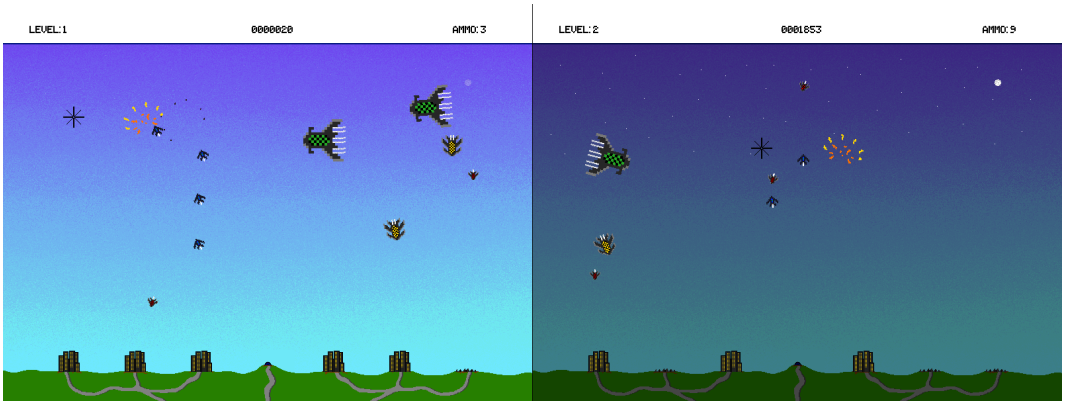
Missile Command

Final Report Assignment

John Segerstedt
sejohn@student.chalmers.se



Chalmers University of Technology
TDA572 - Game Engine Architecture
March 16, 2020

Contents

1 Introduction **1**

2 Gameplay **2**

3 Implementation Overview **4**
3.1 Main 5
3.2 Engine 5
3.3 Sub-systems 5
3.4 Utility 5
3.5 Game Objects 6
3.6 Components 7
3.7 External Libraries 7

4 Sub-systems **8**
4.1 Globally accessible sub-systems 8
4.2 Delegated sub-systems 10

5 Further Improvements **11**
5.1 High Scores 11
5.2 Collision Handling 11
5.3 Line-drawing 11

References **12**

1 Introduction

This report describes the implementation of the 2D game Missile Command, using the SDL2 (SDL n.d.) library and following the architectural design patterns and methods taught in the course TDA572 - Game Engine Architecture. The source code and executable files are submitted separately.

The original Missile Command was launched by Atari in 1980 for arcade machines and was later ported over to multiple other platforms such as the Atari 2600, Atari 5200, Atari ST, Nintento Game Boy, etc. A modern version of the game was announced in the beginning of March 2020, aimed at the mobile smartphone market (McWhertor 2020). In the game, the player is controlling a defence system in an attempt to defend cities from enemy projectiles. Further details on the gameplay of the original game can be found in Section 2.

2 Gameplay

Missile Command consists of the following different game objects:

- *Cities* - Six cities are spread out across the bottom of the screen. These are destroyed when being hit by enemy projectiles and the game is over when all six cities have been demolished.
- *Enemies* - Different forms of enemy projectiles travels across the screen towards the cities. These projectiles can be intercepted by player rockets. Enemy projectiles are spawned in waves, creating levels of increasing difficulties. The original game featured small particle sized missiles and later adaptations featured more advanced aircrafts that were able to spawn additional hostile entities while on the screen.
- *Rockets* - The player can spend ammunition to fire rockets at the incoming enemy projectiles as to hinder them from reaching the cities. All rockets are fired from the military defence facility positioned at the middle bottom of the screen and explode when reaching their target if they have not collided with anything on their way there. The explosion caused by rockets linger momentarily and can intercept more enemy projectiles during this duration. *Note - In the original game, ammunition is capped at 30 and resets between levels. In this implementation, as to allow for more enemies in each level, there is no reset of ammunition. The player starts with 10 ammunition and replenishes 1 after each enemy that gets destroyed.*
- *Crosshair* - Except for firing rockets, the only input the player has to the system is moving the crosshair, towards which the rockets are fired. In the original arcade game, the crosshair was controlled by a large trackball.

The game lacks a win-condition and therefore continues indefinitely until the loss condition is met; all cities are destroyed.

Figure 1: Image from the original Missile Command game
The six blue objects are cities. The three incomming yellow lines are particle sized missiles traveling towards the cities. The green circle is a defensive rocket that just exploded and the white brick is a crosshair with which the player aims. Player rockets are sent from the red pyramid shape in the center. The black box in the lower left and the smaller ones in the bottom center are representing player ammunition. Source: (World-of-Longplays 2013).

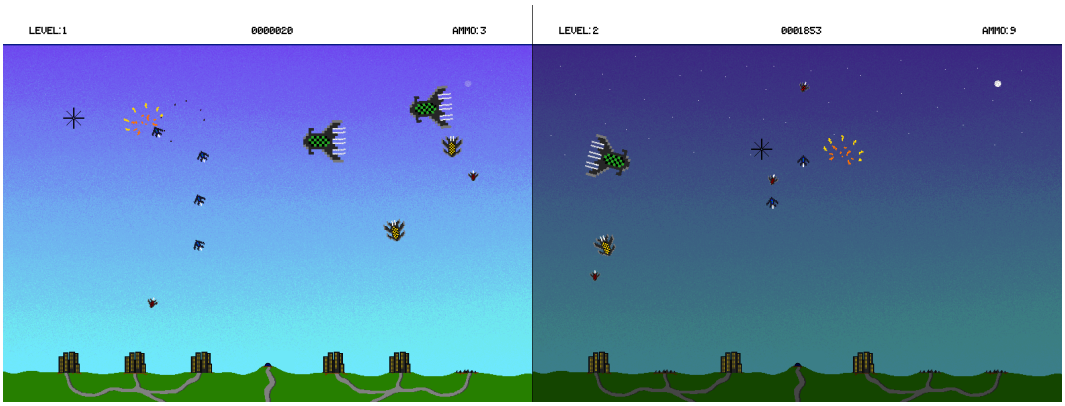


Figure 2: Image from this project's implementation of the Missile Command game
All original game objects are present: cities, rockets, crosshair, explosions, and enemies. This implementation features different types of enemies. See Section 3.5.

3 Implementation Overview

The implementation of Missile Command was based of the reference solution of *Lab 4 - Space Invaders* and therefore, the two share many similarities in the code architecture. Such shared code structure will be covered more briefly.

This Section aims to give an overview of the implementation architecture by briefly discussing each project class.

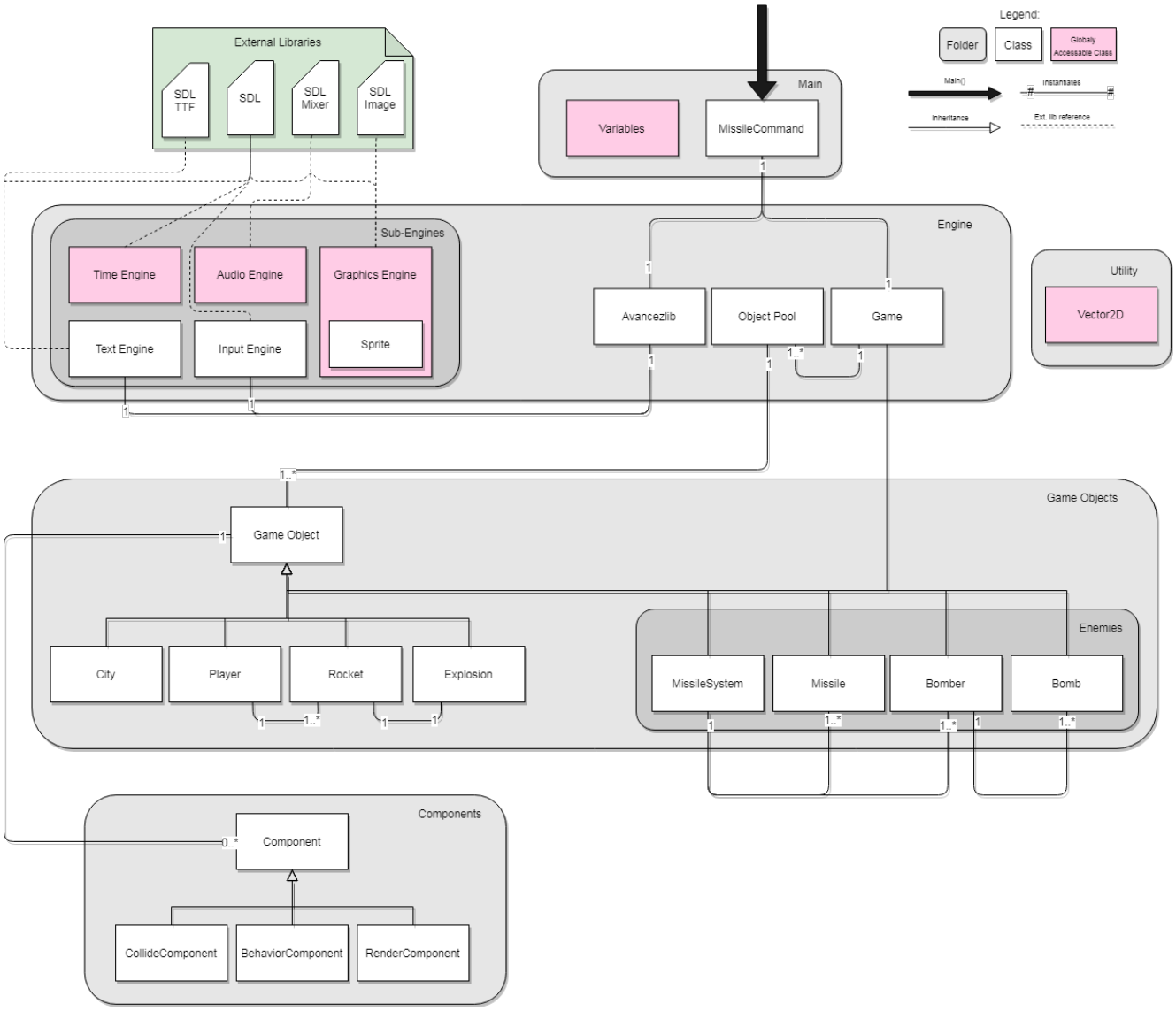


Figure 3: Bare-bones UML diagram of the Missile Command implementation

3.1 Main

The intent behind this folder is to group together the files that should be visible outside the project, if necessary. For an example, this could be the case if one where to implement multiple games within a games portal and wanted the ability to start a game with a particular set of settings.

The Missile Command class contains the Main() method of the project and therefore runs it and maintains the overall project game loop (Nystrom 2014b). It instantiates a Game and a AvancezLib object and gives a reference of the latter to the former.

The Variables file is a globally accessible .h file intended to contain constant input parameters that needs to be visible throughout the project, such as SCREEN_WIDTH and SCREEN_HEIGHT. The intent was to separate these variables as to avoid unnecessary coupling between classes. Additionally, by having all input variables neatly organized in a single file, it becomes much easier to potentially change them at a later date.

3.2 Engine

Here is where the main game and engine classes Game and Avancezlib lives.

The Game class is responsible for creating ObjectPools of all GameObjects and holds the reference to all activated GameObjects. When game.update() is called, all activated GameObjects are also updated (Nystrom 2014h). The Game class also utilizes the state design pattern (Nystrom 2014g) as to handle the game being paused, being over, being in start-up, etc.

Avancezlib is the main interface between the project and its sub-systems, see Section 4.

Here is also where the ObjectPool (Nystrom 2014c) lies.

3.3 Sub-systems

The intent behind these sub-systems is to be a separation of concerns of the Avancezlib engine. For more details on the sub-system architecture of the code base of the project, see Section

3.4 Utility

This folder contains all utility classes and data structures not inherent to any specific implementation area. The only utility class used in this implementation is Vector2D, based on the class with the same name in *Lab 5 - Spatial data structures*.

3.5 Game Objects

The GameObject represents each type of object visible in the game and contains the type of data all such objects share, such as size and position. GameObjects can have multiple different Components, see Section 3.6. Additionally, the GameObject is able to send and receive messages between each other following the observer design pattern (Nystrom 2014d). All GameObjects are allocated using ObjectPools (Nystrom 2014c).

The types of GameObjects are:

- *Game* - Is a GameObject as to be able listen in on the others' messages.
- *City* - Representing each of the six cities. Has two states; destroyed or not destroyed.
- *Player* - Represents the player's crosshair.
- *Rocket* - Defensive rockets shot by the player. Gets spawned by the Player's BehaviourComponent.
- *Explosion* - The animation when rockets explode. As the explosion still is collidable, it is considered a GameObject. Gets spawned by the Rockets's BehaviourComponent.
- *Missile System* - The main AI system. Spawns Missiles and Bombers in proportion to the current level.
- *Missile* - The red and smallest enemy. Spawns at the top of the screen and is shot towards cities or the ground.
- *Bomber* - The green and largest enemy. Spawns at the side of the screen and travels horizontally across it. Periodically spawns Bombs.
- *Bomb* - The yellow and middle-sized enemy. Has a fake gravity aspect to it which makes it curve downwards.

3.6 Components

Components (Nystrom 2014a) are small classes that are attached to a GameObject. Multiple Components may be attached to a shared GameObject. When a GameObject's .update() method is called, it calls each of it's components' .update() methods. Each Component has a clearly defined subset of functionalities. The three used in this project was:

- *Render Component* - Holds the sprite of the GameObject and renders it at the GameObject's position when the Component's .update() is called.
- *Collision Component* - Holds a reference to the ObjectPool of GameObject that the attached GameObject can collide with. Checks collision between these each frame. Hitboxes of all GameObjects are defined as circles.
- *Behaviour Component* - Handles movement and rotation of the GameObject. Also in charge of spawning other objects, such as the Bomber Behaviour spawning Bombers.

3.7 External Libraries

SDL, Simple DirectMedia Layer, is being used for input controls and time management (SDL n.d.).

SDL_Mixer, a sample multi-channel audio mixer library, is used for all audio management (SDL mixer n.d.).

SDL_Image, an image file loading library, used to manage sprite PNG:s (SDL image n.d.).

SDL_TTF, a TrueType fonts sample library, used for writing text on the screen (SDL ttf n.d.).

4 Sub-systems

The intent behind these sub-systems is to be a separation of concerns of the Avance-zlib engine. Consider Figure 4, the idea is that each of the engine parts would be represented within the code as a single sub-system class; a GraphicsSystem, AudioSystem, InputSystem, etc. By separation-of-concerns, one avoids coupled code and more easily allows for adjustments to other implementations or libraries. For an example, if the functionality of how audio is played is to be changed, only the AudioSystem class needs to be updated.

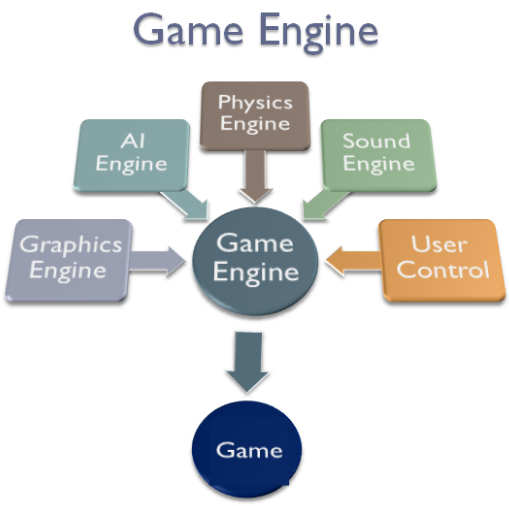


Figure 4: Bare-bones UML diagram of the Missile Command implementation

The five sub-systems in the implementation can be devided into two groups; globally accessible sub-systems and delegated sub-systems, each with their own benefits and limitations. The reason why this project features both types of sub-systems is not just because one isn't inherently better than the other, but to also show that both are viable options as proven by this project. If one of the two types of sub-systems would be preferred, refactoring is not a major issue.

4.1 Globally accessible sub-systems

These sub-systems are singletons (Nystrom 2014e), meaning that only a single instance of each are allowed at any point in time. Additionally, these are all publicly available to other parts of the code base in the form of public getInstance() methods.

```
void Explode() {
    Explosion * explosion = explosion_pool->FirstAvailable();
    if (explosion != NULL) {
        explosion->Init(go->horizontalPosition, go->verticalPosition, go->orientation);
        game_objects->insert(explosion);

        AudioSystem* audio_system;
        audio_system->getInstance()->PlaySound(ROCKET_EXPLOSION);

    }
}
```

Figure 5: Example of globally accessible syb-system in use

The benefits of globally accessible sub-systems is that their functionality is available wherever they are needed. However, there is no control over which parts of the code base is allowed to call on these sub-systems. Therefore it is important that the public interface of these sub-systems do not share memory allocations or include non-constant data variables.

The three globally accessible sub-systems are:

- *TimeSystem* - This class was inspired by the globally accessible Time class in Unity (Unity 2019). When writing in C# in Unity, one often needs to call Time.deltaTime as to get the last frametime. However, separating all time management within this project became almost completely obsolete as the last frametime is being sent as a parameter to each Update() functions of every GameObject and Component. However, if the code base ever is to be refactored, then this class neately handles all time-related implementations. Currently uses SDL, see Section 3.7, to keep track of time.

- *AudioSystem* - The audio sub-system has a public enum class of sounds it is able to play, e.g; ROCKET_SHOT, CITY_DESTROYED, etc. By using the PlaySound(Sound) method, any part of the code can easily play any of these sound effects whenever appropriate, see Figure 5. The AudioSystem uses SDL_Mixer to play audio, see Section 3.7.

- *GraphicsSystem* - This sub-system is in charge of any and all sprite rendering on the screen. The internal class Sprite uses SDL_Image for simple sprite rendering, See Section 3.7.

4.2 Delegated sub-systems

In contrast to the globally accessible sub-systems, these are neither singletons nor able to be references globally. Delegated sub-systems are code only intended to be accessible from the AvancezLib engine class, which therefore acts as an inherent interface between the sub-systems and any external actors.

For an example, for a class to be able to print text on the screen using the TextSystem, then it needs to have been given a reference to the AvancezLib instance. This allows for more control over which classes has access to the sub-systems as they specifically need a reference to the engine class in advance. However, by making it harder to get access to these sub-systems one also introduces the issue of having references to the highly vital engine instance sent around the project.

The two different delegated sub-systems used in this project is:

- *TextSystem* - Can be used to print text messages on the screen on a given pixel coordinate. Uses SDL_TTF, See Section 3.7
- *InputSystem* - Can be used to see what key bindings are currently being activated, such as: UP, DOWN, LEFT, RIGHT, FIRE, PAUSE, etc. The mapping of bindings to keyboard keys are separated into the Variables file as to more easily allow for the possible future extension of custom user key bindings. Uses SDL for key fetching, See Section 3.7

5 Further Improvements

In software, no implementation is perfect. This sentiment also includes this project. Following are some aspects the project could be further improved and extended.

5.1 High Scores

An essential feature of arcade games is the fact that they have a leaderboard where players submit their high scores to compete. For an online leaderboard, one would need to implement a database and ensure that one's arcade machines will have access to it; something that was not possible back when arcades were popular. For a local leaderboard however, a game only needs to write down the high scores in a text file between game sessions to be restored at next start-up.

Currently, there is no saving of high scores between runs in any form and this would be a clear extension for the project with a direct user value.

5.2 Collision Handling

Currently, all sprites are visibly somewhat circular and therefore all collision considers each collidable game object as a circle. This simplification of the collision handling was also deemed sufficient for the scope of this project. However, for a commercial release in a professional environment, then a more accurate collision handling needs to be implemented.

Additionally, the current implementation of collision is rather naïve as it checks all active enemies versus the player crosshair and each city. However, since there are so few game objects on the screen at any point in time, it is not clear whether using, for an example, a spatial data structure (Nystrom 2014f) to separate game objects into would actually provide a benefit in computation time.

5.3 Line-drawing

In the original game, enemy missiles where followed by smoke trails. This was deprioritized during the implementation. However, if one would want to make a more accurate clone of the original, one would need to use line-drawing to draw the smoke trails. Another solution would be have smoke sprites being rendered behind each missile in a line, simulating a smoke trail.

References

McWhertor, M (2020). *Atari revives Missile Command with 'modern take' for mobile*. Accessed on 12 March 2020. URL: https://www.polygon.com/2020/3/3/21162288/atari-missile-command-recharged-android-ios.

Nystrom, R (2014a). *Game Programming Patterns - Component*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/component.html.

— (2014b). *Game Programming Patterns - Game Loop*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/game-loop.html.

— (2014c). *Game Programming Patterns - Object Pool*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/object-pool.html.

— (2014d). *Game Programming Patterns - Observer*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/observer.html.

— (2014e). *Game Programming Patterns - Singleton*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/singleton.html.

— (2014f). *Game Programming Patterns - Spatial Partition*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/spatial-partition.html.

— (2014g). *Game Programming Patterns - State*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/state.html.

— (2014h). *Game Programming Patterns - Update Method*. Accessed on 12 March 2020. URL: http://gameprogrammingpatterns.com/update-method.html.

SDL (n.d.). Accessed on 12 March 2020. URL: https://www.libsdl.org/.

SDL image (n.d.). Accessed on 12 March 2020. URL: https://www.libsdl.org/projects/SDL_image/.

SDL mixer (n.d.). Accessed on 12 March 2020. URL: https://www.libsdl.org/projects/SDL_mixer/.

SDL ttf (n.d.). Accessed on 12 March 2020. URL: https://www.libsdl.org/projects/SDL_ttf/.

Unity (2019). *Unity Documentation - Time*. Accessed on 12 March 2020. URL: https://docs.unity3d.com/ScriptReference/Time.html.

World-of-Longplays (2013). *Atari 2600 Longplay [005] Missile Command*. Accessed on 12 March 2020. URL: https://www.youtube.com/watch?v=uJijGLGHRTE.