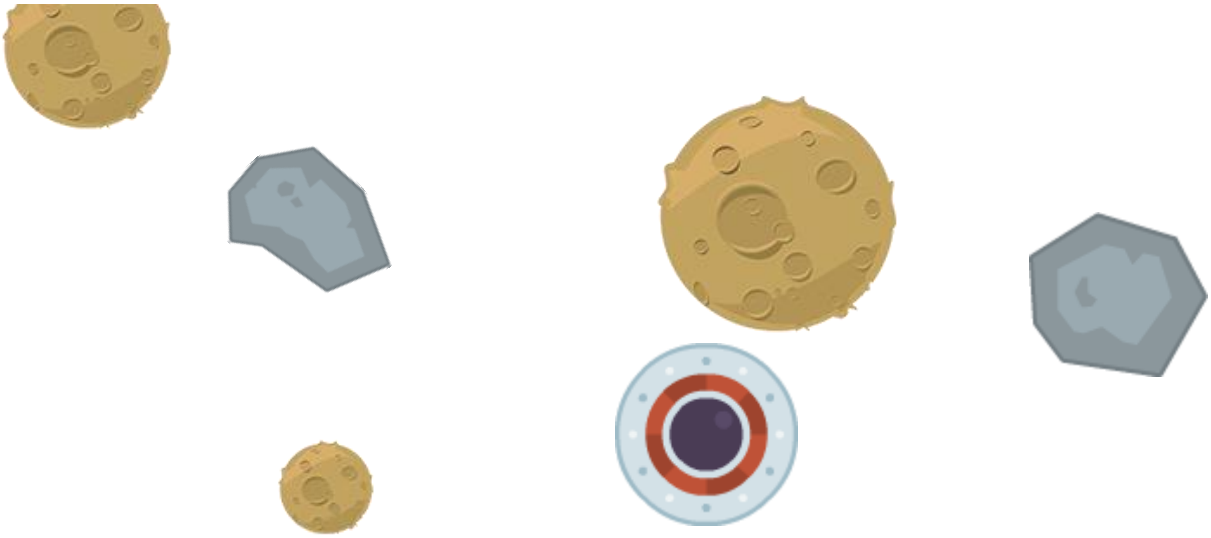
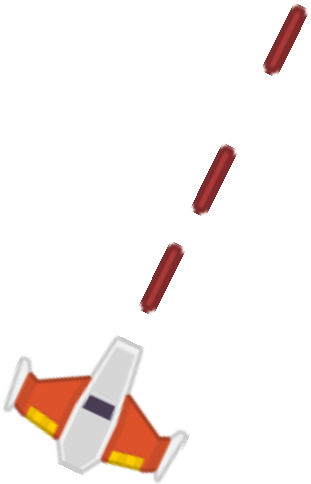


INM379: COMPUTER GAME ARCHITECTURES

Space Adventures

Flight shooter side scrolling game using XNA and C#



Filippos Karagiannis
Acnf581

Table of Contents

Overview

The game is developed using XNA and C#, based on tutorial templates taken from the course's labs. The aim of this project was to let me gain experience on producing a fully functional game demonstrating sound architectural principles in separating game engine and game code whilst implementing design patterns and data driven architecture. The game was developed for academic purposes and is not protected from copyright infringements.

"SPACE ADVENTURES"

"Space Adventures" is a 2-dimensional endless side scrolling shooter game. The game setup takes place in the galaxy, somewhere really far away from Earth. Zeus, the main character of the game, is a spaceship coming from planet Earth which tries to prevent alien races from invading into our planet and ravage humanity. The objective of the game is to collect as many points as possible whilst trying to survive from enemy attacks and incoming obstacles. The player can collect points by destroying enemy objects while he can maintain the spaceship health by picking up collectibles that regenerate a specific amount of health. The game is evolving on an increasing difficulty basis. That is, as the game time progresses, enemies become stronger and harder to destroy whilst their spawning time decreases gradually. As the game progresses, the player can benefit from different type of pickups, i.e. stronger weapon, giving the player the opportunity to last longer and score more points. Finally, the game maintains a high score table in order to introduce the competition element and make the game more fun to play.

0.1 The controls

The game uses basic keyboard controls.

ACTION	KEY
Main Menu	Backspace
Start Game	ENTER
Enter High Score Tables	"H"
Quit Game	ESCAPE
PLAYER CONTROL	
MOVE UP	↑
MOVE DOWN	↓
MOVE LEFT	←
MOVE RIGHT	→
Shoot	SPACEBAR ␣

0.2 Additions after presentation

At the moment of demo presentations, the game was half the way complete. I demonstrated the basic functionality and concept of the game and I pointed out my future implementations and plans for the game development. At the moment the game is complete concerning the project requirements, although there is still space for improvements and additions. The latest version of my game, compared to the demo version, now includes data driven architecture, FSM for both the NPCs and the game screens as well as event listeners and complete overall game-play and presentation with Main menu, High-score table and game over screen. The basic functionality of the level can be found inside the level.cs while the general game functionality can be found in game1.cs. The object animations, the background layers as well as the game menu screens all created using Photoshop CS5. A more detailed explanation of the implementation for each requirement is listed below.

Part 1.

1.1 Resource management strategy.

Resource management strategy is a very important aspect of the contemporary game development techniques. A well-structured resource management can ensure increased performance as well as more specialized control of game assets. Although one wouldn't probably notice any difference in games with small amount of assets, it is always a good practice to adopt such strategies. Owing to the nature of the game which include a few different objects and a single themed level, I am using two different content managers to load and unload the assets. The first content manager is used to manipulate the general game assets, that is, every asset that is not related to the gameplay (i.e. intro sounds, the menu screen textures etc.). In order to load the gameplay assets, I use a different content manager for the game level which handles the assets that are necessary for the gameplay. Once the gameplay is over, the level assets are unloaded properly.

1.2 Event-driven architecture for game controls.

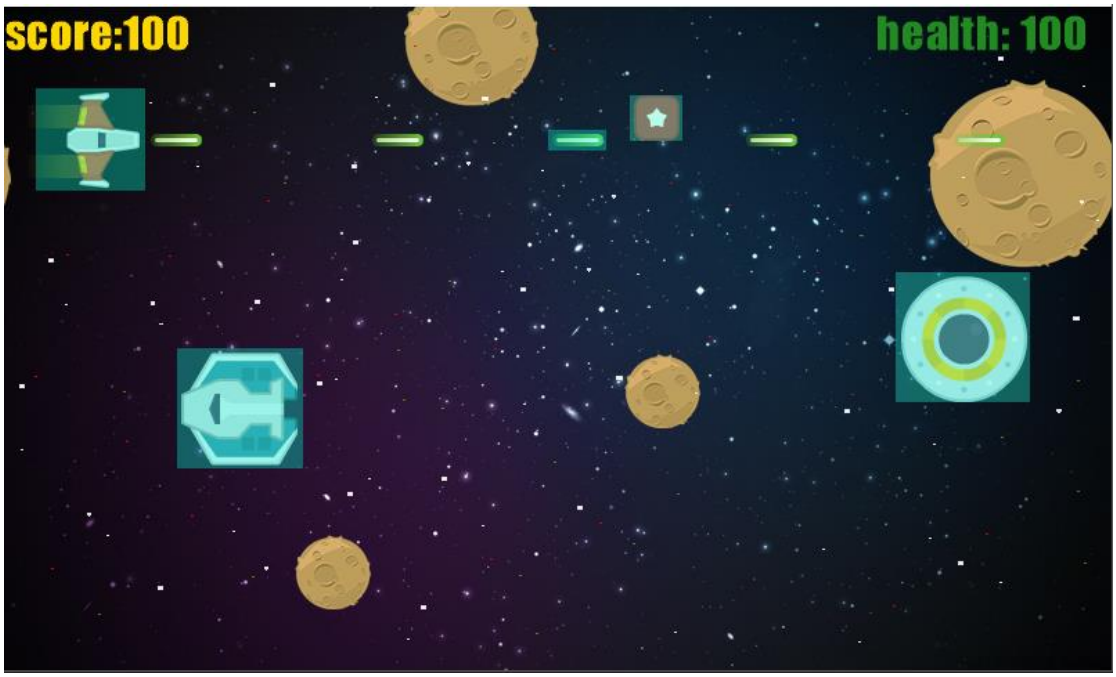
For controlling the player I use keyboard input controls. However, event driven architecture is implemented in order to separate the input hardware from the responding code. Based on the Chase camera example from the labs I created three classed that manage the input events. The KeyboardEventArgs.cs introduces the keyboard events and holds the previous state and the current state for each key passed in the constructor. In order to catch the keyboard events, I use the InputListener.cs class, where I define the delegates for each state of the keys. Finally,

the commandManager.cs class manipulates the thrown events. I created a hashSet of keys in order to store the keys I need to use in the game, and a keyboard binding list to store what each key does on each state. The key bindings are defined inside the Game1.cs class, so I can control the game menus before the level is started. I define the player actions inside player class, and I bind the corresponding function with the key I want inside game class. This kind of implementation abstracts the input system from the engine. XNA provides multiple input libraries, (touch screen inputs, Xbox controller inputs, mouse inputs) making it easy to add different input devices at any time by just defining the corresponding keys and binding them with the event functions.

1.3 Collision detection

The collision detection between the different objects of the game is based on basic brute force techniques. At the moment I create an object in the scene, I create a shape around the object, which will be responsible to interact with other object shapes and trigger collision events. On each update every active object in the scene is scanned for collision with the other objects, and when collision is detected, specific events happen. However, this kind of collision detection can be extremely expensive for the game performance, especially if the game scene is populated with a huge amount of collidable objects, and the engine has to loop every single one of them to check for collisions. Other than that, a collision event could even be missed if the number of collidables exceed the amount of objects that the machine can loop and check in a single update. In order to save computing power and avoid such detection failures, the game objects should be categorized, so the objects that don't need to be checked for collision at a specific update loop, will be skipped.

Due to the small amount of collidable objects in this game, the collision management is achieved easier. Based on the collision detection template from the course's labs, I categorized all active objects in the game scene as collidables. By doing this, I let them inherit the same collision test functions and on collision response functions. In order to manage collisions effectively, I introduced the CollisionManager.cs class. Collision manager creates a list which will hold all collidable objects in it. At the moment an object is created, I push it in the collidables list until it becomes inactive, either due to collision response or because it is out of game boundaries, when I remove it from the list. This enables the game to loop over active collidable objects only.



1.4 Moving and animated game elements with frame-rate independent loop control.

XNA utilizes a fixed framed rate loop of 60 frames per second. However, using frame dependent animations and forces can be disastrous for the game. More powerful machines can affect the animations and forces applied to objects if they can run in more frames than the machine used to develop the game. Likewise, a weaker machine would make the game run slower intoxicating the smoothness of the game with lags and spikes. A frame independent implementation is vital for games development in order to overtake the huge technological gap between the user machines. In order to achieve that I set the FixedTimeStep equals to false and used the GameTime as time parameter in order to update the object animations and forces. After successful testing using TargetElapsedTime = TimeSpan.FromSeconds(); and passing different time values I feel confident enough to say that the game is frame independent.

Part 2

2.1 Data driven approach

Being able to configure the game without having to mess around with the code at all is a huge benefit for a game developer. Data driven approach enables developers to modify and expand their games effectively by just changing the game values, game looks or even the game world itself from a single external file. An endless side scrolling game enables you to implement this approach by configuring the attributes of the game elements. To achieve that, I use a single XML file which includes the attributes from every single object in the game. From changing the variable values to choosing different textures, data driven approach let me configure the game world dynamically as the game progresses. For example, while I use a single class to create enemies (Enemy.cs), I populate the game scene with 3 different types of enemies by just passing different values in the enemy constructor from the XML file. Moreover, I managed to implement the progressive difficulty of the game by changing the enemy values according to game time.

2.2 Collision response based removal of game elements

Object collision and object disposal are two different functions that run separately and alongside. ResolveRemovals() function which lies in Level.cs is responsible for resolving the game elements. Every collidable element include a flag variable, flagForRemoval, which becomes true when the object needs to be removed after a collision event. ResolveRemovals() runs in every single game loop checking the collidable flags. When a collidable is detected with its flag equals true, the collidable object is pushed inside a new toDelete list. After that, the toDelete list is looped and every collidable object that is inside this list, gets resolved. The fact that all the elements of the game inherit from the same collidable class, makes this function more effective, since I don't need to differentiate between game elements, I just add them in the toDelete list and delete them no matter what their type is.

2.3 Scoring system demonstrating use of event listeners

Complex games may combine many different scoring occasions. The more the scoring occasions, the harder it is to manipulate them on collision management. In order to implement a clear and efficient scoring system, the use of event listeners seems unavoidable. Similar to user input management architecture, I introduced a score manager that handles the

score events that are fired after destroying an enemy. Enemies are the only objects that can throw scoring events at the moment. Whenever I create an enemy, I add an enemy binding in the score manager, which will notify the manager when a score event is thrown. Each enemy holds a Boolean field, isDestroyed, which will trigger the scoring event when the enemy is destroyed. Event driven scoring system, enables the game.Update() function to run faster while letting a separate manager take care of score updates. Finally, event driven architecture in scoring system, creates space for easy and effective addition of multiple scoring events. The scoring system implementation can be found in ScoreEventArgs.cs, ScoreListener.cs and ScoreManager.cs classes.

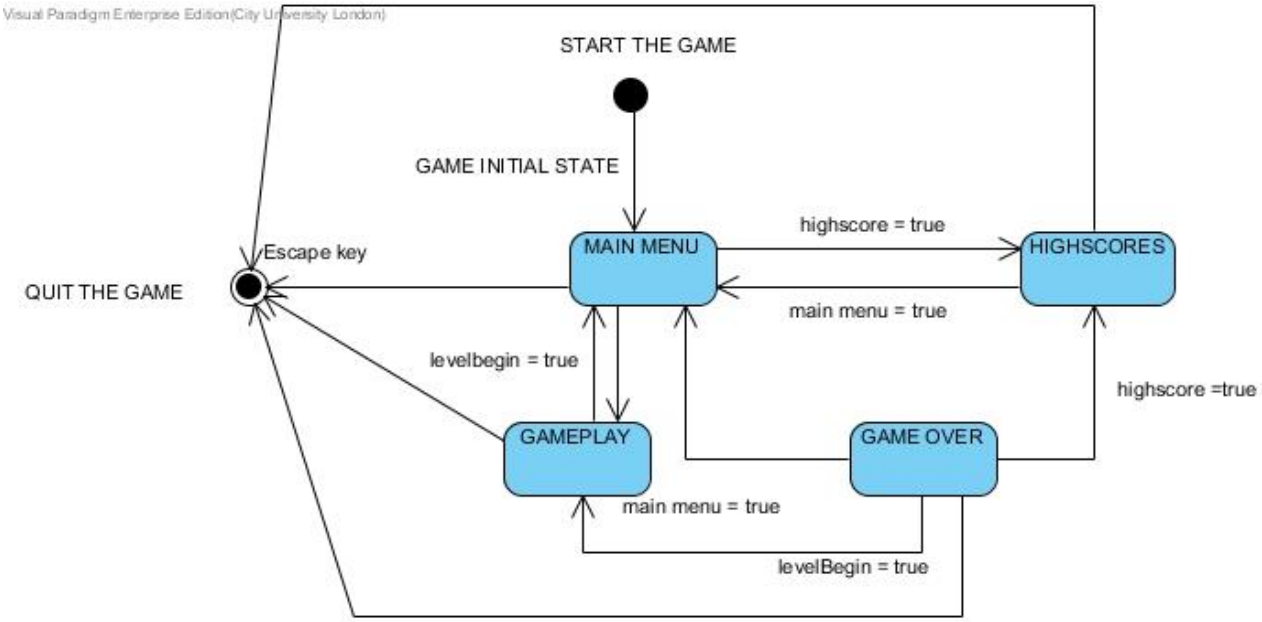
2.4 High Score table demonstrating use of serialization

Based on the platformer template from the labs, I took advantage of the Loader class to implement the high score table using serialization. The game reads and writes the high scores from a text file inside the content folder. I implemented a WriteToTextFile function inside Loader which let me write inside the txt file. Each time I need to update the high scores table, I read the txt file and then I update the entries. All the functionality can be found inside Game1.cs class where you can see the ReadHighScores/writeHighScore functions which I call in different states of the game menu.

Part 3

3.1 FSM with game loop for game Menu

In order to switch between the different game states effectively I implemented a state pattern for the Game to control the game flow. Based on the FSM template code from the labs, I created an FSM class as well as a State class. The FSM class handles the all the states I need to introduce. State class file consist of two classes, the State class and the Transition class. The State class provides template functions for the states which can be modified relating to the functionality I want to give on each state. Transition class provides me with the functionality to transit between different states. The Game menu consist of four different states. The Main menu state, which is the intro screen that shows the keyboard controls, the high score state, which is where the high score table is shown, the gameplay state, which is active when the level is running, and the game over state, which is active when the player dies and the game is over. While you can transit from main menu state to any other state, except from game over state, all the rest transitions are restricted. The initial state of the game is set to main menu. When the transition condition is met, the game state changes. There is a separate draw function for each state, so that only the current state's contents will be displayed. The full transition diagram is shown below:

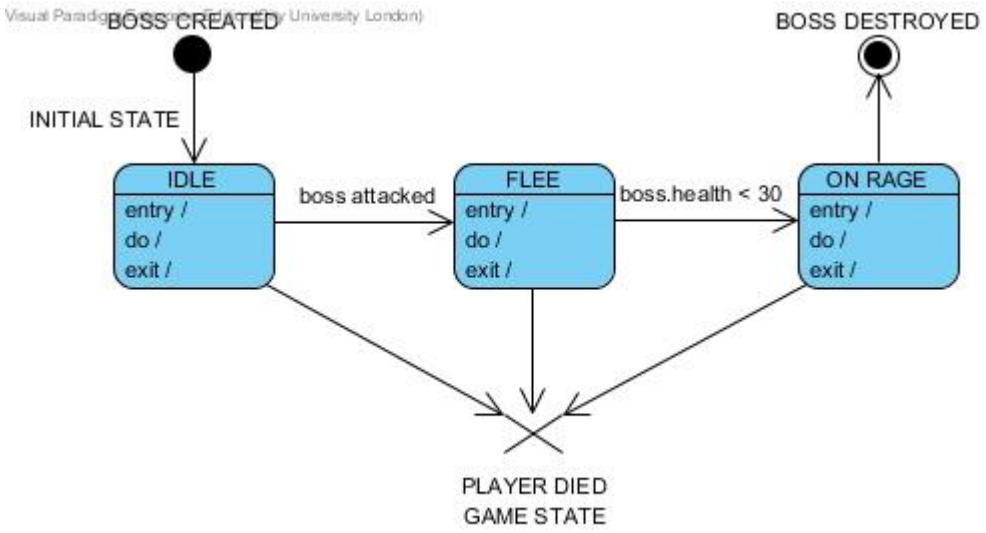


3.2 Power-ups

Power-ups are collidable objects as well. Collision detection and response is as discussed above in parts 1 and 2. When the player collects a power-up, the player health is updated with the amount of health I set inside the XML file. After colliding with the player, the pickup is set as inactive as well as flagged for removal. The inactive flag enables me to delete the pickup from the active pickup list while the flagforremoval enables the resolveRemovals function to take care of its colliding box. The same implementation can be applied for additional pickups, by creating new type of pickups flagging them differently, and introducing effects on collision with the player.

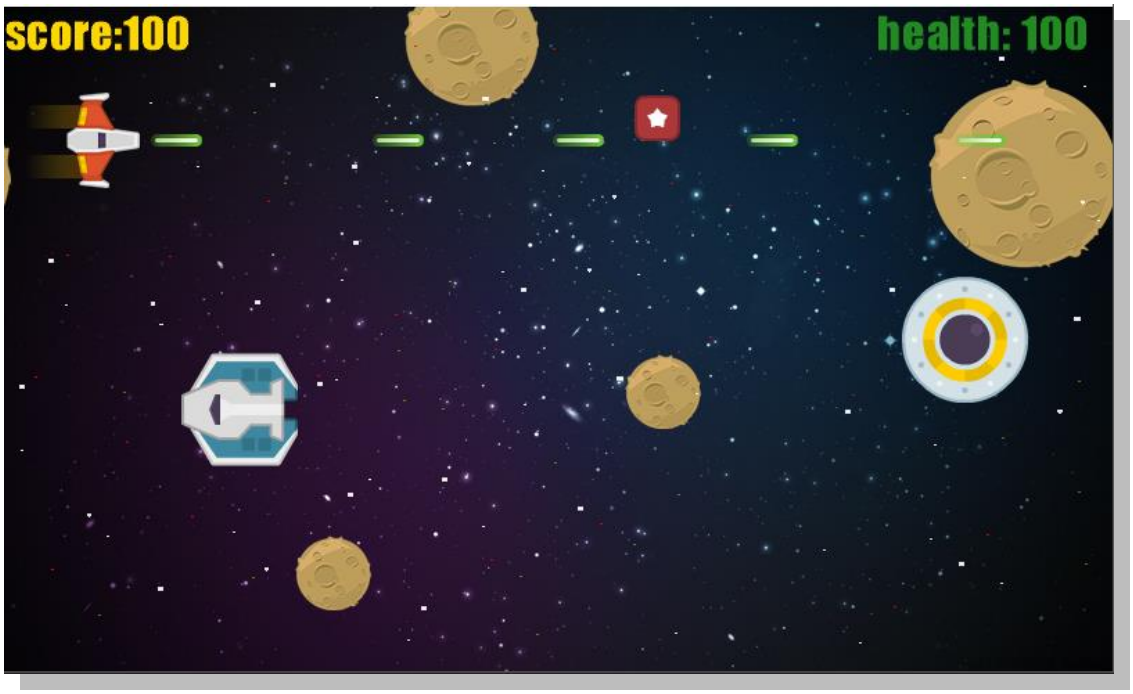
3.3 NPC opponents demonstrating FSM control of game objects

The object that is implemented with FSM control is the enemy type called boss. I generate different enemies passing different index values in the enemy constructor. Bosses have index value = 2 and are generated in the game after a specific amount of game time from level.cs. Bosses are the most powerful enemies and can only be one boss active in the game at any time. The boss incorporates three different states. The idle state, the flee state, and the rage state. When the boss is created the initial state is the idle state. You can notice the boss staying idle on its position waiting for a transition statement to get activated. Once the player attacks the boss, the flee state is activated, when you can notice the boss moving around the scene trying to hit the player. The boss stays always inside the window boundaries. In flee state execute function, the boss changes direction after a specific time. If the boss collides with the player the game is over. As soon as boss's health gets lower than a specific threshold, the onrage state is activated. In this state, the boss moves faster and changes directions earlier making it harder to destroy him. The interesting thing about the boss's states is that all the states transit one way. That means if the boss leaves one state, it can never go back to this state. The relating code can be found in Flee.cs, IdleState.cs and Rage.cs as well as the states initialization in enemy class for the enemy with index number 2. The boss state transitions are shown below.



3.4 Overall game play.

In order to make the game challenging and interesting I populate the scene with three different type of enemies. Enemy ship is the most common enemy object in the scene. It is generated depending on a time step which I set inside the XML file. The enemy ship is moving towards the player. The player can either kill the enemy ship or avoid it. The player receives the score value of the enemy ship only when it is destroyed. If the player collides with the enemy ship, the ship gets destroyed instantly and the player receives the value. However, the enemy damages the player. The other enemy on the game is the meteor. The meteor is moving slower than the ship, has higher health value and gives more points when destroyed. Similarly with the enemy ship, when the player collides with the meteor it gets destroyed. The meteor does higher amount of damage to the player. Last enemy on the game is the boss. The boss is a single enemy that is generated after a specific time step. The boss has the most life of all enemies and cannot be destroyed on collision with the player. Contrariwise, the player dies instantly and the game is over. Once the boss is generated, it never leaves the scene unless it gets destroyed. If the boss dies, the player receives a big amount of score. The last element that populates the scene is the pickups. The pickups restore a specific amount of health and only if the player health is below 80%. The challenges of the game is to score as high as you can. To achieve that you have to stay alive as long as you can. As the game progresses and the difficulty increases, the player will struggle to stay alive for long.



Part 4

4.1 Using profiling software to improve performance of the game engine

Nowadays, in order to create games with realistic graphics and interesting game play, companies in gaming industry use advanced graphics, game worlds with a huge amount of assets as well as complex AI algorithms. While these techniques lead into creating appealing games for the consumers, it is also a nightmare for game programmers to accomplish high performance and efficiently running code. Studies have shown that the human brain can realize difference in frame rates below 30 frames per second. That means that is necessary for contemporary games to be able to run all the game assets including graphics and sound as well as complex AI algorithms with at least 30 frames per second in order to deliver an enjoyable experience.

An interesting fact about computer programs is that usually the 10% of the code is responsible for the 90% of clock time is spent running the program. Thus, optimizing this part of the code would lead into remarkable increase of performance. Therefore, the most vital step for code optimization is finding which part of the code needs to be optimized. To achieve this, programmers use profilers.

Profilers is a form of dynamic program analysis that measures how much time is spent in each function inside the code as well as how many times each function is called at runtime. By using a profiler one can easily determine whether the low performance of a function depends on frequent calls of a function or the use of a slow algorithm. One can use profilers in many

different ways. Statistical profilers operate by sampling the program's call stack at regular system break points. Statistical profilers are less accurate because the resulting data are a statistical approximation. However, statistical profilers do not affect the execution speed of the program that much, enabling the target code to run at the same speed. Although statistical profilers' nature makes it difficult to capture the difference between an efficient function that is called often with a slow one that is called rarely, they remain a good option when the programmer needs a general view of the program performance however.

One can use instrumenting profilers in order to achieve more accurate results. This technique inserts instructions inside the functions that need to be monitored. Instrumenting profilers provide better control of what information will be gathered but can affect the program performance significantly. When using instrumenting profiler one need to be confident about what information will be collected as well as their level of detail in order to keep performance changes as low as possible.

5. REFERENCES:

Content:

1. Space shooter redux, game sprites and assets:
http://kenney.nl/assets/space-shooter-redux {last access 5/4/15}

2. Background layer 0:
http://www.allmacwallpaper.com/get/iMac-27-inch-wallpapers/Galaxy-Desktop-2560x1440/418-10.jpg {last access 3/4/15}

3. Background layer 1 &2, high scores background: originally created using Photoshop cs5.

4. Meteor used for layer:
http://space.customcards.biz/images/rg1024_Moon_in_comic_style.png {last access 5/4/15}

5. Explosion animation:
http://xbox.create.msdn.com/en-US/education/tutorial/2dgame/getting_started

Supporting materials:

6. Moodle lab tutorials:
http://moodle.city.ac.uk/course/view.php?id=16727

7. Xna support:
https://msdn.microsoft.com/en-us/library/bb203894.aspx, {last access 12/4/15}
https://msdn.microsoft.com/en-us/library/bb200104.aspx?f=255&MSPPError=-2147217396 {last access 12/4/15}

8. Sample code examples from “Rob Miles’ C# (Yellow) Book”, Rob Miles, Bananas edition 7 September 2015. : http://www.csharpcourse.com/ {last access 6/4/15}

Profilers:
http://everything.explained.today/Profiling_%28computer_programming%29/
https://en.wikipedia.org/wiki/Profiling_%28computer_programming%29