Modeling HUD Layouts for Selection Based Video Games

MODELING HUD LAYOUTS FOR SELECTION BASED VIDEO GAMES

BY

MANIVANNA THEVATHASAN, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2014) McMaster University

(Software Engineering) Hamilton, Ontario, Canada

TITLE: Modeling HUD Layouts for Selection Based Video Games

AUTHOR: Manivanna Thevathasan

B.Eng., (Software Engineering)

McMaster University, Hamilton, Ontario, Canada

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: x, 81

In loving memory of my grandfather.

Abstract

Display technology continues to become increasingly varied, with new combinations of screen sizes and resolutions introduced every year. In order to design or adapt Heads Up Displays (HUDs) to these screens, manual design processes are preferred, since they afford designers modeling flexibility. However, there are aspects of this process that can be automated to reduce the burden on the designers.

In order to help designers, this thesis presents a semi-automated approach to model, optimize and prototype HUDs for selection based video games. Specifically, this solution produces HUD layout prototypes that are ideal for tower defense games. This approach consists of an algorithm operating with a task efficiency model, which enables flexible modeling capabilities with automated evaluation. The result helps reduce the development time and cost of creating these HUDs. In addition, this work demonstrates the viability of using semi-automated techniques in the design process of video game HUDs.

Acknowledgements

I would like to thank my supervisor Dr. Jacques Carette for his expertise and guidance throughout the entire duration of this work. His knowledge and advice were essential in forming the motivation and reaffirming the direction of this work. Our discussions with regards to this topic were invaluable. All of which, I am eternally thankful for.

I would also like to thank my family and friends for their encouragement and support. They continue to inspire me towards greater accomplishments.

Contents

List of Figures

Chapter 1

Introduction

In recent years, displays possessing a variety of screen size and resolution combinations have been rapidly introduced, causing video game designers to rethink the design of their user interfaces to suit these new displays. Many video games exist on multiple platforms supporting a plethora of display combinations and so adapting the user interface in a flexible but efficient manner is a topic of great research. These issues combined with the already long iterative development process involved, has led to increased development times and costs. In order to solve this problem, methods of automation have been explored as a possible solution to help designers during the development process of video game user interfaces.

Attempts at using automation and/or code generation on video games, as a whole, has been explored before [13, 9]. In addition, automation in the development of graphical user interfaces for general applications has also been explored [26, 10, 21]. However, there has not been much research into applying those techniques towards video game user interfaces specifically. A possible reason for this is due to the unique design of each interface for each game. Heads Up Displays, (the primary interface

used to interact with the game), in particular, possess a unique set of requirements that are tied to that specific game's mechanics and genre. This unique quirk about video games leads us to believe it is unlikely that fully automated techniques will replace the creative aspects of human game designers. However, automation can still be used in other stages of the development process to help reduce the development time and cost of creating video game user interfaces.

This work proposes a solution for one such method of automation. This solution involves appropriately modeling, optimizing and prototyping Heads Up Display, (HUD), layouts within tower defense video games, given the goal of task efficiency. Through the use of Fitts' law, blackbox optimization software and game genre specific knowledge, an algorithm is produced that can automatically solve and present HUD layouts. These layouts, represented as wire-frame images, have been optimized with the goal of minimizing the time it takes players to perform game-specific actions. This algorithm combined with effective modeling provided by the designer, can help reduce development time, and thus cost, due to an increased number of viable layouts automatically produced possessing the designer's goal of task efficiency.

While the algorithm presented here applies to a specific subgenre of games, it can be extended to other genres of video games with additional genre specific knowledge. In addition, it is important to note that the specific software developed for this paper is used to analyze and provide evidence that the presented algorithm satisfies the intended goals. The software developed does not facilitate the end goal of automation in and of itself. However, the algorithm presented can be adapted for use in any other piece of software to facilitate that end goal.

Organization of the thesis

In order to present how to effectively model task efficient HUDs as well as the algorithm developed to solve this problem, Chapter 2, first provides an overview of the necessary background knowledge. Chapter 3, will then introduce and explain how to appropriately model task efficient HUDs. Following which, Chapter 4 will introduce the refinements we made to the model, justify our design decisions and detail our algorithm for producing task efficient Heads Up Display layouts. Chapter 5 then provides an overview of the software implementation used for the algorithm, including an introduction to the blackbox optimization software NOMAD, used to optimize our results. In Chapter 6, an analysis of our results of several example models will be presented providing evidence that the algorithm does indeed work. Finally, Chapter 7 will contain a look at future applications the work presented, the contributions of this work as well as conclusions of this endeavor.

Chapter 2

Prerequisite Knowledge

In this chapter, background information is provided on the important components of the final solution. Specifically, information related to the motivation behind this work, previous work conducted in similar research areas and prerequisite knowledge about tower defense games and Fitts' law are presented in the following sections. In addition, clarification on some of the terminology used in this paper is provided.

2.1 Setting The Stage

Display technology continues to evolve with increasing combinations of screen sizes and resolutions. With the introduction of wearable technologies possessing smaller sized screens, to mobile technologies possessing high resolution medium sized screens, to the large sized screens of televisions possessing resolutions all over the spectrum; the set of displays that a user interface must adapt to, in order to provide a *native* look and feel, is ever growing.

This is an even bigger issue for user interfaces in video games. In addition to

providing a native look and feel, they must also appropriately and efficiently convey gameplay related information to the player. Players actively depend on this information in order to play and succeed in the game. Thus, slow or confusing access to this information can cause the player to have a poor experience, potentially leading them to stop playing the game.

Since the Heads Up Display, (HUD), is the primary interface between the player and the game, it is the most important UI to consider adapting to various screens, to provide a better experience to players. As a result, developers not only have to prototype HUDs to ensure a pleasant gameplay experience but must also prototype to ensure a native experience. Combined with this increase in display technology, the prototyping phase is becoming a much longer and tedious task.

Automation could help solve the problem here, by allowing the designer to specify the requirements of the HUD once and then allow software to adapt the interface to other display combinations. Automation can also help designers prototype initial native interfaces whilst they are still developing the final set of requirements. This allows them to experiment with several ideas before beginning formal development of those HUDs.

HUDs, in particular, are a special case of user interfaces. Graphical user interfaces used in general applications have become much more standardized compared to the early days of personal computing. Terms exist for common UI elements such as buttons, checkboxes, text fields, forms, etc. In addition, the intricate details of these elements have become standardized within platforms as well; details such as the minimum/maximum size of these elements, the amount of spacing between them and other UI elements, the logic behind how they operate, how they provide feedback to

the user, etc. However, in the case of video games, while there does exist a loose standard set of terms for UI elements, the standardization of their look, feel and operation does not exist. This is obvious considering video games from different genres will possess different game play experiences and the creative talent developing the game will wish to embed their own creativity in the game. These unique aspects of each game will then naturally need to be reflected in the HUD. In addition, it is worth noting that corporate branding is another factor to keep in mind, as companies will wish to preserve the aesthetic that differentiates their games from others. For these reasons, we believe it is unlikely that a complete standardization of video game user interfaces will occur. For that reason, we tackle another issue in HUD development that could see benefits from the use of automation.

We view the HUD as possessing three parts: the *skin* or aesthetic of the individual element, the logic behind each element and the layout consisting of the size and position of the HUD element relative to every other element. The first two aspects of the HUD are the parts we believe are not generalizable as they are quite specific to the particular game genre and to individual games themselves. Thus, we tackle the last aspect of the HUD, the layout. We believe that with the appropriate use of automation, the development of the HUD layout for a game within a specific genre can be accomplished with limited user input. This would not only speed up the development process by increasing the number of viable prototypes, but it would also allow the designers to focus their creative energies on the tasks that are not generalizable, the aesthetic and logic.

The goal of this work is to introduce a method of automation that, combined with input and key decisions from a user interface designer, can be used to speed up the

development process of HUD layouts. In order to facilitate this goal, this method of automation consists of three parts: modeling, optimizing and prototyping a HUD given the goal of task efficiency. In our case, we are specifically looking at modeling, optimizing and prototyping HUDs for tower defense games. We choose to focus on the game genre of tower defense games as there exists a well defined way to model the actions that occur in this type of game.

Modeling is performed by the user interface designer, and consists of the designer identifying a particular model of task efficiency for their particular game. These models of task efficiency rely on models of cognitive tasks, (in our case just selection), and a task model which, we posit, describes the actual patterns of use in a game HUD. Fitts' law directly corresponds to selection tasks performed by players and so it is used to describe the cognitive aspect of the model. The task efficiency model, which we introduce, was developed with the particular genre of game in mind, to ensure that game specific actions and their use related with the HUD could be utilized as part of the optimization process.

Once a model is developed by the designer, we validate this model by building a prototype which uses an optimizer to find task-optimal layouts. This is facilitated with the use of an objective function that encapsulates the notion of task efficiency and an interrelated algorithm that incorporates other required design goals. Finally, this process culminates in a prototype that allows the designer to visually inspect the results in order to decide whether or not the prototype is acceptable for their use. As a result of this process, a designer can specify the task-related requirements of their HUD and allow a piece of software to prototype several different optimal results in a much shorter period of time.

2.2 Previous Work

Previous research conducted in the area of applying automation during the development of video game user interfaces is quite sparse. While there are several tools that exist to aid in the development process for video game UI, we are specifically looking at how software can be used to handle some of the arbitrary decisions made by developers in a more defined manner. Hence, throughout the duration of our work, no such previous work involving automation during the development of video game user interfaces was found. Despite this, there were alternate instances where automation of some kind was applied to the development of user interfaces within other domains. The work conducted there provided the initial motivation towards seeking out application in the domain of video game user interfaces.

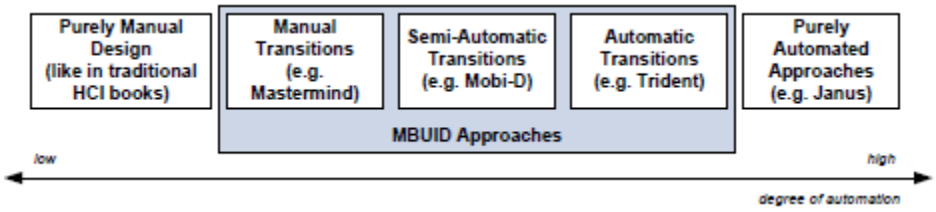

Figure 2.1: Degree of Automation in the UI Design Process [21]

The type and use of automation in the development of Graphical User Interfaces, (GUIs), for general applications, has been detailed and explored before [21]. As seen in Figure 2.1, the authors in that paper highlight a spectrum of capable methods of automation, for use in the development of GUIs. Purely manual fesign refers to requiring a human designer to manually design the UI from a given set of requirements. On the other end of the spectrum, purely automated approaches refers to any tools or pieces of software that handled the complete end to end process of developing a UI.

The problem with the former is that, designers may not consider existing standards for the particular domain or are free to creatively yet arbitrarily decide on a particular design choice. The problem with the latter is that, automatically generated UIs are typically specific to a very well defined narrow domain but the quality of their UI may not be up to par. As a result, it appears that a tool in the middle of this spectrum, a semi-automated approach, would be an ideal solution. This approach would allow software to handle solving the major aspects of the resulting UI whilst allowing the designer to make critical design decisions. For this reason, it is intended that the resulting solution presented in this thesis be usable with a semi-automated approach.

DSLs and code generation are a possible semi-automated approach for introducing automation towards the development process of HUDs. Previous work has been attempted looking at application of code generation towards the development of video games in [9, 13]. There was specific mention of requiring a HUD DSL in [9], however, no further work was found in that regard. Whilst the use of DSLs and code generation would be an important step towards further automating the development process, we chose not to focus on this specific technique. The reason being, DSLs focus on converting a set of inputs into a set of outputs for a given domain. The core logic in DSLs is thus focused on conversion. However, our desire was for the core logic to focus on optimization and evaluation, in order to alleviate the burden from the designer. For this reason, we instead chose to focus on producing some core logic that could be used in an optimization process. This logic, which turns out to be embedded in an algorithm, can always be later adapted into another tool to facilitate the use of DSLs.

The primary influence behind our work, the Auckland Layout Editor [26], provided us with initial motivation when determining how to use a semi-automated approach to solve our problem. Using constraints, the Auckland Layout Editor, (ALE), allowed a designer to quickly design a GUI containing various UI widgets for a general application, whilst ensuring that certain requirements were always met. These requirements were stated via the use of constraints. For example, a designer could set a constraint ensuring all widgets do not overlap any other widgets, or another constraint ensuring the position relationship between two widgets is always maintained. This tool would automatically check the list of specified constraints whenever a new widget was introduced to the GUI, always ensuring that the resulting layout met all the specified constraints. Although the tool required frequent interaction with the designer, the time to develop GUI layouts was reduced because invalid layouts were automatically rejected by the tool. We originally sought to utilize constraints as part of the semi-automated approach to developing HUDs, in a manner similar to the ALE. However, as discussed later in the thesis, HUDs produced a different set of initial requirements that prevented this from being the case.

Another major influence to our work, that utilized an optimization process and coincidentally also looked at an application of Fitts' law, was [23]. Here, the researchers sought to answer a question, how would Fitts' law be used in an optimization algorithm to solve optimal button placements in a GUI layout? An interesting question, since we were interested in asking a very similar question. They were able to solve GUI layouts for optimal button placement with the use of an objective function consisting of a weighted Fitts' law formula, a penalty function ensuring non-overlapping buttons and an optimization method using simulated annealing. However, there were

several additional conditions imposed on their problem space that restricted the scope of their result's application. These conditions included the use of only square buttons and the total area of all buttons being restricted to a predetermined value. The result is a layout that while valid, is impractical for general UI use.

Upon observing the previous work in this area, we decided to find an objective function embedded within an algorithm and used with some optimization software, to determine task efficient HUD layouts. This approach remains a semi-automated approach that improves on the work done in [26] by taking advantage of the domain specific knowledge to eliminate the need for constraints. Our solution also improves upon the work done in [23], by reducing the number of conditions imposed on the input, thereby increasing the scope of the result's applications. This helps produce practical HUD layouts that are both valid and usable in our domain: tower defense games.

2.3 Tower Defense games

Video games span a wide variety of genres and possess game mechanics that may have been developed for one genre in mind, but has since been adapted to another. This allows each individual game the potential to provide a gameplay experience that is completely unique, even if the genre has some predefined, almost standardized, requirements to it. As this is the first step towards the goal of automation, we found it was necessary to focus our efforts on one genre of game. By focusing on one genre, we can utilize genre specific knowledge to better refine our solution and ensure that for the chosen genre, the results are valuable. In our case, we have chosen to focus on tower defense games. However, we posit that the ideas, design decisions and results

can be applied to many, if not all, HUDs in 2D selection based games with minor changes to reflect the genre specific requirements. In addition, with additional domain knowledge, the same can be said about HUDs in any other 2D game.

Tower defense games are a sub genre of strategy video games. The goal of the player is to use towers, traps and other abilities to stop enemy units from crossing the map and reaching their given goal. Typically played in real-time, tower defense games can be played individually or as part of a group. Towers, enemies, traps, player abilities and other tower-defense specific game mechanics usually possess varying abilities, levels and costs to add complexity to the game. In addition, tower-defense games typically split into two categories when it comes to tower placement. The first allows the player to place towers anywhere on the game map so long as a path still exists from the enemy spawn point to their end destination. The second requires players to place towers in pre-determined locations. Some examples of existing popular tower defense games are *Plants vs. Zombies*, *Kingdom Rush*, and *Fieldrunners*.

Figure 2.2: *Plants vs. Zombies* [22]

Plants vs. Zombies is a popular tower defense game created by Popcap Games, originally released in 2009 [22]. Instead of having towers as traditional buildings, they chose to represent towers as plants with unique abilities. By collecting sun resources, players are allowed to place plants in one of the many squares on the grid. Their goal is to eliminate the waves of enemies attempting to reach the left side of the map. This was one of the games that provided the original motivation for the application of this work.

Figure 2.3: *Kingdom Rush* [14]

Kingdom Rush is another example of a tower defense game created by Ironhide Game Studio originally released in 2011 [14]. In addition to placing towers on the map at preset locations, players get to control individual units called heroes and use special skills to help them succeed in preventing enemy units from traversing the map. These additional game play elements are all selection based, requiring the user to select an option and select a location on the map to use their selected option.

Figure 2.4: *Fieldrunners* [24]

Fieldrunners is a tower defense game created by Subatomic Studios originally released in 2008 [24]. This is an example of a tower defense game where players are presented with a map, where the entirety of the space is allotted for them to place towers. While enemy units progress from one or more sides of the screen to defined end locations, the player can place towers so as to create a maze. This is done in the hope that lengthening the time it takes enemies to reach their destinations, will increase the chance that the player's towers will succeed in eliminating them. The only requirement of placing towers is that there exists a minimum amount of free space to allow enemies to progress through the map without getting stuck.

While observing these tower defense games, it is convenient to provide our definition of HUD elements. HUD elements refer to the various components of the HUD that describe individual game mechanics. These include, but are not limited to, buttons, progress bars, timers, icons, text and more. In the case of this work, the use of the term HUD elements specifically refers to buttons. The reason being, buttons are the only HUD element that guarantee the use of selection. The other elements tend to be involved with perception. There has been research to indicate Fitts' law applies to eye saccades [25], which would suggest that it could be used to find the time it takes the eye to move between HUD elements. However, there is not enough evidence to support its use in predicting perception time. Thus, we solve HUD layouts for only selection HUD elements which, for tower defense games, would be buttons. With another objective function that accounts for perception time, this algorithm could be extended to other HUD elements as well.

Tower defense games were chosen because they heavily depend on selection tasks. Playing the game, regardless of the underlying device and its input modality, requires selection. While there are other aspects of the game that require perception, we focus on selection as there currently exists a method to model selection tasks.

Genre specific knowledge, in this case knowledge about tower defense games, is essential in order to derive a semi-automated approach to designing task efficient HUDs for these games. The knowledge gained from observing existing games in this genre helped develop the modeling process presented in Chapter 3.

2.4 Fitts' Law

In 1954, Paul Fitts introduced a model of human movement that predicted the time it took to select a target, effectively modeling the act of selection [8]. This model, now known as Fitts' law, has since been utilized in a plethora of work examining various aspects of Human-Computer Interaction. These include, but are not limited to, tilt-based interaction [17], eye tracking [25], keyboard design and optimization [18] and extension to advanced navigation tasks [3]. There is an overwhelming amount of evidence, in the field of HCI, that suggests Fitts' law accurately predicts selection times for given tasks. Due to this, it is used as the objective function to optimize as part of our algorithm.

Fitts' law is mathematically described as follows:

$$MT = a + b \log_2 (\frac{2D}{W}) \tag{2.1}$$

In this linear equation, MT refers to movement time, (in seconds). a and b are empirically determined non-negative constants representing the specific pointing device's capabilities. They are determined by plotting movement data gathered from the pointing device, and fitting a straight line to the gathered data. Thus, they correspond to the intercept and slope, respectively, of the fitted line. The logarithmic term in this equation is referred to as the Index of Difficulty, (ID), and has units of bits. It represents the relationship between the distance, (D), to the target and the width, (W), of the target.

Other formulations of Fitts' law exist, the most commonly used version being the Shannon Formulation introduced by Scott Mackenzie [15]. Here ID is rearranged to

be:

$$ID = \log_2 (\frac{D}{W} + 1) \tag{2.2}$$

Since negative distances or widths are not possible, this formulation is preferred because it eliminates the possibility of negative ID values, (also note that width is a non-zero term). Thus, at the bare minimum ID will equal $log_2(1)$ which is 0. This then ensures that selection time results are also non-negative.

Fitts' law, specifically the ID portion, descriptively states that a target that is smaller and/or further away takes longer to select than a larger and/or closer target. As a result, a reduction in the selection time of a given target can be considered to have improved the user's efficiency. It is this aspect of the model that motivates our goal of task efficiency, and further contributes to the decision to use it as the foundation of the algorithm.

Limitations of Fitts' Law

Although Fitts' law has been shown to work for several different cases, even beyond normal selection tasks, there are some limitations with it that should be pointed out. These limitations are important to keep in mind when designing an algorithm to take advantage of Fitts' law.

The first limitation is that Fitts' law is a 1 dimensional model, since both the distance to the target and the target's width were measured along the same axis. The scenario under which this model was developed is illustrated in Figure 2.5. In Fitts' original study, users were asked to navigate a pointing device along the x axis between two targets. The widths of the targets were altered between tests in order

to figure out the relationship of target width and distance traveled.

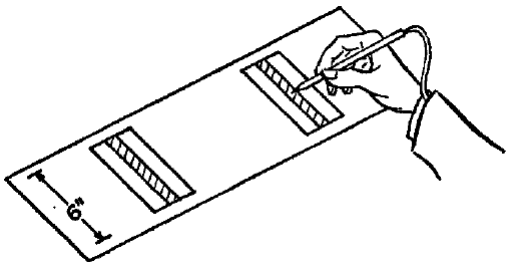

Figure 2.5: Original Fitts' Task from Paul Fitts' paper [8]

Work has since been done to extend Fitts' law into two dimensions. A comparison of various different two dimensional models was completed in [16], and they found that a good approximation for rectangular targets involved using the minimum of the target's width or height. Here the consensus of evidence shows that the model is still valid. Despite the fact that Fitts' law started out as a one dimensional model, its application is still valid in this two dimensional case.

Another limitation to keep in mind is the fact that Fitts' law was not specifically developed for computer usage. As a result, in display setups with constrained selection devices, the edges and corners of the screens provide exceptions to Fitts' law. One common workaround users can perform to select targets along the edges or corners of the screen, is to *throw* their mouse cursor in the general direction of the target. The screen bounds then constrain the mouse cursor and this helps direct the user to their target.

This workaround has unintended positive and negative consequences. Negatively, this workaround circumvents Fitts' law and thus significantly impacts the accuracy of the predicted selection times, for targets located along the edges or corners of

the screen. Some assumptions that are used to get around this issue are to assume that these particular targets possess infinite width or to assume that these particular applications do not encompass the entirety of the display. A positive consequence is that designers can take advantage of this knowledge when developing their interfaces to account for this. For example, menu bars are typically located along the top or bottom edges of the screen for the benefit of even faster selection time.

To reiterate, this is only a problem with a constrained selection device, i.e. a mouse, on a traditional display setup, i.e. a monitor. In other cases where the selection device is not restricted, such as in touchscreen devices or motion controllers, the screen edges and corners do not provide a constrained bound for the selection device. Thus, precise selection is required and so Fitts' law remains valid. In the case of this work, the algorithm and its results apply to any selection device that strictly adheres to Fitts' law.

Finally, another limitation to be aware of, but not worry about, is the fact that human limitations exist too. Objectively using Fitts' law will always produce shorter selection times for smaller and/or closer targets. However, targets that become extremely close and/or extremely small may not be perceptible to human users. Conversely, targets that are extremely far away and/or extremely large may not be possible to select via conventional means. Thus, in extreme cases, Fitts' law may be impractical to use. Solutions to this limitation include introducing minimum and maximum sizes for targets to ensure that the ability to perceive the target is not an issue. There are several examples of minimum User Interface target sizes specified in the mobile development space, [11, 4, 19]. Research done, both in the development of these guidelines and in external research, has shown that a 7 - 9 mm target size

for buttons is an acceptable range. However, these ranges are not absolute and it is possible to find other recommended ranges. Regardless, whether they choose to utilize industry recommendations or develop their own set of UI size standards, the designer can specify their own minimum and maximum target sizes for use with our developed algorithm.

Chapter 3

Modeling Task Efficient HUDs

We need to first describe modeling task efficient HUDs, before we can introduce automation. Improper modeling will lead to incorrect results, causing the designer to question the validity of the automation process. We focus on modeling and optimizing selection tasks, thus the use of Fitts' law as an objective function is necessary. This chapter contains an overview of our previous attempts at modeling task efficient HUDs as well as an explanation of our finalized modeling process.

3.1 Some Bad Models

Our previous attempts at modeling task efficient HUD layouts with Fitts' law, involved exploring constraint modeling languages (MiniZinc) [20], non-linear optimization solvers (APMonitor) [12] and Algebraic modeling languages (AIMMS) [7].

While these were powerful tools in their own right, none of them suited the exact scenario we were trying to model. The use of constraint modeling languages foreshadowed constraint space explosion, whereby the number of constraints involved for

an increasing number of HUD elements would become unmanageable. Non-linear optimization without constraints, always produced results where an overlapping clump of HUD elements were positioned around the center of the screen, (a very undesirable choice). If constraints were needed, but constraint space explosion was not desired, then we needed some additional control structures in place to help solve the problem.

Algebraic modeling languages initially held promise, especially AIMMS. It allowed the use of external programming languages to control the evaluation of the objective function. However, AIMMS still utilized mathematical based solvers to optimize the end results, (i.e. non-linear, mixed-integer, etc.). The external piece of software could evaluate the objective function, but then AIMMS would use its solver to optimize this result. Thus, trying to optimize results for a given task model was not possible with AIMMS.

We preferred to use an approach similar to AIMMS, whereby we could use an external piece of software to evaluate the given task model. The reason being, a large part of deriving the final objective function and algorithm used in this thesis, was a result of experimentation. By containing our HUD layout evaluation algorithm and objective function within an external piece of software, we could change it at will through numerous iterations to fine tune it. This could be done without having to worry about changing the optimization process at all.

In order to solve our problem given an external piece of software containing our own defined method of evaluating the objective function, we needed a tool that could essentially optimize a blackbox. It needed to provide input values representing the position and size values of each required HUD element. Our piece of software would then evaluate that choice. It also needed to keep trying another set of input values

until it reached an optimal result. In the end, NOMAD was the best choice for our needs. NOMAD is further detailed in chapter 5.

In addition, our previous attempts at deriving objective functions that utilized Fitts' law yielded several useful discoveries that were incorporated in the final solution. With Fitts' law at the core of our objective function, we quickly ran into problems of overlapping HUD elements, since Fitts' law is solely used to evaluate selection times. The use of constraints may seem like a logical solution, however, they are actually a poor choice for this application. In addition to the issue of constraint space explosion, constraints would strictly prevent overlapping HUD elements. What we actually desired was the ability to dissuade overlapping HUD elements, as there are HUD layouts that may prefer a slight amount of overlap to occur. This could be due to aesthetic preference or even due to the reduced selection times that would be afforded due to Fitts' law. In order to solve this, we figured that weights of some kind would be necessary, in the objective function, in order to dissuade overlapping elements while still favouring task efficiency.

Several attempts were made at augmenting our objective function with additional terms that represented weights or area, in the hopes of skewing the objective function away from overlapping elements. However, this raised the issue of normalization. How would the effective importance of both portions of the objective function, (the Fitts' law portion and the portion that was preventing overlapping elements), be balanced? Through this experimentation, we determined that the notion of non-overlapping elements needed to be combined with Fitts' law, in order to ensure that it was still valid whilst also disregarding issues of normalization.

The key benefit of our previous attempts was the realization that the ability to

correctly model the situation at hand was incredibly important. That is to say, the ability to model not only selection tasks but also game specific actions was required in order to ensure the results produced task efficient HUDs.

3.2 Task Efficient HUDs

In order to model task efficient HUDs for tower defense games, we took advantage of the game genre specific knowledge we had at hand. Mainly, tower defense games heavily rely on selection tasks. In order to model these selection tasks, we chose to use Fitts' law as it accurately provides the selection times for a given selection task. However, to speak about task efficiency, we must also consider the fact that, in video games, there is a reason for the player to perform these selection tasks, i.e. the result of playing the game. Embedding this knowledge that players will perform certain frequent actions, that involve selection, will yield better task efficient results.

We define tasks as the transitions between HUD elements that the player performs as they are playing the game. Selection tasks are then defined as the transitions that require selection at the beginning and the end of that transition. In the case of tower defense games, an example of a selection task is visualized in Figure 3.1. Here, the action of selecting a button, moving to a position on the map and then selecting that position to place that tower is considered a task.

Figure 3.1: An example of a task in *Plants vs. Zombies*.

In order to ensure that the resulting HUD layout is task efficient, the selection tasks that players perform as they transition between HUD elements needs to be minimized.

3.3 A Task Model for Tower Defense Games

An important design aspect of the HUD to consider is the tendency to group similar HUD elements together, or rather HUD elements that depend on similar or interdependent game mechanics. This is commonly done to reduce the amount of time the player will have to travel between these HUD elements that are somewhat related. In

order to capture this design aspect, task flows are introduced.

A task flow is a chain of tasks listed sequentially, that the game designer expects players to commonly perform when playing the game. In our case, since HUD elements are considered to only be buttons, task flows represent the sequential transition between buttons for common in-game actions. Different in-game actions possess a different task flow. By modeling task flows for each expected in-game action, the resulting HUD layout is better suited for the intended goals.

Figure 3.2: An example of a task flow in *Plants vs. Zombies*.

Consider the game *Plants vs. Zombies*, a common action for a player would be to select the shovel button, destroy a plant, select a plant and then place it on the

game grid. This task flow, as illustrated in Figure 3.2, can be represented as (Shovel → Plant Position on Game Grid → Plant Button → Plant Position on Game Grid), where each item corresponds to the respective HUD element. The result from our solution is that these three elements are kept closer together than other HUD elements not listed in this task flow, since reducing the distance between tasks in a given task flow will increase the efficiency of the player.

Once task flows are defined, a Fitts' law calculation between the game space grid and the first HUD element of the task flow is performed. Thereafter, Fitts' law is calculated between the current HUD element and the next HUD element in the task flow. This continues until the last element of the task flow. Since task flows involve Fitts' calculations between multiple HUD elements in a sequential manner, each subsequent HUD element will be placed as close as possible to the previous element and its size will be scaled according to both available space as well as the best width to minimize the result of that particular Fitts' law calculation.

In summary, a designer creates a task model for their HUD by creating a set of expected task flows. Each task flow contains a sequential chain of tasks that players are expected to perform with the corresponding HUD elements. This task model is then presented to our algorithm, combined with an optimizer, to determine a task efficient HUD layout.

Chapter 4

Refining The Model

With the notion of tasks defined with respect to tower defense games, we sought to further refine the model that could be described by designers when using our solution. With further experimentation involving our previous attempts with other solvers, we discovered several problems with the sole use of Fitts' law as our basis for task efficient HUDs. Additional design details were required in the solution in order to ensure that results were not only consistent with Fitts' law but that they also produced a viable solution that would actually be used by designers. This chapter contains our final solution consisting of our objective function, our developed algorithm and additional details about our design decisions embedded in our solution.

4.1 Set Of Required Assumptions

Fitts' law requires knowledge of precisely where the user pointed on the screen; knowledge which is unknown at design time. A designer would be specifying the unique characteristics of their HUD at design time, but the evaluation of the specified scenario

would only occur at run time of our algorithm. In addition, during the optimization of the specified HUD layout, no interaction with the player would be conducted. So, we needed some way of modeling where the player points in the HUD at design time in order for Fitt's law to evaluate the choice of HUD layout at run time. To solve this problem, a set of assumptions were developed and used during our experimentation phase to simplify the optimization process, constrain the scope of the problem space and yet allow Fitts' law to remain applicable to our results. The list of assumptions is presented in Figure 4.1.

1. The default position of the user's gaze and selection device are located at the center of the screen.
2. Button clicks occur at the center of the button, even though the size of the button affects the ease of selecting a button. This applies to subgrids as well.
3. Buttons possess a predetermined width to height ratio, ensuring their size appearance stays relatively the same as their absolute size changes.
4. The game space is represented as a grid containing a specified number of subgrids.

Figure 4.1: Final set of assumptions

The first assumption assumes that the player is at the center of the screen before a given task, in order to standardize a starting point for all tasks. The second assumption assumes that the user selects the centers of the screen and individual elements as they are deemed an average case. This assumptions allow us to simplify the optimization process while allowing the results of Fitts' law to provide an average case solution, where the user does not under or over shoot the center of a given target.

The third assumption was required in order to ensure buttons do not possess abnormal appearances in the results, while also constraining their area so that they

do not grow too large in one dimension alone. Recall that minimizing Fitts' law implies maximizing the target's width, so we need to ensure that their appearance remains relatively the same as their absolute size increases or decreases.

The final assumption was one that requires the game space to be represented as a grid containing subgrids, the number of which is specified by the designer. This is required in order to enable several of the features embedded in the developed algorithm, but also reflects the inability to determine where the user selected at design time. Further details about the use of a grid is provided in Section 4.3.1.

4.2 Our Solution

From a high level perspective, the objective function that we wish to minimize is a weighted sum of Fitts' law calculations, where each Fitts' law calculation evaluates the task performed between HUD elements. Figure 4.2 contains a high level description of our objective function.

$$TotalTime = OverlapPenalty * (\sum_{i=0}^{n} Fitts(SourceElement_i, TargetElement_i))$$

where $Fitts$ is represented as:

$$Fitts(SourceElement, TargetElement) = \alpha + \beta * ID$$
$$ID = log_2(\frac{Distance(SourceElement, TargetElement)}{TargetElementWidth} + 1)$$
$$Distance(A, B) = \sqrt{(A.x - B.x)^2 + (A.y - B.y)^2}$$

Figure 4.2: Total Time objective function and the inner components.

This objective function is embedded within an algorithm that handles task flows as well as adds a few other features detailed in the following sections. The Task Efficient HUD Layout Algorithm is presented in Figure 4.3 below. It describes the

logic that is performed when a list of required task flows is presented.

Algorithm 1 Task Efficient HUD Layout Algorithm

```
for Each HUD element specified in the Task Flows do
    if Current Task Flow length > 1 then
        Result += Fitts (Game Space Grid, First HUD element)
        if next HUD element is Game Space then
            Result += Fitts(Current HUD Element, Game Space Grid)
        else
            if Next HUD element is SubGrids then
                for Every SubGrid do
                    Result += Fitts(Current HUD Element, SubGrid)
                end for
            else
                Result += Fitts(Current HUD Element, Next HUD Element)
            end if
        end if
    else
        if Current Task Flow length = 1 then
            for Every SubGrid do
                Result += Fitts(Game Space Grid, Only Element)
                Result += Fitts(Only Element, SubGrid)
            end for
        end if
    end if
end for
return OverlapPenalty * Result
```

Figure 4.3: Task Efficient HUD Layout Algorithm presented in pseudocode. Result is the cumulative sum of all Fitts' law calculations.

The first check is to determine whether the current task flow consists of one or multiple HUD elements. If the current task flow consists of one HUD element, then it is handled as an asymmetric task. If the current task flow consists of more than one HUD element, then another series of checks is performed to determine the target of

the subsequent Fitts' law evaluation. The designer is free to model any of the three situations in the case of task flows. The subsequent target in a task flow can be the individual subgrids, the entire game space grid or another HUD element. By allowing the designer the choice to specify, they are provided with more flexibility in modeling a larger variety of HUD layouts but are also burdened with ensuring their description matches their requirements.

4.3 Additional Features

The algorithm that we developed contains four special features that enable designers to describe a larger set of applicable models whilst receiving more viable results. These features: the game space grid, the overlapping penalty function, the alternate grid centers and asymmetric tasks arise from solving problems that arose when developing the algorithm. These problems as well as the details of each feature are detailed below.

4.3.1 Game Space Grid

We define the game space as the part of the screen where the main gameplay occurs. In the case of tower defense games this can be considered the grid or map where the player places down the towers as enemies traverse through it.

One recurring problem that involved much discussion and experimentation was whether or not to consider the game space as a HUD element. The reason being, in tower defense games, the player is often traveling back and forth between buttons located in the HUD and the game space. Thus, this question directly implied asking

whether or not the game space should be involved in the distance and size calculations, as part of Fitts' law, between another HUD element and itself.

If the game space was to be considered a HUD element, then its center could be used in the distance calculations to calculate the time it takes the player to go from a given button to the game space. In addition, if the game space was the target of a given task, then its width could be involved in the Fitts' calculations thereby determining its size. This would be beneficial since the position and size of the game space grid could be determined during the optimization process.

If the game space was not considered a HUD element, then how would the distance calculations work? Would they be between a button's center and the center of the screen? Remember that Fitts' law would seek to minimize the distance between each HUD element and the center of the screen. Since a size would no longer be needed, (as this would not be a HUD element), what would prevent other HUD elements from congregating around the center of the screen? This would certainly hinder the player's ability to view and play the game itself.

Figure 4.4: Visualization of the game space grid containing subgrids. In this case 16 x 9 subgrids are contained.

Through much deliberation, the former design choice was made whereby the game space would be considered a HUD element and represented as a grid. Based on our domain analysis of existing tower defense games, the decision to encapsulate the game space to a confined area was a suitable choice. The reason being, many of these games provided ambient aesthetics or a greater view of the game outside the area of central gameplay to immerse the player. Effectively, this meant that many existing tower defense games did not utilize the full available space of the screen for the main gameplay and so our results would reflect the game design decisions made in this genre of game.

The game space is represented as a grid containing a predefined number of subgrids, as visualized in Figure 4.4. Each of these subgrids are considered to be a button, i.e. a viable place within the Grid where the player may select. This is possible, because in tower defense games, interaction with the game takes place in the game

space regardless of whether it is explicitly shown as a grid or a map. Remember from Chapter 2, the game space in tower defense games can be in one of two categories; either predefined locations are available for selection or the entire space is available for selection. Our subgrid design decision enables the designer to model either of the two by allowing the player to select any subgrid within the game space or in specific subgrid locations.

The only requirements from the designer with respect to the game space grid are the number of subgrids they wish to embed as well as the minimum and maximum sizes for each subgrid. With these two pieces of information, along with the proper modeling of task flows, the algorithm would determine the optimal size of each subgrid, thereby determining the optimal size and position of the game space grid as a whole. The game space grid as a whole has been fixed, centered with respect to the screen.

The use of a game space grid allows a designer the flexibility of specifying the grid as a whole, individual subgrids or every subgrid as part of a task flow. This allows them to model more complex tasks they may expect or desire players to perform when playing their game. While it is possible to allow the subgrid height and width to vary independent of each other, the models presented in this thesis all assume that the subgrid width and height are equal to each other. In other words, each subgrid is a square. This allows the solver to solve for one less variable while also allowing both dimensions to scale. If it is so desired, the subgrid width and height can be made independent by a designer.

4.3.2 Overlapping Penalty Function

Utilizing a *raw* version of Fitts' law as the sole objective function in a given optimization solver introduces the overlapping elements problem, also discovered in [23]. It's obvious to see, as shown in Figure 4.5, that given the formula and the goal of minimization, the best result would be to position each element's center overlapping one another. This makes the distance, traveled between elements, zero and would thus provide the minimum possible value. In practice however, if these buttons were to overlap one another, then it would not be possible for the player to select the buttons hidden underneath. Thus, this is an impractical solution and overlapping elements must be prevented.

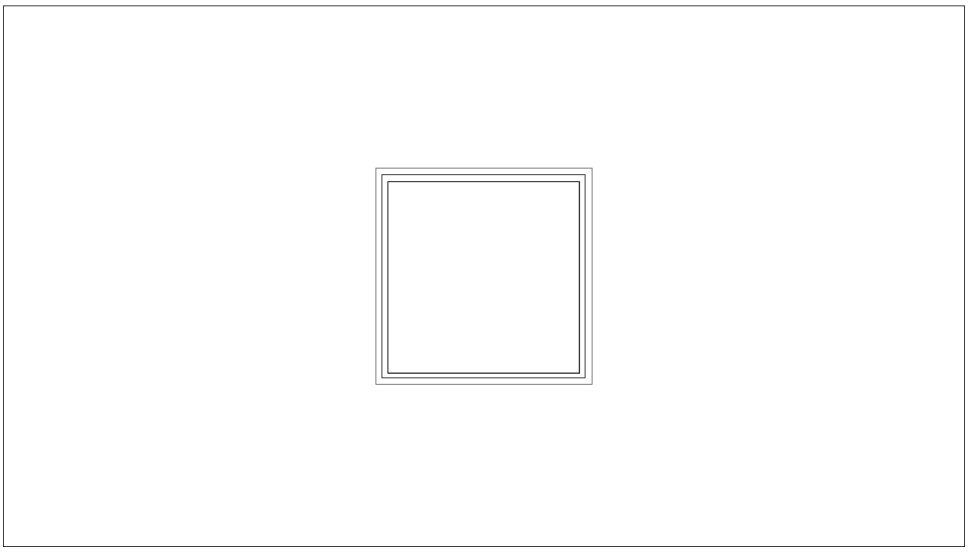

Figure 4.5: A scenario with several square HUD elements overlapping. It may be hard to notice but the inner square possesses thicker lines because several elements actually possess the same position and size.

Constraints utilizing rudimentary collision detection checks could be introduced ensuring that elements do not overlap. Ultimately however, the use of constraints

is a poor choice to solve this problem. The number of constraints needed will grow non-linearly as the number of elements introduced grows linearly. This results in an explosion of the number of constraints required. This is an even bigger problem in optimization packages that do not allow any sort of control structures in the definition of the optimization problem, since the user would have to manually input a new, increasingly larger, set of constraints for each element added. In order to solve this problem, some method of preventing overlapping elements was needed that could appropriately scale regardless of the number of elements introduced. We chose to use a penalty function that was not based on constraints or collision detection.

The penalty function used, labeled as *OverlapPenalty* in the algorithm presented in Figure 4.3, applies a weight to the resulting Fitts' calculations, if even a single pixel of overlap occurs in the resulting HUD. It overwhelmingly prefers layouts without overlap by producing a significant impact on the rest of the objective function. However, in the case of no overlap, it has no impact at all on the objective function. The impact can also be changed by the designer, if for some reason they don't mind a minute or complete overlap situation.

The *OverlapPenalty* algorithm is described below:

- Begin with a representation of the screen and its individual pixels, essentially a virtual layer. Each *pixel* in this virtual layer consists of a weight. At the beginning each weight is set to 0.
- Virtually draw each HUD element onto its corresponding position, filling out its corresponding size on this virtual layer.
- When drawing each HUD element, check the weight of each individual pixel. If

it is 0, then change it to 1. If it is greater than or equal to 1, then multiply that individual pixel's weight by a penalty amount.

- After all the HUD elements are drawn, sum up the values of the pixels that contain weights greater than 1.
- This sum is the value of the *OverlapPenalty* that is applied to the objective function. If there were no overlaps, then its value will be 1, causing no impact on the rest of the objective function. If there were overlaps, then its value will significantly impact the objective function.

By virtually drawing these HUD elements onto a virtual screen, collision detection checks are avoided. Typically these checks involve checking each side of the HUD elements against all other sides, which does not scale well when a larger number of HUD elements are used. In addition, it does not require knowledge of where the collision occurs; just that it occurs. The benefit then is that by running the objective function multiple times with a different set of inputs each time, the objective function will take care of ensuring non-overlapping HUD elements are preferred each time. Note that this doesn't mean overlaps are impossible. Depending on the circumstances, it may be the case that a minor amount of overlap is preferable. In such a case, the designer must decide if the amount of overlap that exists is too large a detriment to their game.

Another aspect to keep in mind here, is that the *OverlapPenalty* is just a weight, it does not possess units. It also is not a separate term, it is embedded as part of the weighted Fitts' law calculation. Thus, normalization of values is not a problem here as it once was with our previous attempts.

4.3.3 Alternate Grid Centers

Fitts' law wasn't developed for use with HUDs, thus weights were introduced to account for overlapping HUD elements. Similarly, another feature was required in order to ensure that the results of the optimization process more appropriately utilized the available screen space. Consider the following scenario, where we are required to place one HUD element. The game space grid and overlapping penalty function with non-overlapping elements preferred are provided. What would the optimal solution look like? Figure 4.6 contains one of only two possible solutions.

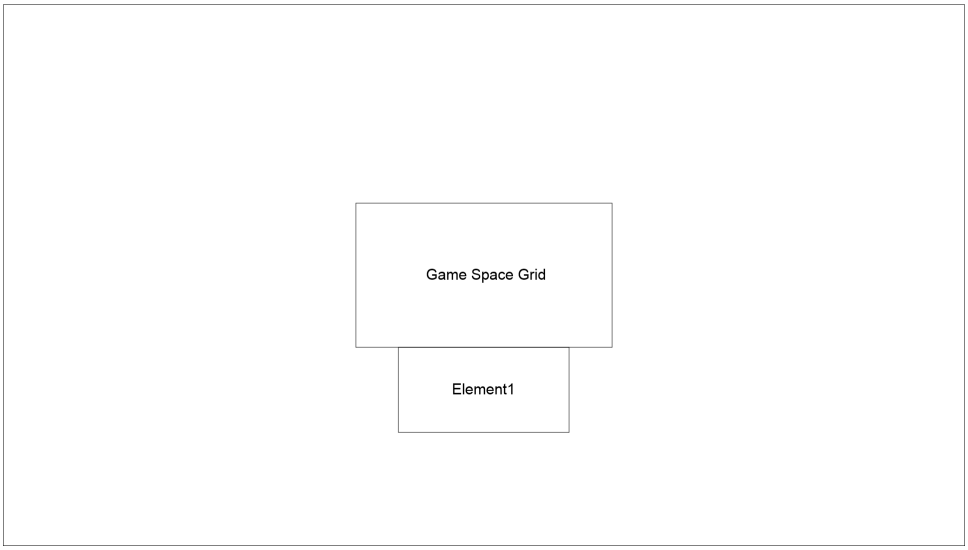

Figure 4.6: Without alternate grid centers, for this scenario, the position of this element is one of only two possible optimal results.

In this widescreen scenario, the best position for specified elements would *specifically* be directly above or below the game space, where the center of the element is aligned with the center of the game space grid. The reason being, the distance between the center of the game space and the center of the HUD element is less along

the y axis than along the x axis. Since the use of Fitts' law prefers elements to be as close as possible and our overlapping penalty function dissuades them from overlapping, these two scenarios will always be provided as results. This is the expected outcome from using Fitts' law since that is exactly what the equation is asking of the solver, however, this is not always desirable to the designer. In order to provide some additional flexibility, alternate grid centers are introduced that help modify the results.

Alternate grid centers are optionally used in Fitts' law calculations for asymmetric tasks. They alter the center position of the game space grid in that particular distance calculation. This allows us to skew HUD elements to certain sections of the screen.

One example of a benefit would involve ensuring that the HUD elements were positioned away from particular sections of the screen, so as to not impede gameplay related information entering the screen. For example, when new enemy units enter from the right side of the screen in a tower defense game, HUD elements are better suited on the other sides of the screen so that the player can focus on the new information whilst interacting with the game elsewhere. This allows designers additional flexibility in modeling their exact requirements of the HUD.

By default, four alternate centers are provided as alternate choices. These alternate centers allow the HUD elements to skew to the top, bottom, left and right sides of the screen, as seen in Figure 4.7. However, any user-defined alternate center could be used as well with modification to the implementation.

Figure 4.7: HUD layouts with alternate grid centers.

4.3.4 Asymmetric Tasks

Task flows consist of a chain of tasks between specific HUD elements. Task flows also consist of a special case, referred to as asymmetric tasks that apply specifically to task flows with only one HUD element. asymmetric tasks involve a secondary motion that involves returning from a target HUD element back to each subgrid in the game space grid. In order to see why this is necessary consider another example from *Plants vs. Zombies*. In *Plants vs. Zombies* selecting a plant then requires the player to select where, on the game space grid, they wish to place it. This secondary motion is a key exploit of this genre of game, that gives way to asymmetric tasks.

Asymmetric tasks refer to the two step motion required for several actions in tower defense games. Frequently, players have to select a button on the HUD and then select a position on the game space grid. The assumption that each task begins from the center of the game space grid is already there. However, this two step action

is asymmetric because the distance between the HUD element and where the player selects on the grid can be different than the center of the screen. In fact, more often than not, it will be different since the assumption that the player can only select the center of subgrids is required. Thus, in order to capture this common action, asymmetric tasks for a given action sum the Fitts' law calculation from the center of the game space grid to the center of the HUD element and then the center of the HUD element to the center of each subgrid. Figure 4.8 illustrates this process.

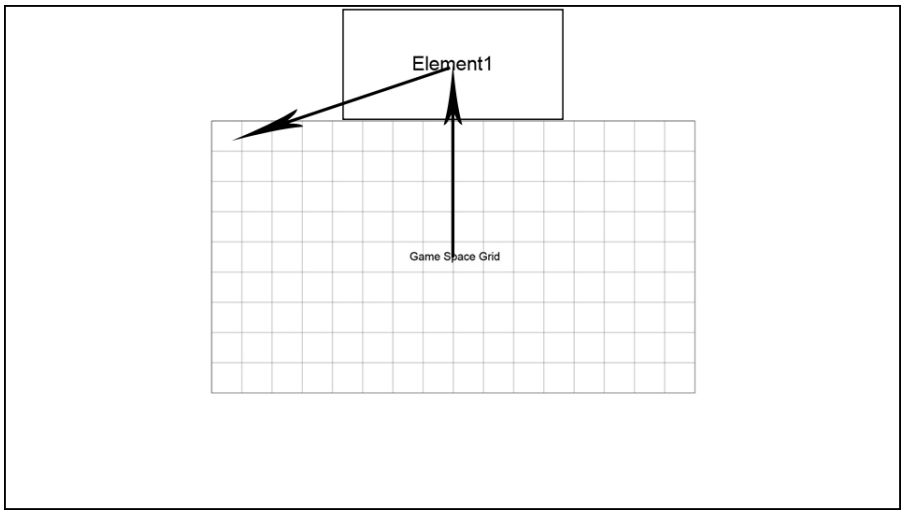

Figure 4.8: This motion is repeated to each subgrid as part of the asymmetric task.

The multiple Fitts' law calculations for each HUD element yields several benefits. Since each subgrid is a potential choice for the player to select in the second step, the first benefit of this approach is that each subgrid has an equal effect on the Fitts' law calculation. This allows the HUD element to be placed anywhere relative to the game space grid, whilst adjusting its size to compensate for any additional distance gained from the ideal location. The benefit becomes much more powerful when combined with the second benefit. Since the second step of the action involves multiple Fitts'

law calculations with each subgrid, where each subgrid is the target, the size of the subgrid is being optimized with each additional HUD element placed. The size of each subgrid will be chosen such that the size of the game space grid is as large as possible whilst simultaneously limiting the size of the game space grid such that other HUD elements can be placed in a non-overlapping as large as possible manner. This then solves the issue of trying to determine the size of the game space grid, since it can be solved by the optimization solver via multiple Fitts' law calculations with each HUD element.

Chapter 5

Implementation

This chapter details parts of the software implementation used to test and verify the solution developed in this paper. In particular, the choice of blackbox optimization software will be detailed and relevant aspects of the implementation that apply to the analysis performed in Chapter 6 will be presented and discussed.

5.1 NOMAD

NOMAD (Nonlinear Optimization by Mesh Adaptive Direct Search), is the specific blackbox optimization software used in this project [1]. It uses a Mesh Adaptive Direct Search, (MADS) algorithm to solve optimization problems that do not possess mathematical properties such as derivatives. The key benefit here is that it handles the external software provided as a black box, meaning it needs no knowledge of the specific application, which is beneficial to us. The specific inner workings of the MADS algorithm is described in great detail in [5]. We provide a high level overview with focus on important relevant points.

Algorithm 1: High-level presentation of MADS

```
Initialization: Let x_0 ∈ ℝ^n be an initial point and set the iteration counter k ← 0
Main loop:
repeat
    SEARCH on the mesh to find a better solution than x_k
    if the SEARCH failed then
        POLL on the mesh to find a better solution than x_k
    if a better solution than x_k was found by either the SEARCH or the POLL then
        call it x_{k+1} and coarsen the mesh
    else
        set x_{k+1} = x_k and refine the mesh
    Update parameters and set k ← k + 1
until Stopping criteria is satisfied;
```

Figure 5.1: Pseudocode for MADS algorithm [6].

From a high level perspective, the MADS algorithm picks a set of values for the set of decision variables we wish to solve for. This is called the *SEARCH* step. It then requests the external software to evaluate this set of values and provide the objective function's result. The algorithm then iteratively attempts to find a better solution than previously found by traversing a mesh over the constrained user-specified domain. This is called the *POLL* step. Once that is no longer possible within a given iteration, a finer mesh is used and the process is repeated. This continues until one of three situations occur: the number of iterations performed has reached a user-defined maximum, the mesh size has reached the NOMAD-defined minimum or an optimal value has been reached prior to either of the previous situations. Once this process is complete, the results are provided indicating the values of each of the decision variables and the objective function's result.

In addition to the implementation and use of the MADS algorithm, NOMAD possesses several additional features to help solve larger specialized problems. The

features that are useful to us are Latin Hypercube Search (LHS), Variable Neighbourhood Search (VNS) and Direction Types.

LHS is a technique used to determine a initial set of values for the decision variables. This is especially useful if an initial set of values is not provided by the user. However, there is no guarantee that this technique will find an initial point in a given run of the program.

VNS is a technique used to avoid reaching a local optima. In addition to the regular set of evaluations the MADS algorithm will perform, VNS performs an independent set of evaluations with the MADS algorithm using a random set of values as its initial point. The hope is that by attempting to use another algorithm run over the same domain from a different start point, the chance of finding a better solution elsewhere can be improved. If a better set of values is found, then the regular MADS algorithm continues its regular run from the newly found point. The extra number of iterations performed by using VNS does slow down the total run time of the tool and thus is typically limited to problems with a lot of local minima solutions. In our case, we use it when we start solving for a large number of decision variables, ($\sim$ greater than 10), in order to increase the chances of finding a better solution before the minimum mesh size constraint is reached.

NOMAD provides three different direction types each with varying parameters that dictate the number of directions they generate. In all, 16 different direction types are provided that can be specified by the user for the MADS algorithm to use during its *POLL* step. These direction types dictate how the MADS algorithm will calculate its list of trial values for the *SEARCH* step. The user can specify multiple direction types to use as they wish. In addition, it is important to note that the

	s1	s2	s3	s4	direction types
1	ORTHO	1			OrthoMADS, 1.
2	ORTHO	2			OrthoMADS, 2.
3	ORTHO				OrthoMADS, n+1, quad model for (n+1)th dir.
3	ORTHO	N+1			OrthoMADS, n+1, quad model for (n+1)th dir.
3	ORTHO	N+1	QUAD		OrthoMADS, n+1, quad model for (n+1)th dir.
4	ORTHO	N+1	NEG		OrthoMADS, n+1, (n+1)th dir=-sum n first dirs.
5	ORTHO	2N			OrthoMADS, 2n.
6	LT	1			LT-MADS, 1.
7	LT	2			LT-MADS, 2.
8	LT	N+1			LT-MADS, n+1.
9	LT				LT-MADS, 2n.
9	LT	2N			LT-MADS, 2n.
10	GPS	BIN			GPS for binary variables.
11	GPS	N+1			GPS, n+1, static.
11	GPS	N+1	STATIC		GPS, n+1, static.
12	GPS	N+1	STATIC	UNIFORM	GPS, n+1, static, uniform angles.
13	GPS	N+1	RAND		GPS, n+1, random.
14	GPS	N+1	RAND	UNIFORM	GPS, n+1, random, uniform angles.
15	GPS				GPS, 2n, static.
15	GPS	2N			GPS, 2n, static.
15	GPS	2N	STATIC		GPS, 2n, static.
16	GPS	2N	RAND		GPS, 2n, random.

Figure 5.2: The 16 different specifiable direction types for the MADS algorithm [6].

Lower Triangular search (LT) is a non-deterministic approach whilst the Generalized Pattern Search (GPS) and Orthogonal Search (ORTHO) are deterministic. Further information on the technical details of these direction types can be found in [2, 5].

Limitations of NOMAD

NOMAD satisfied a lot of our requirements for an optimization solver that could handle our external software. However, it too possesses some limitations that are important to be aware of when attempting to solve our problem. These limitations are more prevalent when dealing with complex cases, ($\sim$ 4 or more HUD elements),

as opposed to many of the simpler cases.

The most important limitation to keep in mind is one that relates to the termination criterion for the algorithm. More often than not, the minimum mesh size will be reached before an optimal solution will be found. Thus, the solution to many complex problems are approximations of the optimal solutions rather than the actual optimal solutions.

Another limitation is the possibility of reaching local minima even with VNS enabled, simply because the problem domain is incredibly large.

However, due to the lack of open source or otherwise readily available blackbox optimization software, we chose to continue using NOMAD as the main purpose of this work was to introduce our algorithm. We note this limitation to reiterate that this is not a limitation of our algorithm but the choice of optimization software used.

5.2 Implementation details of our software

The software developed was written in C++ so as to take advantage of NOMAD. The implementation consists of three important parts:

- Algorithm for evaluating HUD layouts.
- NOMAD blackbox optimization software.
- Layout rendering code.

The algorithm for evaluating HUD layouts has been detailed in Section 4.2, while the description of NOMAD was provided in the previous section. The last component consists of code that writes an HTML5 file. The canvas in HTML5 was chosen to

render the resulting HUD layout, as it was a trivial way to render wireframe rectangles without the use of additional third party code.

In order to run our implementation, two pieces of input are required:

1. A separate parameter file for NOMAD.

2. Specific design choices embedded in the C++ code for the algorithm to use.

The first piece of input required is a parameter file for NOMAD, highlighting the different choices made by the user for the solver specific variables. This would include the decision to use NOMAD techniques such as VNS or LHS, indicating the number of parameters that need to be solved for as well as their upper and/or lower bounds, indicating any initial point coordinates for the optimization process and other NOMAD configurations.

The second piece of input required is embedded in the C++ code containing our algorithm. Designer decisions such as the width to height ratio of buttons, subgrids as squares or rectangles and the number of subgrids contained in the game space grid. Optionally the use of any alternate grid centers for asymmetric tasks could be provided here as well. Each HUD element possesses three decision variables that need to be solved for by the solver. These variables are the x and y position of the HUD element and the width of the HUD element. Since the width to height ratio is provided by the designer, our algorithm can figure out the HUD element's height, reducing the burden on the optimizer.

Once the optimization process is completed, each wireframe rectangle is labeled and correctly sized/positioned in the resulting layout within the HTML5 file. The exact pixel values used for each HUD element's size and position can also be viewed

within the HTML5 code. In addition, in our implementation we store the objective function's result, so that a designer could objectively determine whether one layout is better than another in a similar scenario.

The design decisions provided in the algorithm and the model-specific parameters provided to NOMAD need to be well formed. This is currently the responsibility of the designer. If the designer provides poor input, then the current implementation has no way of determining whether to reject that input or suggest alternatives. It will only seek to evaluate and optimize HUD layouts for the provided input. For example, if the designer specifies that the minimum size of their HUD elements is equivalent to the entire screen's size, then the resulting layouts will have buttons as large as the screen. There does not exist a current standard of design guidelines for this particular application area, so it is currently left as open as possible. If guidelines were developed, then they could be incorporated into the solution as well. For now, we provide a list of our suggested guidelines, that were followed in the scenarios explored in Chapter 6.

- The number of subgrids along both the x and y axis, combined with the size of each subgrid, is chosen such that the game space grid maintains the same aspect ratio as that of the screen. This provides a more aesthetically pleasing result, but also allows the maximum size of the game space grid to be the entirety of the screen. For example, in a scenario where a game space grid has 16 x 9 square subgrids, the maximum size of each subgrid would be 120px. This would allow the game space grid to be as large as the screen, if that is the best option.

- The minimum size of each HUD element should be easily perceptible to the end user. For example, choosing the minimum size of each HUD element to be 2px

by 2px, would be too small for them to perceive. There exists some guidelines on minimum selection sizes in other domains, and these could be used to influence the chosen minimum sizes [4, 11, 19].

- Square, or close to square, HUD elements seem to produce more realistic and aesthetically pleasing results than rectangular HUD elements with large width to height ratios.

A tedious detail of this particular solver is the fact that changes to the number of decision variables need to be accounted for in both the NOMAD parameter file as well as the code containing our algorithm. The input and output to the optimization process is not the focus of this work, so the fact that they are currently geared towards a developer, (since changes to the model and changes to the input are done within the code), instead of a designer is not of concern. Our algorithm, representing the core logic of this optimization process, is the important aspect to focus on. If desired, this algorithm can be adapted to another tool, that provides a more user-friendly interface for a designer or a more expressive output, at a later stage.

The C++ code developed can be viewed at `http://www.cas.mcmaster.ca/~thevam/HUDImplementation/`.

Chapter 6

Analysis

This chapter contains an analysis of the algorithm used to solve task efficient HUD layouts. Each of the following sections present a different scenario with increasing complexity to demonstrate the results of the algorithm.

For each of the following scenarios, the following input is provided.

- The NOMAD parameter file describing the features of NOMAD to use: the number of decision variables, the minimum/maximum values of each decision variable and the initial starting point for the solver, if any.
- The standard required information of our software: the screen dimensions, the number of subgrids, the number of HUD elements & decision variables, a list of task flows where each HUD element is mentioned at least once and width to height ratios for each HUD element.
- Optionally the designer can also specify the use and choice of any alternate grid centers and the intensity of the overlap penalty.

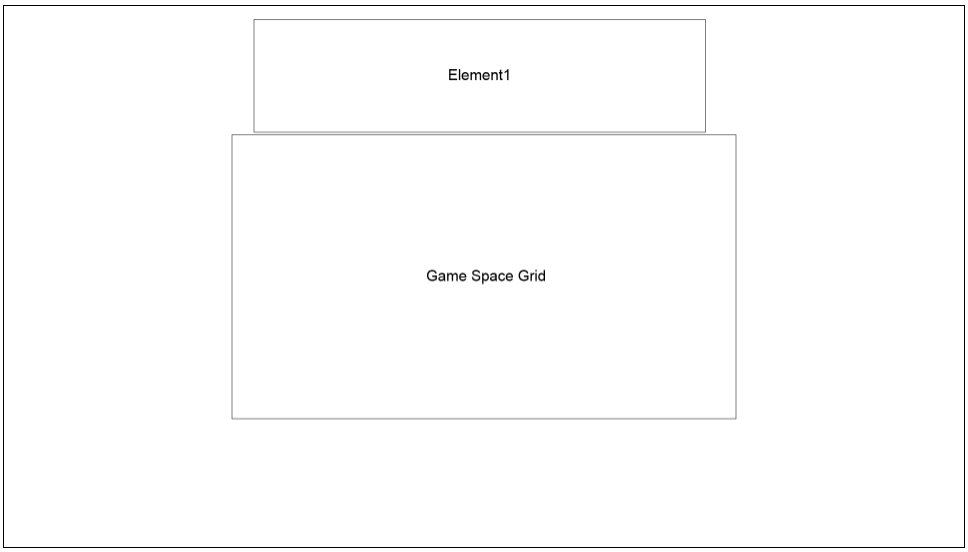

Figure 6.1: One HUD element case. Task flow: (1)

6.1 One Element Case

The first case to test consists of the game space grid and one HUD element. This is the simplest case where one button needs to be placed relative to the game space grid. Since there is only one element presented, the task flow is considered as an asymmetric task. The task flow input consists of just one HUD element, as an asymmetric task.

The results of this case appears in Figure 6.1. For a one element case, there are really only two positions for the element, since this is a widescreen scenario. Either the HUD element would be placed above or below the game space grid whilst its center is aligned with the game space grid's center. The particular choice of above here is entirely dependent on the solver's path to the best feasible result.

The nature of this asymmetric task is apparent in the size results. The size of the HUD element and the game space grid arrive at a point of balance, where their sizes are chosen such that they influence minimal Fitts' Law results. If the second part

of the asymmetric task were omitted, then the result would be a HUD element with the largest possible width. Since non-overlapping elements would still be a factor, the size of the game space grid would then shrink to accommodate the larger HUD element. The result, more often then not, would be the game space grid that is as small as possible, since it is never the target of any task. It is for this reason, a task flow consisting of just one HUD element is considered an asymmetric task. This allows optimization of both the game space grid as well as the HUD elements.

6.2 Two Element Case

When two or more HUD elements are needed, specifying task flows become a possible option. There are two cases to analyze; Case 1: one where each element is independent of the other and Case 2: both elements are part of a task flow. For Case 1, the task flow input would consist of two tasks where each task is an asymmetric task consisting of one HUD element. For Case 2, the task flow input would consist of one task flow where both HUD elements are listed in a particular order. For each of the following cases, each HUD element's width to height ratio was chosen to be 4. This results in rectangular buttons. Figures 6.2 and 6.3 contain the results of Case 1 and 2, respectively.

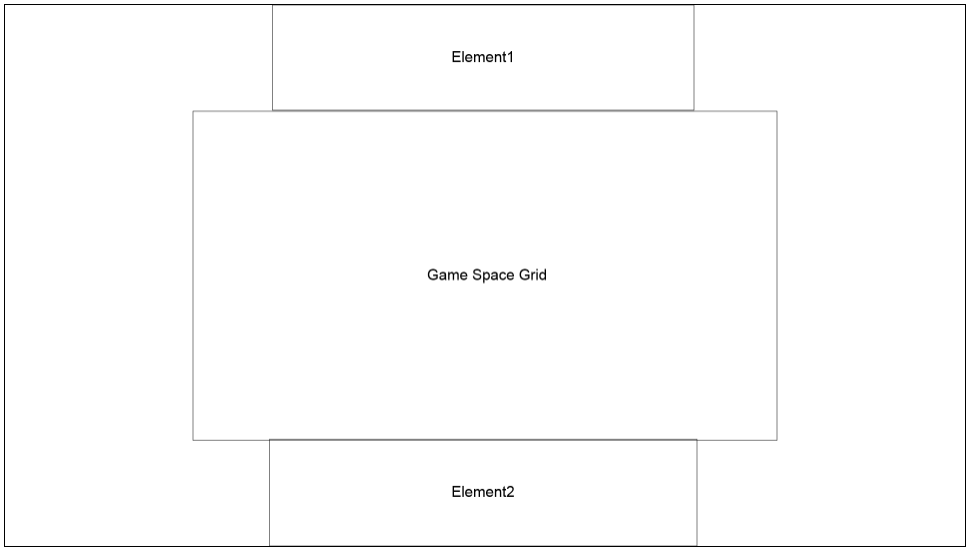

Figure 6.2: Case 1: Two asymmetric tasks. Task flow: (1), (2)

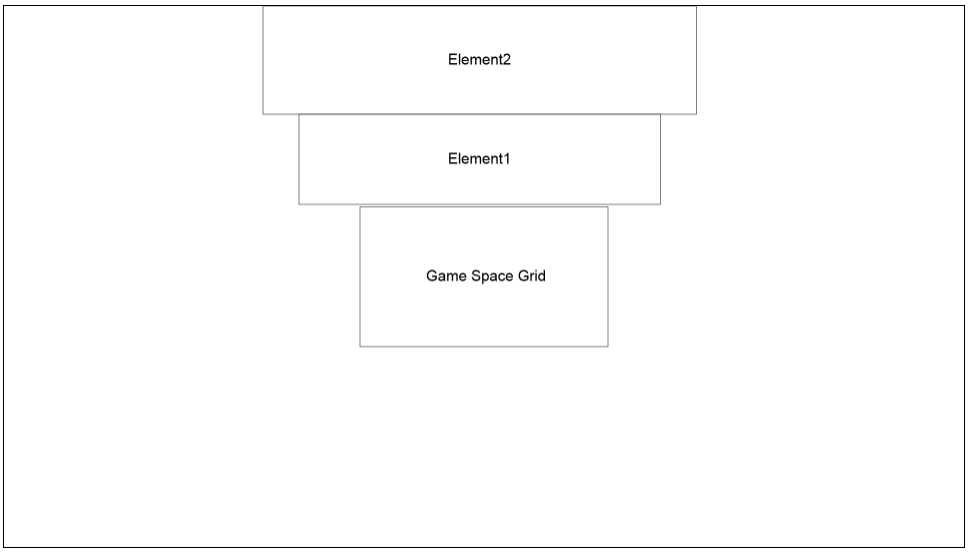

Figure 6.3: Case 2: One task flow consisting of two HUD elements. Task flow: (1 $\rightarrow$ 2)

The results of these two cases highlight the difference between individual asymmetric tasks and a task flow. In Case 1, both tasks are independent asymmetric tasks which seek to minimize the distance between them and the game space grid whilst also making both them and the game space grid as large as possible. In a wide screen scenario, this results in a solution like in Figure 6.2.

In Case 2 the task flow consisting of two elements provides a different result. task flows, as mentioned before, seek to optimize the efficiency of performing the specified tasks in the specified order. Thus, the use of a task flow in Case 2 seeks to minimize the distance between the first and second element, whilst maximizing the size of the second element. This results in a solution like in Figure 6.3. If the order of the task flow was specified in the opposite way, i.e. from the game space grid to the second element, to the first element, then the result would be similar to Figure 6.3 but with both elements switching size and place.

Caveat of Task Flows

One important caveat to note of task flows containing more than one element: there is no implicit return to the game space grid or its subgrids. If either of these are required, then they need to be explicitly specified in the task flow by the designer.

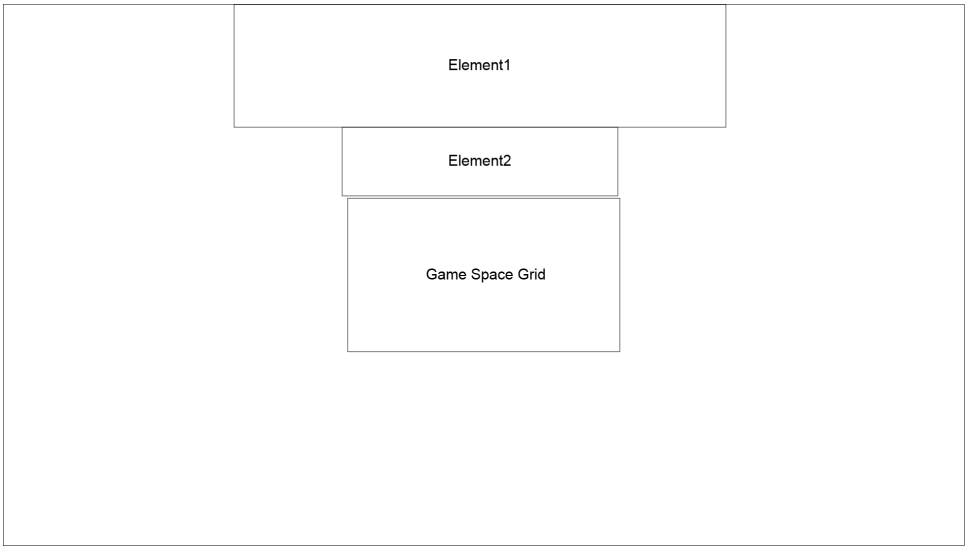

Figure 6.4: Modification to the previous Case 2. The task flow now consists of three elements: element 1, element 2 and the game space grid. Task flow: ($1 \rightarrow 2 \rightarrow$ game space grid)

In the case of returning to the game space grid, the final Fitts' law calculation will evaluate the task moving between the previously last HUD element and the game space grid, using the grid's center in the distance calculation. Observing Case 2 again, the additional task of returning to the game space grid results in a HUD layout like in Figure 6.4. Element 2 seeks to be as close to element 1 and the game space grid, thus it is placed between the two elements. The first task in the task flow also prefers to have element 1 as close as possible to the game space grid, but since this is not possible its width is made larger in an attempt to offset the increase in distance between the two.

Figure 6.5: Modification to the previous Case 2. The task flow now consists of three elements: element 1, element 2 and the subgrids. Task flow: ($1 \rightarrow 2 \rightarrow$ Each subgrid)

In the case of returning to the subgrids, the final Fitts' law calculation will evaluate, for each subgrid: the task moving between the previously last HUD element and each subgrid. Observing Case 2 again, with the addition of returning to the subgrids, results in a HUD layout like in Figure 6.5. Note that since this case involves returning to the subgrids, the second task is between element 2 and *every* subgrid. This overwhelmingly causes element 2 to be placed in the optimal position, but since there are several Fitts' law tasks from element 2 to each subgrid, each subgrid's width will be maximized. Thus, the game space grid as a whole will increase in size. This therefore forces the size of element 2 to reduce, to compensate for the distance increase to the center of the game space grid.

These two cases highlight the fact the flexibility of modeling afforded to the designer but also reiterates the importance of correctly modeling the desired scenario in order to get viable results. With the various possibilities of modeling scenarios

with only two elements specified, we can now look at multi-element cases; the most common yet most complex scenarios to produce results for.

6.3 Multi-Element Case

Now that the base case and the slightly more advanced two element cases have been explored, we now look at a greater multi-element case. In the following examples, we tested 7 HUD elements under varying scenarios to observe whether or not the results produced a valid result and, more importantly, a viable usable result.

6.3.1 Multi-Elements with Asymmetric Tasks

To start off, a simple multi-element case is conducted. Here, all 7 elements are provided as asymmetric tasks. For input, all 7 elements are specified as asymmetric tasks, an initial point for the solver is specified by us, the width to height ratio of HUD elements is set to 4 and the maximum width of each HUD element is constrained to equal the entire width of the screen. Figure 6.6 contains an example of a result for this case.

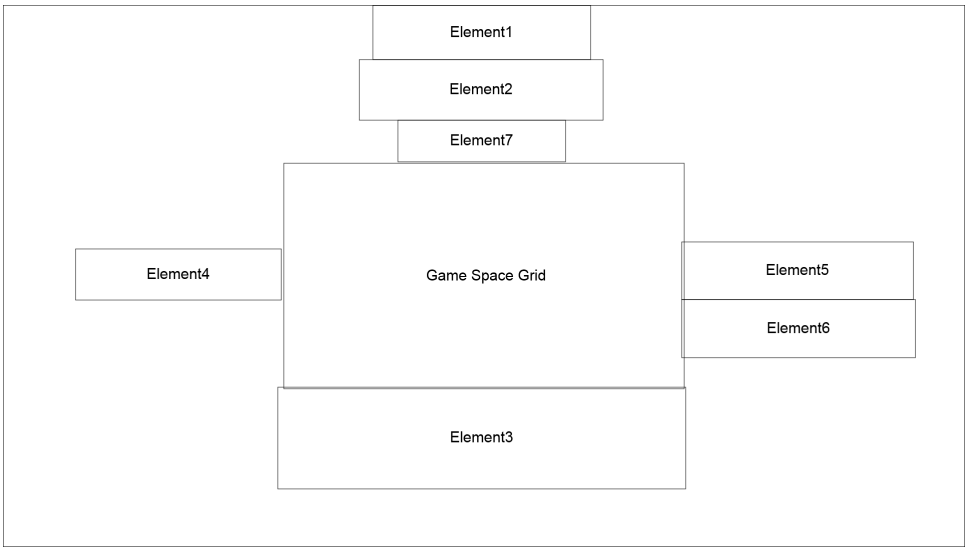

Figure 6.6: Seven HUD elements, all considered as asymmetric tasks. Task flow: (1), (2), (3), (4), (5), (6), (7)

With more than two elements, the results provided by the algorithm become harder to verify. We already know that the best location, in this widescreen scenario, for a given HUD element is centered above or below the game space grid. The next best locations for HUD elements, start to differ based on their size. If they are small enough, placing them in a diagonal position as close to the center above or below the game space grid is ideal. If they are larger however, placing them centered to the left or right of the game space grid is ideal. Their position is entirely dependent on the Fitts' law calculation for that specific position and size combination. Looking at Figure 6.6, we see there are a few elements centered above, below, to the left and to the right of the game space grid. How do we know if this is correct?

In this case, since we specified a width to height ratio of 4 for each HUD element, and since minimizing Fitts' law maximizes the targets's width, these buttons are too wide to be placed on a diagonal. Each of them are better placed centered relative

to the game space grid. We know that the result of Fitts' law maximizes the target HUD element's width, and the return to the subgrids forces the game space grid to increase in size as well. But we do not know that this is the optimal result. Running the same scenario a second time produces a result shown in Figure 6.7.

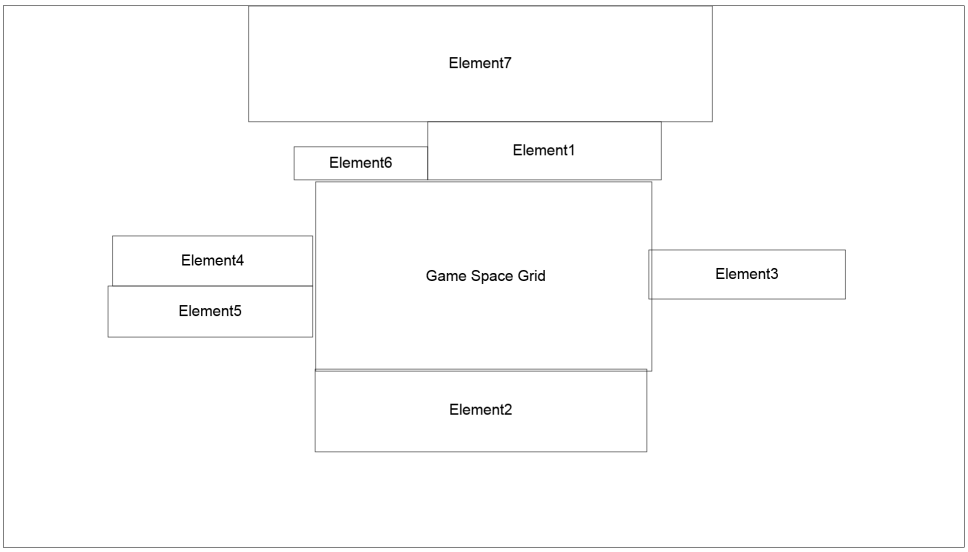

Figure 6.7: Seven HUD elements, all considered as asymmetric tasks. A second attempt.

Notice how elements 6 and 7 in Figure 6.7 possess sizes that are different than element 1 and 7 in Figure 6.6. Note that even though they are different element numbers, since they are not part of any task flows, their position relative to other elements are chosen entirely by the solver as it evaluates the objective function. The point however, is that elements further away from the game space grid are larger whilst the ones closer are smaller. In multi-element cases the differences in sizes relative to their distance are more apparent. Still the results appear peculiar. For example, in Figure 6.6, element 1 is further away but it is not larger. This is due to two reasons. The first reason is due to the minimum mesh size constraint embedded

in NOMAD, resulting in NOMAD terminating its optimization process when this limit is reached. This results in a layout that is the best feasible result found by NOMAD in a given run. The second reason, related to this minimum mesh size problem, is the fact that we provided an initial point for the NOMAD to start from. The problem is that our initial point may not be near an optimal solution at all. Combined these two problems result in NOMAD traversing possible solutions from our initial point, but terminating before any optimal solutions because the minimum mesh size is reached. This occasionally produces results like Figure 6.6, where certain elements do not necessarily seem to adhere to Fitts' law. However, this is not a fault of the algorithm.

This is easiest to verify by noting that the result of minimizing the objective function in the second attempt was a smaller value than in the first attempt. So long as we notice this to be true, then we know that the algorithm is working fine, while the limitations of the solver are at fault. Another way to verify this, is to remove the initial point provided by us, and request NOMAD to try and determine one for itself. In this case, we get a result like in Figure 6.8.

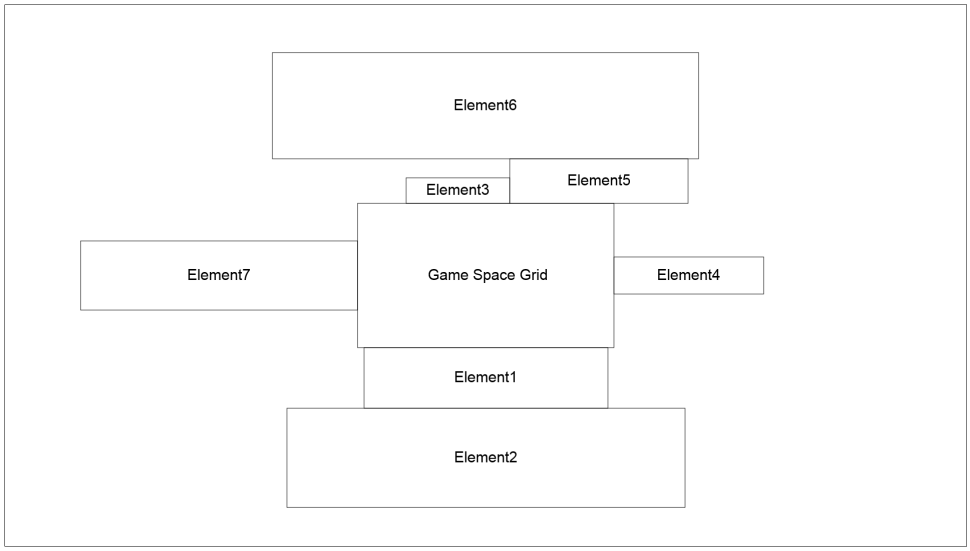

Figure 6.8: Seven HUD elements, all considered as asymmetric tasks. No initial point provided by the designer.

Here it can be observed that the correct expectations of the size of each HUD element relative to their position, due to Fitts' law, is met.

Now one could ask why these results still appear peculiar, as it would be hard pressed to find a game with a HUD layout like the previous three figures. In the previous three runs, we specified that the HUD elements were rectangular in nature with a width 4 times that of their height. If we change that condition to consider each HUD element as a square instead, we get a result like in Figure 6.9.

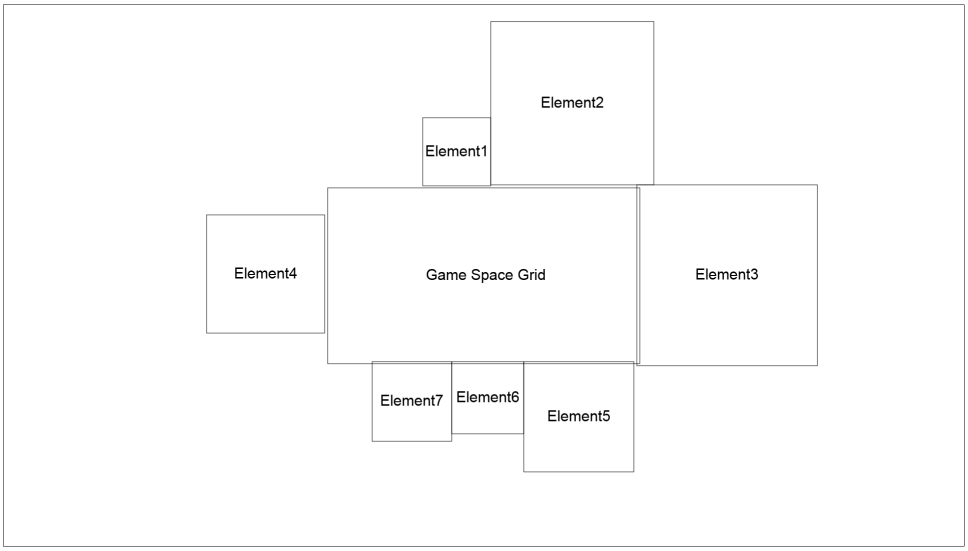

Figure 6.9: Seven HUD elements, all considered as asymmetric tasks. Initial point provided by the designer, each HUD element restricted to squares.

In this layout, each HUD element retains its square shape whilst scaling its size in both dimensions when its position is not directly centered with respect to the game space grid. However, the range of sizes for each HUD element is not constrained beyond the size of the screen. A UI designer may wish to constrain the button sizes to something more appropriate. If the designer specifies a minimum and maximum size for each HUD element, then a result like in Figure 6.10, could be produced. In this example, the minimum size of each HUD element is restricted to be 32px x 32px and the maximum size of each HUD element is restricted to be 120px x 120px.

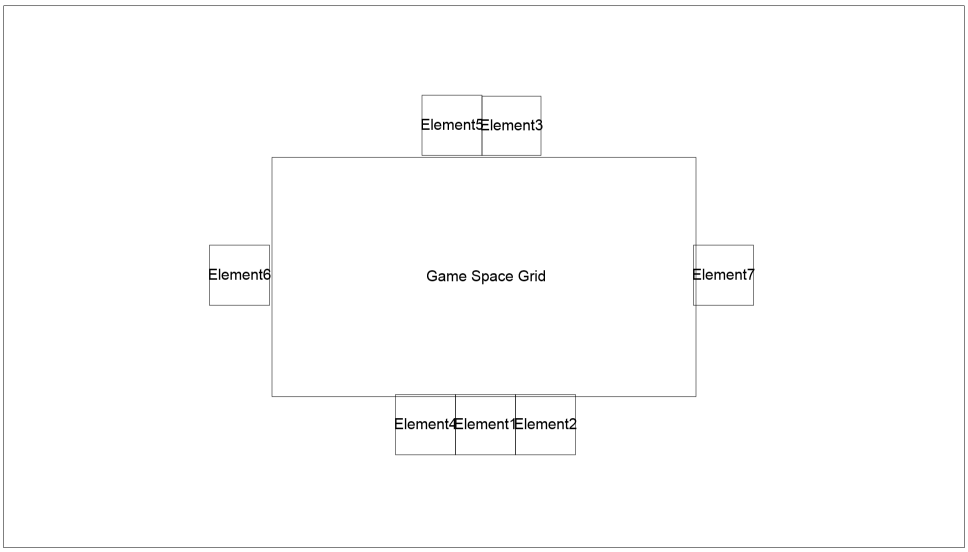

Figure 6.10: Seven HUD elements, all considered as asymmetric tasks. Initial point provided by the designer, each HUD element restricted to squares, with a smaller preset width to height ratio.

This result appears much more desirable as it reflects a possible realistic choice for a HUD layout. The game space grid is slightly larger, most of the buttons are chosen to be their maximum value and diagonal placement, of these buttons, relative to the center of the game space grid seems more favorable than the sides. In order to observe this, let us again remove the initial point provided by us to the solver and request NOMAD to try and choose one itself. In this case, a result like in Figure 6.11 is produced.

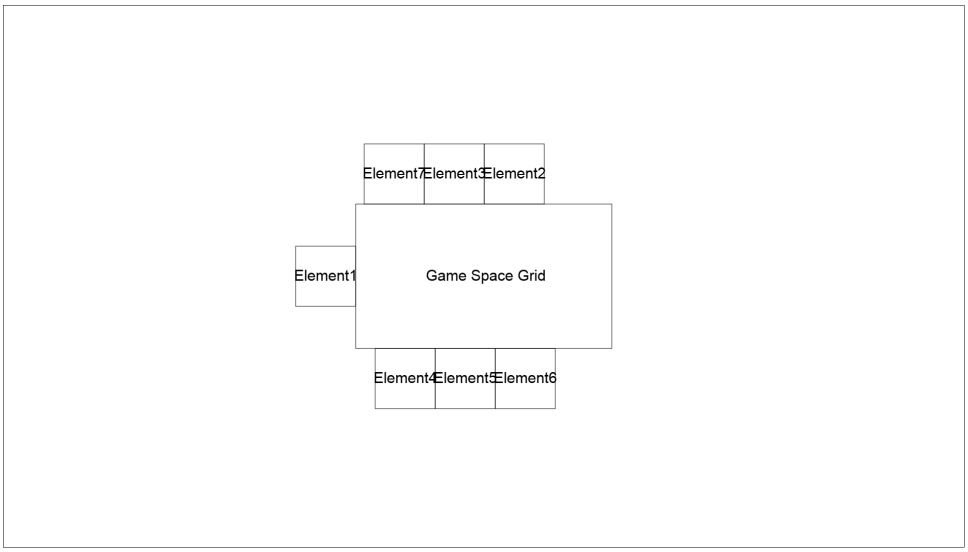

Figure 6.11: Seven HUD elements, all considered as asymmetric tasks. Initial point chosen by NOMAD, each HUD element restricted to squares, with a smaller preset width to height ratio.

An even better choice, though perhaps still not optimal. The buttons are laid out above and below the game space grid, save for one. The layout shown in Figure 6.11 is objectively a better layout, according to our algorithm, than the layout shown in Figure 6.10. The objective function result of the former case is 1.30984e+06 whereas it is 1.26372e+06 in the latter case.

In order to gain the maximum benefit of this algorithm, given the choice of solver, the designer must be very specific in their input choices in order to properly model the requirements they have. If done correctly, with multiple runs to overcome the limitations of NOMAD, HUD layouts can be produced that are optimized for task efficiency and still meet the expectations of the designers.

6.3.2 Multi-Elements with Task Flows

With multiple HUD elements, each specified in their own asymmetric task, we can begin to look at the most complex cases; multiple elements with multiple task flows. To start off, observe a simpler case akin to the simple multi-element case with 7 asymmetric tasks. Here, let us change 7 asymmetric tasks to instead request one task flow consisting of 7 HUD elements. In addition, in this case the size of each HUD element is left to be as large as possible, the elements are chosen to be rectangular, and the solver begins with its own initial point. The result of this is produced in Figure 6.12.

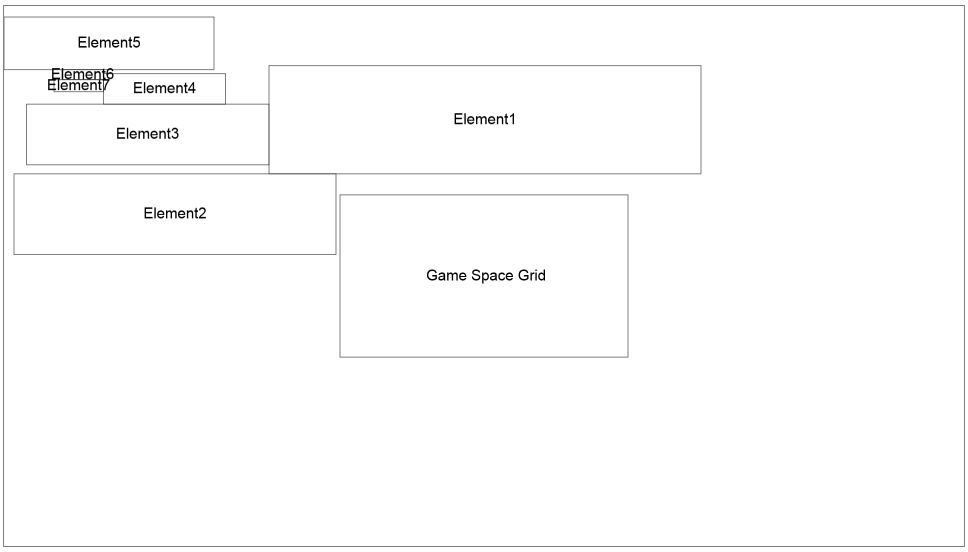

Figure 6.12: Seven HUD elements, all contained in one task flow. Task flow: ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$)

This task flow states that the player will perform a task from each element numbered 1 through 7 sequentially. The expected result then is that the distance between sequential elements is minimized and subsequent elements have larger widths. The

result, in Figure 6.12, shows the first expectation with sequential elements no longer being placed as close to the game space grid, but rather as close as possible to other HUD elements. Since elements are rectangular, their widths are chosen to be as large as possible. Note, however, that since we have non-overlapping elements as a requirement, their size naturally becomes constrained based on the available space. This is why, for example, element 7 is quite small. Again, the minimum mesh size in NOMAD is reached, and so the objective function result for this scenario is 2286.23.

For comparison sake, we limit the maximum width of HUD elements to 120px, in order to guarantee space for each element and thus verify that a better solution is placing elements in a line like manner, close to each other with each their maximum size. Allowing the solver to again choose its own initial point, we produce a result like in Figure 6.13. Here the objective function result is 2275.83; indeed smaller and thus a better choice than the previous scenario. However, if NOMAD could determine the optimal solution, then the best result would have all of these elements center aligned with the center of the game space grid.

Figure 6.13: Seven HUD elements, all contained in one task flow. The width of each element has been restricted to 120px.

Similar to the previous section with asymmetric tasks, the next scenario involves square HUD elements in a task flow, (sequentially 1 through 7). For comparison, the size of each HUD element is allowed to be as large as possible. The solver is again allowed to choose its own initial point. The result is contained in Figure 6.14.

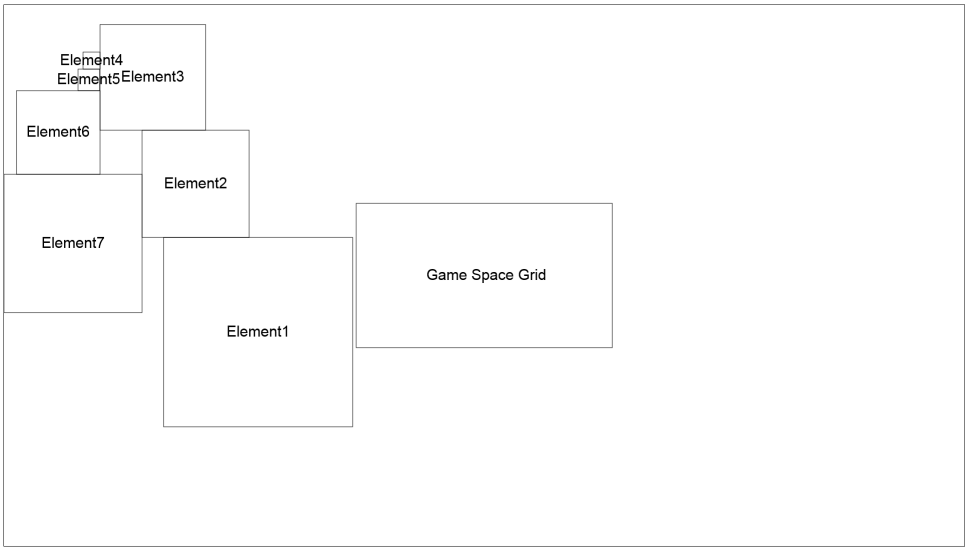

Figure 6.14: Seven HUD elements, all contained in one task flow. Each element restricted to a square shape. Task flow: $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7)$

Next, the same scenario is conducted but with a maximum width of 120px enforced on each element. Three different variations of the same scenario are presented below, to highlight the vast differences that can occur with a simple task flow requirement. Figure 6.15 contains the results.

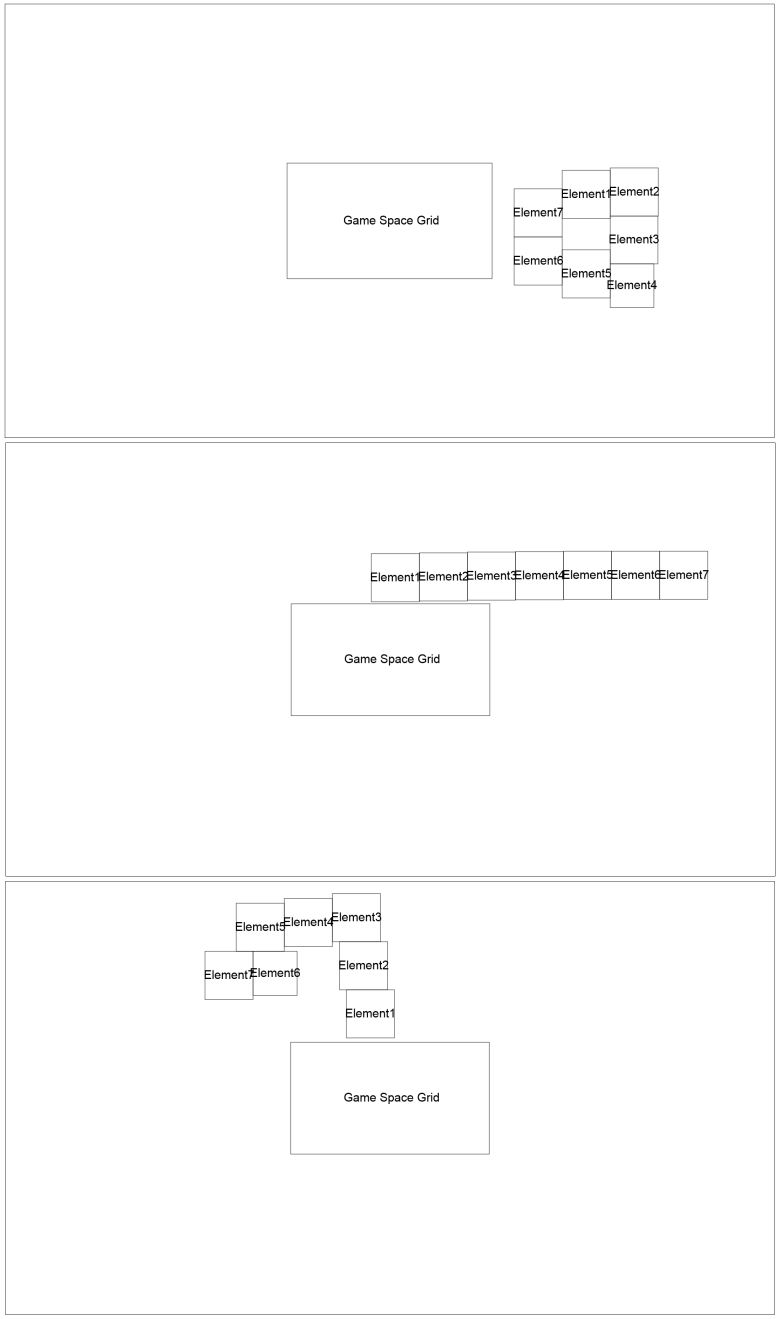

Figure 6.15: Seven HUD elements, all contained in one task flow. Each element restricted to a square shape. Task flow: $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7)$

Expanding to more complex multi-element task flow cases with multiple task

flows containing multiple tasks, in addition to asymmetric tasks, we begin to see the limitations of our optimizer more clearly. In the following case, all HUD elements are restricted to be 120px by 120px squares, in their respective task flows.

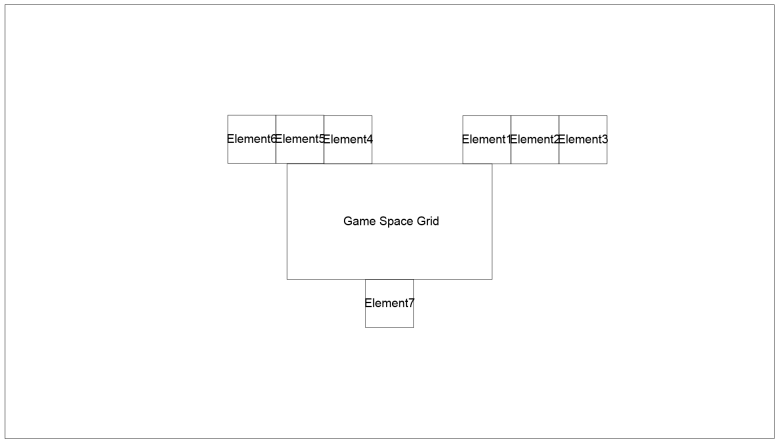

Figure 6.16: Three task flows: $(1 \rightarrow 2 \rightarrow 3), (4 \rightarrow 5 \rightarrow 6), (7)$

After several runs of the optimization process, the best result produced is presented in Figure 6.16. Here we see that elements 1 through 3 and 4 through 6 are kept as close as possible to each other, whist element 7, the only asymmetric task, is kept centered to the game space grid; that being the best position for it. The objective function result of this scenario is 176966. Running the same scenario with a better initial point, specified by us, produces a result in Figure 6.17 with an objective function result of 176889. Here the result is much more aesthetically pleasing, while also reflecting the optimal solutions that could be produced with the use of Fitts' law.

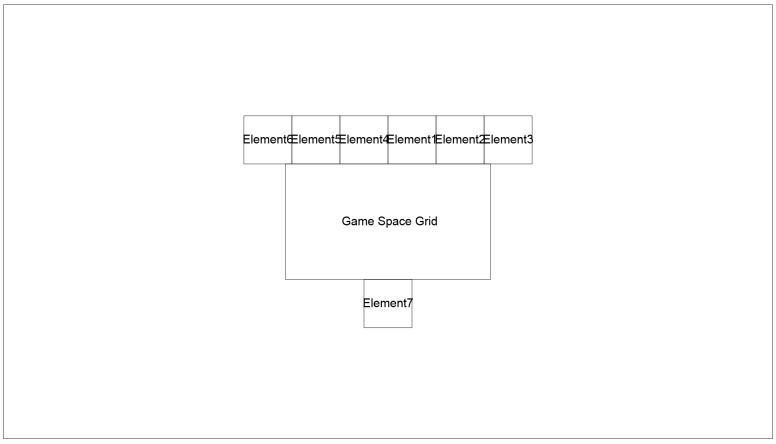

Figure 6.17: Three task flows: $(1 \rightarrow 2 \rightarrow 3), (4 \rightarrow 5 \rightarrow 6), (7)$

As can be observed, the optimizer quickly becomes unable to solve the increasingly complex problems because the number of variables and results of changes to these variables become incredibly large, leading to several local optimal or best feasible results in the given run. To combat this, we had to make several NOMAD-specific changes to allow it to derive a better solution. In the worst circumstances, we had to force the solution just to verify that our algorithm was working correctly. However, in each situation we tested, it was clear that the limitations were not contained within our algorithm. Our algorithm embeds our goal of task efficiency as defined by Fitts' law and our task model. Should another better suited piece of optimization software be used, together with our algorithm and combined with proper modeling from the designer, we believe these complex scenarios can be solved without further designer interaction.

Chapter 7

Conclusion

As display technology continues to advance and displays continue to be produced with multiple screen property combinations, the cost and time required to design and develop video game user interfaces will continue to rise. In order to reduce this development cost and time, automation will be necessary during the development process. In this work, one such method of automation was introduced. Utilizing Fitts' law, blackbox optimization software and game genre specific knowledge, an algorithm was developed to solve task efficient HUD layouts. By producing more viable layouts that adhere to the designer's specifications, rapid prototyping of multiple layouts is possible with reduced cost and development time.

7.1 Contributions

This work contributes towards the goal of introducing automation in the development process of video game user interfaces, by providing an algorithm that produces task efficient HUD layouts for tower defense games. Our algorithm consists of our

assumptions, design decisions and domain specific knowledge. Through the use of Fitts' law, NOMAD - a blackbox optimization software, and game genre specific knowledge, our algorithm can produce wireframe prototypes of HUD layouts that are optimized with task efficiency in mind. This algorithm can be adapted and included in other designer's specific development software/processes to solve their own HUD layout problems.

7.2 Future Work

As mentioned at the beginning, this work introduced one method of automation that could be used to help automate parts of the development process for video game user interfaces. There are of course additional steps that can be taken to enhance and further develop the algorithm presented here. A list of possible future work is included to indicate the next steps that can be taken.

- As mentioned before the algorithm presented here was influenced by the choice of application: tower defense games. However, with domain knowledge of other genres of video games, the algorithm can be adapted to develop task efficient HUDs for those genres.

- The algorithm presented here was influenced based on the choice of optimization software as well. One of the problems with this choice of optimization software was that, in some cases the minimum mesh size would be reached before the optimal solution was found. With the use of another piece of optimization software, the algorithm can potentially be upgraded to handle more complex cases with a higher degree of accuracy and less interaction with the designer.

- The algorithm was the key contribution of this work, whilst the software developed was used to test and verify the results of this algorithm. Thus, it was not designed as a user-friendly accessible tool. With further work, a Domain Specific Language combined with a Code Generator can be developed to provide an end user with a simple interface towards solving task efficient HUD layouts. In addition to solving these layouts, code generation could be used to create them in the designer's programming language of choice.

- User studies could be conducted on the resulting HUD layouts to observe common task flows. This could be used to help further refine the results.

Bibliography

[1] Abramson, M., Audet, C., Couture, G., Dennis, Jr., J., Le Digabel, S., and Tribes, C. (Accessed: 26-03-2014). The NOMAD project. Software available at `http://www.gerad.ca/nomad`.

[2] Abramson, M. A., Audet, C., Dennis Jr, J. E., and Digabel, S. L. (2009). Orthomads: A deterministic mads instance with orthogonal directions. *SIAM Journal on Optimization*, **20**(2), 948–966.

[3] Accot, J. and Zhai, S. (1997). Beyond fitts' law: Models for trajectory-based hci tasks. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 295–302, New York, NY, USA. ACM.

[4] Apple (Accessed: 16-06-2014). Apple developer guidelines: Icons and image sizes. `https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/IconMatrix.html`.

[5] Audet, C. and Dennis, Jr., J. E. (2006). Mesh adaptive direct search algorithms for constrained optimization. *SIAM J. on Optimization*, **17**(1), 188–217.

[6] Audet, C., Le Digabel, S., and Tribes, C. (2009). NOMAD user guide. Technical Report G-2009-37, Les cahiers du GERAD.

[7] Bisschop, J. and Roelofs, M. (2006). AIMMS - User's Guide. http://www.aimms.com/downloads/manuals/user-s-guide/.

[8] Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, **74**, 381–391.

[9] Furtado, A. W. and Santos, A. L. (2006). Using domain-specific modeling towards computer games development industrialization. In *The 6th OOPSLA Workshop on Domain-Specific Modeling (DSM06)*.

[10] Gajos, K. Z., Weld, D. S., and Wobbrock, J. O. (2010). Automatically generating personalized user interfaces with Supple. *Artificial Intelligence*, **174**(12), 910–950.

[11] Google (Accessed: 16-06-2014). Android developer guidelines: Metrics and grids. http://developer.android.com/design/style/metrics-grids.html.

[12] Hedengren, J. D., Shishavan, R. A., Powell, K. M., and Edgar, T. F. (2014). Nonlinear modeling, estimation and predictive control in APMonitor. *Computers & Chemical Engineering*.

[13] Hernandez, F. E. and Ortega, F. R. (2010). Eberos GML2D: a graphical domain-specific language for modeling 2D video games. In *Proc. 10th SPLASH Workshop on Domain-Specific Modeling, Aalto-Print*.

[14] Ironhide Game Studio (2011). Kingdom Rush. http://www.ironhidegames.com/.

[15] MacKenzie, I. S. (1989). A note on the information-theoretic basis for Fitts law. *Journal of Motor Behavior*, **21**(3), 323–330.

[16] MacKenzie, I. S. and Buxton, W. (1992). Extending Fitts' law to two-dimensional tasks. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 219–226. ACM.

[17] MacKenzie, I. S. and Teather, R. J. (2012). FittsTilt: The application of Fitts' law to tilt-based interaction. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, pages 568–577. ACM.

[18] MacKenzie, I. S. and Zhang, S. X. (1999). The design and evaluation of a high-performance soft keyboard. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 25–31. ACM.

[19] Microsoft (Accessed: 16-06-2014). Windows phone developer guidelines: Interactions and usability with windows phone. `http://msdn.microsoft.com/en-us/library/windowsphone/design/hh202889%28v=vs.105%29.aspx`.

[20] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer.

[21] Pleuss, A., Botterweck, G., and Dhungana, D. (2010). Integrating automated product derivation and individual user interface design. *VaMoS*, **10**, 69–76.

[22] PopCap Games (2009). Plants vs. Zombies. `http://www.popcap.com`.

[23] Schmitt, A. and Oel, P. (1999). Calculation of totally optimized button configurations using Fitts' law. In *Proceedings of HCI International (the 8th International Conference on Human-Computer Interaction) on Human-Computer Interaction:*

Ergonomics and User Interfaces-Volume I - Volume I, pages 392–396, Hillsdale, NJ, USA. L. Erlbaum Associates Inc.

[24] Subatomic Studios (2008). Fieldrunners. `http://subatomicstudios.com/`.

[25] Ware, C. and Mikaelian, H. H. (1987). An evaluation of an eye tracker as a device for computer input2. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, CHI '87, pages 183–188, New York, NY, USA. ACM.

[26] Zeidler, C., Lutteroth, C., Sturzlinger, W., and Weber, G. (2013). The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, NY, USA. ACM.