AMBIENT CROSS-MEDIA BOARD GAMES:
GAME ENGINE, TERRAIN EDITOR, EXTENSIBLE INTERFACE AND
ADAPTIVE PLUGGABLE INPUT

by

YANNIS LILIS

MASTER’S THESIS

Heraklion, March 2008
Abstract

The focus of this Thesis is the marriage of ambient interactions and computer games towards a form of socially-engaging and group-entertaining activities. To this end, three primary requirements have been put forward: (i) to support cross-media presentations, adaptive pluggable input with accessible add-ons, and extensible user interface system; (ii) to focus on a generic game genre suitable for ambient delivery, in particular board games, emphasizing social dynamics, political correctness, and cross-cultural acceptance; and (iii) to deliver an integrated development platform, comprising the game engine and a full-fledged terrain editor.

The key outcome of this work is the on-board! game platform comprising: (a) a board game engine suited to ambient setups, supporting configurable roundtable player positions, adaptive pluggable input infrastructure, extensible in-game dialogues and remote interface-feedback services; (b) an integrated terrain editor providing a directed graph model for board terrains consisting of game-path vertices, while facilitating the authoring of terrain components’ libraries (sub-graphs as bitmaps with connectivity information), game-board composition from path elements with game items, and game logic scripting in the Delta language. Finally, a fully working board game entitled The Four Elements has been built using the on-board! platform.
ΔΙΑΧΥΤΑ ΠΟΛΛΑΠΛΩΝ ΜΕΣΩΝ ΕΠΙΤΡΑΠΕΖΙΑ ΠΑΙΧΝΙΔΙΑ:
ΜΗΧΑΝΗ, ΠΕΡΙΒΑΛΛΟΝ ΣΧΕΔΙΑΣΗΣ, ΕΠΕΚΤΑΣΙΜΗ
ΔΙΕΠΑΦΗ, ΚΑΙ ΠΡΟΣΑΡΜΟΣΙΜΗ ΕΙΣΟΔΟΣ ΜΕ PLUG-INS

ΙΩΑΝΝΗΣ ΛΙΛΗΣ

Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Η παρούσα εργασία εστιάζει στο πάντρεμα διεπαφών για περιβάλλοντα διάχυτης ευφυϊας με ιππαγιδία υπολογιστών προς μία μορφή ψυχαγωγικών δραστηριοτήτων με έμφαση στην κοινωνικότητα και ομαδικότητα. Για το σκοπό αυτό, τρεις βασικές απαιτήσεις οριοθετήθηκαν: (i) η υποστήριξη παρουσιάσεων σε πολλαπλά μέσα, προσαρμόσιμη είσοδος με τη μορφή plug-ins υποστηρίζοντας επιπλέον προσβασιμότητα, και επεκτάσιμο σύστημα διεπαφής, (ii) η επικέντρωση σε μία γενική κατηγορία παιχνιδιού πρόσφορης για απόδοση μέσω διάχυτης διεπαφής, ειδικότερα τα επιτραπέζια παιχνίδια, με έμφαση στην κοινωνική δυναμική, πολιτική ορθότητα, και διαπολιτισμική αποδοχή, και (iii) η παροχή μίας ολοκληρωμένης πλατφόρμας ανάπτυξης, περιλαμβάνοντας τόσο τη μηχανή του παιχνιδιού όσο και το περιβάλλον σχεδίασης παιχνιδιών.

Το κύριο αποτέλεσμα της εργασίας είναι η πλατφόρμα παιχνιδιών on-board! που περιλαμβάνει: (α) μια μηχανή επιτραπέζιων παιχνιδιών κατάλληλη για διάχυτες διεπαφές, που υποστηρίζει προσαρμόσιμες θέσεις παικτών «γύρω από ένα τραπέζι», υποδομή καναλιών προσαρμόσιμης είσοδου μέσω plug-ins, επέκταση των διαλόγων του παιχνιδιού, και δικτυακές υπηρεσίες ανάδρασης της διεπαφής, (β) ένα ολοκληρωμένο περιβάλλον σχεδίασης που παρέχει έναν κατευθυνόμενο γράφο ως μοντέλο δομής του χώρου του παιχνιδιού, αποτελούμενο από κόμβους διαδρομών παιχνιδιού, προσφέροντας ταυτόχρονα τη δημιουργία και διαχείριση βιβλιοθηκών δομικών τμημάτων (υπογράφων με γραφικά και πληροφορία συνδεσμότητας), τη
σύνθεση παιχνιδιών από δομικά τμήματα και αντικείμενα παιχνιδιού, και την υλοποίηση της λογικής του παιχνιδιού στη γλώσσα Delta. Τέλος, ένα πλήρως λειτουργικό επιτραπέζιο παιχνίδι με τίτλο The Four Elements έχει αναπτυχθεί μέσω της πλατφόρμας on-board!.
Ευχαριστίες (Acknowledgements)

Θα ήθελα να ευχαριστήσω τον επόπτη της μεταπτυχιακής μου εργασίας Αντώνιο Σαββίδη για την συνεχή καθοδήγηση και υποστήριξή του τα τελευταία τρία χρόνια στο πλαίσιο της συνεργασίας μας στο Εργαστήριο Επικοινωνιών Ανθρώπου-Μηχανής, του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας και ειδικότερα στο πλαίσιο της εκπόνησης της μεταπτυχιακής μου εργασίας.

Θα ήθελα επίσης να ευχαριστήσω την οικογένειά μου και τους φίλους μου για τη βοήθεια και τη στήριξη που μου παρείχαν όλα αυτά τα χρόνια.
# Table of contents

Abstract ......................................................................................................................... iv
Περίληψη ....................................................................................................................... v
Ευχαριστίες (Acknowledgements) ................................................................. vii
Table of contents ........................................................................................................ viii
List of tables ................................................................................................................. xii
List of tables ................................................................................................................. xii

1. Introduction ............................................................................................................ 1
2. Related work .......................................................................................................... 4
3. Game Engine .......................................................................................................... 7
   3.1 Software Architecture ................................................................................ 7
   3.2 Generic Terrain Structure ........................................................................ 8
   3.3 Board Game Elements ............................................................................. 10
      3.3.1 Cross-Points ................................................................................. 10
      3.3.2 Collectable items and Player Inventory ...................................... 11
      3.3.3 Decorations ................................................................................... 12
      3.3.4 Game Events ................................................................................ 12
      3.3.5 Preconditions ............................................................................... 14
      3.3.6 Game Variables ........................................................................... 15
   3.4 Script Management ..................................................................................... 15
   3.5 Game Loop and Play-State Transitions ................................................. 17
   3.6 Board Game Table and Player Positioning ............................................. 18
4. Terrain Editor ....................................................................................................... 20
   4.1 Interactive Pattern Element Design .................................................... 20
      4.1.1 Terrain Node Construction ......................................................... 20
      4.1.2 Pattern Element Intra- and Inter-Connectivity ............................ 21
   4.2 Organizing Pattern Element Galleries ............................................... 23
   4.3 Game World Construction ................................................................ 25
      4.3.1 Terrain Editor Architecture ..................................................... 25
      4.3.2 Terrain Grid Area ...................................................................... 27
      4.3.3 Toolbox ...................................................................................... 27
      4.3.4 Pattern Element Gallery Manager .......................................... 30
      4.3.5 Property Manager ..................................................................... 30
List of figures

Figure 1 – Ambient interface ......................................................................................... 3
Figure 2 – Game engine core architecture ........................................................................ 7
Figure 3 – Board Game Engine Macro Architecture ..................................................... 8
Figure 4 – Visual representation and graph structure of a pattern element ................. 9
Figure 5 – Script Management Organization ............................................................... 16
Figure 6 – Task scheduling state transitions ................................................................ 18
Figure 7 – Support of customized interface based on player positioning .................... 19
Figure 8 – Terrain Element Editor: Corner insertion using edge gravity ................... 21
Figure 9 – Terrain Element Editor: Inner and outer connectivity ............................... 23
Figure 10 – Tile Set Editor Interface ........................................................................... 24
Figure 11 – Terrain Editor Architecture .................................................................... 26
Figure 12 – Terrain Editor Interface ............................................................................ 26
Figure 13 – Insertion of a pattern element ................................................................ 28
Figure 14 – Copy semantics of inner pattern element items for the copy operations.. 29
Figure 15 – Property Manager interface ...................................................................... 31
Figure 16 – Terrain Editor and Sparrow interaction interface ..................................... 34
Figure 17 – Input dialogue automata .......................................................................... 35
Figure 18 – Dice rolling animation................................................................................ 37
Figure 19 – Path selection logical dialogue structure (automaton)............................... 38
Figure 20 – Path selection input dialogue ................................................................... 38
Figure 21 – Path tracing using highlighting ................................................................ 39
Figure 22 – Inventory control logical dialogue structure (automaton)........................ 40
Figure 23 – Inventory management through the inventory input dialogue ................. 41
Figure 24 – The Dice Rolling Service Interface .......................................................... 42
Figure 25 – The Game Monitoring Interface ............................................................... 43
Figure 26 – The Inventory Service Interface ............................................................... 44
Figure 27 – The Default Inventory Service Graphical User Interface ......................... 45
Figure 28 – The Media Service Interface .................................................................... 45
Figure 29 – The Mini Game Service Interface ............................................................ 46
Figure 30 – Virtual Input Device Meta-Architecture ................................................. 48
Figure 31 – Virtual Pointing Device Architecture ...................................................... 51
Figure 32 – Input dialogues and Scanning Interaction ................................................ 52
Figure 33 – Virtual Switch Pad Device Architecture .................................................. 53
Figure 34 – Remote Input Device Architecture ........................................................... 54
Figure 35 – A sample Remote Input Client providing all available input controls .... 55
Figure 36 – Adaptation process used for automatic input device selection .......... 60
Figure 37 – Input device adaptation logic in DMSL ................................................... 61
Figure 38 – “The Four Elements” game world ............................................................ 64
List of tables

Table 1 – Command interface for native Input Dialogues...................................................36
1. Introduction

Playing games has always been an important activity in our lives. Many classic board games like Chess or Go have literally thousands of years of successful history, providing an educational and enjoyable activity for people of all ages. Recently, with all the technological advancements in the field of computers, vast possibilities have emerged regarding new interesting computer games. Being able to perform enormous complex calculations and having so realistic graphics and sound effects, computer games can take us into a totally new gaming experience, carrying us into believable immense game worlds. With such an edge on the gaming experience computer games have dominated the game market. Still, traditional board games keep going strong and outsell their computer rivals in many countries around the world [1].

This actually is not proved to be as absurd as it initially seems. The reason is that both types of gaming have their unique elements that make them surpass the other in certain aspects, making them both worth playing. Traditional board games clearly have the edge on the social interaction they provide. Almost all of these games are made for multiplayer use and each game session is usually organized as a social event, where friends get together and pass their time in a pleasant cohesive manner. It is exactly this human-oriented interaction that makes playing board games such a pleasant social activity [2]. Of course, when advancing to more complex game rules and dynamically changing worlds, the static nature of a traditional board game restricts it from being able to provide the desired gaming experience. This is where computer games come in. With their rich visual and audio support (amazing 3D graphics, video playback, sound effects, narration, etc.) and their ability to deliver dynamic content, the possibilities to create new interesting games are endless. The focus here is on delivering a better game-play, but that comes with the cost of having the interaction shift towards the computer side. Both in single player computer games as well as in multiplayer ones that provide co-located multiplayer modes, the interaction is still primarily mediated by a computer screen.

The goal is to try and combine the best elements of the two worlds forming a new class of hybrid games – often called augmented computer board games – that will
benefit from the advantages of computing technology while emphasizing on social interaction. To preserve the social situation and the group dynamics of face-to-face game sessions, the game board on a table surface should remain the primary interface while being augmented with beneficial features known from computer games [3].

The primary objectives of this Thesis are:

- To design and develop a generic board game framework.
- To design and develop an interactive designer tool able to produce game world specifications for instantiating specific board games through the framework.
- To provide an extensible interface both directly through the framework itself as well as externally, through the support of an artificial avatar that is aware of the game process and can interact with players during game-play.
- To provide inherent framework support for adaptive pluggable multimodal input.

We focus on ambient board games, emphasizing social dynamics, political correctness, and cross-cultural acceptance. With respect to the desired focus on the social interaction, we avoid supporting computer players; all participants of the game are human players interacting with each other.

Regarding the setup of the ambient environment where our framework is to be deployed, a couple of choices are viable. The first one is the roundtable setup, where the players sit around a table that incorporates the basic board game interface. This table can uses top or bottom projection for displaying the game world, or just provide a large display for it. Additionally, there is the ambient interface extension, facilitating extra displays around the table that can be used for additional player feedback and interaction. Additionally, we have the gameland setup where the game world is projected on a wall and the ambient interface extension is provided either on the same wall or on some other successive wall. The targeted setups are illustrated in Figure 1.
Specifically for the board game engine and its interface, we focus on games that can be represented by some graph-based terrain structure and have boards of less or equal size with the resolution of the screen. A game world larger than the screen resolution would require scrolling of the game terrain in order to display every part of it. Such an approach is however problematic, as it fails to consistently show all players at once (for instance if four players have their pawns at the four corners of the table) and as it makes the use of physical pawns inconsistent with respect to their relative and absolute positions. The former approach is therefore selected, providing a mapping of the game’s board directly to the screen.
2. Related work

The idea of combining the physical and the virtual world in the area of games has been around for some time and there are quite a few examples of this idea being implemented in various ways. One of the first examples is *PingPongPlus* [4] developed and presented by Ishii et al. in their attempt to augment a traditional ping pong game table with several output modalities such as sound and graphical effects projected at the table surface. Later on, Schneider and Kortuem [5] described *Pervasive Clue*, an augmented version of the classic live-action role-playing game *Clue*. This environment of the game was designed to enhance live-action games providing a testing ground for sociability enhancing applications. At about the same time, Björk et al. [6] presented a hybrid game system called “*Pirates!*” that does not use a typical game board but integrates the whole world around the players, transforming physical locations and player movements into virtual analogs that are inserted into a classic computer game. In this sense, “*Pirates!*” integrates virtual and physical components in the game application allowing physical interaction between the players and their environment whilst keeping the computer game benefits.

In [2], Mandryk et al. exploring the space between board and video games, created a hybrid game that leverages the advantages of both physical and digital media. A custom sensor interface promotes physical interaction around the shared public display while the un-oriented tabletop display encourages players to focus on each other rather than on the interface of the game. Sometime later, Andersen et al. [7] presented an augmented reality board game called “*BattleBoard 3D*” that features on-screen projection of the virtual battles between the physical pieces of the two sides while in [8], Wagner et al. presented a system architecture for interactive, infrastructure-independent multi-user augmented reality applications and developed “*Invisible Train*” a mobile, collaborative interactive multi-user game, in which players control virtual trains on physical terrain with railroad tracks. Finally, in [9] Magerkurth et al. presented STARS, a platform for developing computer augmented board games that integrate mobile devices with an interactive table. The aim of STARS is to augment traditional board games with computing functionality, but
without sacrificing the human-centered interaction dynamics of traditional tabletop games.

Comparing the work described so far with our own, we conclude that while a lot of work has been performed in the field of both general augmented games and augmented board games in particular, none of that has resulted in an extensible and reusable board game framework like on-board!. In this sense, our work takes the process of creating augmented board games to the next level, abstracting carefully selected critical elements of standard board games and providing them as an open framework for the creation of any game encompassing similar elements. Additionally, this is the first case of an augmented board game able to provide an extensible user interface and support an artificial avatar that interacts with the players and game state. Finally, regarding the use of multimodal input we provide the same functionality as the one used in state of the art input mechanisms, using input interface layers and plugging actual device using these interfaces.

Regarding augmented board game design, [10] compares various possible electronic augmentations of traditional board games:

- Integrated digital rules
- Error protection and prevention
- Animated game examples
- Electronic teaching
- Various randomizations (e.g. changing game board)
- Automated administrative tasks
- Time information
- Score information
- Game movements
- Player statistics
- Audiovisual feedback
- Artificial intelligence
- Simulation of players
- Suggestion of moves
• Save and restore games
• Physical feedback
• Cross-media applications
• Communication between game instances
• Transparent added complexity.

This work concludes that the most wanted enhancements are the randomly changing board game, the board composition, the integrated digital game rules that can enable automatic error protection and prevention as well as the simulation of additional players. Additionally, [12] explores existing examples of augmented games and specific tasks associated with them, resulting in a categorization of some common board game tasks and giving hints on how to augment them. This information can be used as a guideline for creation of augmented board games and is applied in the design of the building blocks that constitute our generic board game framework.
3. Game Engine

3.1 Software Architecture

The on-board! Game Engine is structured upon a 2D based Game Engine Core. The Game Engine Core features the use of sprites that are displayed through rendering of animation films (i.e. bitmaps containing the object’s visual states and transitions) and can be animated in various ways – specified by the animations – through use of external animators. The architecture of the Game Engine Core is depicted in Figure 2.

Building on this core, the Board Game Engine is enriched with various specific board game elements (e.g. inventories, collectable items, cross-points, nodes, etc) and terrain structure support. Additionally, it provides a game event system for triggering actions based on player input or game state, along with a precondition system able to restrict or allow triggered event execution. Moreover the flow of the board game is divided into semantic task activities and a task management module is responsible for managing and orchestrating such tasks. The Board Game Engine also encompasses sophisticated input management, supporting pluggable and multimodal input, as well as scripting support for implementing the game logic. Finally, it features an inherent mechanism for deploying external services, thus providing an extensible interface. The macro architecture of the Board Game Engine is depicted in Figure 3.
3.2 Generic Terrain Structure

The Game Engine provides a generic terrain structure applicable to any board game. This structure consists of a directed graph, with each of its nodes representing a higher level game node (i.e. a board game “square”), while its connectivity also reflects the connectivity of the game nodes, in the sense that for a player to cross from a game node to another there must always be an edge in the underlying graph connecting the respective terrain nodes. In order to have a visual representation of the graph for the terrain to be displayed, the whole terrain or smaller parts of the terrain should be associated with bitmaps illustrating the underlying nodes and terrain structure. The case of having a single bitmap for the whole terrain provides no adjustability for the terrain structure and is therefore rejected. In order to provide multiple possibilities for the board layout as well as reuse bitmaps for parts of the terrain graph, a tiling technique is applied. Specifically, we divide the terrain board into small parts, called tiles (a typical size for such a tile is 64x64), turning it into a tile grid. This grid can then be filled with elements that have dimensions multiple of one tile, called pattern elements. These pattern elements are associated with a bitmap used for visualizing and some polygon areas representing terrain nodes and their corresponding game nodes. Additionally, they contain connectivity information.
regarding the way the inner nodes are connected to each other as well as how they can be connected to other nodes. Figure 4 illustrates the internal (i.e. graph) and the external (i.e. visual) representation of a pattern element.

![Visual representation of a pattern element](image1) ![Graph structure of the pattern element](image2)

**Figure 4** – Visual representation and graph structure of a pattern element

In this sense, each pattern element represents an internal sub-graph that can be connected with other such sub-graphs through its external edges. This is performed by placing pattern elements to the terrain tile grid in such a way that an outgoing edge of one element matches an incoming edge of another. These two pattern elements are therefore connectable, forming a joined graph. In the same way all pattern elements inserted to the terrain grid are combined together, thus forming the complete terrain graph.

The game nodes, each of which is associated with a node of the graph, are critical elements of the board game proving the terrain positions that players can access and move through, while providing a placeholder for additional game elements like cross-points, collectable items and game events. Specifically, a game node can contain any number of collectable items but at most one cross-point and can have any number of game events associated with it. These events can be triggered whenever a player passes over a node through the course of their movement (On Pass events) or when the player ends their movement onto some node (On Terminal events).

One last concept related to the terrain of the game is that of a *path*. A path is an ordered list of continuously connected game nodes. Paths are used throughout player movement providing traversal of the terrain nodes through which players pass to get from the beginning node to the ending node of their movement.
3.3 **Board Game Elements**

3.3.1 **Cross-Points**

Cross-points are specific interaction points located at terrain nodes that allow or prevent passage to a player based on some precondition. Any time a player tries to pass from a cross-point, its associated precondition is evaluated for the given player to determine whether or not they will be allowed to pass. Cross-points are divided into two categories: the *plain cross-points* that have an empty precondition and always allow passage and the *doors* that always have a non-empty precondition associated. When a player reaches a door, the system tries to decide if the player is able to pass through the door by looking if its associated condition can be pre-evaluated. For instance in the case of a precondition that needs a certain score or possession of a specific item the system can pre-evaluate the precondition immediately upon reaching the door, while in the case of preconditions that involve rolling a certain dice throw or depend on scripted functions there is no way to decide before actual evaluation is performed. When being able to pre-evaluate a player’s failure to meet the precondition of a door, the movement is stopped and the player is informed of the failure reason. In case of pre-evaluated success or inability to pre-evaluate the precondition, the system proceeds to ask the player if he/she wants to pass through the door notifying him regarding the associated precondition. When the player chooses to pass, the precondition evaluation is actually performed deciding to permit passage or not. Upon successful passage the player is allowed to continue with the remaining movement, while in any other case he is notified of the failure cause and his movement (as well as turn) ends.

Upon successful passage from a cross-point, that cross-point is added to the player’s *pass-list* used to determine if the player has passed from specific terrain nodes. Additionally, a game event is triggered by the cross-point, namely a *Cross-Point Pass* or a *Cross-Point Fail* event, corresponding to whether the player managed to pass or not. For plain cross-points the former is always triggered while the latter is never triggered.
3.3.2 Collectable items and Player Inventory

Collectable items are game elements that can be gathered from the terrain or awarded to a player as the outcome of an event. The gathering of an item from the terrain can be performed either while passing from a node that contains the item or upon ending the move of the specific node, based on each node’s settings. Additionally, an item located on the terrain may have an associated precondition that a player has to meet in order to collect it. Whenever able to collect an item from the terrain the user is asked if he actually wants to collect it being also notified about its potential precondition. If there are more than one items present at a node, the player is asked for each one separately.

Each item is stored in the player’s inventory, a placeholder for inserting collectable items. The inventory has a fixed capacity (i.e. total weight) – adjustable through game events – regarding the items it can hold while each item has a certain weight associated with it. Through their inventories, players can select an item and interact with either by dropping to the terrain or by using it. Dropping an item simply places it on the player’s current node for the same or another player to collect it. Regarding usage, each item has a certain number of times it can be used before it is consumed (that can also be infinite). Each time a player uses an item the amount of its usages is decreased. When an item is consumed, it is deleted from the player’s inventory relieving it from its weight.

Several game events can be associated with a single item. Specifically, the events that a collectable item can trigger are the following:

- *Get Item Event*, triggered upon collecting the item from the terrain.
- *Use Item Event*, triggered upon using the item.
- *Drop Item Event*, triggered upon dropping the item.
- *Lose Item Event*, triggered when a game event causes the player to lose the item. The item is removed from the inventory but not put back on the terrain.
3.3.3 Decorations

The decorations are simple bitmaps or animated films that can be placed throughout the terrain. Decorations have no inherent functional role regarding the game process. Their sole purpose is to enhance the visual representation of the game world, helping to build the virtual game world and creating a better game atmosphere for the player. This property makes them quite important elements of the board game engine. An interesting thing to note is that even though the decorations have no built-in functional role, the game designer is free to check their position and animation status at any time and perform any kind of event using this information as input. This way, the designer can extend the functionality of a core game engine element through the use of scripting. The same thing also applies to all visual game elements, but is more likely to be used in the context of a decoration item since it has no functional role, in contrast to other game elements that already have some game specific role.

3.3.4 Game Events

Game events are triggered actions that can occur during game-play. Such events can be triggered as a result of player interaction, when a player reaches some terrain interaction point or upon performing an item related action (e.g. dropping an item on the terrain, using an item, etc.) but also as a result of the current game state, for instance at the beginning of a turn, or when a player skips their turn due to a trap. In particular, there are four element categories through which a game can be triggered:

- Node related Events
- Item related Events
- Cross-point Events
- Turn Events

The first three categories are analyzed in their respective sections and thus no further explanation is necessary. The turn events can be triggered whenever a player skips a turn or gets to play an extra turn, as well as when the turn of player begins or ends.
Besides the variety of elements able to trigger an event, there is also a wide selection of available game event types. These types include the following:

- **Function Events** (C++ or scripts), performing the function specific with the current player and game context. Can have arbitrary results.
- **Mini Game Events**, instructing the player to play an external mini game. The result of such a mini game is usually information regarding the player’s performance in the mini game, possibly used by an additional event.
- **Multimedia Events**, instructing an external service to deliver multimedia content.
- **Commands** toward a player.
- **Combination of the above (Binary And Game Event)**

The Commands can be further categorized as follows:

- **Movement to specific terrain node**
- **Gain or loss of a number of turns**
- **Score change** (negative or positive)
- **Inventory Capacity change** (negative or positive)
- **Get or lose a collectable item**
- **Gain or loss of life points**
- **Change of a game variable state**
- **Instant win or loss**

The game event types described above are actually **Game Event Classes**. In this sense a command that gives a player a specific score bonus forms a whole game event class representing score bonus of the specific amount. In order for these events to be actually triggered, we need to instantiate **Game Event Instances**. Each such event instance has an associated event class containing the type of the event to be performed, a number of total activations it is able to produce as well as some precondition enabling selective triggering of this event instance. Finally, specifically for event instances corresponding to node events, a bitmap or an animated film may also be applied to provide the visual context of this game event.
3.3.5 Preconditions

Throughout the course of the game there are game actions or events that need to be performed that are based on some condition. Such game conditions are called precondition in the sense that they are the prerequisite for the triggering of an event or game action. A precondition may need to be evaluated upon crossing a door, getting an item or upon activation of a game event. The types of preconditions supported by the Game Engine are the following:

- **Have Score Precondition**, requiring the player to have some score amount.
- **Have Life Precondition**, requiring the player to have some life total.
- **Have Item Precondition**, requiring the player to have a specific item.
- **Use Item Precondition**, requiring the player to use an item.
- **Drop Item Precondition**, requiring the player to drop some item.
- **Pass Cross-point Precondition**, requiring the player to have passed from a specific cross-point.
- **Win Mini Game Precondition**, requiring the player to play and win an external mini game.
- **Dice Throw Precondition**, requiring the player to achieve a certain roll of the dice.
- **Game Variable Precondition**, requiring an internal game variable to have a specific value.
- **Function Precondition**, that refers to a C++ or a script function while requiring that function to provide a successful evaluation based on the current player and game context.
- **Combination of Precondition** (*Binary And Precondition*, *Binary Or Precondition* and *Unary Not Precondition*), requiring the player to satisfy the specified precondition combination.
3.3.6 Game Variables

Game variables are designer defined named variables that can store a runtime value. The value of a Game Variable can change as a result of a game event, while it is mainly used for evaluating preconditions of the corresponding type. Strictly, there is no need for explicit usage of such variables, since their function is directly provided by variables existent in some script. In this case, a Game Variable Precondition could be replaced by a Function Precondition querying about the value of the script variable and likewise an event causing a Game Variable to change state could be replaced by a Function Event performing the corresponding change within the script. Nevertheless, they are essential in the construction of board games that use no scripting, but describe their game logic using only built-in game event types. In such cases, using a Game Variable is the only possible way to store a game value. If on the other hand a board game uses scripting, Game Variables are simply used as a convenience, minimizing additional script functions that would otherwise be used for getting and setting the value of a script variable.

3.4 Script Management

The Game Engine provides a mechanism for extending the game logic of a concrete game through the use of scripting. The component responsible for loading compiled scripts and executing them upon request of the Game Engine is the Script Manager. The language selected for the scripting support is the Delta Programming Language. The Delta programming language [13] is an imperative scripting language that encompasses (a) dynamically typed variables, (b) runtime classes, (c) functions as first-class values, (d) unnamed functions, (e) dynamic handling of actual arguments, (f) extensible operator semantics and (g) dynamic inheritance. The selection of the Delta language was based both on the variety of feature it provides as well as on the fact that its implementation is in C++, thus allowing easier integration within our Game Engine also implemented in C++.
In order to provide scripting support, the board game has to export any interface that the developer of a script may need regarding game interaction. This includes the Game Core Engine with all its rendering and animation interfaces as well as most of the Board Game Engine components such as Collectable items, Game Events, Precondition, Nodes, etc. Exporting of the interfaces to be used by the scripting logic may seem a great overhead at first, but pays out through the later use of the language features that make coding the game script much more efficient that coding in the native language.

Upon initialization of the game world, the Script Manager is provided with a folder path containing the script files associated with the game world. The folder given is actually the workspace directory of the Delta IDE used to create the game scripts (see section 4.4 for more details on the scripting environment) containing the projects and the scripts of the game world in a hierarchical way. Nevertheless, the Script Manager loads recursively all compiled scripts found in the given directory. Each of these scripts is also executed to perform any necessary initializations. During the course of the game, the Game Engine requests the invocation of delta function through scripted function game events. The Script Manager simply interprets the given data to locate the corresponding active script and function to call and performs the invocation. The described organization regarding script Management is illustrated in Figure 5.

![Figure 5 – Script Management Organization](image-url)
3.5 Game Loop and Play-State Transitions

The game loop of a traditional Game Engine consists more or less of the following basic elements:

- Input Management
- Artificial Intelligence Management
- Animation Progress
- Collision Checking
- Rendering

The same principle applies for the Board Game Engine as well slightly modified to best fit its specific properties related to a board game. Input management, Animation Progress and Rendering all apply here as well so they remain unchanged. Since there are no artificial characters in (the board game is played between actual players) the artificial intelligence management is dropped. Additionally, there are no game elements subject to collision checking, so this is dropped as well. Moreover, there is a need for a new component able to orchestrate the next game action to be performed, handling some sort of a game automaton with game states being its state transitions. This new component is named Task Scheduler and is responsible for orchestrating the game related tasks. It is therefore important to classify that various game tasks by analyzing the game flow. Deploying such a classification, the following entities are realized as semantic tasks: (a) Turn Beginning, (b) Dice rolling, (c) Path Marking, (d) Movement Step, (e) Player Action, (f) Player Getting Item, (g) Door Crossing and (h) Game Event Handling. Some of these tasks can also be further classified into sub-tasks, like the crossing of a door which requires first checking if the player is able to cross, then ask him if he wants to cross and finally performing the door crossing. However, the former categorization is more compact and providing an adequate categorization while minimizing the state transition of the task sequence automaton.

The turn of a player begins with a Turn Beginning task. It is then followed by a Dice Rolling task and a Path Marking task. A series of Movement Step tasks follows that continues until finishing the movement (at which time we return to a Turn Beginning task) or until some game or user input schedules one of the Player Action, Player
*Getting Item, Door Crossing or Game Event Handling* tasks. After the scheduling and completion of each of these tasks, the Movement Step task continues as usual again until it is interrupted or until it finishes. The task scheduling state transitions are also illustrated in Figure 6.

**Figure 6** – Task scheduling state transitions

### 3.6 Board Game Table and Player Positioning

The targeted setting for experiencing board game created with the Board Game Engine is an actual table board where the players can play using actual pawns. The input device to be used for player interaction can be either a touch screen recognizing cursor (simulated through hand pointing) and pawn positioning or some normal screen combined with some external vision mechanism capable of providing the same kind of input. In the absence of necessary hardware, the board game can also be played on a standard personal computer using the mouse to provide any pointing input needed. In this case physical pawns are not supported; instead virtual on-screen pawns take their place while maintaining their original functionality.

The Board Game Engine supports a total of ten players to simultaneously participate in a board game session. The wider sides of the table support up to 3 players each, while the others support up to 2 players each. Based on player positioning, the Board
Game adapts itself in order to provide the best possible interface for each player. In particular, the players’ inventories and dialogues are positioned right in front of them while being properly rotated to face them directly. This is the case regarding the setting of the board table. When the game is played on a computer screen, the positioning of each player items’ remains the same, but no additional rotation is applied as all players are facing the screen from the same side. Figure 7 illustrates the support of multiple players and the customized interface the Board Game adapts for each one of them.

Figure 7 – Support of customized interface based on player positioning
4. Terrain Editor

4.1 Interactive Pattern Element Design

The first step in the process of creating a board game is the creation of the pattern elements that will later be used for the terrain construction. This is performed using the Terrain Element Editor tool. Using this tool, the designer inputs bitmaps that have dimensions multiples of a single tile (for instance the single tile can have dimensions 64x64 pixels) along with the node and connectivity information corresponding to the given bitmap and gets the pattern element specification file as the output. This file contains dimension of the pattern element relative to the single tile, the node information as a set of point sequences (i.e. polygons), the inner connectivity as set of directional edges between nodes and the outer connectivity as directed external edges beginning at an inner node and going through a part of a side of the pattern element (number of parts each side is divides to is based on the pattern element dimensions), all encoded in xml format.

4.1.1 Terrain Node Construction

Having selected the bitmap to use for a specific pattern element, it is presented to the screen for the user to interactively enter the node information. Each node is represented by a polygon whose corners are specified by the user. The addition of corners to the polygon is simply performed by left-clicking at some point of the bitmap, while the sequence of the corners is determined by the clicking sequence following a clockwise style. Likewise, deletion of a corner is performed by right-clicking on an already existing corner while the polygon is active. In order to increase efficiency of polygon creation as well as minimize the time needed for this task (adding a single corner would require reconstruction of the whole polygon following the previously mentioned technique) we use an edge based gravity technique to determine exactly where the newly added corner is to be placed regarding the polygon corner sequence. Specifically, when the mouse passes over some edge of the active
polygon, this edge becomes the active edge for insertion meaning that addition of a corner would place the new corner between the two corners forming the active edge. This is visualized to the user by a pair of dashed lines connecting the active edge corners with the mouse point position giving a preview of the new polygon after the oncoming corner addition. This behavior is illustrated in Figure 8.

Figure 8 – Terrain Element Editor: Corner insertion using edge gravity

The gravity technique also applies to the corners of all polygons (not only the active one). This is used for placing a polygon corner at exactly the same point as a corner of another polygon, something required to automatically determine the adjacent polygons (two polygons that have two or more common corners are adjacent). Specifically, when the mouse gets within the range of an existing corner, it is automatically gravitated to that corner, thus allowing selection precise positioning of the new corner.

4.1.2 Pattern Element Intra- and Inter-Connectivity

Apart from the node creation, the user can also interactively enter the connectivity information for the pattern element. The connectivity information is divided into intra-connectivity, which is the connectivity between the nodes of a single pattern element (i.e. inner connectivity with respect to a pattern element) and the inter-connectivity, which is the connectivity between two pattern elements (i.e. outer connectivity with respect to a single pattern element).
The inner connectivity between two adjacent nodes is visualized by highlighting the adjacent edge (or edges). Let node1 and node2 be two nodes having a common edge (i.e. are adjacent). There are three possible cases for them:

1. \( \text{Node1} \rightarrow \text{Node2} \) and \( \text{Node2} \rightarrow \text{Node1} \) are both valid movements. The adjacent edge is classified as bidirectional for both nodes (green highlighting).

2. \( \text{Node1} \rightarrow \text{Node2} \) is valid but \( \text{Node2} \rightarrow \text{Node1} \) is not. The edge is classified as outgoing for node1 (blue highlighting) and as incoming for node2 (red highlighting).

3. \( \text{Node2} \rightarrow \text{Node1} \) is valid but \( \text{Node1} \rightarrow \text{Node2} \) is not. Likewise, the edge is classified as outgoing for node2 (blue highlighting) and as incoming for node1 (red highlighting).

The user can set and change the current connectivity setting between two nodes by simply left-clicking on the adjacent edge, toggling between the three possible states, or remove the connectivity information by right-clicking on it.

Outer connectivity, used for connecting pattern elements with each other, matches a node with a specific side of the pattern element (i.e. left, right, up or down) and a specific part of the side (based on the dimension of a pattern element each side has a specific number of parts). In this sense, a pattern element with outer connectivity only from the down side should only be placed directly above a pattern element with outer connectivity from the up side (the specific parts of each side also have to match but this depends on the exact positioning of both pattern elements on the terrain grid upon terrain construction – see section Game World Construction 4.3 for more details). Similarly to inner connectivity, the type of outer connectivity can be incoming, outgoing or bidirectional, maintaining the same color highlighting (red, blue and green respectively) and again the user can set and change the current connectivity setting by left-clicking on the according part of the desired side, toggling between the three possible states, or remove the connectivity information by right-clicking on it.

The node corresponding to this side part is automatically associated by finding a polygon with an edge close to the specific side part. In case of an unsuccessful matching, the closest node is selected and the user is warned to check that this is actually the desired connectivity setting.

The visualization of both inner and outer connectivity is depicted in Figure 9.
4.2 Organizing Pattern Element Galleries

For a board game with a large number of pattern elements it is essential to organize them in some fashion in order to handle them in the most efficient way. This organization can be performed using the Tile Set Editor tool. Using this tool, the user can easily create and manage galleries containing pattern elements.

In order to create a gallery the user can create new pattern elements from bitmaps using the Terrain Element Editor (which is incorporated into this tool), or import already existing pattern elements. Of course, loading of existing galleries is also supported. The Tile Set Editor supports multiple open pattern element galleries and has a fully customizable graphical user interface allowing the open pattern element galleries to be placed at a variety of docking areas. This way, the user can easily move of copy pattern elements from one gallery to another using the drag and drop technique (simple drag and drop for move, drag and drop holding down control for copy). Moreover, the user is able to hide and show specific pattern elements at will as well as reorder the gallery elements. Finally, the Tile Set Editor provides theme support for its galleries. The user can add any number of themes for each gallery simply by providing a folder with the alternative pattern element bitmaps and the rest
work is automatically performed. However, there are some restrictions related to the theme support. Firstly, the bitmaps provided for each new theme should have the same names as the original bitmaps in order to automatically incorporate them into the gallery and secondly the insertion of new pattern elements becomes a bit complicated having to provide a bitmap for each theme (or alternatively keep the same bitmap for all themes and then manually make the replacement with the proper one). Nevertheless, theme support is a very useful and helpful feature, eliminating the need for replicating pattern element information and providing an additional organization layer for the pattern element galleries.

With all these features the Tile Set Editor makes pattern element gallery management an easy task. Some additional of its features that also make it user friendly include the universal drag and drop support (dropping existing galleries open them, dropping existing pattern elements adds them to the current gallery and dropping of supported bitmaps adds them to the current gallery creating pattern for them), the auto adjusting grid display of the gallery based on the size of the window as well as the keyboard binding for all possible actions. The interface of the Tile Set Editor is illustrated in Figure 10.

![Figure 10 – Tile Set Editor Interface](image-url)
4.3 Game World Construction

Using the pattern element galleries created from the Tile Set Editor, we now have the building blocks to begin the construction of the game world. The tool for this construction is on-board! Terrain Editor, where the user can put together pattern galleries to form a concrete terrain structure, add collectable items, cross-points and decorations on the terrain, mark specific nodes for interaction and associate preconditions and events with the game elements. Section 4.3.1 gives an overview of the Terrain Editor architecture, while the following sections focus on its most important Graphical User Interface components.

4.3.1 Terrain Editor Architecture

Since the Terrain Editor is a tool for interactive authoring of a game world terrain, the Terrain component is naturally fundamental regarding the architecture of the system. It is the component that coordinates all other components to provide the desired functionality of the system. It uses the Toolbox to be aware of the current interaction mode (insertion of some object, deletion, selection, etc.) in order to interpret each user action accordingly. Upon pattern element insertion, it uses the Pattern Gallery Manager to get the selected Pattern Element for the insertion, while for item insertions, it creates a new item adding it to the terrain and registering it to the Property Manager, through which the user can later customize its properties. Additionally, when moving or copying and pasting pattern elements, it uses the Clipboard to assist performing the task. Finally, any of these operations are registered as undoable operations – referring either to terrain items or pattern elements – for the support of the undo/redo mechanism. The architecture of the Terrain Editor is depicted in Figure 11, while Figure 12 illustrates its interface and the close relation between its architectural and its Graphical User Interface components.
Figure 11 – Terrain Editor Architecture

Figure 12 – Terrain Editor Interface
4.3.2 Terrain Grid Area

The core of the Terrain Editor interface is the central area of the window containing the open terrains of the game worlds the user is editing. Each terrain consists of a tile grid of fixed width and height specified at construction time. This grid is the placeholder for inserting pattern elements at specific positions in order to form the desired path structures. On these elements, and specifically on the nodes they contain, the user can later add collectable items and game cross-points, while also begin able to place decoration elements at any position of the grid. Additionally, based on the current operation mode the user is able to move or copy and paste selected terrain elements along with the items they contain. Such copying and pasting can also be performed between different game worlds, allowing the user to reuse specific path structures previously created for another terrain or game world. In order to facilitate the copying and pasting process this component uses an internal Clipboard able to hold intermediate terrain element items.

4.3.3 Toolbox

The Terrain Editor uses a shared Toolbox for its various opened terrains. The operations supported by this toolbox are: insertion of terrain objects (specifically pattern elements, collectable items, cross-points and decorations), selection of terrain objects, deletion of terrain objects, and terrain panning.

When the pattern element tool is active, the user can select a pattern element from the open pattern galleries (or import a new gallery and select one of its patterns) and insert it to the terrain at a specific position. While moving the mouse to reach the desired position, a preview of the resulting placement is always available to the user. Additionally, when placing a pattern element on top of one or more existing elements, all overlapping elements are removed from the terrain (i.e. are overwritten by the newly placed element). This also applies to any occasion at which one or more pattern elements are placed onto the terrain, like pasting and moving of pattern element selections. The insertion of a pattern element is illustrated in Figure 13.
Apart from the pattern element insertion, the user can also choose to insert collectable items, cross-points or decoration items to the terrain. Each of these insertions can be performed by selection the corresponding tool from the toolbox. Collectable items and cross-points can only be inserted at a node of a pattern element, as they are interactive game play elements. Specifically for the cross-points, only one may be placed at any node therefore addition of a second one at a specific node overwrites the first instance. Decoration items, as non-interactive game play elements, can be placed at any point of the terrain, regardless of the existence or not of a specific node or even pattern element. After placing any of these object items to the terrain, the user can then select them using the selection tool and customize its properties through the Property Manager. Any graphical change performed in the properties of terrain object is directly reflected to the terrain grid interface.

The selection tool can be used to select any number of pattern elements along with any collectable items or cross-point they may contain to allow performing massive operation onto a whole set of pattern elements. Such massive operations include copying (copy-paste as well as drag and drop style), cutting, moving (both simple and justified move) and deleting the entire selection. Whenever a selection of elements contains additional items, the copying, cutting and moving operations have the expected semantics regarding the cloning or not of each item present in the element selection. Specifically, upon copying an element selection, all additional items are
also copied to the newly created elements, cloning the attributes of the corresponding original items and getting a different unique identifier (i.e. copy by value is performed). Upon cutting an element selection, the additional items remain unchanged for the first paste (i.e. copy by reference) and get copied for each additional paste. Finally, upon movement of a selection, the items also remain unchanged. This behavior is illustrated in Figure 14.

![Figure 14](image)

**Figure 14** – Copy semantics of inner pattern element items for the copy operations

Continuing with the selection tool, the user can also select a collectable item, cross-point or decoration item from the terrain as well as a specific named terrain node, while activating the corresponding item property page in the Property Manager. Specifically for the nodes, there is no tool for explicit named node creation; instead a named node is created upon first selection and further customized from the Property Manager. Finally, using the selection tool the user can specify a set of pattern elements and change their current theme setting choosing one of their gallery’s supported themes.

The eraser tool can be used to delete any terrain object simply by clicking on it. The eraser first checks if the clicked position corresponds to a collectable item, cross-point or named node (and in this exact order in case of overlapping objects) removing it from its terrain node. If no such item is found, the eraser then checks for possible decoration item and even if that fails, it deletes the pattern element present on the clicked position (if any).

The last toolbox element is the pan tool. Using this tool, the user can navigate through the terrain grid by changing the current terrain view through dragging and releasing of the terrain. While not an essential tool for inputting data to the game world, the pan tool provides a fast and efficient way of terrain navigation, especially helpful for large terrains and small screen resolutions.
4.3.4 Pattern Element Gallery Manager

In order to insert pattern elements to the terrain, the user must first have pattern element galleries to choose from. The Pattern Gallery Manager is used to import existing pattern galleries previously created with the Tile Set Editor and provide them to the user for selecting the appropriate pattern element to insert to the terrain. The interface of the Pattern Gallery Manager resembles the one of the Tile Set Editor, providing management of multiple pattern element galleries along with their adjustable interface positioning and the ability to show only specific pattern elements of each gallery. The Pattern Gallery Manager is shared between the currently open terrain instances containing all pattern galleries centrally. The first time a terrain uses a pattern element from a new – for this specific terrain – gallery, that gallery is automatically appended to the terrain’s pattern gallery dependency list. This way, whenever a terrain is loaded, the pattern galleries it depends on are also opened automatically eliminating the need for additional user actions to construct the gallery working set. In the same sense, a pattern gallery should never be removed from the Manager while there is still some active terrain depending on it, meaning that when closing a terrain, it is not always proper to close of all its gallery dependencies. This can be easily overcome by looking for common gallery dependencies between a terrain being closed and all the active ones.

4.3.5 Property Manager

Each collectable item, cross-point, named node or decoration item placed on the terrain has various attributes that need to be customized for it to perform its functional role in the game world. Additionally, there are the preconditions and game events that need to be declared and customized, along with some other general properties regarding the game world. The interface component responsible for supporting all these features is the Property Manager, featured in Figure 15. The Property Manager contains a list of property pages for each object category, namely for collectable items (i.e. inventory items), cross-points, named nodes, decorations, preconditions, game events as well as an additional property page for some general game world properties. Each of these lists contains all active elements of the specified category. This way, the
user has all the element of a specific category gathered together and organized in such a way that allows quick customization of their properties. Specifically for items that are inserted to the terrain, the selection of an item property page is also reflected in the terrain grid area with the selection of the corresponding item. Regarding the preconditions and game events that are not items not inserted to the terrain, the user can create them simply by selecting the “New…” item in the corresponding list of items and then further customize them as usual. Specifically for the game events, the property page interface also provides the link to the game scripting support with the interpretation of project, script and function properties of a *scripted function* event as hierarchical project structures in an Integrated Development Environment (IDE) and the ability to create and manage scripts and functions using this IDE simply by interacting with the game event’s property page. Further details about the scripting support is provided in section 4.4 regarding Game Logic Scripting.

Figure 15 – Property Manager interface
4.3.6 Undo / Redo Mechanism

An important feature of any editing application, especially one that features construction of game world, is the support of an undo mechanism that is capable of undoing and redoing the effects of any operation that changes the current state. In order to provide such a mechanism for a given set of operations, we have to ensure that each operation $A^+$ in the given set has an inverse operation $A^-$ that if applied right after the original, nullifies its effect. Following this technique, all terrain related operations (like inserting or erasing pattern elements or selectable items, moving or copying them, etc.) are implemented to always be accompanied by their inverse operation. For instance, the reverse of an insert operation is a delete operation (possibly affecting more than one element), the reverse of a move operation is another move operation that restores the original position of an object, etc.

Each game world instance has its own state and therefore a private undo queue containing the inverse operations of the ones performed during its editing. Each additional operation performed inserts the invert operation in the undo queue. When the user chooses to undo a number of operations, the corresponding operations from the undo queue are removed and performed in the exact inverse order in which they occurred, thus restoring the desired state. Meanwhile, whenever an operation is removed from the undo queue it inserts its invert operation (i.e. the original operation) in another queue, the redo queue used for redoing the original operation effects. In the same manner, when the user chooses to redo an operation, it is removed from the redo queue and performed while inserting again its inverse to the undo queue. This way, the user is able to undo and redo any action for a game world instance, allowing him to focus on the constructive editing of the world rather than worry about negative consequences of an accidentally mistaken action.
4.4 Game Logic Scripting

Having completed the game world construction, positioned the various collectable items, cross-points and decorations, as well as registered the game preconditions and events, the only thing left to complete a concrete board game is to implement the game logic corresponding to the function events registered during the construction. This game logic can be implemented in native C++, linked together with the Board Game Engine library to form the final game executable, or using a scripting language, loading the game scripts at startup and executing them when needed. A combination of the two implementation methods is of course supported. As mentioned in previous sections, the language selected for the Board Game Engine scripting support is the Delta language. Additionally, in order to provide interactive scripting support while registering the game events in the Terrain Editor, we use an Integrated Development Environment (IDE) developed for Delta language, the Sparrow IDE. Sparrow, described extensively in [14] and [15], provides a remote deployment interface through which a remote client is able to interact with the IDE components. Extending this interface and adjusting it to the specific needs of the game event scripting, we end up with a Board Game Remote Deployment Interface ready to be used by the Terrain Editor to provide interactive scripting support.

When the Terrain Editor is launched, it also launches the Sparrow IDE for the scripting support (upon failure to launch Sparrow, the user is simply notified that interactive scripting support is turned off). Upon creation of a new game world, a correspondingly named workspace is automatically created and loaded in Sparrow. Additionally, any action that changes the current game world, as loading an existing game world, or switching between already open game worlds in the Terrain Editor interface, is also reflected in the Sparrow interface. In this sense, the active Sparrow workspace always corresponds to the active game world of the Terrain Editor. The interactive scripting support is triggered any time the user interacts with a game event of scripted function type. The user can hierarchically organize a scripted function, explicitly declaring its project within the workspace, script within the specific project as well as function name in the specific script. References to inexistent project, script, or function objects result to automatic creation of the entire hierarchy in the Sparrow
workspace. Likewise, renaming the project, script or function related to a scripted function event results in the renaming of the corresponding object in Sparrow. Finally, the user is able to jump from the Terrain Editor interface directly to the code segment of a script that contains a specified function simply by double-clicking on the function name property. Similar functionality can be found in IDEs supporting Graphical User Interface (GUI) Designers able to directly jump to the code segment of a UI component upon its interaction through the Designer. The big difference between the two cases is that such an IDE is aware of both its GUI Designer as well as Code Designer, while in our case the Code Designer (i.e. Sparrow) is actually a third party application incorporated into an existing GUI Designer (i.e. Terrain Editor). Still, the combination of the Terrain Editor and the Sparrow IDE yields a user friendly interaction interface making even the game logic scripting an easier task.

Figure 16 illustrates the interaction between the Terrain Editor and the Sparrow IDE.
5. Extensible User Interface

5.1 Animated User-Defined Input Dialogues

An Input Dialogue is a graphical user interface component able to interact with the user through pointing input (i.e. the user selects a dialogue item by pointing and clicking it). Each Input Dialogue contains a list of clickable items able to fire commands upon being clicked. It also provides an interface for registering game events to be mapped for each command fired. This way the Game Engine being aware of the current state of the game can register specific actions for each dialogue item of an Input Dialogue, show the dialogue to the player and let him make a choice, automatically triggering the correct chain of game events based on the clicked item. In this sense, each dialogue is associated with a state transition automaton. For each transition we have a corresponding event and overall an event system enabling callbacks to be associated for key dialogue progress points. The callbacks are responsible for activating dialogues, which is essentially the overall logic for the entered state. The connection between actual input, input dialogues and final output provided to the player is illustrated in Figure 17.

![Figure 17 – Input dialogue automata](image)

The Input Dialogues are fully customizable and extensible by any user of the Game Engine (i.e. developer of a specific game instance). The user can define new input dialogue types, specifying a name and animation film (i.e. image containing the various states of the dialogue) for them along with a list of commands. Each such command consists of a rectangular area corresponding to the bounding box of the
represented clickable item, a frame to be displayed upon clicking the specific item and of course a name to be used for identifying this item when registering game events for this Input Dialogue. The name of each command can either have a user defined value enabling the registration of custom game events, or be one of the reserved input commands that trigger built-in game events automatically handled by the dialogue itself. The currently supported reserved commands are: move and close. The move command is used for re-positioning a dialogue to a different position of the game surface, while the close command is used to hide a dialogue (only applicable to modal dialogues – dialogues initiated from the flow of the game).

With this extensive customization the user can create a wide variety of dialogue types based on needs of the specific game. However, the use of Input Dialogues is inherent in the Game engine and consequently, the Game Engine expects some specific dialogue instances (and additionally of specific types) to be existent in order to have native Input Dialogue support. The native Input Dialogue set consists of the “path selection” dialogue, the “get item” dialogue, the “pass door” dialogue, the “or precondition” dialogue, the “inventory management” dialogue and the “inventory show” dialogue. The command interface that the native Input Dialogues have to implement is illustrated in Table 1.

<table>
<thead>
<tr>
<th>Dialogue Name</th>
<th>Dialogue Type</th>
<th>Dialogue Command Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path selection</td>
<td>Path selection</td>
<td>previous, next, select, [move]</td>
</tr>
<tr>
<td>Get item</td>
<td>Boolean selection (yes/no)</td>
<td>yes, no, [move]</td>
</tr>
<tr>
<td>Pass door</td>
<td>Boolean selection (yes/no)</td>
<td>yes, no, [move]</td>
</tr>
<tr>
<td>Or precondition</td>
<td>Boolean selection (1/2)</td>
<td>first, second, [move]</td>
</tr>
<tr>
<td>Inventory management</td>
<td>Inventory management</td>
<td>previous, next, use, drop, hide, [move]</td>
</tr>
<tr>
<td>Inventory show</td>
<td>Inventory show</td>
<td>show, [move]</td>
</tr>
</tbody>
</table>

Table 1 – Command interface for native Input Dialogues
Each native Input Dialogue is instantiated once for each player and assigned a corresponding player identifier. Based on where each player is sitting, his dialogues are positioned close to him and also rotated in order to provide him with their best possible view. When a dialogue needs to be shown to the player, it appears from the edge of the table (at the position where the player is sitting) performing an animated move until it reaches the desired position. Likewise, upon hiding the dialogue it animatedly disappears to the same edge of the table. While the dialogue is active its items can be clicked by the player using pointing input. When checking for pointing input to determine whether or not a dialogue item was clicked, the player source of the input device must match the player identifier of the dialogue for the “hit test” to become true. This way, each player can only interact with his dialogues disallowing situations when one player would accidentally (or not) click on a dialogue item of another player. Additionally, in a case when two or more active dialogues of a specific player happen to overlap (this should not be the case in the default configuration, but the players are free to move their dialogues around so such a situation may actually occur), a dialogue item of the one in the front takes higher priority in the pointing input checking as would be the case in any common dialogue based system.

5.2 Rolling Dice, Path Selection and Path Tracing

Each turn begins with the current player rolling a dice to determine the amount of nodes he can traverse during his current turn. The rolling of the dice is by default implemented using a virtual dice that provides a random movement amount, while showing an animation of a dice being rolled (see Figure 18).

Figure 18 – Dice rolling animation
After rolling the dice, the player has a movement amount of the game squares (i.e. nodes) he can traverse during his current turn. According to the movement amount, a set of the possible paths is calculated using a depth first search like algorithm on the directed graph created by the terrain nodes. The user is not allowed to pass more than once per turn from a specific node. Such moves are invalid and therefore not shown to the user. The calculation of these paths does not take into account game events that can occur during a movement of the player along the path or some door that the player may not be able to pass; all possible paths will be presented to the user. With the set of paths ready, the player can select the path he wants to follow by inputting direct keypad or switch pad events or by using pointing input to activate the path selection input dialogue (see section 6 for details on user input). The players can iterate through the possible paths by selecting to view the previous or the next path and start movement by selecting the current path. Figure 19 presents the path selection control automaton, illustrating interaction between the user input and corresponding game engine actions, while Figure 20 shows a sample implementation of the path selection input dialogue.

![Figure 19](image19.png)  
*Figure 19 – Path selection logical dialogue structure (automaton)*

![Figure 20](image20.png)  
*Figure 20 – Path selection input dialogue*
During the path iteration, the user can see the full path for his movement (not only the starting and ending points) through an animated highlighting of the path nodes. This is useful in case there are more than one routes resulting to the same position, probably containing different collectable items as well as associated game events.

After successfully selecting a path, the movement of the player begins through the use of a highlighted path tracing technique. Using this technique, a highlighting animation is initiated at the beginning of the path, starting to move along the path until it reaches its end. This is illustrated in Figure 21. During this animation, whenever a game event is triggered, or whenever user input is required, the animation is paused until any pending event is completed and then it resumes, continuing its move along the path. Such pending events include for instance events triggered by passing through a node, event triggered by collecting an item, events triggered by the usage of an item (this applies to all players using some item) or event events recursively triggered by some other event. The path tracing continues in the same way until the movement is completed at which point the turn of the player ends.

As mentioned in previous sections, the Game Engine supports user input through the use of a physical (or virtually simulated) pawn. The pawn can be used to mark specific nodes as interaction points during the movement of the player. This way, the player can move his pawn at some node along in the path, forcing the animated path tracing to stop when it reaches the specific node and ask for user input. The same technique can be used for the path selection as well. When a given path is highlighted as the current path, the player may place his pawn at any node within the path. Doing so sets the selected node as an interaction point, while implying the selection of the current path (which leads to the specific node), so the path tracing begins and will stop at the selected node to ask for user input. The pawn positioning mechanism relies
on the use of pointing input. Short of pointing input, marking an interaction point is restricted to only marking the current player node and can be performed using keypad input.

5.3 Inventory Control Activities

A player is able to interact with his inventory at all times. This interaction can be performed through direct player keypad or switch pad input or by using pointing input to activate the inventory input dialogues (see section 6 for details on user input). Input given by the user is interpreted as a logical command, which then triggers game engine actions and subsequent invocations of remote feedback service methods. Figure 22 presents the inventory control, illustrating interaction between the game engine logic and the actions performed by the remote feedback inventory service, while Figure 23 shows the visual representation of the default inventory implementation.

Figure 22 – Inventory control logical dialogue structure (automaton)
Regarding inventory interaction, the player can first of all show or hide his inventory at will, allowing him to browse through it whenever necessary as well as protecting it from opponent peeping (the latter applies in the case of a public inventory display either on the board screen or at some shared external display). When a player’s inventory is visible, he can navigate through the items it contains by moving to the next or previous item. Additionally, he can interact with his inventory as well as the rest of the game by selecting to use or drop an item from the inventory, triggering any event associated with the specific item interaction. It is important to note that this can be performed at any time, even during another player’s turn. This way, all players can potentially interact with the game flow rendering them active throughout the whole game duration. Such behavior provides a sense of real time game interaction that combined with the traditional turn taking applied amplifies the player’s interest and focus on the game, providing him an improved gaming experience.

5.4 Remotely Deployed Feedback Services

The Game Engine has been developed with an inherent support of remotely deployed feedback and interaction services. In this sense, the Game Engine has an a priori knowledge of some service interfaces and is able to interact with concrete instances of these interfaces that can be dynamically plugged into the system. To support such a mechanism, the use of the abstract factory software design pattern was deployed while combined with the use of dynamically linked libraries, thus forming a plug-in mechanism (for implementation detail regarding plug-ins see section 6.3.2). Each service that can be deployed to the system is plugged in with some additional parameters for its usage. In particular the number of the active instances each service
supports (i.e. if it is a singleton or not) as well as if the existence of each service is mandatory for the Game Engine to function. In case of a mandatory service failing to successfully plug in to the system, the Game Engine terminates with a corresponding error message. If on the other hand the service is optional the Game Engine uses a default version of the service that resides within the Engine itself. This default version can provide a full functional service component, or provide partial or even no functionality (i.e. empty service implementation). The pluggable services supported by the Game Engine are presented in the following sections.

5.4.1 The Dice Rolling Service

The Dice Rolling Service provides a mechanism for determining the number of “squares” the player can move his pawn for his current turn. This “dice roll” can be virtual, accompanied by some animation of a dice being rolled (or the spinning of a wheel, or any other process producing a movement amount) shown either at the board screen or a player’s PDA or even projected at an external shared screen. It can also be physical, with players actually rolling some custom dice and a vision system recognizing the output of the roll and giving it to the Game Engine. The Dice Rolling Service is semantically a singleton service and each of its implementation should implement the interface illustrated in Figure 24. Additionally, it has a default implementation that provides a visualization of a virtual dice throw that returns a random movement amount. For testing and demonstration purposes, in the default implementation the user is also allowed to cheat a bit by explicitly specifying the number of the dice roll.

```cpp
class DiceRollingInterface {
public:
    virtual void RollDice (void) = 0;
    virtual bool RollFinished (void) const = 0;
    virtual unsigned GetRoll (void) const = 0;
};
```

Figure 24 – The Dice Rolling Service Interface
5.4.2 The Game Monitoring Service

The Game Monitoring Service provides a monitoring mechanism for the various game events. Such a mechanism can be used for profiling purposes regarding the timing and number of occurrence of each game event. More importantly, it can be used to provide feedback to an external ambient system capable of interpreting the game events and transforming them into higher level game independent events (i.e. some player is the leading one, a player was lucky in the past few turns, etc.) and able to interact with the players through socially-oriented game narration and emotional feedback by keeping a history of such high level events and interpreting them based on some adaptable logic system.

```cpp
class GameMonitoringInterface {
public:
    void NotifyGotoNode (unsigned player, string node, bool stop, time t);
    void NotifyScoreChange(unsigned player, int score, time t);
    void NotifyExtraTurns (unsigned player, int turns, time t);
    void NotifyLifeChange (unsigned player, int life, time t);
    void NotifyInventoryCapacityChange(unsigned player, float capacity, time t);
    void NotifyVariableChange(string variable, string value, time t);
    void NotifyTerminal ( unsigned player, bool win, time t);
    void NotifyGetItem ( unsigned player, string item, bool playerInitiated, time t);
    void NotifyLoseItem ( unsigned player, string item, time t);
    void NotifyDropItem ( unsigned player, string item, bool playerInitiated, time t);
    void NotifyUseItem ( unsigned player, string item, bool playerInitiated, time t);
    void NotifyNativeFunction ( unsigned player, string function, string data, time t);
    void NotifyDeltaFunction ( unsigned player, string project, string script, string function, string data, time t);
    void NotifyMiniGame ( unsigned player, string game, Difficulty difficulty, string category, time t);
    void NotifyMultimedia (unsigned player, string type, string media, time t);
    void NotifyDiceRoll ( unsigned player, unsigned dice, time t);
    void NotifyPossiblePaths (unsigned player, unsigned paths, time t);
    void NotifyDoorFail (unsigned player, string door, string reason, time t);
    void NotifyDoorPass (unsigned player, string door, time t);
    void NotifyTurnBegin (unsigned player, unsigned turn, time t);
    void NotifyTurnEnd (unsigned player, unsigned turn, time t);
    void NotifyScheduleEvents (string type, unsigned amount, time t);
    void NotifyTogglePause(bool paused, time t);
};
```

Figure 25 – The Game Monitoring Interface
The Game Monitoring Service is semantically a singleton service and each of its implementation should implement the interface illustrated in Figure 25. Additionally, it has a default implementation that simply logs each incoming event to a file to be used for statistical purposes.

5.4.3 The Inventory Service

The Inventory Service provides a visual representation of a player’s inventory. Again, such visualization can be performed on the board screen, close to the location the player, locally on a player’s PDA, or globally in a shared external screen. Whenever a game event causes the inventory of a player to change state, or whenever the player himself changes the state his inventory, all effects are mirrored in the Inventory Service for the visualized inventory to reflect the actual in-game inventory. The Inventory Service should be instantiated once for each player, therefore is not a singleton service. Any concrete Inventory Services should implement the interface shown in Figure 26, while the default one displays each inventory on the board screen close to the player position, presenting it as a list of item pages each of which contains a fixed amount of items. The default inventory service interface is illustrated in Figure 27.

```cpp
class InventoryInterface {
public:
    virtual void Next (void) = 0;
    virtual void Previous (void) = 0;
    virtual void PageUp (void) = 0;
    virtual void PageDown (void) = 0;
    virtual void Select (string item) = 0;
    virtual string GetSelected (void) const = 0;
    virtual void Add (string item, string category, string description, unsigned timesOfUsage, string film) = 0;
    virtual void Remove (string item) = 0;
    virtual void Show (void) = 0;
    virtual void Hide (void) = 0;
    virtual bool IsVisible (void) const = 0;
    virtual void UpdateUsages (string item, unsigned usages) = 0;
};
```

Figure 26 – The Inventory Service Interface
5.4.4 The Media Service

The Media Service provides an interface for remotely deploying any type of game related media content. For instance, upon reaching some specific game state, there may be the need for showing a cinematic scene to the players, showing them some game related images or playing a specific sound or some music. Such a service would typically get the media information from the Game Engine and perform the request using external screens and sound output (especially in a tabletop configuration located in an ambient intelligence laboratory). The Media Service is semantically a singleton service and each of its implementation should implement the minimalistic interface illustrated in Figure 28. The default implementation supports text and images that are displayed on the board screen for a short period of time as well as sound that is played through the computer’s speakers.

```cpp
class MediaInterface {
public:
    virtual void Play (string type, string id) = 0;
};
```

Figure 28 – The Media Service Interface

5.4.5 The Mini Game Service

During the course of the main game, it is interesting for a player to enter a totally different quick game that can however affect the progress of his main game. Such games, called mini games, are totally orthogonal and independent of the Board Game Engine, allowing for their dynamic deployment through the use of the Mini Game
Service. This service acts as a game server at which the Game Engine connects at specific time instances (i.e. when a triggered game event instructs a player to play a mini game) for playing a specific mini game and getting back some result that can possibly be used to trigger additional events in the flow of the main game. The result of a mini game is an arbitrary string able to encode information about the player’s performance in the mini game. Such encoded information can then be passed to a game script able to interpret it, thus affecting the game flow in a way specified by the designer of the game world.

The Mini Game Service is a singleton service whose concrete instances should implement the interface depicted in Figure 29. A default implementation for this service also exists, but it is a null implementation, discarding any mini game requests and reporting back that no mini game was actually played. A designer creating a game that depends on the existence of mini games is responsible for having knowledge of the mini games provided by the Mini Game Service as well as marking the service as mandatory for the Game Engine. This way, it is guaranteed that each request for a mini game will be dispatched to the correct service, therefore retrieving meaningful game data whose interpretation will finally trigger potentially necessary game events.

```cpp
class MiniGameInterface {
public:
    enum Difficulty { EASY, MEDIUM, HARD, ANY };

    virtual void Play (string game, Difficulty difficulty, string category) = 0;
    virtual bool IsFinished (void) const = 0;
    virtual string GetResult (void) const = 0;
};
```

Figure 29 – The Mini Game Service Interface
6. Adaptive Pluggable Multimodal Input

6.1 Virtual Input Devices

6.1.1 Software Meta Architecture

The Game Engine supports various input devices that can be dynamically plugged into the system. In order to achieve this, we define a set of events that the Game Engine is interested for as well as an abstract interface for input devices. Additionally, we have the Virtual Input Device, a component that acts as a logical input device wrapping all actual input devices and holding information about the mapping of input events to game events (i.e. event callbacks). In this sense, when the Virtual Input Device is polled for input, it respectively polls all its actual input devices collecting input events and performs the actions registered for these events.

The Game Engine registers actions to be performed based on input events to the Virtual Input Device unaware of the actual input devices used in the system. On the other hand each input device only has to propagate its event to the Virtual Input Device unaware of any action that has to be taken based on the event. This way, we can have any number of input devices plugged into the system and even have multiple devices for a single player or one device for all players.

The meta-architecture described above is depicted in Figure 30.

This architecture applies for all possible input device types. Specifically for the Board Game Engine we have three types of input devices:

- Keypad Devices
- Pointing Devices
- Switch Pad Devices

These input device types are described in the following sections.
6.1.2 Keypad Device

A keypad device is an input interface that maps the event of a keystroke or pressing of button directly to a game event. The approach followed to implement the keypad device handling follows the exact architecture of the Virtual Input Device. In this sense, we have the Virtual Keypad Device wrapping a variety of concrete keypad devices that implement an abstract keypad device interface as well as the various keypad events fired by the devices that result into performing the corresponding actions registered by the Game Engine. The varying element is to select the set of events that best describes the input requirements of the Game Engine.

A player can give direct input to the Game Engine in three cases. Firstly, during path selection the player can choose to highlight the previous or the next path, or select the currently highlighted path. Secondly, the player can manage his inventory at any time, giving him the option to show his inventory, select the next or the previous item in the inventory, use the selected item, drop the selected item to the terrain or hide the inventory. Finally, sometimes the user is prompted for a boolean selection. This happens to ask the player whether or not he wants to pass a door or get an item or to let him select which precondition to satisfy if he is given the choice. Taking into account the cases of direct player input as well as the time instances at which they can occur we conclude the set of keypad events as follows:

**Figure 30** – Virtual Input Device Meta-Architecture
The used command provides an extensible game specific command to be bound to a keypad event. Such a user command is identified as a custom user defined keypad command type and is accompanied by an encoded string given the user command value. Additionally, the Virtual Keypad Device provides an interface for registering external handlers for such events (both native and scripted handlers are supported). Consequently, such input given by the user is forwarded to an external Game Engine independent component able to interpret the command of the user.

6.1.3 Pointing Device

A pointing device is an input interface that allows the player to input spatial (i.e. continuous and multi-dimensional) data as well as physical gestures like pointing, clicking and dragging, to the Game Engine. Supporting pointing input is essential for the interaction with the input dialogues. Additionally, the Game Engine, being designed specifically for Board Games, should provide an inherent support for positioning a (traditionally) physical pawn at any place on the board.
In order to determine the Pointing Event types we take into account the variety of devices that can be used for pointing input along with their input model. For instance, the most typical pointing device for computers is the mouse that supports pretty much any event we would be interested in. On the other hand a tabletop touch screen can also be used as a pointing device but able only to provide a limited range of events (depending on the specific touch screen this can be as limiting as a simple click event corresponding to the touched position of the screen). Taking all these into account, we come up with flexible but expressive set of pointing events that supports both a cursor pointing for dialogue interaction and a pawn positioning for the physical pieces. The pointing input events are:

- Pawn Move At
- Cursor Move at
- Cursor Move Relative
- Click
- Release

In order to implement a generic pointing input device framework, we follow the Virtual Input Device architecture again, adjusting it slightly to incorporate the input dialogues. Just like before, game actions are registered to the Virtual Pointing Device to be performed any time a pointing event occurs to the. The registration of such actions is actually performed to the Input Manager who serves as a wrapper for the virtual input devices. This way the Input Manager is able to intercept each pointing event and determine whether or not it was addressed to an input dialogue. If the event was addressed to an input dialogue the corresponding event of the dialogue is triggered, thus performing any action the Game Engine had previously registered for the commands of the input dialogue. Otherwise, the event is propagated normally directly to the Game Engine for additional handling (for instance a click event could occur when a player marks a node as an interaction point, or a pawn move event is triggered when a player moves his physical piece). As for the other end of the Virtual Pointing Device, various pointing devices can be plugged into the system ranging from the typical mouse, keyboard (i.e. using the arrows) or joystick (i.e. using a stick) devices to more sophisticated ones like a tabletop touch screen, a special laser pointer or even some vision system able to recognize gestures and translate them into
pointing input. Figure 31 illustrates the architecture of the Virtual Pointing Device component.

**Figure 31 – Virtual Pointing Device Architecture**

### 6.1.4 Switch Pad Device

A switch pad device is an input interface providing scanning interaction techniques. Such devices support motor- or cognitive-impaired user access, as well as access in other situations in which more complex input devices (i.e. keyboard, mouse, joystick, etc.) cannot be used.

In [16], graphical user interface elements are classified into categories depending on the dialogue policies they require. The user interface elements of the Game Engine able for scanning interaction are the input dialogues and according to the aforementioned categorization they fall into the container objects and the button objects. Each dialogue acts as a container object containing button objects each of which has an associated event to be triggered upon its activation. At any given time, at most one dialogue is the active scanning container providing sequential scanning of all its contained button objects. The currently selected button object is denoted by the use of a highlighter (see Figure 32).
The scanning interaction techniques are based on two fundamental actions: SELECT and NEXT. When navigating through the objects of a container, NEXT advances the currently selected object, while SELECT activates it causing the event associated with this object to be triggered. When reaching the end of the container, we enter a state transition automaton with two states, entry and exit. NEXT toggles the selection of the current state, while SELECT exits the automaton and returns to container iteration mode. When exiting from an entry state the active scanning container (i.e. dialogue) remains unchanged and starts iteration from the beginning. Respectively, when exiting from an exit state the next available container becomes the active one and iteration is started from its beginning. In order to simplify the interface, we can use two additional actions: OUT and PREVIOUS. PREVIOUS is the symmetric of NEXT and can be used to iterate the containers in the reverse order. OUT is used to directly move to the next container without the need to enter the state transition automaton. In this case, when reaching the end of the container the NEXT action (as well as the PREVIOUS if available) wraps around and reiterates the container from the beginning.

The SELECT and NEXT actions are mandatory, meaning that a switch pad device should always provide their functionality. The OUT and PREVIOUS actions are optional in the sense that a switch pad device can be fully functional without them, but they help providing a more flexible and easier to use interface. The switch pad configurations supported are:

- 2-switches – SELECT and NEXT actions
- 3-switches – SELECT, NEXT and OUT actions
- 3-switches – SELECT, NEXT and PREVIOUS actions
- 4-switches – SELECT, NEXT, OUT and PREVIOUS actions
Of source, the final interface provided to the user depends on the actual switch devices used and the configuration they support.

Once more we use the Virtual Input Device architecture to implement a generic switch pad device framework. The Virtual Switch Pad Device acts as a wrapper for various actual switch pad devices that are dynamically plugged into the system and can trigger SELECT, NEXT, OUT and PREVIOUS events. The additional component here is the Switch Dialogue Manager, an add-on component on top of the existing input dialogues of the Game Engine. The Switch Dialogue Manager is aware of the active dialogues for each player and is responsible for showing the currently active scanning container along with the currently active button per player as well as invoking the corresponding command upon button selection. This is performed by receiving all switch events and interpreting them according to their configuration for each player. For instance, if a player has two switch pad devices active, the one supporting only the mandatory actions and the other supporting all possible actions, the virtual switch pad device wraps the two devices as one device supporting all possible actions. Consequently, the interface for this player will use the 4-switch configuration even if the user is currently operating through the first more limited switch pad. To enable the 2-switch interface the player would have to first unplug the device supporting all possible actions.

The architecture for the support of scanning input through the Virtual Switch Pad Device and the Switch Dialogue Manager is depicted in Figure 33.

Figure 33 – Virtual Switch Pad Device Architecture
6.2 Remote Input Device

All input devices discussed so far are devices plugged locally to the computer hosting the Board Game. We would also like to support remote input devices in the sense of having devices not locally plugged but able to provide input through the network. In order to achieve this functionality, we first define a new type of input device, the Remote Input Device that wraps all functionality of the input device types of our system. Specifically, the Remote Input Device can be seen and used as a Keypad, Pointing or Switch Pad Device triggering any kind of Keypad, Pointing or Switch Pad events and sending them to the corresponding Virtual Input Device type for further dispatching. The Remote Input Device is responsible for decoding incoming network messages and triggering the appropriate event. These network messages are propagated to the proper Remote Input Device by the Remote Input Server. The Remote Input Server is the component that listens to a port for incoming connections from Remote Input Clients and maps them to Remote Input Device instances. It is also responsible for managing inactive connection (i.e. when a remote input client is no longer available) as well as receiving messages from the active ones propagating them to the Remote Input Device instance mapped for the specific connection. The protocol used for the message exchange between the Remote Input Device and the Remote Input Client is a custom lightweight TCP/IP protocol based on xml encoding. The Remote Input architecture is shown in Figure 34.

![Remote Input Device Architecture](image)

**Figure 34** – Remote Input Device Architecture
The absence of a Game Engine component highlights that the remote input management is orthogonal to the rest of the system and can be dynamically deployed on top of the existing framework.

The Remote Input Client on the other hand is running on a remote host, connecting to the Input Server at any time instance and feeding it with input data packed as messages to be interpreted by the Remote Input Device as input events. Typically, such a client will be running on a PDA or a smart-phone used by one of the players for giving input to the Game Engine. A sample implementation of a Remote Input Client for a PDA providing all available input controls is illustrated in Figure 35.

![Remote Input Client](image)

**Figure 35** – A sample Remote Input Client providing all available input controls

### 6.3 Plug-In Mechanism

#### 6.3.1 Input Device Interfaces

In order to support dynamically pluggable input devices, we first have to define a specific interface that each of these devices should provide. The most fundamental abstract interface for a generic input device consists of a single `Poll` function invoked to query the device for any pending input and through which the device actually fires Input Events to the appropriate Virtual Input Device. Additionally, there should be some functionality for registering the player using this specific input device as well as retrieving the registered player information. Both Keypad and Pointing Devices use exactly this simple interface, while Switch Pad Devices need an additional function
for checking which Switch Pad Events are supported from the specific device. This additional information is used by the Switch Dialogue Manager to determine the type of interface to be used for the scanning interaction.

### 6.3.2 Implementation Details

The implementation of the Plug-in mechanism is based on the *factory* software design pattern combined with the use of dynamically linked libraries. Each concrete input device is implemented as a dynamically linked library (i.e. dll) containing two additional specific functions. These are the *Constructor* and the *Destructor* functions that play the part of the factory, respectively creating and destroying object instances.

When instructed to load an input Plug-in, the Game Engine loads the library file and queries for these two special functions. Upon successful location of the functions, the plug-in is considered to be successfully loaded and further instantiation of the plug-in instances can be performed through the factory interface. However, since the number of active instances of a plug-in may vary (for instance a mouse plug-in can only have one active instance while a joystick plug-in can have multiple active instances) additional input is required specifying the number of active instances. This input is provided along with the library path through a configuration file containing the plug-ins to be loaded by the system. After successful creation of a specific input device, registration to the corresponding Virtual Input Device component should be automatically performed (it is the plug-ins responsibility to register itself to the appropriate component). From this point on, the appropriate Virtual Input Device component polls the newly registered input device at regular intervals using the corresponding abstract input device interface, thus receiving input events from the actual device. The only additional action for the Game Engine is to keep the input device instance in order to release its resources upon unloading of the plug-in using the special *Destructor* function.
6.4 Standard Plug-ins

The Virtual Input Device architecture has been used for all input device types in order to achieve an extensible and orthogonal input model supporting decoupled implementation of actual input device based on a simple interface as well as their dynamic plugging to the system. In order to have a fully functional Input Management mechanism though, some standard input device plug-ins are essential. For this purpose, we provide some already implemented input devices based on typical hardware as the keyboard, the mouse and the joystick. These are:

- The Keyboard Plug-in (Keypad & Pointing Device using the keyboard)
- The Joystick Plug-in (Keypad & Pointing Device using a joystick)
- The Mouse Plug-in (Pointing Device using the mouse)
- The Switch Pad Plug-in (Switch Pad Device using keyboard based switches)
- The Remote Input Management Plug-in

6.4.1 Keyboard Plug-in

The Keyboard Plug-in is an implementation of the Keypad Device and Pointing Device interfaces using the keyboard. Pointing Events are supported by assigning four keyboard keys for cursor movement (i.e. left, right, up and down) and two additional keys for cursor clicking and pawn clicking. Keypad Events are supported by directly mapping a keyboard key to the desired event. In order to minimize the amount of keyboard keys needed to support both Keypad and Pointing Events, only one input interface is active at any given time instance and one extra key is used to toggle the active input interface between Keypad and Pointing. This way less keyboard keys are needed for a single Keyboard Plug-in instance, allowing additional Keyboard Plug-in instances to be active using the same keyboard. It is important to note that each Keyboard Plug-in instance has different key bindings declared in a configuration file used upon instantiation.
6.4.2 Joystick Plug-in

The Joystick Plug-in is an implementation of the Keypad Device and Pointing Device interfaces using the joystick. Pointing Events are supported by assigning a joystick stick for cursor movement (i.e. left, right, up and down) and two joystick buttons for cursor clicking and pawn clicking. Keypad Events are supported by directly mapping a joystick button to the desired event. In order to minimize the amount of buttons needed to support both Keypad and Pointing Events (joysticks only have a limited number of buttons), only one input interface is active at any given time instance and one extra button is used to toggle the active input interface between Keypad and Pointing. A joystick cannot be shared between players, but many joysticks can be plugged into the system, each one corresponding to a specific player (a player may even have two or more active joysticks). Additionally each Joystick Plug-in instance can have different stick and button bindings configurable through a configuration file used upon instantiation.

6.4.3 Mouse Plug-in

The Mouse Plug-in is an implementation of the Pointing Device interface using the mouse. Movement related events are directly associated with the movement of the mouse, while cursor clicking and pawn clicking are assigned to the mouse buttons. Since the target platforms support only one mouse this is a singleton plug-in associated to a single player upon instantiation.

6.4.4 Switch Pad Plug-in

The Switch Pad Plug-in is an implementation of the Switch Pad Device interface using keyboard based switches. The pressing of each switch simulates the pressing of a corresponding keyboard key (this is the functionality provided by the hardware switches and drivers they come with). Each of these keyboard keys is then associated with one of the four Switch Pad Events: SELECT, NEXT, OUT and PREVIOUS. A switch pad cannot be shared between players, but many switch pads can be plugged
into the system, each one corresponding to a specific player (a player may even have
two or more active switch pads). Additionally, each Switch Plug-in instance has
different key bindings configurable through a configuration file used upon
instantiation. Adjusting the key bindings for any switch pad we can simulate any of
the 4 supported switch pad configurations described in the Virtual Switch Pad
Section. The only concern with this implementation is that specific keyboard keys
should be reserved for switch pad input (supposing switch pad devices are present)
and restricted from the Keyboard Plug-in. Nevertheless, the extensive configurability
of both the Switch Pad Plug-in and the Keyboard Plug-in ensure their non-conflicting
co-existence.

6.4.5 Remote Input Management Plug-in

The Remote Input Management Plug-in is actually not a concrete input device
implementation, but an input device placeholder enabling remote input device
instances to be dynamically plugged into the system. It is nevertheless a plug-in itself
giving the Game Engine the flexibility of enabling or disabling the Remote Input
Management at will based on some configuration or network state. For instance, in a
computer configuration without network support it would be useless and needlessly
resource-consuming to setup an entire Input Server when no connection would ever
be established. In this sense, it is important that the Remote Input Management is
implemented as a plug-in giving the Game Engine the privilege of loading it
dynamically at any time.
6.5 Adaptive input

An important requirement regarding user input devices is adaptivity. In this sense, we want to support automatic selection of the best-fit input devices for a specific player. In order to achieve such behavior, we define a set of rules containing the input adaptation logic that are based on player characteristics. These rules are then evaluated for each user, resulting in the activation of the appropriate input devices. For this purpose, we use the Decision-Making Specification Language – DMSL [17]. DMSL is a domain specific rule-based language for making decisions on activation / deactivation of components during the run-time of a system. It is based on a compiler that loads and compiles files containing DMSL adaptation rules, and an interpreter that evaluates the compiled rules based on the profile of each player. The output of the decision making process is a set of activation and deactivation commands that can be interpreted by the system to perform the appropriate adaptation actions. Specifically for the board game, we incorporate the use of the DMSL compiler and interpreter, providing a logic specification of input adaptation and taking as a result the activations and deactivations of specific input device modules for each user (see Figure 36). Figure 37 shows a sample program in DMSL, describing the decision logic for this purpose. The program takes the decision on which input modules should be activated, according to specific characteristics of the player’s profile.

![Diagram](image.png)

**Figure 36** – Adaptation process used for automatic input device selection
//-- Sample Decision Making Logic for Input Devices

def Joystick params.user.joystick
    component input [  
        if Impaired then  
            activate "KeyboardSwitchpad data/switchpad.xml"
        
        if CanUseMouse then
            activate "MousePointing"
        
        if CanUseJoystick then  
            activate "JoystickKeypad " + Joystick + " data/joystick.xml"
        
        if PrefersKeyboard then  
            activate "KeyboardKeypad data/keyboard2.xml"
    ]

Figure 37 – Input device adaptation logic in DMSL
7. Deployment – Making a Board Game

In order to evaluate the Board Game Engine as well as the tools accompanying it, it is essential to deploy it in the construction of a concrete board game. Such a game should illustrate the potentials of the Game Engine, while still being a simple entertaining board game to play and not just a test case demonstration of all the Game Engine features.

7.1 Board Game Design

The game selected for creation is inspired from the four nature elements and called “The Four Elements”. The storyline is based around the Element Temple, a place where once all four elements coexisted peacefully until they were separated by mysterious forces and scattered across the land. The task of the game is to collect the four elements from the four corners of the terrain and bring them back to the Element Temple located in the center of the terrain in order to restore balance in the world.

There are multiple paths to reach each element area and various items and interaction points located in between. In particular, there are doors that need some key to be opened, others that need some switch to be pressed, as well as some collectable items that can be used performing specific events:

- A trap that upon usage is activated and causes the next player that passes from it to lose his turn
- A teleportation item allowing a player to instantly go back to the temple
- A coin that can be dropped in a wishing well for something to happen
- Two keys, each opening one door
- One reset button item, used for returning each collectable item to its original place.
- The four elements that need to be gathered for the game completion that additionally work as portals to instantly go back to the element’s area.

Additionally, there are the following interaction points:
- Two nodes containing switches, each opening one door (the switch is toggled each time it is pressed)
- A wishing well where a player can throw the coin for something to happen
- The four elements’ areas that provide the player with the corresponding element upon passage.

The strategy followed for all usable items is that they are initially present at some point on the terrain, can only be used once, and that upon usage they are restored to that initial point. The only exception is the case of the element items that can be used an infinite amount of times, never being consumed. Additionally, each player has an inventory able to hold 6 items.

The game plays out with players trying to gather the 4 elements while obtaining any useful item they encounter on the way, trying to make his path easier and the paths of the opponents harder. Competition between players is enforced by the nature of the game but player cooperation can also occur. For instance, a player may drop an item for some other player to take it, or some players may team up against an opponent closer to finishing the game focusing only on how to make his path more difficult.

### 7.2 Board Game Implementation

Beginning with the game construction, we start creating the pattern elements, and pattern element galleries to be used for the terrain construction later. It is evident that graphic design benefits greatly from the use of the Terrain Element Editor as well as the Tile Set Editor, minimizing user effort needed for pattern element management. Continuing with the terrain construction, we use the Terrain Editor for creating our game world. The Terrain Editor proves to be very efficient and user friendly, with all its features greatly improving the construction process. Most of the work is performed interactively using the well-constructed user interface, allowing for a quick creation of the game world setting. The resulting game world is illustrated in Figure 38.
In order to complete the game construction some coding is needed in order to provide the scripting logic for the various game events. The developer is assisted in this part of the process as well, using the Sparrow IDE through the Terrain Editor interface and having all project management performed automatically. Additionally, the actual code to be written is reduced even more taking into account that the Game Engine provides built-in support for the most commonly used event (again through interactive interface) as well as the fact that the event system used in Game Engine is quite general and flexible, allowing for event scheduling at virtual any time during the game. Specifically, for the game “The Four elements” described here, the implementation of the whole game logic scripting does not exceed 200 LOCS. Additionally, the time spent on creating the game world setting through the Terrain Editor accounts for more or less 4 hours total. These two numbers alone highlight the increased efficiency achieved by the use of the Board Game Engine and its tools in the creation of a concrete game. As for playing the game, it presents an enjoyable experience that preserves the player-centered interaction, while clearly benefiting from the computing technology it uses.
8. Summary, Conclusions and Future Work

In this Thesis we have presented a framework for creating augmented computer board games. Initially we discussed about the architecture of the Board Game Engine, presenting the elements that it consists of. Then we presented the Terrain Editor, an interactive editing tool used for creating game worlds for our framework. Additionally, we showed the inherent ability of our Game Engine to support external services thus providing an extensible user interface. Moreover, we presented the input management techniques used in order to support pluggable multimodal input. Finally, we deployed the tools developed, creating a concrete augmented board game instance on top of the Board Game Engine.

From the work carried out, the main conclusion is that we should always try to detect common behavior among things and try to abstract it, providing the most generic view possible. In this sense, developing some board game is fine, but the real benefit is in the development of the Board Game Engine that abstract common board game element and is capable of instantiating any number of concrete board games. Only this way can one comprehend the current state of their research field and then take it one step further.

The Board Game Engine along with its terrain editing tools provides a fully functional framework for the construction of augmented computer board games. Nevertheless, there are several features that could be improved or added to the system, providing either efficiency or additional functionality. One interesting feature would be the support of dynamically repositioning terrain pattern elements. In this sense the graph of the game would be able to be restructured at runtime based on the game flow, providing true dynamic board composition. Another possible feature would be the support of a network game-play mode. Such a mode would target interconnecting various ambient environments, allowing the participation of all their players in a joined game session. This would also be an appealing addition, providing a setup that would possibly imply additional external services like remote player communication. Another attractive feature to be added into the system is the formal analysis of the reachability of the possible game states. In this sense, the terrain editor would be able
to automatically identify all possible game states and game state transitions and determine if some of them are unreachable. Of course, in order to provide this kind of automation to such a dynamic system that also supports use of scripting to specify the game logic, the designer would have to some formal specification model for any scripting behavior, thus allowing the editor to conclude possible script event outcomes and then identifying the possible game states by applying all possible combinations based on the starting game configuration. Finally, an important addition would be a generic Save and Load system able to store the current state of the game and restore it at any time later. Research-wise, this is a very interesting task as it involves not only storing and loading the local game data, but also the execution state of each active virtual machine of the scripting engine, as well as potentially the state of the external services. As an added bonus, the addition of such a Save and Load system would be a very useful feature for the game engine as well.
9. References


