Defender Project Document

Catherine Honegger 566247 and Christian Kamwangala 685344

School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg

Abstract: This report documents the design and implmentation of a PC based remake of the classic arcade game Defender. A layered object-oriented approach has been used to design the game implemented in ANSI/ISO C++ in conjunction with the SFML 2.3.1 graphics library. The application layers have been separated using a combination of the Model-View-Controller and the Model-View-Presenter design patterns. The 'model' layer contains the game entities and is modelled using inheritance. The 'view' renders all of the game objects. The 'controller' encapsulates all of the core game logic and links the 'model' and the 'presenter' layers. The 'presenter' creates a link between the 'controller' and the 'view' layers. Basic game functionality has been achieved with a scoring system, player lives, smart bombs, power-ups and a mini-map with screen-scrolling as extra major and minor features. Testability of the solution is increased by the use of separation of concerns. Design flaws in the model exist and lead to inappropriate class functionality, as can be seen in the Logic class. Possible functionality improvements could include the addition of Bombers, Humanoids and game levels. The design promotes the maintainability of the framework due to the high level of abstraction throughout the code.

Key words: Agile, Defender, Layered, Object-Oriented Design, MVC, MVP, Separation of Concerns, SFML

1. Introduction

Williams Electronics developed and released the classic arcade game, Defender in 1981. Defender was so popular that it has been released on numerous other platforms, including Antari 2600 and PlayStation Portable. The game became one of the highest earning arcade games of all time, and it inspired multiple other scrolling shooter games [1].

Defender is a side-scrolling shooting game set on the surface of a mountainous planet. The player flies a spaceship above this planet shooting at alien invaders. The main aim of the player in this game is to protect the humans from being abducted. In order to advance to the next level, the player must destroy all of the aliens and save all of the humans. The player is given three lives at the start of the game and once all of these lives are lost, the game ends. The player loses a life by colliding with aliens or their bullets.

This report provides a solution to a design of a computer based game called Defender Arcade V4.0 that recreates some of the features seen in the original Defender game using the C++ programming language and the SFML graphics library.

A brief project background including the requirements, constraints, and success criteria of the project is presented in Section 2.. An overview of the design techniques and game architecture is given in Section 3.. Section 4. provides the structure of the code presenting key abstractions and object interactions. Finally, the paper presents a critical analysis of the solution in Section 5..

2. Background

2.1 Project Specifications

The aim of the project is to design and create a PC-based adaptation of the Defender 1981 arcade game using C++.

2.2 Project Outcomes

It is expected that on completion of this project, the developers will be capable of [2]:

- Performing Object-Oriented (OO) analysis given software problems that involve numerous interacting objects.
- Designing and Implementing the solution in C++.
- Understanding and making use of existing software libraries in their own code.
- Providing a unit test suite verifying the behaviour of the software solution.
- Providing the necessary documentation for a software project, including an automatically generated technical reference manual.

2.3 Project Constraints and Assumptions

The project has the following constraints:

- The developed solution must be coded in ANSI/ISO C++ using the SFML 2.3.1 graphics library.
- The game must run on the Windows platform.
- The game needs to display correctly on a screen with a maximum resolution of 1600 x 900 pixels
- The use of OpenGL and any libraries that are built on top of SFML cannot be used for this project.

In addition to the constraints above, the developers need to meet the following Agile iteration deadlines:

- Simple game and test functionality - 7 September 2015
- Basic functionality plus one major feature - 21 September 2015
- Basic functionality plus two major features and two minor features - 4 October 2015

It is assumed that future developers may build upon the code, thus it is crucial that the solution is well designed, documented and easy to understand.

2.4 *Game Functionality*

The game has the following functionality:

1. The game entities that exist are: the player's ship, the player's laser, Landers and Lander missiles.
2. The player's ship can switch directions and move horizontally and vertically. The player's ship can fire its laser. The player's ship has four lives that are displayed on the screen.
3. Landers appear and move randomly in space and move. They are able to fire missiles directly at the player's ship in order to destroy it. Lander's are killed by the player's laser, and the player loses a life when the Lander's missile hits it, or when a player collides with a Lander.
4. There is a scoring system that displays the current score during the game, and the game's high score at the end of the game.
5. The player has three smart bombs that are displayed on the screen and destroy all aliens on the screen when they are released.
6. The screen is capable of scrolling and a mini-map exists at the top of the screen which displays the player, aliens, landscape and highlights the particular section which is currently being shown on the main screen.
7. The game has power-ups that appear and disappear. When the player's ship collides with a power-up, it shoots four heat-seeking missiles that kill the Landers.
8. The game ends when all Landers are destroyed, or if the player has no more lives.

2.5 *Success Criteria*

The final solution of the game is considered successful if it meets the following criteria:

- The functionality as listed in Section 2.4 is achieved before the deadlines as given in Section 2.3
- The game makes use of good OO practices including composition and inheritance.
- There is a valuable unit test suite that encompasses a great deal of test code coverage.
- The appropriate documentation is created, including: an adequate user manual, a technical reference manual, a testing report and a version history file.

3. Design Techniques

3.1 *The Agile Approach*

The design method used for this project is the Agile approach from the 2001 Agile Manifesto. The Agile approach to software development encompasses planning and documentation, as well as flexibility with regards to working. The method, like that of the spiral approach, is an iterative method that creates its final product based on multiple iterations of the Waterfall model given in Figure 1 [3].

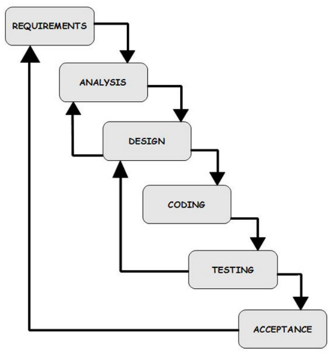

Figure 1: Waterfall Model

3.2 *Design Architecture*

A combination of layered and OO architectures was used in the project. Functionality within each layer is related by a common role or responsibility. Using a layered architecture allows for objects to be individually tested and manipulated within the code. This approach supports flexibility and maintainability of code [4].

In order to use an OO design, all entities of the game are identified and the behaviours of these entities are outlined in Table 1.

Table 1: Entities and their behaviours

Entity	**Behaviours**
Bullet	Moves left & right. Collides with Enemy entities.
Enemy	Moves randomly left, right, up & down. Shoots EnemyBullet entities. Collides with Bullet & PlayerShip entities.
EnemyBullet	Moves diagonally left & right, up & down. Collides with the PlayerShip entity.
HeatSeek	Moves in all directions. Collides with Enemy entities.
PlayerShip	Moves left, right, up & down. Shoots Bullet & HeatSeek entities. Collides with Enemy & EnemyBullet entities
PowerUp	Collides with the PlayerShip entity.

These entities have common behaviours and can be characterised by the same object information: an identifier, a position and an object size. Thus an OO approach using abstraction and class inheritance has been used to manage shared functionality.

3.3 *Separation of Concerns*

A fundamental software technique, is that of separating concerns, namely the presentation, logic and data layers of code. In software the presentation layer consists of the Graphical User Interface and its control, solely focusing on the user interaction. The logic layer encompasses all of the processing, logical decisions and calculations within the software. The data layer is responsible for the storage and access of the software's data.

The Logic layer acts as an intermediary layer between the presentation and data layers as shown in Figure 2.

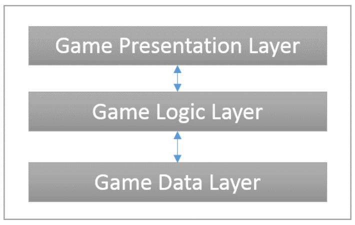

Figure 2: Software Design Layers

Decoupling of the three layers allows for independent implementation and testing which in turn improves software quality, usability and maintainability. The separated layers ensures that there is minimal overall impact of the software when code is changed.

Model View Controller/Presenter

In order to completely isolate the Presentation layer and the logic layer, a connection must be set up to communicate between the different layers. The two most common design patterns used to separate the presentation layer from the logic layer are: Model-View-Controller (MVC), and Model-View-Presenter(MVP). A combination of MVC and MVP design patterns have been used in this project [5].

Model

The 'model' defines how the data is represented. This corresponds to the data layer of the software. This is achieved in the code by making use of inheritance as the base classes have no knowledge of their functionality, they only store the state of the model, i.e their speed, object identifier, position and size. Their functionality is implemented by the 'controller'.

The Entity base class and its derived classes fall into the 'model' aspect of MVC.

View

The 'view' handles the visual representation of the data, the user inputs, and the updates of the game view state. This corresponds to the presentation layer of the software. The presentation layer receives commands from the player, and relays these to the 'presenter'. The presentation layer receives updated states from the 'presenter', and renders these on the screen.

The EventHandler and GameView classes make up the 'view' aspect of MVP/C.

Presenter

The 'presenter' communicates between the 'controller' and the 'view'. The GamePresenter class encompasses the 'presenter' aspect of MVP.

Controller

The 'controller' acts as the liaison between the data and the presenter. The 'controller' deals with the data, handles input processes, and completes the relevant actions. The logic layer is represented by the 'controller', and it acts as the Create Read Update Delete part of the system. The logic layer has no knowledge of the 'model', it only uses the data supplied from the model, i.e. positions and object identifiers.

The CollisionDetecton, Controller and Logic classes form the 'controller' part of MVP/C.

The complete decoupling of these layers allows the logic layer and the data layer to be independently tested from the presentation layer. This enables valuable unit testing to be implemented.

4. Structure of the Code

The Include Dependency Graph of main.cpp is shown in Appendix A Figure 8. This gives a good overview of how the classes interact with one another.

4.1 Utility Classes

Utility classes are interfaces that supply other classes with resources and functionality that they need rather than performing any specific game related functionality.

DataTypes

This header file defines essential enumerations and classes that are used throughout the game framework. One such essential class is the Vector2d class that creates a co-ordinate class of floats which is used by other interfaces to store object positions. The DataTypes class stores all variables that are used in more one class, this facilitates readability and makes it easier to modify variables in the code. The class gets passed up the architecture through inclusion and inheritance.

RandomPositionGen

The RandomPositionGen class generates a random Vector2d position within the game world boundaries. This class is used for PlayerShip re-spawning and for Enemy and PowerUp entities to appear randomly on the screen.

ResourcesHolder

The ResourcesHolder class is a helper class that deals with external resource management, specifically for the presentation layer. The class loads all resources the presentation layer needs to carry out functions. The interface loads textures of sprites and loads fonts using the SFML 2.3.1 library. The class maps object identifiers to their image. This allows the presentation layer to pass an object identifier to the ResourcesHolder and in return receive the corresponding image.

SfmlDataTypes

This header file stores custom typedefs for SFML data types used in the game.

StopWatch

The StopWatch interface provides simple timing operations. This class is used by the logic layer to ensure that a separation of the SFML library and the logic layer is upheld.

4.2 Model Classes

The following classes create the 'model' layer of the MVC architecture used.

Entity

The Entity class models all objects that are able to be rendered to the screen. The Entity thus forms the base class of the hierarchy used in modelling the 'model' layer. It is the class that encompasses all the shared attributes of data objects and is a polymorphic class from which specific entities inherit. The primary client of the Entity class is the Logic class. The Entity class provides all the stored information about the object to be rendered.

All derived classes must be able to get and set their position, rotation, and current direction. The classes must be able to return their speed, velocity and object properties (including object identifier and object size). The entity must be able to return 'true' or 'false' if they are active or inactive, and update their active property if they have been killed. The constructor initializes the velocity to 0, the rotation to 180 and the isActive boolean to true. All entities must be able to move. The Entity class inheritance graph is given in Appendix B Figure 9.

Bullet

The Bullet class models the player ship's laser. The class inherits from the Entity class, and is thus required to fulfil the contract stipulations of the Entity class. The Bullet needs to be able to update its position and update its status if it has collied with an Enemy or if it goes out of bounds. The Bullet needs to start at the same position as the PlayerShip.

Enemy

The Enemy class models the Landers of the game that are controlled by the AI. The Enemy class is derived from Entity and is thus required to honour the contract stipulations specified in the interface. The Enemy class must be able to move and shoot, update its position, and update its current status if it collides with a Bullet, or the PlayerShip.

EnemyBullet

The EnemyBullet class models the Lander's missile. The class inherits from the Entity class, and is thus required to fulfil the contract stipulations of the Entity class. The EnemyBullet must start at the same position as the Enemy. The EnemyBullet class calculates the velocity of the missile by using $\Delta Y/\Delta X * EnemyBulletSpeed$. The EnemyBullet needs to be able to update its position and update its status if it has collied with the PlayerShip, or if a certain amount of time has passed without collision.

HeatSeek

The HeatSeek class models the heat-seeking missile that the player ship shoots if it has picked up a power-up. The class inherits from the Entity class, and is thus required to fulfil the contract stipulations of the Entity class. The HeatSeek needs to be able to update its position, update its status if it has collied with an Enemy, and start at the same position as the PlayerShip.

PlayerShip

The PlayerShip class models the player's ship in the game and is controlled by the user. The class inherits from Entity and is required to honour the contract stipulations specified in the interface. The PlayerShip must be able to update its position, rotation and number of lives left. The default constructor takes in the initial position of the PlayerShip. The speed of the PlayerShip is initialized in the constructor list, and it remains constant. The number of lives of the PlayerShip is initialized to four in the constructor list, and the PlayerShip must be able to decrease its number of lives when it is killed until it ceases to exist.

PowerUp

The PowerUp class models the power-ups in the game. The class inherits from the Entity class and is thus required to fulfil the contract stipulations of the Entity class. The PowerUp needs to be able to update its position and update its status if it has collied with the PlayerShip, or a certain amount of time has passed without collision.

4.3 View Classes

The view classes deal with displaying objects from the data level, and relaying user input to the logic layer. The player interacts with the game using a qwerty keyboard. A use case diagram for the player is given in Figure 3.

GameEngine

The GameEngine's sole purpose is to run the game loop via a single start( ) method. The GameEngine instantiates the GameView class calling it's game loop functionality. The main reason for having a GameEngine class is to ensure that the player can play a new game immediately without having to exit the game completely and start again.

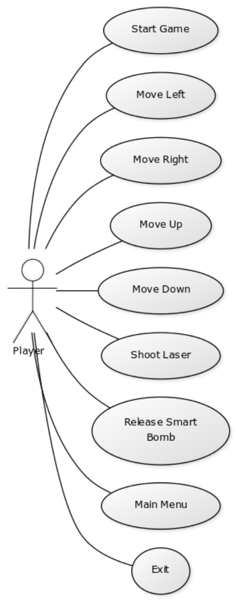

Figure 3: Use case diagram for game

GameView

The GameView class uses all of the other classes to create the game loop. There can only be one instance of the GameView class present in the game at all times to avoid memory leaks. The class provides a means of presenting the game world to the user by rendering all game objects to the screen. The interface also receives polled user input and passes this to the EventHandler class if the key pressed was not 'ENTER', 'M' or 'ESCAPE'.

GameView has a public function run( ) which starts the game loop. The class renders a welcome screen which allows the user to press the 'ENTER' key to start the game, or to press the 'ESCAPE' key to close the game. The class sets the variables and initialises the objects and positions of the sprites from the Game Presenter class.

The interface constantly checks the keys being pressed by the user and performs the corresponding actions as described above. The class updates its variables from GamePresenter during each iteration of game loop. The class then renders the screen and displays all sprites on the screen. This is completed by looping through a shared pointer vector of mapped object identifiers to sprites in ResourcesHolder. The GameView UML diagram is presented in Figure 4.

EventHandler

The EventHandler class provides information about user input commands to the Logic class. The class interprets user keyboard interactions via a polled event from GameView. The interface checks the keys being pressed and sends the corresponding enum Direction to the Logic class. The class acts on 'W', 'A', 'S', 'D', 'SPACE' and 'RIGHT SHIFT'. Using the EventHandler class to pass an

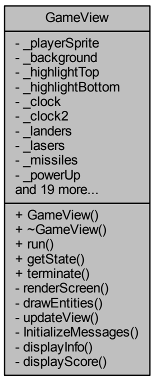

Figure 4: Use case diagram for game

enum Direction to the Logic class separates SFML keys from logic.

ScreenManager

The ScreenManager class handles all the screen specific operations and transitions. It deals with the creation and deletion of game screens, and it determines which credit splash screen to display. If the player wins or gets a high score, a different screen will be displayed than if the player was killed.

4.4 Presenter Classes

GamePresenter

The GamePresenter class is used to separate the display from the implementation. The GamePresenter serves as a link between the GameView class and the Logic class. This creates a flexible design, as logic has no idea what graphics library is calling it.

4.5 Controller Classes

The controller classes determine how objects behave given specific conditions. The classes ensure that a certain set of rules is adhered to, and the game state is essentially determined here.

Controller

The Controller class handles the movements of all Entity objects. The class ensures that all objects remain within the boundaries of the game world, and it updates the position of all entities in the game. The constructor takes an argument of a shared pointer of Entity type. The movements of the entities are achieved by accessing their speed and adding an offset in the direction of movement.

CollisionDetection

The CollisionDetection class detects if two entities have collided. The class returns a boolean of true if the two objects have collided. The CollisionDetection class makes use of the bounding box algorithm to check for overlaps between the bounding boxes (taking the top left corner co-ordinates, the object's width and height into account).

Logic

The Logic class overseas all of the game's processing. The class contains all of the information on the current object states in the game. This class creates all game entities in share pointer vectors and controls them through the Controller class. The main functions of this class are to instantiate game entities and their controllers, respond to user commands accordingly and to update the GamePresenter class with the entities to be displayed.

The Logic class makes use of the StopWatch class to shoot EnemyBullet entities after a certain amount of time, and to destroy these entities if they have not collided within a certain time frame. Using the StopWatch class upholds the separation of SFML and the logic layer. The Logic class also ensures that the player shoots the correct weapons when he has a power-up or not. The class enforces the rule that the PlayerShip is not able to fire a new bullet until the heat-seeking missile has hit an enemy.

Logic is able to check if entities have collided and change their status accordingly. If the PlayerShip and an Enemy collide the PlayerShip loses a life, and the Enemy dies. If the PlayerShip collides with an EnemyBullet, the PlayerShip loses a life, and the EnemyBullet dies. If the PlayerShip collides with a power-up the player's score increases by 1000 points. If the Enemy collides with a Bullet the Enemy and the Bullet die, whilst the player's score increases by 150 points. If a HeatSeek collides with an Enemy, both entities die and the player's score increases by 150 points. The smart bomb is a special case, as it destroys all Enemy entities present on the screen and only increase the score by 500 points no matter how many Enemy entities were destroyed.

The Logic UML diagram is given in Figure 5, and the include dependency graph is given in Figure 6

Score

The Score class is used to calculate the current score and store the high score. The interface increases the player's current score depending on certain actions. If the player kills a Lander their score increase by 150 points. If the player uses a smart bomb, their score increase by 500 points and if the player collects a power-up their score increases by 1000 points. The class is able to read a high score text file and determine whether the player has broken the high score or not. If the player has achieved a new high sore, the Score class writes the new high score to the text file.

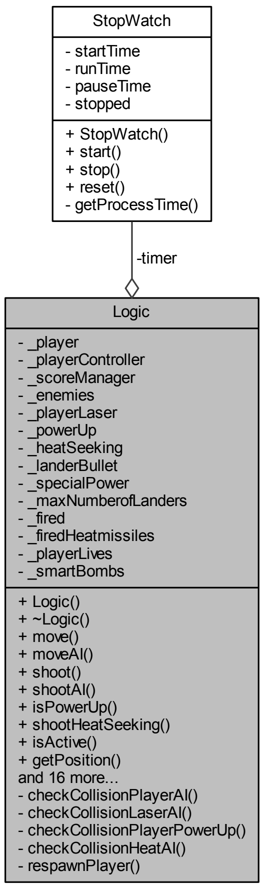

Figure 5: Logic UML Diagram

4.6 *Key Object Interactions*

The PlayerShip needs to be controlled by the player. The framework must therefore be designed to monitor for specific keyboard inputs from the player. The input must be processed by the GameView and EventHandler classes and the action transmitted to the Logic class.

The GameView class detects a keyboard event using the SFML framework. If the event is not 'ESCAPE' or 'ENTER', the EventHandler class will check if the key was 'W', 'A', 'S', 'D', 'SPACE' or 'RIGHT SHIFT'. If one of the required keys is pressed, the EventHandler will send the corresponding information to the Logic Class.

Figure 7 shows a sequence diagram for the dynamic class interaction between how the key input interpretation in EventHandler updates the Logic class.

5. Critique

5.1 *Functionality*

The designed solution's functionality meets the requirements as laid out in Section 2.4.

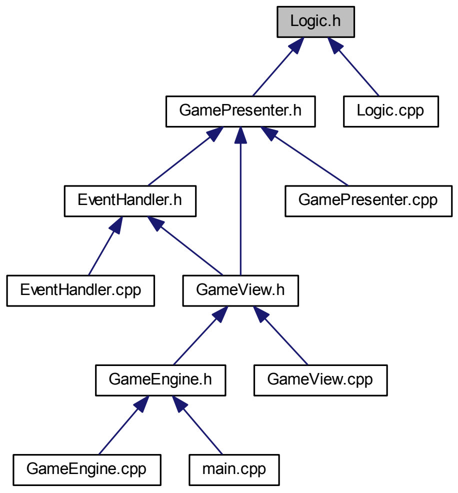

Figure 6: Logic include dependency graph

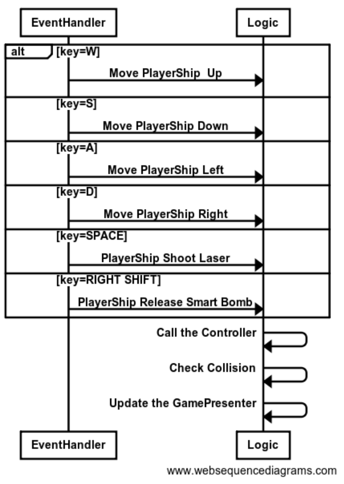

Figure 7: Key interpretation and Logic Sequence Diagram

The game has been tested extensively proving very few bugs to exist in the code. Despite achieving the required functionality, there is however sometimes a bug that when the player re-spawns the PlayerShip is able to exceed the boundaries by a small increment. Although the player can still be seen, it is a bug that needs to be fixed in further versions of the software.

5.2 *Design Analysis*

The code currently uses vectors which can be quite expensive during deletion of elements, thus for future development these vectors should be changed to lists. The Logic class needs to be re-factored as it appears to be a monolithic class. The inheritance needs to be extended as there are some classes that require the base class to have more functionality than originally anticipated.

The memory management of the code is well handled with the use of C++14 smart pointers. The code makes good use of separation of concerns by implementing a combination of MVC and MVP design patterns which enables the code to be more flexible, testable and easily maintainable. The code makes good use of re-usability through the use of inheritance to successfully model objects in the data layer. This prevents code duplication and results in smaller more manageable classes.

The Resource Acquisition Is Initialised rule is followed by ensuring that all properties of classes are initialized in their constructors and the use of destructors to released dynamic memory has been used.

5.3 *Maintainability*

Currently the code has some specific collision detection methods in Logic that would force more code to be written if a user wanted to add on to the functionality of the code. Apart from the Logic class, the design promotes the maintainability of the framework due to the high level of abstraction throughout the code. Inheritance allows for new functionality to be added to the game with little to no changes made to the existing code. Another example of the flexibility of the code due to the use of MVC and MVP is that the software can accommodate for easy switching of the graphics library.

5.4 *Future Improvements*

The game could be further developed to include different levels, Bombers, Humanoids and sound. The code could also be adapted to include an implementation of real-time keyboard input to allow for the spaceship to fly and shoot at the same time.

6. Conclusion

The solution presented was a success as it meets all the success criteria outlined in Section 2.5. The solution was developed iteratively making extensive use of C++ OO design to manage and control the data, logic and presentation layers and the SFML library is used to render the screen. The final design presents a very flexible code that allows for further development with little room for corrupting the code. Many improvements and bug fixes can still be made to enhance the code, but overall the solution has been implemented well and with its extensive testing and functionality the design has an upper-hand on its competitors.

References

[1] Wikipedia. *Defender (1981 video game)*. https://en.wikipedia.org/wiki/Defender_(1981_video_game), Last accessed: 03/10/2015.

[2] Levitt, S. *Project 2015-Defender*. 13/08/2015, University of The Witwatersrand.

[3] Fowler, M et al. *Manifesto for Agile Software Development*. http://www.agilemanifesto.org/, Last ac-

cessed: 03/10/2015.

[4] Microsoft. *Chapter 3: Architectural Patterns and Styles.* https://msdn.microsoft.com/en-us/library/ee658117.aspx, Last accessed: 03/10/2015.

[5] Emmattay, J. *Differences between MVC and MVP for Beginners.* http://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners, Last accessed: 03/10/2015.

Appendix

A Main include dependency graph

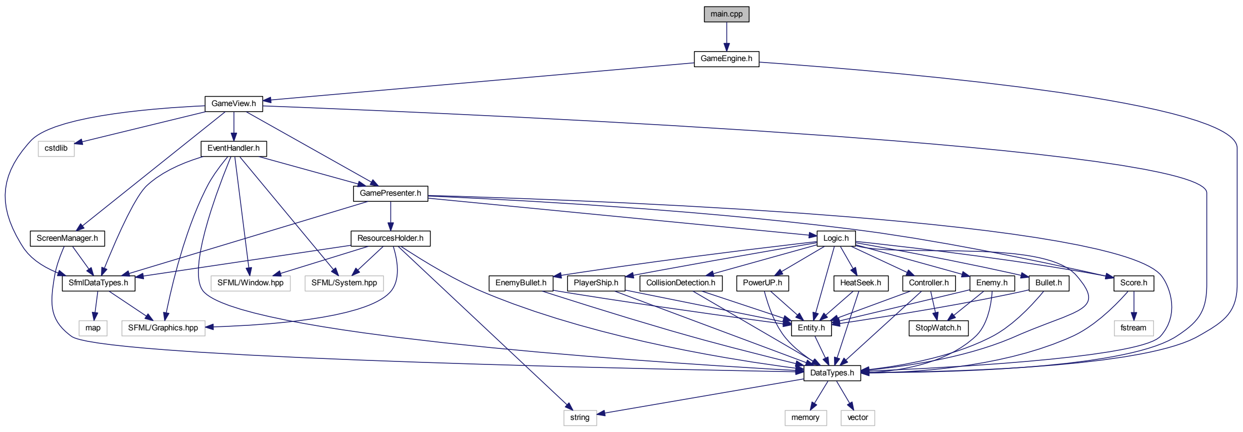

Figure 8: Main include dependency graph

B Entity class inheritance graph

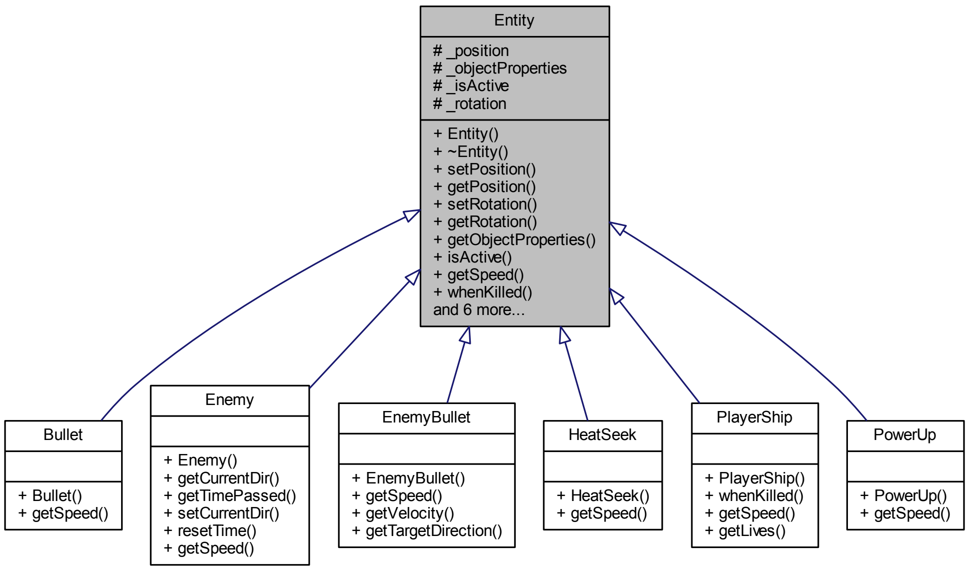

Figure 9: Entity class inheritance graph