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

ELEN3009 - Software Development II

”Alien Annihilation” Object Oriented Design Project

Group 37
Miroslav Minev (0500767K)
Andrew Russell (0506290K)

October 22, 2007

Abstract

A sound understanding of the design and implementation of Object Oriented programming is important to electrical and information engineers. This report documents the design and implementation of an arcade game using the methods of Object Orientated Analysis and Design. A low level breakdown of the written classes is discussed, paying special attention to the levels of abstraction and the hierarchy that exist in the design. The functionality of each seperate class with relation to how all the classes interact is analysed. A UML diagram of the class design is included as an Appendix which shows an in depth look into what functions and variables exist in the various classes. Analysis of the design reveals that the solution is prone to a number of risk issues including exceptions and buffer overflows - methods for minimizing these risks are suggested.

Contents

1 Introduction

The best solutions for projects follow a process of analysis of the requirements and developing a design that satisfies them - for projects of large complexity the methods of Object Oriented Analysis and Design (OOAD) result in the most efficient solutions [1]. It is for this reason that a sound understanding of the theory of Object Oriented software design and the techniques used to achieve high level abstraction is vital to the modern electrical and information engineer.

This report documents the design process, from the conceptual solution to the final implementation, of a version of Tubular Invaders titled Alien Annihilation, using the methods of OOAD. The solution documented in this report is aimed at not only fulfilling the specific requirements of the project but also addressing the broader problem domain of creating a design that is flexible and extendable.

Section two of the report outlines the success criteria of the project in terms of non-behavioral and behavioral (or functional) aspects. The conceptual solution is developed in section three while section four documents the high level entities that constitute major constructs of the solution. Section five describes the individual objects and object relationships used to implement the conceptual solution. Section six covers the conceptual and project specific benefits of the design. Section seven covers the risk issues and assumptions made and provides recommendations on how to minimise these risks. Section eight concludes the report and summarises the design.

2 Success Criteria

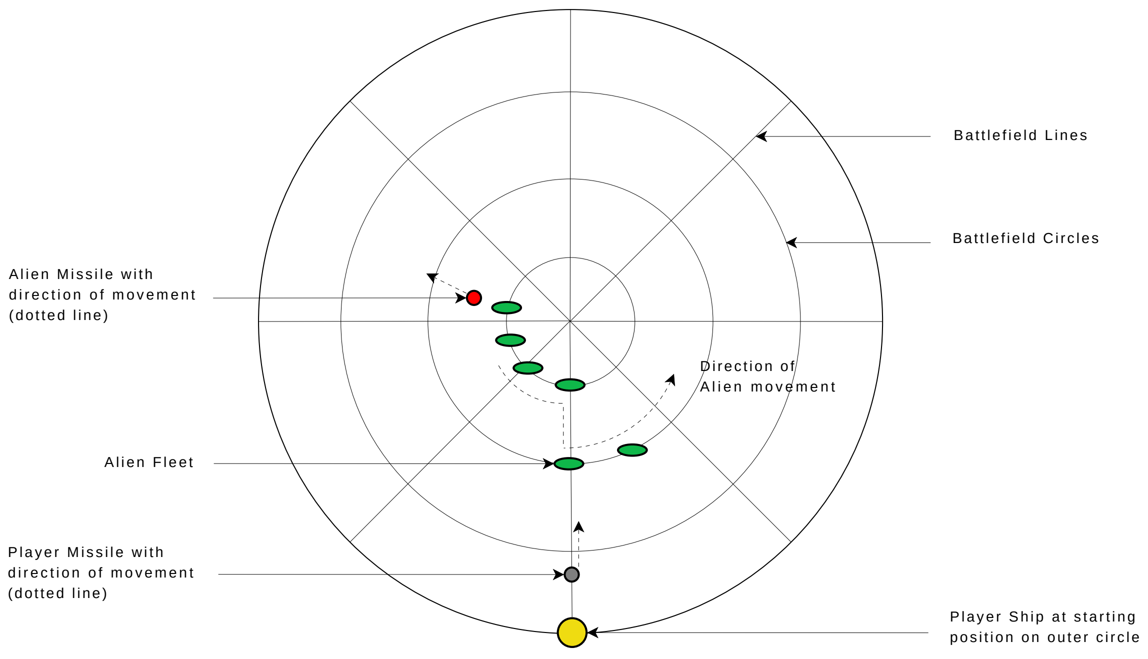
2.1 Non-Behavioral

A number of non-behavioral requirements must be met in order for the solution to the project to be declared successful:

- The game developed must obey certain fundamental rules.
- ANSI/ISO C++ along with the cross platform multimedia library Simple Direct Media Layer (SDL) are to be used to develop the project solution.
- Libraries built on top of SDL, other than the font and primitive graphics libraries available in SDL_gfxPrimitives.h, may not be used.
- The design is to make use of abstraction through Object Oriented code.
- A source code management system is to be made use of throughout the duration of the project.

2.2 Behavioral

A schematic of the required solution is shown in figure 1. The game consists of an armada of alien ships arranged along concentric circles around the centre of the battlefield. The alien armada start near the centre of the battlefield and move in a counter clockwise direction until each ship has completed a full revolution, after which each ship moves outward to the next concentric circle. The object of the game is for the player to shoot all the alien ships whilst avoiding alien shots and preventing an invasion. An alien invasion constitutes the situation whereby one of the alien ships in the armada reaches the outer most circle of the battlefield on which the player base is situated [2].


Battlefield Lines
Battlefield Circles
Alien Missile with direction of movement (dotted line)
Direction of Alien movement
Alien Fleet
Player Missile with direction of movement (dotted line)
Player Ship at starting position on outer circle
Figure 1: Schematic of Alien Annihilation in-game view

Collisions in the game are governed by three rules [2]:

- If a player's shot hits an alien both the alien and the shot are destroyed.
- If an alien's missile hits a player's laser base then both the laser base and the missile are destroyed provided that aliens have been equipped with the ability to shoot - as this is not a basic minimum requirement of the solution.
- If a player's shot hits an alien's missile both are destroyed.

The game-over condition is met when either the alien armada reaches the player ship base, or when the player ship is destroyed by an alien shot, or when the armada of alien ships is destroyed by shots from the player ship.

3 Conceptual Solution

The conceptual solution relates to not only the project-specific problem but also to the broader problem domain of arcade games in general. The conceptual solution has the responsibility of identifying the project requirements and forming a high level design that satisfies them.

In concept the following objects are needed to fully implement a 2D arcade game:

- A field of play - in this case the field of play is a circular battlefield; in the case of Pac-Man the field of play is a maze [3].
- A player character controlled by user input - be that single or multiplayer, and has some variation of fire power.
- Opponents or enemies - in this case the enemies are alien ships that use an algorithm for movement.
- Some variation of an object from which a "collision" can occur - Space Invaders has missile collisions while Pac-Man has a player colliding with food and enemies.

The conceptual solution to the project identifies and addresses all of the above game traits. The field of play is realized as a Battlefield object in this solution with all other objects in the game having some reference to it. The concept of a user input is addressed via a player ship object coupled with a class dedicated to streaming in input via the keyboard. The opponents in the conceptual solution are realized as alien ships - these alien ships are controlled by an algorithm which dictates their movement and ability to shoot. The final conceptual trait, that of collisions, is realized in the solution with the game being able to detect collisions between the alien missile objects and the player ship, player missile objects and the individual alien ships and finally between alien missile objects and missile objects shot by the player ship.

The conceptual solution developed deals with the broader problem domain in that the game structure can be extended to other arcade-type games without having to re-design the entire solution. The solution caters for a player that moves around the screen, has a specific position in relation to the field of play and opponents, and has the ability to launch an object with the intent of a collision with an opponent - in this case launching a missile to hit an alien ship.

The solution also caters for the implementation of other arcade games; it is however optimized for ship-type games such as Space Invaders and Alien Annihilation as opposed to arcade games such as Metal Slug which has human characters [4]. These types of games can, by all means, be extended from the solution developed but this would involve either modifying the *Ship/PlayerShip* class to have human attributes rather than only ship attributes or to derive a human character class from the ship class - the latter is counter intuitive. An alternative solution could be to deepen the level of abstraction by conceptualizing a *Player* class containing non specific properties such as coordinates from which a *PlayerShip* or *PlayerHuman* are derived.

4 High Level Entities

This section describes the high level entities of the solution. High level entities are objects, or groups of objects, that constitute major constructs of the design [5]. The high level entities of the solution divide the total of eleven classes that make up the solution into four groupings and form the bridge between the conceptual solution and the actual implementation in C++. The high level schematic shown in figure 2 in Appendix A and the UML diagram in figure 3 in Appendix B illustrate the high level entities and class structure of the solution respectively.

4.1 Interacting Objects

The first high level entity groups the classes that are of a character type and missile type. The role of this entity in the project is to create the platform for a user controlled character and for the AI of the game characters. The inclusion of missile-type objects allows for actual game play. This group consists of one base class *Ship* from which *PlayerShip* and *AlienShip* are derived. The *PlayerShip* class, when instantiated forms the user controlled character while the *AlienShip* class forms the basis for a single alien ship, or enemy which is controlled by an algorithm governing movement and shooting characteristics. The armada of alien ships consists of multiple instances of the *AlienShip* class stored in the vector STL container class. Access to the stored instances of the *AlienShip* class is achieved using the vector iterator. One instance of the user character *PlayerShip* has been implemented in the design; there is however functionality for the development of multiple user characters via the same method used to create more than one alien ship. Multiple instances of the *PlayerShip* would need to be initialized, stored in a vector and accessed via the vector iterator.

The design implements the concept of collisions in much the same way that the character objects are realized. The *Missile* class forms the base class from which two classes are derived - *PlayerMissile* realizes missiles shot by the player ship while *AlienMissile* realizes missiles shot by the alien ships. Instances of the player missile class are initiated when the user presses a specific "launch" key on the keyboard while instances of the alien missiles are initiated according to a specific algorithm. As in the case of the armada of alien ships, numerous instances of the two missile types are stored in separate STL vectors.

The method of creating numerous *Ship* and *Missile* objects is important as it allows for flexibility in the number of objects that can be created. This method also reduces the amount of code repetition as the *AlienShip* and two *Missile* classes are defined only once and then instantiated multiple times as opposed to writing out new definitions each time a new alien ship or missile is required.

4.2 Field of Play

The field of play is another high level entity that forms an integral part of the project solution.

The field of play is realized in Alien Annihilation as a Battlefield upon which all objects move. All these objects have their coordinates referenced to the Battlefield and know where they are on it. The Battlefield is designed with the intent of remaining as close as possible to the specifications given in the project brief as outlined in section two above. The Battlefield consists of concentric circles extending towards the outer perimeter of the screen while radial lines also extend outward from the centre of the screen.

The blueprint of the *Battlefield* is achieved with the use of two general structures namely *Circle*, which is characterized by a centre Cartesian coordinate and a radius, and *Line* which is characterized by a starting Cartesian coordinate and an ending Cartesian coordinate. Both structures share less-fundamental traits such as color and transparency. These structures are instantiated a number of times, each with specific Cartesian coordinates, and stored in separate *Line* and *Circle* vectors to achieve the total structure of the Battlefield.

These structures form the blueprint for a the field of play; it is important to note that they store only the properties of circles and lines which are then passed to the SDL graphics class - no drawing to the screen is done in this module of the design. This approach creates flexibility: the structure of the Battlefield is altered by simply changing the number of instances of the Line and Circle structures. Extending on this, the structure of the field of play can be altered by simply changing the structures that it comprises of, e.g. squares and lines for checkers while still retaining an x-y reference to other objects. This attribute is important as it enables the structure of the field of play to be limitless in design, allowing the same entity to be extended to other games regardless of the specific structure that is required.

4.3 Control

The control entity of the solution encompasses a number of classes, the most important of which is the *Game* class around which the *Input* and *Settings* classes operate. The *Game* class contains the game loop - the process of repeating over and over until the game is quit [6]. The game loop within the control entity has an administrative role and performs little logic - it merely calls internal methods from *Game* and methods from other classes in a specific sequence. The game loop operates in the following sequence:

1. Read and process an input
2. Update the elements of the game
3. Check for collisions
4. Display game elements
5. Check for victory/loss conditions

User inputs from the keyboard are streamed in and processed via the *Input* class. Similarly the settings for any instance of a game are streamed in and processed via the *Settings* class.

The control entity of the solution performs almost no logic itself - it calls member functions of other classes that perform a specific logic. This approach enhances the modular design and encapsulates a single idea of the conceptual solution. Two logic functions are performed within the *Game* class itself - one with the purpose of detecting collisions in the game and one to check for the game-over or victory/loss condition. An alternative is to define the collision detection function outside *Game* in a separate class with the soul purpose of checking for collisions. This approach is aimed at enhancing the object oriented aspect of the solution and removing responsibility of performing logic from the *Game* class. This approach also allows for easy extensions of the collision detect logic if the rules for which collisions are detected change.

When considering the logic function of checking for game-over conditions one may be inclined to say that this functionality should, like the collision detection be removed from the *Game* class. While this would enhance the modular structure of the solution it would not constitute as a benefit in terms of coding overheads as the function is relatively simple comprising of only a few lines of code. Separating this function from the *Game* class has merit in the event of the number of game-over conditions being extended to more than three.

4.4 Graphical Output

The final high level entity of the solution controls all aspects of the graphics of the game and maps the blueprints of all objects mentioned above to the screen. The *Graphics* class performs no logic functions; it only contains functions that accept properties such as x-y coordinates and colours and use the SDL library to draw these to the screen.

The graphics entity takes form in the game as a single class called *Graphics* and is important as it limits exposure of the SDL interface to a single class - no other class directly uses the SDL library. This approach enhances the object oriented aspect of the design and creates a separation between the logic of the game and the actual drawing to the screen. This approach also allows for flexibility of the graphics of the solution - if future application requires another graphics library to be used, for example Open GL or Direct X, only the *Graphics* class would need to be re-written as opposed to the whole game needing modification (if the SDL library were exposed in other classes of the game).

5 Lower Level Class Design

5.1 Game

All the fundamental game logic and control is implemented in this class. The game loop is contained within a function titled *RunGame*, which must be called from 'main' to begin the game.

Private methods perform actions such as shooting and updating positions depending on user inputs, check if any collisions have occurred, check if one of the three game-over conditions have been met, save the current game, and display the current game being played. These methods are private as they are only ever called from within *Game* itself.

The initialization of the *Game* class involves passing it two parameters - the integer values of the screen width and height. These parameters are then used to create the *Graphics* object. All the other game objects are created from *Game* once the player chooses the appropriate option in the main menu.

5.2 Graphics

The *Graphics* class uses the Simple DirectMedia Layer multimedia library with its primitive shapes for drawing circles, lines, boxes and text to the screen. All of the functions that draw various objects and shapes to the screen are contained within this class. SDL also provides for keyboard event sensing, which is implemented in a public function inside this class.

The *Graphics* class is constructed by passing it the integer values of the width and height of the screen that the game will be displayed on. In the constructor a pointer is assigned to the screen that is created, which is stored as a private data variable and is passed when any object is drawn to the screen.

5.3 Input

This class creates the interface between the user and the game and calls the functions of the *Graphics* class to check if a key is pressed. This is done as the functionality for checking keyboard inputs is contained within the SDL library. Depending on which key is pressed the Input class returns an Action enumeration to the *Game* object where it is handled.

No specific constructors are implemented since the class does not store variables, and the only logic that is exercised is in the functions.

5.4 Settings

The purpose of the *Settings* class is to read in parameters which influence the difficulty of the game from a text file and return them to the game to be used for level construction. This class is used only once and only by instantiating an object of it when the user specifies a level to be played. Passing no parameters to the constructor calls the default constructor which reads in the first settings text file. One can also pass the file name (e.g. settings2.txt) as a string to the constructor, which reads the parameters stored in that specific file. If the settings text file is not read in correctly or if it doesn't exist, an error is reported to the user, after which the game performs a restart. The parameters read in from the text file are the integer values for the number of circles and lines. This can be expanded to stream in more parameters for external control of the game and levels.

5.5 Battlefield

The *Battlefield* class contains the number of *Circles* and *Lines* that make up the field of play. Also contained is the radius of the outer *Circle*, which is used in limiting the battlefield to a certain radius from the centre of the screen. There are functions that use these variables to create organised vectors (in terms of radius or angle) of *Circles* and *Lines*, which are then iterated through in the *Graphics* class when required to be displayed. This is a more efficient approach to storing their characteristics as opposed to storing each one in its own separate variable or structure as they are all contained within one vector and any *Circle* or *Line* can be accessed via an iterator instead of having to search for or pass a specific variable every time.

Battlefield also contains the centre coordinates of the game screen which are used by many of the other classes to calculate their positions relative to the centre of the battlefield.

The *Battlefield* object is instantiated with integer parameters representing the size of the screen and the number of *Circles* and *Lines* to make up the battlefield. Using these, the constructor sets all the private data variables and creates the vectors of *Lines* and *Circles*.

5.6 Ship

This class forms the base of a general stationary spaceship that can be positioned anywhere on the battlefield. *Ship* forms the base class for player and alien ship since both are ship-types and require similar functionality and abstraction, such as state, coordinates, health, size, colour and constructors. The variables for state, positioning, health, colour and size are contained within this class, as well as functions for reducing the health and altering the state. Instances of the *Ship* class are not created as they would be useless in terms of the game interactivity since *Ship* contains no movement functionality.

The importance of the *Ship* class is that it reduces the amount of repetition that would've existed if each player or alien ship class had its own variables for coordinates, health and other properties. These properties can simply be inherited from the *Ship* class. This in turn introduces a level of abstraction where multiple ships of various characteristics can be created by inheriting from the parent class *Ship*.

5.7 PlayerShip and AlienShip

These classes inherit from *Ship* class all the variables that are used to store Cartesian and polar coordinates on the battlefield and colours and size of a certain *Ship*. These classes create a level of abstraction as they contain the movement functions specific to the *PlayerShip* and *AlienShip* individually. These functions are similar to each other in that they move the object along the circumference of the *Circles*, but the *PlayerShip's* movement is controlled by the user and the *AlienShip's* movement is automatic therefore each is unique enough to be in its own function.

An object of *PlayerShip* or *AlienShip* is constructed by instantiating it with parameters that represent its starting position, colour and size. The starting position coordinates are passed on to the inherited constructor of *Ship* which handles and stores them accordingly, while the colours and size are set within the class's constructor, since *PlayerShip* and *AlienShip* have different colours.

5.8 Missile

This class forms the base of a general stationary *Missile* that can be positioned anywhere on the field of play. *Missile* forms base class for player and alien missile since both are missile-types and require similar functionality and abstraction, such as state, coordinates, size, colour, and constructors. The variables for coordinates, colour and size are contained within this class, as well as a single function for altering the state.

An alternative to this design structure is to implement a base class from which *Ship* and *Missile* are inherited from as these types have similar properties. In implementing this approach a great deal of code repetition that exists in the *Missile* and *Ship* classes may be reduced.

5.9 PlayerMissile and AlienMissile

These classes inherit all the variables that are used to store its state, Cartesian and polar coordinates on the battlefield and its colours and size from the parent class *Missile*. They also create a level of abstraction from the *Missile* class as they contain the separate movement functions specific to *PlayerMissile* and *AlienMissile*, which differ only by the fact that *PlayerMissile* objects move towards the centre of the battlefield while the *AlienMissile* objects move outwards.

Separate functions are implemented for moving the missiles along discrete points on each of the *Circles* or continuously with no reference to the *Circles*. The implementation of two movement functions aids flexibility and allows for the dynamics of the game to be altered.

Player or *Alien Missile* objects are instantiated by passing parameters representing the starting point of the *Missile* i.e. from where it was fired to the constructor. These are obtained from the current coordinates of the ship using accessor functions defined in the *Ship* class.

6 Benefits

6.1 Conceptual

The conceptual solution makes use of abstractions that are flexible and not Alien Annihilation specific. This approach to the broader problem allows for implementation of the solution to other applications of a similar nature. The design also incorporates levels of abstraction which allow for the future extension of the solution without those developers needing in-depth knowledge of how the classes' functionality is achieved - all that would be needed is an understanding of how the present classes interact at a high level, knowledge of their actual implementation is not required.

The separation of the various roles that the conceptual solution fulfills is a benefit of the design as it means that should future application require that only one section of the solution be changed, only the class pertaining to that specific role needs to be re-written. An example of this would be a future application requiring a different media library such as Direct X as opposed to SDL - in this case only one new class would need to be written as the only class that SDL is exposed in is the *Graphics* class.

6.2 Project-Specific

The solution exceeds the requirements of the project brief with additional feature of the alien ships having fire-power. The AI which controls which alien ships shoot and at which times consists of two nested loops both of which contain random functions. The first loop generates a random number that determines when an alien missile should be fired while the second is used to select a random alien from the vector to fire the missile. This AI for the alien ships is used as it is simple to implement yet still creates an element of surprise in the game and remains effective in an application of such a limited size. The flexibility of the class structure allows for alternative algorithms governing the behavior of the alien ships to be implemented in future development if required.

The solution has the additional feature of having three difficulty levels: rookie, experienced and veteran. This additional feature extends the playability of the game. The level difficulty parameters that are variable are the number of circles and lines. These parameters, while simple to alter, extend the game play drastically. The modular structure of the design is important as the difficulty levels are handled by an individual *Settings* class - should future development require that the structure of the difficulty levels be altered only this class will need to be redesigned.

The solution has the additional advanced feature of being able to save at any point in the game and load the saved game at a later stage. This save and load functionality is achieved by streaming the relevant data to and from a text document. This feature provides flexibility and enhances the desirability of the final product to end users.

The solution is aesthetically pleasing using only primitive shapes - this is an effective example of how high-quality graphics of an application can be implemented without the use of libraries built on top of SDL.

7 Assumptions, Risk Issues and Recommendations

7.1 Exceptions

In general the solution overlooks the risks of exceptions - a fatal oversight that makes the game vulnerable to both internal and external exploits. A general approach to solve this problem is to define an *Exception* base class from which all specific exception classes are derived. Within these classes methods are defined that handle exceptions that are likely to occur in the solution. A second preventative measure is to enclose pieces of code where exceptions are most likely to occur within *try* blocks and deal with exceptions that are generated in *catch* handlers [7]. This method should report the exception to the developer and create a time-stamped log for future reference.

7.2 Buffer Overflows

The inherent risk of the solution comes from the fact that C and C++ provide no built-in protection against accessing or overwriting data in any part of memory. More specifically, they do not check that data written to an array or vector is within the boundaries of that container [8]. The solution is particularly vulnerable in this area as no buffer overflow checks have been implemented which results in the possibility of exploitation and memory mismanagement occurring. An important factor is that the design does not limit the number of alien ships, alien missiles or player missiles. All instances of these objects are stored in vectors - this results in the risk of buffer overflow occurring if the size of each vector is not managed properly. This is dangerous as it allows for the possibility of memory access exceptions and possible program termination. The same risk also poses a security threat as buffer overflows present the opportunity of altering the program flow and exploiting the entire system via the stack [9]. Numerous safe-libraries which have been developed for the purpose of buffer management in C and C++ should also be considered [8].

A player score is implemented in the solution. The score of the player operates in such a way that a constant integer (20 in the case of Alien Annihilation) is added to the running total each time a player missile collides with an alien ship. The container for this score is a signed integer that has no limitations - there is thus a risk that an integer overflow may occur if the score runs up to a high enough value. A method of protecting against possible integer overflow involves limiting the score to a certain value after which it is reset to zero. Setting the type of the score value to an unsigned integer also eliminates the unexpected case of the value ever reaching a value less than zero.

7.3 Save and Load Functionality

The application has functionality to save and load games. An initial assumption made was that the application will always have a saved game to load. This assumption does not account for the situation in which there is no saved game to load or if the saved game data - a text file containing the coordinates and health status of each alien - is not correctly located in the project folder or has been moved/deleted unknowingly by the user. This situation constitutes a load-error in the game. The latest version of the solution caters for this possibility by alerting the user to the fact that a load error has occurred and instead of terminating, the game is reset to the main menu.

Building onto the previous risk, the design of the save and load feature of the application operates under the assumption that programmers working on the application will not tamper with the layout of the saved file or the way in which the application streams information to and from this file. Should an uninformed programmer alter the dynamics of this saving mechanism, the application may act unexpectedly as the expectation is that the data that is streamed in from the file is the coordinates and health status of each alien in that specific order.

The above risk may be minimized with the use of the *assert* function to check if the data that is streamed in is within the bounds of the expected values. The *assert* function is optimal for this type of error checking - if a future developer changes the structure of the save file a compiler error will be generated directing him to the line of code that is performing the violation [10]. This approach limits any future programmers from deviating from the set-template of saving.

7.4 Game Speed and Delay

Testing of the game on a few computers of varying age and processor speed showed that the rate at which the game runs is inconsistent and different on all machines. SDL has a built in Timer that returns the number of milliseconds that pass since the initialization of SDL. This can be used to create a delay in the infinite game loop which permits the loop to run only after a certain amount of time has passed, thereby controlling the refresh rate of the game. One disadvantage is that the logic functions will also be delayed by the same amount and might slow the game down slightly more [11].

Another trade off of not having any delays in the code is that the processor dedicates itself to running the game at 100% speed and efficiency. This can be seen on the Windows Task Manager Performance chart (if running Windows) which shows the CPU current usage as a percentage. Once the game is started most of the CPU's that were tested get instantly boosted to over 70% usage. This results in overheating of the CPU and slower operation of programs running in the background, and may lead to CPU damage and malfunction in the long term. To reduce the usage and protect the CPU one can implement the SDL built-in delay function, SDL_Delay(), inside the frame-rate delay loop (as discussed previously). This function allows the processor to perform other tasks or to do nothing i.e. cool down, for a certain amount of time while the game is being delayed; and since these delays will be in the range of milliseconds they will be too quick for the human eye to notice [11].

8 Conclusion

The optimal solutions for projects follow a process whereby the requirements are analysed and a design is implemented that satisfies them - for projects of large complexity the methods of Object Oriented Design and Analysis result in the most efficient solutions. The solution developed makes use of object oriented design techniques to successfully implement a 2D arcade game titled Alien Annihilation.

The most valuable, non-functional benefit of the solution is the portability of the design to other arcade games of a similar nature. The solution meets and exceeds the project functional specifications with the additional features such as alien fire power and three difficulty levels being implemented. The solution also incorporates the advanced feature of saving and loading a game. A number of risk issues were investigated. It was found that the solution is particularly vulnerable to buffer overflow and causes over-usage of the CPU. Various methods of minimizing these risks were discussed with the most general preventative measure involving the development of a class that specifically handles exceptions in the program such as overflows. A standard SDL delay function can be implemented to give the CPU time to rest or to perform other necessary tasks for the operating system. This will reduce overheating keeping the processor in a healthy and controlled state.

9 References

[1] H. M. Deitel and P. J. Deitel. *C++ How to Program*, chapter 1, page 25. Prentice Hall, Upper Saddle River, New Jersey, 2005.

[2] Dr K. Nixon. ELEN3009 Software Development II Project Brief 2007.

[3] Pac-man. http://en.wikipedia.org/wiki/Pacman, Last accessed October 2007.

[4] Metal slug: Super vehicle-001. http://en.wikipedia.org/wiki/Metal_Slug, Last accessed October 2007.

[5] Scott Hackett. "How to Write an Effective Design Document". http://blog.slickedit.com/?p=43, Last accessed October 2007.

[6] Geoff Howland. "How Do I Make Games? A Path to Game Development". http://www.gamedev.net/reference/design/features/makegames/page2.asp, Last accessed October 2007.

[7] H. M. Deitel and P. J. Deitel. *C++ How to Program*, chapter 16, page 813. Prentice Hall, Upper Saddle River, New Jersey, 2005.

[8] Buffer overflows. http://en.wikipedia.org/wiki/Buffer_overflow, Last accessed October 2007.

[9] Aleph One. "Smashing the Stack for Fun and Profit". http://www.phrack.org/archives/49/p49-14, Last accessed October 2007.

[10] H. M. Deitel and P. J. Deitel. *C++ How to Program*. Prentice Hall, Upper Saddle River, New Jersey, 2005.

[11] Aaron Cox. "Getting the Time with SDL". http://www.aaroncox.net/tutorials/2dtutorials/sdltimer.html, Last accessed October 2007.

Appendix

A Game Entities

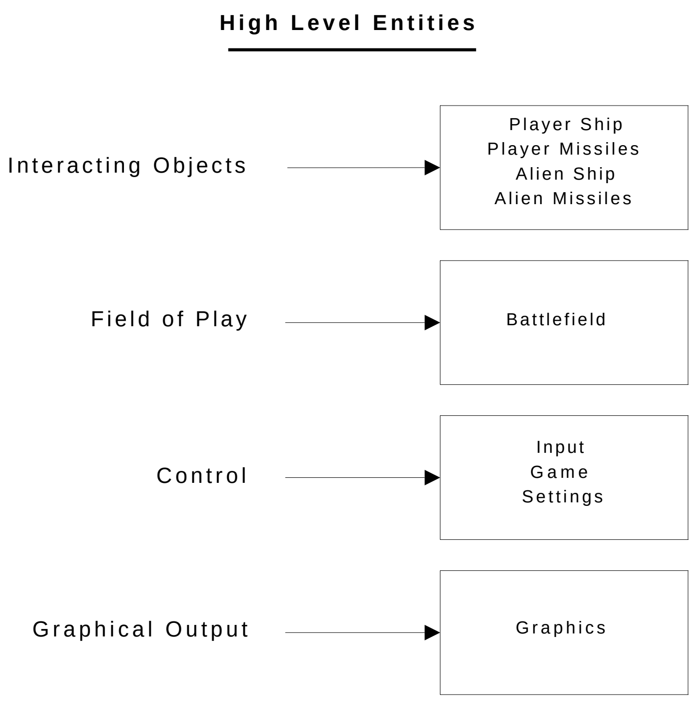


Figure 2: Schematic showing grouping of classes for high level entities

B UML Diagram