto keyword was also used in conjunction with these iterators. A distinguishing feature of the overall design is the use of smart pointers. These play a crucial role in the proper management of memory.

The code does not make use of templates or template functions. This caused several instances where the DRY principle was violated. Certain portions of code, although not explicitly repeated, could have been replaced with template functions. This would reduce the number of unnecessary functions.

Structs are used in many places within the source code. The structs fulfilled the needs of the software sufficiently well. However, the usage of classes instead of structs would have been better in terms of object oriented design. One of the reasons why structs were used in the first is because they are faster to implement than classes.

All the stucts are included in a single file. Any class that requires a struct simply has a header which includes that file. The problem with this is that the classes now have names of structs that are polluting its namespace. These classes now have a slightly limited selection of variable names.

9.4 Effective use of Inheritance

Inheritance is used effectively throughout the game program. This facilitated the use of polymorphism which is a strong point in the game design.

However, there is an instance where inheritance was not used correctly. A refused bequest has the potential to occur in one of the model classes. The barrier class inherits two functions from the Collidable abstract based class for which there is no implementation. These are the setBlocked and setUnblocked functions. A barrier is not capable of being blocked because it does not move in the first place. Only other game entities may be blocked by a barrier.

9.5 Monolithic Classes

The Game class can be considered to be monolithic in nature. It also does not conform to the standard of a single responsibility class. One of the reasons for this is due to the fact that there is a lot required of the Game class. Even though the class has multiple functions, most of these are private functions. In a sense, the Game class does have a single responsibility; to run the game world. However, the class could have been constructed to be more modular.

A major responsibility of the Game class is the creation of new entities. Each entity needs to be created as a shared pointer, then cast as a weak pointer and finally be added to each of the managers' vectors. This requires a fair amount of code.

9.6 Long Functions

A side effect of having a monolithic class is that it is prone to long functions. The Game class is not the only one with long functions. Some of the managers, namely Collision and Tracking, had long manage functions. These long functions violate the DRY principle in many instances.

The reasons for long functions can not only be attributed to poor design or forethought. Time constraints limited the amount of refactoring that could take place.

9.7 Coupling of Classes

The following of the MVC architecture ensured a loose coupling of classes and major program modules. An example of where tight coupling did occur was with the Orientation class. This class was developed so that SFML would not be used in the model classes. Also, it helped ensure that code was not repeated. The trade-off with this was that a tight coupling occurred.

10. RECOMMENDATIONS

The following recommendations relate to improving the existing game design. Changes to existing design are not given.

To address the performance issue a simple conditional statement can be included in the collision detection algorithm. More specifically, an algorithm exit conditional statement can be put at the beginning of the algorithm. If two entities that are being compared are further apart than the maximum sprite dimension, then collision detection does not need to occur. This means that object that are far apart with not be checked for a collision. The change does not require any changes to interface. All the required information for this additional conditional statement is accessible from with the algorithm scope.

The monolithic nature of he Game class can be reduced by including a Map class. This class would take away the Game class responsibility of processing map data. The inclusion of a Creation manager would also lessen the responsibility of the Game class. It would also absorb the long function which adds new game entities. The Game class would then have a single responsibility, that is, to run the main game loop. These changes will have no effect on other modules of the program. This is because only a small portion of the Game class consists of public functions.

Violations of the DRY principle can be considerably reduced if templates functions are used. Also, simply spending more time on refactoring the source code would reduce repetition.

The implementation of classes instead of structs is also recommended. This would not result in any changes to interface.

11. CONCLUSION

The final game design is based of an adapted version of MVC. Basic functionality with effective decoupling of SFML was achieved. The game software is in a position to be ported. Overall game performance was found to be slow due to an order n^2 collision detection algorithm. An additional early exit condition statement in this algorithm is suggested. The reason for a bug in the turret firing algorithm was found to be caused by the exclusion of a specific unit test. Suggestions are given on how to reduce the number of violations of the DRY principle. It is proposed that two additional classes be added to ensure the Game class is single responsibility.

REFERENCES

[1] M. Fowler. "GUI Architectures.", 2006. URL `http://martinfowler.com/eaaDev/uiArchs.`

html#HumbleView.

[2] "Arcade Perfect Port.", 2014. URL http://tvtropes.org/pmwiki/pmwiki.php/Main/ArcadePerfectPort.

[3] "Agile Methodology.", 2014. URL http://agilemethodology.org/.