

Journey Infinita

**An editor with an embedded
programming language**

**By
Kristian Spencer
Michael Bo Magling**

23rd of May - 2012

**Supervisor
Joseph Roland Kiniry**

Abstract

Abstract

When developing a game that allows the user to create their own content, it is necessary to create a competent editor to work with the game. The project is based on the classic JRPG (Japanese Role Playing Game) genre, and investigates how one would define a platform for such a system, along with the creation of a domain specific language needed for the editor. Based on the analysis of two existing editors, an editor with an embedded programming language was created, and used in conjunction with a JRPG system.

A basic RPG platform was designed systematically, using BON and implemented in C#, and an editor was created based on WPF windows, assisted by XNA framework capabilities. A programming language was created for the editor, utilizing functional programming with F#. The main characteristics of a domain specific language were the lexing, parsing, static checker and compiler to create executable bytecode that a virtual machine can execute at runtime. It was found that implementing a compiler and virtual machine for a game developed on a high-level platform is inefficient, and should be avoided, while a well typed, statically checked programming language is very helpful for users to avoid flaws in their code. To get user generated content into the game, a compilation process was created, using XML, ZIP, and a data architecture split into two parts. Using a Resource Manager was essential to allow serialization to work, and to simplify the control of resources.

Preface

This project started out with the goal of creating the baseline work for a large, commercial product focusing on what is essentially a JRPG with game content made exclusively by users. Waiving this goal into our Bachelor Project was a way for us to work on something interesting, and which can actually be further developed to make an actual product.

In the semester before the bachelor project started, a number of songs and graphical assets were developed that we think are required in a JRPG, which are all included in the example game. Along with these, we began designing what to include in the system, and how the final game should look. However, a large unknown factor was how to make the editor for the users to create content with, along with how to define a programming language to be used with such a system. Therefore, it makes sense to investigate the possibilities for our bachelor project.

The project is a collaboration between Kristian Spencer, who wrote the editor part of program, and Michael Bo Magling, who wrote the programming language part of the program. Report, data structure and other parts have been developed co-operatively.

We would like to thank Joseph Kiniry for supervising the project, as well as Rune Møller Jensen and Mette Holm Smith for coordinating the bachelor project course.

On the CD

- The report, in color
- The source code for the project solution
- A compiled version of the editor and game client
- An example journey project file
- BON specifications
- Data Architecture

Table of Contents

Introduction	5
Use of Tools	6
Related Work	9
JRPG systems	9
Game Maker	10
RPG Maker	12
Derived Features	13
Game Maker Language	14
Academic Work	16
Overall Architecture	17
Code Metrics	18
The Editor	19
Data Architecture	19
Editor Architecture	25
The Programming Language	30
Overall Design	30
Lexing and Parsing	35
Code Checking	39
Compiling	40
Virtual Machine	43
Testing	45
Conclusion	46
Appendix	
1 - Glossary	48
2 - Bibliography	51
3 - Editor Images	52
4 - Task Descriptions	57
5 - Virtual Windows	83
6 - JourneyScript	91
7 - JourneyScript, Backus-Naur Form	102
8 - E/R Model Explanation	106
9 - User Manual	107

Introduction

The ability to have user created content for games has become an important part of gaming for the last ten years. Some of the most popular games ever made, like Counter Strike or DOTA, has started out as simple modifications of existing games, using tools provided with the game. This gives the games an ability to evolve over time, fueled by community support and with little to no resources required from the developers.

One genre that has not taken advantage of this is the classic JRPG (Japanese Role Playing Game) genre. While tools exist for making such games from scratch (popular examples includes Game Maker and RPG Maker), there is no overall system that allows for crowdsourcing of game content within a certain framework. Such content would include game artwork, monster and character designs, and actual gameplay with story.

For this project, we seek to create a 2D JRPG system that allows users to create their own Journeys, while still allowing for players to create their own character to use for such a journey.

Problem Statement

For the project, we wish to answer the following questions:

- What would be the main characteristics of a domain specific language for a classic JRPG editor?
- What are the most important factors when developing a JRPG editor?
- What is necessary in order to transfer user created content into a game?

Project Focus

For the scope of this project, we focus on the editor, programming language, and using the editor to get data inside of a game. Creating a large online gaming platform that supports user created content being played through caching from an online server, as well as in-game playability, is outside the scope of this project.

Use of Tools

For the project, several tools have been used.

F#

F#^[1] is a .NET programming language that has functional programming properties. Functional languages are very useful for describing programming languages due to some of the high-level features included. Also, F#, through the F# Power Pack^[2], contains lexing (FsLex) and parsing (FsYacc - F#'s Yet Another Compiler-Compiler, a LALR parser) that are used for defining the programming language.

To use the lexer and parser, a .fsl and .fsy files are required. To generate the automatically generated .fs lexer and parser files, the following commands are used in the Visual Studio command prompt, with the power pack added to the path:

```
fslex --unicode JSLex.fsl  
fsyacc --module JourneyScript.JSPar JSPar.fsy
```

Which will create the lexer and parser used in the programming language.

Visual Studio

For the project, we choose to use Visual Studio^[3] for the general support for multiple projects, along with general support for project development. The tool is especially useful for its support of C# and F# - the two programming languages used in the project. The solution on the CD should be easily opened in a Visual Studio environment.

NUnit

For testing, NUnit^[4] is used. NUnit is a simple testing module for C#. While Visual Studio can use NUnit internally for C#, there is no support for using it along with F#. Therefore, an external testing tool is used in order to run the tests, the testing tool being the NUnit executable included. Also, for ease of integrating test cases in F#, FsUnit is used. FsUnit^[5] is nothing but a wrapper for the NUnit features, that instead use the F# features for constructing test cases.

XML

XML^[6] is an important data format used in the project. XML is used due to the ease of import and export through serialization, where only a few attributes are required to save and later restore data, to and from a single file.

XNA

The Microsoft XNA framework^[7] is a game framework for Windows, using C#. It is used in the project for displaying images in the editor, and for the basic underlying engine for the game. There are alternatives, but XNA is a well-known framework that was desired to attempted used for this specific project.

Systematic interface design

Systematic interface design^[8] is a technique by Søren Lauesen used for creating user interfaces with a high level of usability. This involves user tasks, a database, and drawn windows, also known as virtual windows. The technique is not used fully, with areas like Usability tests and understandability being skipped. We focus on the creation of workable windows and less on usability tests, as testing the program with users would be a lengthy process due to the size of the editor. Also, Function design is separated from the window design, so that actual presentation functionality is designed more in-depth and separately from the actual window. However, as the editor is used almost exclusively for setting the data, the function design is included in the same phase as the virtual windows in order to save time.

WPF

The Windows Presentation Foundation^[9] is a user interface system for rendering on Windows. It is used in this project due the advantages over Windows Forms, as it is easier to create better looking user interfaces.

BON

BON^[10], Business Object Notation, is a tool for describing the framework of a program. We use the notation to describe the data of the program. However, due to XNA, WPF, F# and XML, there is a significant problem with using BON throughout the program. The issue is caused by writing Code Contracts for F# and some of the frontend coding of WPF which is not clearly visible. Another issue with the BON notation is that it overlaps with the systematic interface design. This leads to very limited use of the notation.

FMOD

FMOD^[11] is an audio library that can play music files, and is used in many video games to provide sounds. We make use of FMOD Ex, the low level engine of the sound platform, along with a standardized C# Wrapper system that controls the external calls to the engine.

Code Contracts

Code Contracts^[12] are a set of methods that can be used to replace simple exceptions calls with guarantees on input on output to prevent exceptions. These fit very well with BON, and the formal part of BON is essentially classes with lists of methods with code contracts. Code Contracts can be checked by analyzing the code for paths that could cause a contract failure, but we do not use this, as previous experience has shown that this approach is time inefficient due to the amount of useless errors that are raised. Code Contracts are a very useful tool, however, to make sure that things go wrong when they should go wrong, and not at some arbitrary point later on at run-time.

Avalon Edit

Avalon Edit^[13] is an open-source WPF based text editor. Avalon Edit is used in this project in order to make use of its internal syntax highlighting features, which are defined in the .xshd files in the Assets part of the editor.

Custom Color Picker

Custom Color Picker^[14] is an open-source GUI element for selecting a color, which in the process is used for selecting a color for zones.

Code Metrics

Code Metrics Viewer is a tool for displaying code metrics, and uses the Code Metrics Power Tool^[15] to generate the code metrics. We use code metrics as a way of displaying metadata and overall quality of our code.

In the Code Metrics section, C# and F# is treated differently. Due to the tools inability to properly use F#, the two F# projects are analyzed manually. Due to the language differences, the aspects are also different between F# and C#. A "member" in C# is a method, while in F# it can either be a type, a member or a function. A "type" in C# is a class, while in F# it is a module. Lines of Code in C# relies on Code Metrics Viewer (which seem to count statements except Code Contracts), and in F# are manually counted Non-Commenting Source Statements (no empty lines, comments or headers). Analyzing maintainability, cyclomatic complexity and class coupling is not possible manually, and are not done for the F# projects. It should be noted, that the metrics includes code not written for the project, more specifically some WPF methods only visible after compilation and most of the FMOD namespace. These have a significant impact on the metrics. FMOD counts for almost half of the cyclomatic complexity in the JourneyInfinitaData project. Lastly, due to the heavy use of manual counting due to the lack of tool support, it is possible that some numbers are slightly incorrect, especially for types and members.

BNF

For describing the syntax of JourneyScript, we use BNF^[16], or Backus-Naur Form, to describe the terminals and non-terminals of the language. It does not display the transition to an abstract syntax, however, but gives a simple view of the layout of the major parts of the lexer and parser in regards to basic design. We use traditional BNF over EBNF, as the FsYacc parser is written similar to BNF, in which case it makes sense to use that.

E/R Model

An entity relation model^[17] is a way of abstractly view data for software. Entity relations involves entities with attributes (which can be considered a class) and relations (which can be considered object references). We use the entity relation model to describe the data used in the game, a description of the specific use of it can be found in appendix 8.

ZIP

ZIP^[18] is a file format that compresses data for archiving purposes. This is used in this project in order to compress a number of files into a single file as an archive.

Related Work

JRPG systems

When designing a JRPG, it is important to note what are special about that genre, and what the different games in the genre have in common. For this, analysing existing, noteworthy games in the genre is important. Three noteworthy games that are all classic examples of the genre are the early Final Fantasy^[19], Dragon Quest^[20], and Breath of Fire^[21] games, all originating from the NES/SNES era. It should be noted that there are many more games that are similar in nature as well, like Grandia, Pokemon, Chrono Trigger and the Mana series.

The games are overall very similar. In all three games, an overworld map is used, where monsters attack the player appears randomly. What monsters appear depends on the location on the overworld map. Also on the overworld map are towns and dungeons that can be entered, opening a new area where monsters may also appear randomly, depending on the area.

The three series uses a leveling system, where experience is gained by slaying monsters, and leveling up increases the player's stats, and gives access to new abilities, both of which are necessary, as monsters grows stronger throughout the games. The specific type of stats differs slightly between the games, but usually includes health, mana, attack, defence, speed, magic, and a luck stat. The attack and defence stats are normally being controlled by equipment, though equipment may alter the other stats as well. Characters has multiple equipment slots, like weapon, armor, shield, helmet and accessories, which are special items that have effects that goes beyond attack and defence.

Besides the simple slay monster, get experience mechanic, an economic system is also included. Monsters drop gold besides giving experience, which can be spent in a shop to gain better equipment or items that can be used in combat, or in an inn to get full health.

The story is a big focus point in these games, and besides the main storyline, all the games includes a significant amount of text that can be read by talking to non player characters, with the content of the text differing based on the current state of the game. For instance, a character may respond depressed in a city captured by an evil empire, but later respond jubilant when the main character has saved it. This state system is also used to prevent the player from continuing until a certain plot-specific scenario has unfolded, like a bridge being destroyed and only repaired later in the game, also known as the Broken Bridge^[22] trope.

There is a noticeable difference in the battle system. Not so much in the technical aspects, as all of them uses a turn-based system using an interface to select an action, like fighting, using an item, using a spell and running. The differences are in the visual style. Breath of Fire uses a diagonal view of the battlefield, Dragon Quest uses a frontal view of the enemy with the player's party invisible, and Final Fantasy has a side view of the battle, with more movement in the attack animations then the others.



(Battle systems for *Breath of Fire*, *Dragon Quest* and *Final Fantasy*)

Another aspect that these games share, is the fact that they have a longer play time than most games. They achieve this through the use of the very simple design. It is easy to include a lot of monsters, as they, like in the earlier games, are only really a set of numbers with a name and a sprite attached to it. Multiple monsters may share the same sprite, but with different colors. The overworld map can also be made very large, as only a very small set of images are necessary in order to construct the entire map. This can be seen in the case of the earlier *Dragon Quest* games, illustrated above, where "forest", "plains" and "mountains" are nothing more than a simple 16x16 tile. Considering that the story is usually also linear, and most of the story is told through text boxes, significant content can be created for relatively little development time.

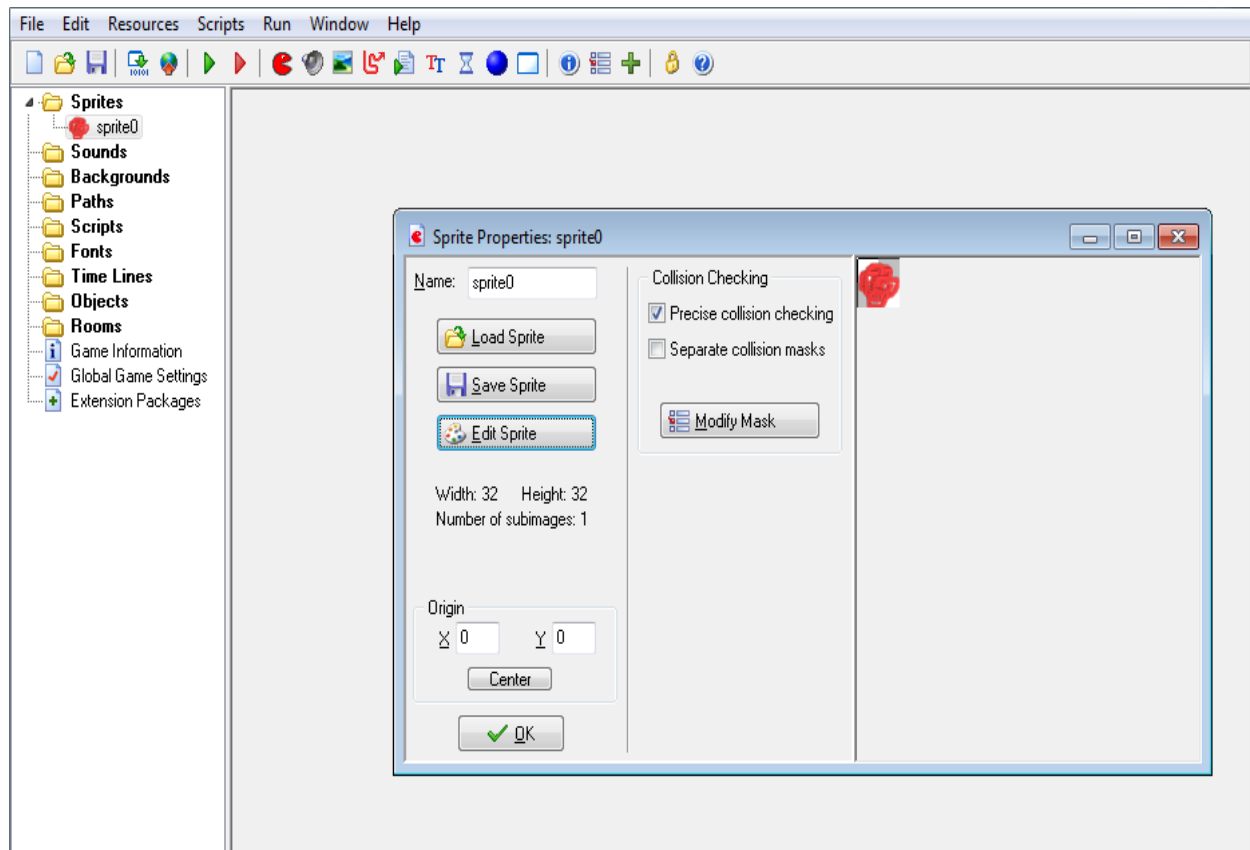
Game Maker

Description

Game Maker^[23] is a software created by Mark Overmars, which was released in 1999. It is developed in Delphi and the purpose of the program is to create simple games. Game Maker has support for multiple platforms. Currently Windows, Mac and HTML5 are supported, with mobile devices being developed as well.

The UI of Game Maker has two modes, a simple and an advanced mode. The simple mode excludes some of the more advanced features like time lines and paths, and is there to help new users ease into the program. Even in the advanced mode, the interface is very simple and easy to approach. Game Maker also includes an image editor for both backgrounds and sprites, and has a wide array of features despite not supporting layers.

One of the most impressive features of Game Maker is the scripting language also known as GML^[24]. The language structure is similar to C, but the language itself has a very loosely typed syntax. Game Maker also provides support for .dll which has resulted in many addons for Game Maker allowing better 3D and Network support^[25]. An Example of GML, along with additional images of the editor, can be found in appendix 3. Game Maker interprets the resources and scripts at run time, which makes it somewhat inefficient unless a .dll replaces any heavy code.



(The main window of Game Maker, with a sprite properties window)

Game Maker is designed to work with a multitude of genres, and does not focus exclusively on creating the classic JRPGs. Therefore, the program does not include pre-implemented features necessary to have in a JRPG, which complicates the process for the user. Also, as there is no focus on one specific genre, there are also very few premade resources that can be used in a JRPG setting.

Pros

- Can develop games for pretty much any genre
- A wide array of standard functions to use in games
- A great tutorial section and help section for new users
- Supports 3D and network
- Very simple UI
- Has Drag & Drop for people who haven't learned GML

Cons

- Interpret on runtime which makes it slow
- Is not built specifically for JRPGs so it requires a lot of backend code
- Very limited premade resources like images and sounds.

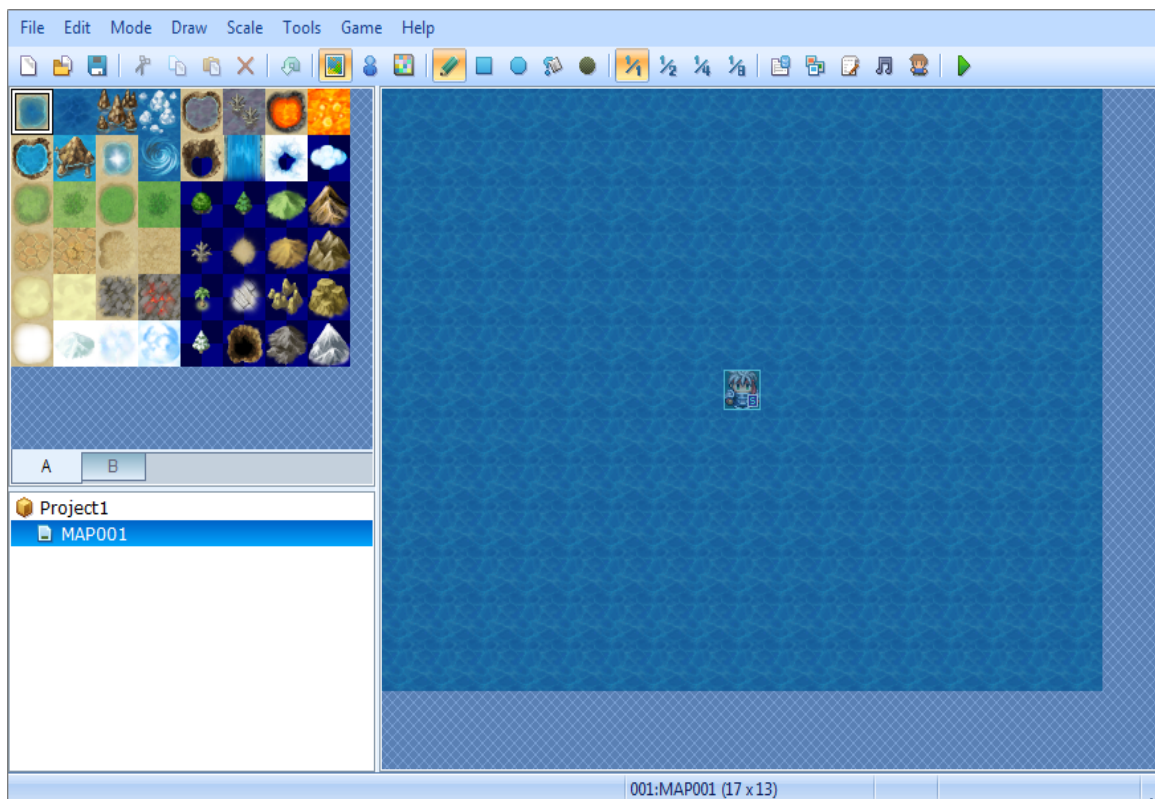
RPG Maker

Description

RPG Maker^[26] is a software developed by a Japanese group known as ASCII. The first version of RPG Maker was released in 1992, and since then over 40 different versions^[26] has been released. ASCII has also released versions of RPG Maker for the PlayStation console. The first Windows version was released in 1997.

As shown in the screenshot, RPG Maker comes with premade resources and is designed around RPGs. To play a game made in RPG Maker, a specific client is required, unlike Game Maker.

RPG Maker also has a scripting language, based on the programming language Ruby^[27]. Unlike Game Maker, Drag & Drop is replaced by inserting code sections at button clicks. This makes creating code significantly more complicated. For instance, to change the player position when he steps on a field, one would have to go to Event mode, create a new event at the field, move to tab 2, choose a movement based event, and finally type in the new destination in the variables section. The game then creates the script, which makes it possible to combine multiple event types together. This process is a lot less direct than Game Maker's Drag & Drop, though, and significantly less user friendly.



(The main window of RPG Maker, which shows the image resources and selected map)

An example of a RPG Maker script, along with additional images, can be found in appendix 3.

In RPG Maker, it is possible to customize the game significantly. You are not locked into one specific type of game. However, the combat system is locked in and little customization is possible without resorting to significant re-coding. This can be especially hard when even simple functionality can take time to implement due to the way the coding is set up.

Pros

- Is made specifically for RPGs and by that supports a lot of features with no backend programming
- Variety which makes it possible to make almost any kind of RPGs
- Has a big resource package with images and sounds.
- Includes a character generator which generates sprites and profile images for customizable characters.

Cons

- The combat system is locked and leaves no room for customization
- It can be very complicated to achieve very simple functionality

Derived Features

When creating our own editor, we have taken inspiration from the editors mentioned above.

We find it very useful to have an Area Editor where you create the different areas, similar to the way both Game Maker makes a room, and how RPG Maker makes the areas. We wanted it so there was always a working area, while in Game Maker a new window is opened up. Game Maker uses a very detailed system for the area, where objects, tiles and backgrounds can be manipulated separately, which we find useful.

The folder system that contains all the project resources, is done very well using Game Maker, giving a very simple overview of all resources in the form of a tree structure. This makes sense to include in our editor, and can be expanded to include other RPG aspects that Game Maker does not support.

The Game Maker scripting language seems much simpler to work with, especially for those new to coding. Especially the use of events is useful for games, which is a feature we also want to focus on.

The editors for manipulating JRPG resources in Game Maker are a source of inspiration, despite not using the same system. RPG Maker did provide us with inspiration on how to set up the windows in the Virtual Windows process. Also, a large amount of premade resources, like in RPG Maker, are very useful to have for a JRPG. We wish to supply the user with premade resources as well.

Game Maker Language

As mentioned, the programming language in Game Maker is a very central feature in the program, and through the wide collection of functions is a very diverse domain specific language. Because of the importance of the programming language in our system, it makes sense to analyse the programming language a little more in depth. The language used in RPG Maker is not analysed in depth, however, as it builds heavily on the Ruby language, and has not been designed specifically for the language in the same way the Game Maker language has been.

The Game Maker Language is constructed as an imperative language, very similar to the style of C. This is interesting, considering that Game Maker was implemented using Delphi^[28], whose style is considerably different. Another interesting design decision is to make semicolons optional at the end of statements, as a newline can be used instead of semicolon to terminate a statement. These decisions are likely done in order to make the programming language more clear to people with little to no programming experience.

What also helps those new to programming is that, while the language is object oriented, it is also a scripting language, reducing its complexity significantly. Unlike other object oriented languages like Java or C Sharp, classes are not defined in the language itself. What is instead the case is, that a "class" is an element defined and manipulated within the editor, with aspects like parenting, sprite, collision masking and area depth being defined outside of the language. This also means, that as the programming language contains no integral class structure, it becomes nothing but a sequence of code to be executed. These sequences are then included inside of a drag-and-drop system, where the programming language are simply a piece of code that can be used in combination with simple command blocks defined with drag-and-drop elements that takes variables. This makes the Game Maker language easy to get into for non-programmers, as it is possible to code an entire game without writing a single line of code.

What really controls the language is not the code, but the use of events. Classes can choose to implement a section of events, with code that are called at runtime, usually by the underlying engine. Events includes keyboard and mouse click events, event called at object creation, collision events, step events which are called once each game loop, events that are triggered by leaving the area and view, along with quite a few more. Using events, Game Maker can couple user created code together with the engine it is build on.

Not everything in the language is without consequences, though. In Game Maker, variables are declared inside the code itself, or in drag-and-drop elements. This is a problem, as variables declared in one event can be used in another. This may potentially be called first, where the variable has not yet been declared, or variables are called that are never created. In essence, this means that a typing system is not possible, as variables are resolved dynamically at run-time. Another thing about the dynamic typing system is that all variables (despite arrays and strings) are values.

The lack of variable typing means, that it is also not possible to efficiently declare booleans. While a boolean in languages like C are also not optimized, taking up the same space as a byte, in Game Maker it takes up the same amount of space as a double would, both being real values. The lack of optimization at such a level does mean that the language itself becomes a lot more inefficient, especially considering that the code is interpreted at run-time.

At compile time, references to editor elements like images or sounds are all resolved to their identifier so they, too, are values. This leads to some very bizarre implications where you can add a sound to an image, or use a reference to an image where a sound is desired, and everything compiles fine. This very loose look at typing has the consequence, that bugs in the system can be hard to find. This is especially the case as there is an option where new variables can be declared at runtime if the name it looks for doesn't exist, meaning that it won't throw an exception, it will only result in odd behaviour. Using an image where a sound is desired will result in the sound being played will be the sound that shared the ID of the image. This can cause weird behaviour that can be really hard to fix, especially considering that the code is very fragmented, being split among multiple code sequences, events, and objects. It is not even possible to really isolate it to one object, as objects can manipulate each other, including change and add variables to each other, breaking the object oriented design.

Lastly, an issue that Game Maker has received critique on, is decompilation. As interpretation is done at runtime, the main code is still inside of the executable. This means that stealing the source code from Game Maker files is possible to do, making the program as a whole less attractive. All in all, this shows that while the Game Maker Language has its advantages, it still has some very significant flaws.

Academic Work

Tim Sweeney

In a presentation^[29] about the direction of future programming languages, Tim Sweeney talks about multiple areas where code is used in games, which includes gameplay simulation, numeric computation and shaders. He also discusses how programming languages can help games in the area of performance, code modularity, reliability in terms of preventing run-time errors, and concurrency, in order to improve parallel performance.

Tim Sweeney's presentation is useful for our project, as it gives focus to what areas that are causing problems for game programmers, including which areas that we might choose to focus on for our programming language. While Tim Sweeney is mostly discussing it from the angle of handling 3D, it is still useful to take into account when dealing with 2D.

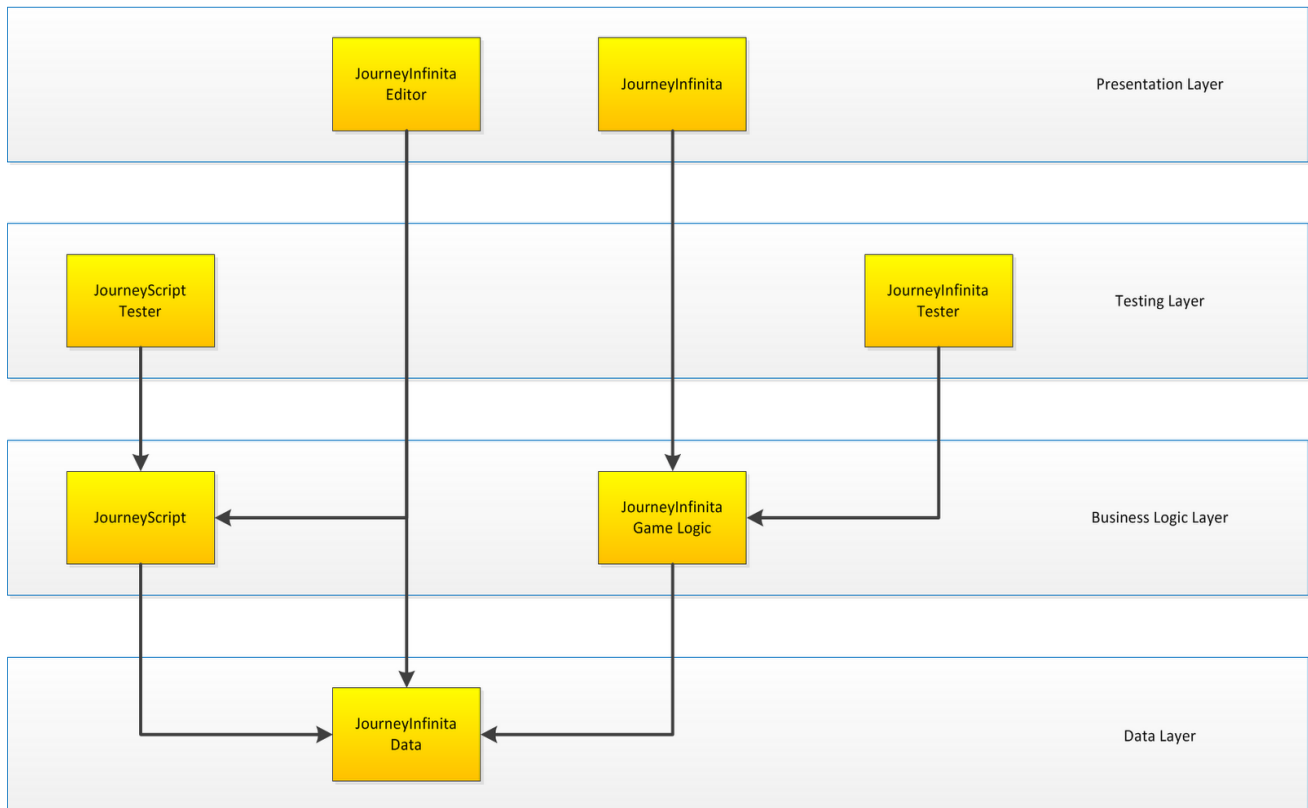
A Domain-Specific Language for Computer Games

In a Master Thesis by Jeroen Dobbe from 2007^[30], the use of DSL in computer games are described, with the focus of finding the bottlenecks and paradigms when applying DSLs to games. In the thesis, Dobbe concludes, that current domain specific languages are not used specifically for game design, as these are implemented in general purpose languages. He describes that DSL for a game needs to take into account objects (that make up the game world and has properties), the interaction with the player (through user control), core mechanic rules (that describe the specific game domain), and the game storyline (should it be present in a game).

Jeroen Dobbess analysis of the important areas that Domain Specific Languages has to deal with are important when we create our own Domain Specific Language.

Overall Architecture

The Journey Infinita solution consists of a number of individual projects, and has a set of dependencies, which are showed in the following layer diagram.



JourneyInfinitaData - The data for the game and editor, including saving to XML and FMOD.

JourneyInfinitaGameLogic - The logic for the game at run-time, including the virtual machine.

JourneyInfinita - The executable for the game, player of compiled game files.

JourneyInfinitaTester - The tester for virtual machine and other game logic functionality.

JourneyInfinitaEditor - The executable for the editor used for managing data.

JourneyScript - The logic for compiling strings to an array of byte code.

JourneyScriptTester - The tester for the compilation pipeline.

Code Metrics

<p>C# Journey Infinita</p> <ul style="list-style-type: none"> • Namespaces: 1 • Types: 5 • Members: 27 • Lines of Code (NCSS): 133 • Types per Namespace: 5,00 • Members per Type: 5,40 • Lines of Code per Member: 4,93 • Maintainability index: 78 • Cyclomatic Complexity: 81 • Class Coupling: 58 	<p>C# Journey Infinita Tester</p> <ul style="list-style-type: none"> • Namespaces: 1 • Types: 4 • Members: 365 • Lines of Code (NCSS): 3511 • Types per Namespace: 4 • Members per Type: 91,25 • Lines of Code per Member: 9,61 • Maintainability index: 60 • Cyclomatic Complexity: 442 • Class Coupling: 46
<p>C# Journey Infinita Data</p> <ul style="list-style-type: none"> • Namespaces: 26 • Types: 144 • Members: 1263 • Lines of Code (NCSS): 4425 • Types per Namespace: 5,54 • Members per Types: 8,77 • Lines of Code per Member: 3,50 • Maintainability index: 92 • Cyclomatic Complexity: 2664 • Class Coupling: 257 	<p>F# Journey Script</p> <ul style="list-style-type: none"> • Namespaces: 1 • Types: 9 • Members: 421 • Lines of Code (NCSS): 2505 • Types per Namespace: 9 • Members per Type: 46,78 • Lines of Code per Member: 5,95
<p>C# Journey Infinita Editor</p> <ul style="list-style-type: none"> • Namespaces: 4 • Types: 47 • Members: 860 • Lines of Code (NCSS): 3604 • Types per Namespace: 12 • Members per Type: 18 • Lines of Code per Member: 4 • Maintainability index: 71 • Cyclomatic Complexity: 2751 • Class Coupling: 355 	<p>F# Journey Script Tester</p> <ul style="list-style-type: none"> • Namespaces: 4 • Types: 16 • Members: 942 • Lines of Code (NCSS): 2859 • Types per Namespace: 4 • Members per Type: 58,88 • Lines of Code per Member: 3,04
<p>C# Journey Infinita Game Logic</p> <ul style="list-style-type: none"> • Namespaces: 1 • Types: 15 • Members: 200 • Lines of Code (NCSS): 858 • Types per Namespace: 15 • Members per Type: 13,33 • Lines of Code per Member: 4,29 • Maintainability index: 87 • Cyclomatic Complexity: 506 • Class Coupling: 54 	<p>Project Total</p> <ul style="list-style-type: none"> • Namespaces: 38 • Types: 240 • Members: 4078 • Lines of Code (NCSS): 17895 • Types per Namespace: 6,32 • Members per Type: 17,00 • Lines of Code per Member: 4,39

The Editor

Data Architecture

The data architecture of the project is used for containing the information that is set in the editor and used in the game. There are multiple design decisions in regards to this data architecture. An E/R model of the data architecture for the engine can be found on the CD with an explanation in appendix 8. It is not in the report due to the size causing the text to be illegible. The formal and informal BON generated for the data architecture can also be found on the CD.

The E/R model contains a significant amount of classes, and the design might seem a bit frantic due to the amount of connections. However, the design can for the most part be focused into a few areas: The profession system, the ability system, the storyline/NPC system, the area system, the object system, the troop system and resources.

The Profession System

To make it possible for a player to customize his character, a profession is used. A profession consists of a set of abilities that can be learned, details on what stats are used by the character, as well as a list of equipment.

Companions also uses the profession system. Besides the profession, a companion also contains a list of abilities learned at a specific level. This is different from how the player incorporates profession, since the player needs to actively learn it when he reaches a given number of accumulating points. This distinction is made in order to have significant customizability for the main character, while simplifying the system as much as possible for companions.

Each profession consists of a set of equipment. The quality of equipment scale by level, and as the level increases, so does the name and description, in order to better describe the increased quality. While it would be simpler for the system to have each equipment at a specific level, it raises the problem that a player may set the equipment he uses to a given level that are not sold in stores. By giving names to ranges of levels instead, this inconsistency can be avoided, as there can be equipment for all levels instead, and the user can make as few or as many different names for levels as desired.

Each profession has a series of sprites that they must have in order to work with the battle system, like a sprite for when the character is dead, or moving right. This is so that combat can be simplified by having a known array of animations to choose from at specific situations, across professions. It also makes it easier for the journey creator, as he can animate the player, no matter what professions he use, as long as these basic sprites are used.

The Ability System

In a JRPG, abilities are a central part of the combat system. A single ability can have animation, and a series of effects like ongoing damage. Also, it is possible that abilities may not be usable at a specific time, like a revive ability that can only be used when a character is dead. In order to include this, and make it possible for the user to program, the ability system uses three main classes, the Ability, the Effect, along with activation and expiration conditions.

Balancing the abilities is important for having players create characters separate from a journey, and still have the difficulty level be somewhat fair. Balancing is done by adding either additive or multipliable cost to each effect, condition and value. This results in better abilities having higher costs. The player only has a set amount of points to buy abilities from, and that increases by level, preventing players from taking very expensive abilities at a very early point

Effects are separated by abilities, in that they apply an effect that may work over time. Using effects, it is possible to create a wider array of abilities than without, especially as effects may themselves apply other effects.

Animation works by having a number of animation elements for a given ability. Each animation element consists of a start and end frame, a sprite with an animation speed, as well as a movement pattern between coordinates that may have an origin at the player, the ability target, or the screen. An essential part of the animations are, that they may use the player's professions standard animations, like a simple attack animation. That should simplify animations for the user when creating a series of abilities. Due to the complexity of animations, however, this is currently not implemented in the editor.

The Storyline System

The storyline system is included in a way to better structure game progress. A storyline has a certain amount of states, and each state have a series of Progress elements. A progress element can change the state of storylines, including itself, and can be set to trigger by talking to an NPC, and for having quest items in the inventory. Implementing this, the amount of time the user has to keep track of storyline in code should be reduced significantly. It also fits together with how NPCs work, and the speech system (what NPCs say when you talk to them) uses the progress elements as well.

The NPCs have other functions than speaking. They are also useful for shops to sell items for the player. Shops sells equipment to be used by the player, set at a given level. What specific items are sold depends on what equipment the player and companion characters can use.

The Object System

The object system controls the actual programming part of the editor, and includes classes, zones and functions. These three classes are where the user can write their own code using the JourneyScript language.

Classes are objects with a state, very similar to classes in Game Maker. Classes have access to basic physics, like movement and acceleration, in order for the user not to have to program it. Classes also contains a sprite, which is used to draw the class in an area, and animated at a certain speed, which can be set. Classes uses events where users can write code that triggers at a certain point in time, and variables, which are additional state for a class that a user can add. Classes may connect to an NPC in order to allow functionality for a standardized speech control system, where calling the speech attached to the NPC of a nearby object becomes simpler for the user, as it is being handled internally in the code.

Zones are similar to classes, as they contain events and variables. However, they have no visual presence in the area, and are, for programming purposes, used only as triggers. Zones are useful as they are more efficient than classes, as they have no physics and collision is simpler.

Functions contains code that can be called in events by classes and zones. The real advantage of functions is that they are defined globally, so that a script can be implemented once and used by multiple events in multiple classes and zones.

The Area System

Areas are the locations in the game outside of battle, including the overworld map, towns and dungeons. An area consists of tiles, which are small sections of images that are used to construct the graphics of areas. In order to contain tiles effectively so that information is not repeated needlessly, tiles are stored inside of grid sizes which describes which sections of images the tiles are located, including their size.

Areas also uses zones and classes as gameplay elements, where zones are used for aspects like collision, random encounters and collision triggers. Zones are shown as colored squares, and the color can be chosen to make it easier to differentiate different zones.

The Troop System

The combat system works by having the party of player characters fight a troop of monsters. Combat can be activated through random encounters, which uses zones. A zone can contain a set of troops, and when the player walks through the zone, there's a chance that a random encounter is activated.

Each monster in a troop has a set of stats, just like a player character, and they also use abilities. On top of this, monsters has affinities to specific elements, like fire or physical damage. This helps differentiate different monsters, and makes it possible to create a plant-like monster that is weak to fire, but strong against water. Stronger monsters can have better affinities, and a points system is used in the editor in order to make sure that the affinities are balanced. On top of element affinities, monsters may also be immune or resistant to specific status effects, like sleep.

Monsters also contains a list of item drops. Drops may be either simple consumable items, equipments for the player or a companion character, and accessories. Monsters may also drop specific quest-relevant items to be used in combination with the storyline system. Items are either retrieved after battle when the monster is defeated, or when a steal ability is used on the monster to get the items, if the item is stealable.

The Resources

The data architecture consists of three types of resources, images, sprites and sounds.

Images are a single image, and are used for tiles, along with backgrounds in areas and zones. If an image is a tileset, it has some options regarding how the tiles are placed in the image. Otherwise, images are just a simple image, and little else.

Sprites, on the other hand, contains a series of images, and includes information about collision and the center of the image. Sprites are used for drawing classes, monsters, professions, and are used for the different animations abilities have.

Sounds are either sound effects that may be played once, or looping background music used either for areas or for combat. The only options available for sounds is if it loops, the volume of the sound as well as the panning of the sound towards the left or right speaker.

Editor Data and Game Data

The data architecture is split into two sections, the engine and the game. This is because the engine and game needs different data stored, and are designed for two specific purposes, working in an editor, and working in a game. For instance, an object's current speed and movement direction is not something that is needed to be known in the editor, and what color a zone is, and what names that have been given to different elements, doesn't matter in the game. Because of this, the entire data architecture is duplicated, with only minor overlaps between the two. This may seem like code duplication, and the majority of the code between an engine class and a game class are significant. While it would be possible to use the same classes for both purposes, it would leave some fields to be left unused in either the editor or in the game. This solution clearly separates a game object and an editor object, and also makes the compilation phase clear. To compile, a game object takes an editor object in its constructor. It also simplifies XML, as what it needs to store depends on if it is used for saving an editor project, or if it a compiled project.

Data architecture with XML

When saving and compiling, serialization to XML is used in order to save the data. This is then packaged along with the resources in a zip package, with a custom file extension. This ensures that the entire journey is stored as a single file, and prevents fragmentation and issues with checking for missing or corrupted files. It also does not influence the speed of the game, except for loading, as the data is unpacked into a temporary folder, so that images and sounds are not all loaded into memory at the same time. The journey file isn't encrypted, however, so it is possible to extract the data.

The project package consists of an XML file for each resource, and images, sprites and sounds are further contained inside of subfolders. Besides the resource files, the project package also contains a folder with the folder trees serialized. This is to maintain the specific folder layout when saving and loading. The package also contains an indexing file, whose only purpose is to keep track of the amount of elements. This is done to be able to quickly load all the data, instead of doing a string comparison to identify them. This also comes with corruption issues, if an element is deleted within the package, it can corrupt the entire package which makes all following element unloadable or shifted.

While it has not been implemented, we did experiment with validation of XML files, using XML schemas. This would make it possible to detect corruption, in cost of a slow down during load. However, as corruption requires one to mess up the file outside of the editor, validation shouldn't really be necessary.

We also considered the possibility of exporting and importing data, especially the professions, since you need a profession for the player which is separate from the journey. This was however later decided against, due to the focus on the editor and not the game. This is definitely something that the editor would benefit from, though, and could even be extended for other sections of editor, allowing for content packages to be exported between different projects. Game Maker does something similar to great effect as well, so it should definitely be considered.

A very interesting peculiarity in the system is how references are approached. Because of how the journey file needs to include the data of referenced objects inside of the object through composition, references are done through an identifier. As the identifier is simply an unsigned integer, there is no problem when saving to XML. While it is possible that the serialization in C# is able to deal with object references without any issues, control over the order of objects being loaded are lost. It is also important to be able to reference elements based on ID, as the IDs are used from the programming language. This means that it is necessary to store the actual objects in a way that makes it possible to easily search by identifier. To do this, the data is all stored in a Resource Manager that contains a list of resources, and sorts the contents by ID. This essentially uses the multiton design pattern, with the key to access the elements being the ID, and they have global access as the Resource Manager itself follows the singleton design pattern. As the lists are sorted over ID, it makes it easy to do binary search to find the ID, reducing the search time from $O(n)$ to $O(\log n)$, a significant improvement. In the game, this is further improved, by loading references at the start of the game in order to reduce it to $O(1)$.

Using the Resource Manager, however, causes problems with serialization due to contracts. The contracts that raise problems state, that when setting an ID, the resource with that ID must exist in the Resource Manager. However, the automatic serialization doesn't add the objects before setting the resources, and breaks the contracts when setting the values. Because of this, loading projects is not possible while using contracts, and the only way to fix that is to manually deserialize the game objects, which is not performed for this project.

Editor Architecture

Preliminary Design

The first decision in following with systematic design of the interface, was to find what tasks a user of the program would require, the result of which can be found in appendix 4. This is build upon the data architecture created, so that tasks are created in order to match the data in the database, in a way that makes sense for the user. Afterwards, virtual windows are created. We developed alternatives to some of the more complicated windows, but only the chosen design was further developed.

The most interesting virtual windows can be found in appendix 5. Many of the details in the windows are repeated in other windows. Mainly because the same functionality is needed for more windows. Like a list of resources or a drop down menu of a resource.

A basic user manual for the editor can be found in appendix 9.

The editor opens with the main window. Due to the main window being the most used, it serves multiple functions, and is split up in several components. The first is the resource view, which has nine tabs, each with one to four folder systems. This is where the resources of the journey are created, removed, renamed, and edited. Moving these important functionalities together in a single area of the program, much like Game Maker. The second part is the area editor, where the areas are edited. The area editor is itself split in three parts. The settings window, the main view of the area, and the view settings. The settings window is where all the settings of the specific area can be set, including tiles, objects, backgrounds, sounds and information about the area itself. The main view is used for dropping the selected tiles, object and zones onto the area, and also removing them. The view settings allows the user to disable zones, objects and tiles that are not on the current depth, and is a useful extra tool for the user.

The editor also contains many additional windows, and most follows a similar style. The resources that have an in-game presence all have name fields, and possibly a description field as well. Also, all windows with the exception of the main window, contains a Save and a Cancel button. Save will maintain any changes made to the current resource while cancel will discard any changes made to the resource. This is the easiest way to implement undo functionality, by having a set of data in the editor that is only saved when save is clicked, instead of having to worry about saving changed data to a list of undo-functions. Simply pressing the cancel button closes the window without any changes to the data before the window was opened. That does mean that clicking cancel accidentally, removes all the recent data, which is a problem with the current program. A way for us to solve this would be to detect if changes has been made or compare current data to saved data, and to give a warning if there is an inconsistency.

The area editor does not use the save and cancel system, however, and instead saves and loads when the current area is changed. This was implemented as the area editor is inserted into the main window, making it impossible to actually close an area, but does require one to save the entire journey. Game Maker does this by having the areas opened in a sub-window in the main window, which can be closed. This solves the problem, but complicates the interface somewhat.

The actual resource data, when opening a resource window, are loaded on initialization of the window. As there would be too much data in memory at the same time, we instead have all the resources on the hard drive, and load it into main memory when it becomes necessary. This ensures that the controls have loaded and are ready to be manipulated. Some custom controls like the XNA controller needs to have an OnLoad method assigned, since it is not fully loaded on initialization of the main window. The OnLoad for the XNA controller will then load the images, if they exists.

Being able to resize the windows is important for usability, as different users have different screen resolutions, and different scenarios may require different window sizes. However, that is not currently possible except for the main window. While the worth of having this feature can be discussed, it is definitely something that should be looked into in the future.

Overall, we focus on a multi-page dialogue. Due to the size of the editor, it would simple be unfeasible to try and consolidate all of the information into a small set of windows that control a large part of the editor. However, steps have been taken to reduce to total amount of windows, by including lesser important windows into the main resource window, which have saved on the amount of windows used, especially for the area and storyline.

Ideally, we should be able to manipulate multiple windows independently from each other, like editing multiple code events simultaneously. This would cause a significant complication, as it would be necessary to include threading in order to be able to operative multiple windows simultaneously. It also complicates the navigation between different windows, as opening multiple instances of the same data should not be allowed. To simplify this, threading between different windows is avoided by locking the action to one specific window. This avoids complications, like deleting a resource that is opened, or editing multiple copies of the same file. It does significantly lower usability of the program, but considering that Game Maker also locks windows in order to avoid complications like this, we consider it an acceptable decision. If one were to attempt to include threading, the first step would be to map all navigation paths using a state diagram, and then decide the proper cause of action for each individual jump.

WPF Controls

Custom controls have been included to make the editor windows easier to make. Most of the controls contain custom properties, which can be set directly inside the WPF documents. This has been done with DependencyProperty and by registering the properties.

The Resource Tree View is used on the main window to contain all the folders. It also includes functionality for creating and removing resources and folders, and how it is implemented can be read below. The tree view uses a checker when a context menu is opened to find the element which was clicked. This is done to access the resource type and actually open the correct resource editor.

The Editable Text Block was made to make the folders and resources editable in the resource tree view. It consist of a text block and a text box, which swaps when the block is being edited. This also checks if the name is empty and in that case, it will change back to the name before the editing in order to prevent the user from entering an empty string.

The Int / Float Text Box was made for all when number fields were needed. The spinners are part of the control, and also checks for key presses like up and down. The control itself contains min, max, increment and large increment properties, which can be set from WPF. It also implements an event that allows for delegates to be triggered when the value in the field is changed.

The Resource Drop Down was created so we could select a resource by showing it as the tree in the resource folders. This also allows for hiding some resources if it is required, like if the resource itself should not be used. The resource drop down implements a lazy load when it comes to setting the selected item, to make sure that the selected item is set OnLoad and not before.

Drawing

For the editor, it is important to be able to draw images. In order to accomplish this, we use XNA. This is because it would be the tool used to draw in the game as well, so this prevents errors from occurring where the different toolsets accepts different forms of graphics. However this approach is not without problems. Due to the way the XNA project is set up, we could not use XNA for the editor, as it would require creating new controls like text boxes and so on. However, we found a guide^[31] which allows for XNA imaging in WPF windows.

The draw method itself runs through a list of XNA Textures and draws the textures. The XNA Texture is a class created with the purpose of filtering the textures to be drawn. It also contains the draw rectangle and the draw partial rectangle. An XNA Texture knows which image it belongs to, and contains a class or background if it belongs to that category. The XNA Texture also contains a depth value which is used for sorting. Since the draw method simply runs through the list of XNA Textures, the list needs to be sorted such that depth has an influence. The XNA Texture filtering is used in the draw method when the area settings are set, so it is possible to skip a certain depth, skip classes or skip backgrounds. This method can cause problems in-game, however, if objects changes their depths frequently, as the list would have to be re-sorted.

Besides the XNA textures, other drawing functions are done directly by WPF. The grid, sprite grid and zone drawer are all WPF drawers used by editors, and this is all acceptable as these will not be used in-game and doesn't need to work with XNA.

The purpose of the grid control is to draw a grid or draw cells, including the spacing. The grid contains properties allowing for customization, such as the thickness and color of the lines. It also contains properties changing the spacing, offset and cell size. The grid can be quite inefficient when using spacing, since it is required to draw rectangles instead of lines, but the grid is optimized so using a spacing of 0 will change the drawing mechanics to lines instead. It also contains a property that allows selection, making it possible to select cells, which is used by the area editor both for the main area and the tiles. The sprite grid is a modified version of the grid, which allows for individual sprites to be shown, as well as the use of indexing. The zone drawer is also a modified version of the grid, but unlike the sprite control, it contains a list of zones to be drawn. The zone control also implements the gray zone, which is drawn when creating a zone. It also contains different filtering fields which allows for hiding different zones.

The main issue with this XNA control is efficiency. The rendering method is called when the parent windows decides to render, even when nothing is changed and can be triggered when the control changes size. Every time a render is called, it has to loop through all the pixels of the XNA Texture and convert it to a writeable bitmap which can be used by WPF. This process takes a long time, especially with larger images. This was solved by implementing an FPS property, which can regulate how many times it should actually render. This increased the performance, but it was still slow when having to draw large images. A ForceRender method was made to combat this problem, which bypasses the render method, but perform the same actions. That makes it possible to restrict the XNA control from rendering (FPS = 0), and then force the render when needed. This increases performance tremendously.

Another issue with rendering is disposing of unused textures. One would assume that a texture could be disposed when a window is closed, but that is not the case. While a window is closing, it will render one or two times more, but at this point the texture would have already been disposed. Due to the rendering being called, the XNA control tries to draw the disposed texture, which results in an exception. What also causes an exception is disposing when the XNA control reaches the drawing code. This causes textures to stay in memory until garbage collected, which results in XNA textures using up a significant amount of memory, causing memory leaks when working on bigger projects with large images.

Another problem with the textures and the writeable bitmap conversion in general, is the fact that a texture is limited to 4000x4000 pixels, which translates to limitation in the area size, background size and even the amount of sprites a single sprite can have due to the way it is represented. This is actually a bigger problem than it would seem, due to the often large areas, and overworld maps from some older RPGs exceeds this size.

The problem with XNA seems to originate from the fact that the editor is not separated from the drawings. In his presentation, Tim Sweeney focuses on the importance of performance, and in splitting the work of the shader from the CPU. We were proved wrong in assuming that working with a 2D game platform would limit the importance of performance, and the slowdowns have been significant. In order to fix this, we have to represent images in an abstract fashion, and then fetch the actual image data when drawing the screen, possibly using the GPU. As it stands, the inefficiency of XNA when using WPF is by far the biggest limiting factor of the editor.

MVVM

The MVVM^[32] (Model View ViewModel) design pattern is a pattern used, when data needs to be shown in a modern user interface. The idea is to separate the user interface from the data by a class specifically designed to contain the data and tell the user interface when updates in the data happens. This is done with the observable collection which notifies when an element is added or removed from the collection. It is important to note, that the contained element inherits from the `INotifyPropertyChanged` interface, so that the user interface knows when data changed inside the collection, as it already knows when the collection itself changed. In WPF, the `DataContext` of the specific control is set to the viewmodel class. This allows for a binding internally to the data, which further allows access to the properties of the data.

We used this design pattern in the folder system for the resource tree view. The folder elements inherits from `INotifyPropertyChanged`, and call it whenever a more important property has changed, like visibility. The folder element class contains many methods to traverse the tree both upwards, to find the root, and downwards, looking for a specific child.

The folder view model contains an observable collection of folder elements, but also controls everything related to saving and loading the folders. On the user interface side, the resource tree view sets its `DataContext` on construction time, to a folder view model, which the WPF then binds to, creating a dynamic list of folders.

Programming Language

Overall Design

Language Basics

The first step when designing the language is to ascertain the basic structure of the language. If the language is iterative or functional, if objects and loops exists, and so on.

There are multiple general ways to create a language interface to the user, and have it work within the bounds of the editor and game. The simplest solution is to reuse an existing language and embed it inside of the editor, using a specific API. The advantage of this approach is in the reuse of code, as the language has the compilation process already implemented. However, this comes at the cost of customizability. On the other hand, implementing a language for this specific domain would allow for greater customizability, but would take additional time.

A major reason why a domain-specific language makes sense, is in terms of usability and type safety. In terms of usability, it becomes a matter of how different the language needs to be from other languages. If going simply by the fact that it should look like a simple programming language, either solution works, however, this can lead to issues with code being overly complicated in order to fit the existing system. For example, an event call could potentially be reduced from "Event.Call(Class(objRef,"ClassName").Events("EventName")),Parameters)" to "call objRef(ClassName) EventName(Parameters)" by being able to define the scope of the language features. This is also important in terms of type safety, as the first example could be implemented using a method, which would not be able to check the length of input at compile time, or check if it matches the actual event, as this code is executed at runtime. On the other hand, when defining a custom parser, it is possible to implement rigid type checking rules by being able to check names and types in the actual editor at compile time. Such a feature is a major plus for the system, and for this reason, a custom made domain-specific language makes sense to use in this project.

Programming Paradigms

When dealing with game entities, they will have certain properties, like their position in the game world. In order to affect these properties, they must be representable in some form within the programming language, making shared variables a core connector type required in the project. Also, as this is a game that is run in a loop, the game engine needs to select when to call the code written by the user. This fits very well with discrete event simulation, as defined by Simula^[33], the first object oriented language. Events runs in a loop in a given sequence, where the user implements code for the event that can be executed at runtime in the system. With events and shared variables, it makes sense that the programming language will be object-oriented.

However, both variables and events can be created without requiring them inside of the language. As the language is used within an editor, the language can exclude the object-oriented aspects, resulting in it being a simple scripting language, just like GML^[24]. By moving the workload of events and variables from the programming language to the editor, it is possible to create an object oriented scripting language. When variables are declared outside of the events, unlike in Game Maker, typing can be more controlled, and the problems with creating new variables at runtime, and calling variables before they are created, are solved.

The programming language has a dependency on state changes. Therefore, an imperative language seems more useful than declarative programming, including functional programming, as side conditions can be used to set state. While it is still possible to describe states using functional languages via monads, this is too indirect an approach to setting values when a user that may not be skilled at programming has to use it. Because of this, imperative programming is the favored paradigm.

As Game Maker has demonstrated, having events has clear advantages in connecting the user generated code and the game engine. Therefore, it makes sense to have the program be event-driven. Each event has a code body, in which an event header is defined. Besides the event body, multiple locally defined functions can be defined, that can only be called from inside the event.

As Jeroen Dobbe points out^[30], a couple of specific points needs to be taken into account when creating a DSL for a game system. The specific game rules for the game are controlled through the different kinds of available functions that can be called, which limits the access to what the user can change. Game objects are included through the use of different kind of types, some with accessible properties. The game storyline is included through the use of the storyline type, which adds some structure to how the player can manage story progression. Player control is taken into account in the same way that Game Maker does, by creating specific events for the different UI options accessible to the player.

This means that the chosen language is an imperative, event-driven object-oriented scripting language, just like Game Maker. A full description of the language can be found in Appendix 6.

Typing discipline

Due to the ability of having a direct access to the defined elements in the editor at compile time, a strong type system can be implemented, even though Game Maker skips this opportunity. As Tim Sweeney points out in his presentation^[29], it is important for programming languages to keep the code at run-time as reliable as possible, which means discarding as much faulty code as possible at compile time. Therefore, there should be relatively few limitations in regards to how typing is used in the language.

Implicit typing is possible to use here (something that Game Maker uses by necessity), but it would require a significantly detailed system of type analysis to guess the type to be used. It does not seem like implicit typing would be worth developing, because of the time it takes to develop.

Dynamic typing makes little sense for this language, despite Game Maker focusing heavily on this. The main reason is, that a desirable feature for the language is to have as many things fail at compile-time as opposed to run time, as this is a considerable weakness in GML. As such, having a static type checker is more ideal.

A strong typing system makes sense in a language with a static checker and explicit typing, as comparing an image with a sound file yields no real value for the programmer. However, in terms of numeric types, it makes sense to assign the boolean "true" to a value, or have an object reference return true if used as the test expression in a control structure. This opens up for two areas useful for weak typing like Game Maker uses: all types can be checked if true, and all numeric types can be used interchangeably through implicit conversions. These two aspects makes a weak type system attractive as a feature in the language.

Nominative typing, contrary to structural typing, makes sense for the system. If the user tries to compare two objects, the most natural way would be if the objects both reference the same instance of the same class, as opposed to their position. This is what Game Maker essentially does too, though both can be considered true as references are all resolved to identifiers. In a similar manner, comparing two images only makes sense if they are referring to the same image in the system, and not the actual content of the image. If nothing else, such an equality analysis could turn out to be crippling at runtime if every pixel in two images would need to be checked.

The programming language chosen is an explicit, nominal, weak, statically checked language.

Interpreter or Compiler

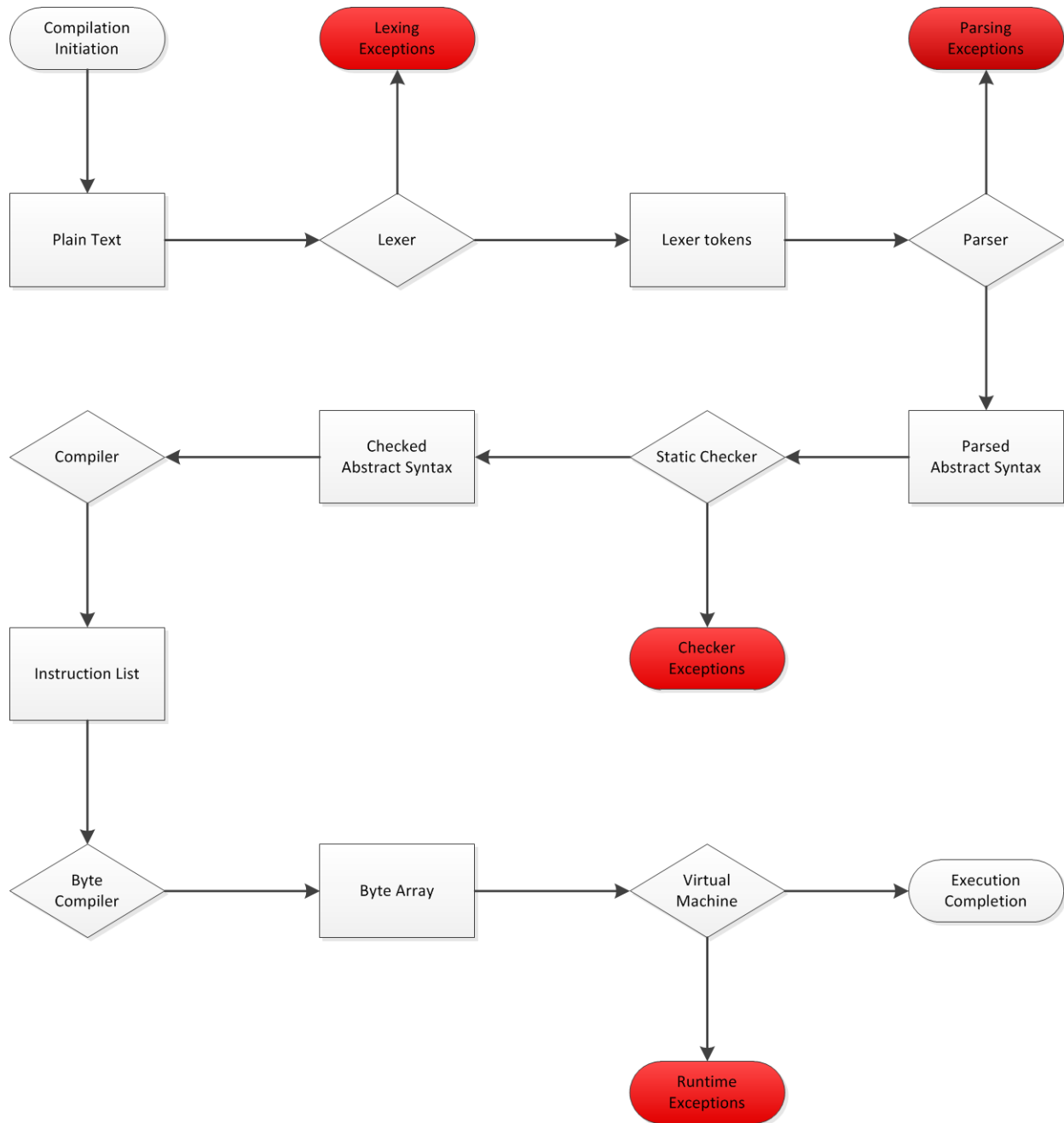
A major decision in regards to the programming language is if it is interpreted at runtime, or if it uses compiled bytecode in a virtual machine. The advantage of having an interpreter is simplicity, which is desirable, and probably why Game Maker does it. However, using a compilation scheme means that code should execute faster than if interpreted, which seems a much more important factor when choosing the programming language.

Another issue with code is decompilation, where someone else is able to reconstruct written code, and essentially get access to the source files through the compiled file. Compilation helps avoid decompilation as code is in the form of instructions, which can't easily be converted back into workable code. While not a specific requirement, it a definite plus when comparing the two approaches. On top of runtime efficiency, compilation makes more sense for the project.

Compilation Pipeline

In order to execute compilation of code from a written string, the code has to go through a series of transformations. First, it is important that the written text is interpreted as a series of tokens, adhering to a specific set of rules through a lexer. This is in order to simplify the parsing process, which follows. The parser converts the code into an F# readable tree structure describing the code, which can then be checked by a static checker, compiled to a list of instructions, and finally saved as a byte array that can be executed by the virtual machine.

The lexer and the parser can throw exceptions if the code is badly written, and the static checker can throw exceptions if the code contains flaws not possible to check in parsing or lexing. The virtual machine can also throw exceptions, should a problem arise at runtime.



View of the compilation pipeline

The differences between the parsed and checked abstract syntaxes lies in their intended use. While the parsed abstract syntax has to be used for conversions from the parser, the checked abstract syntax is used for preparing for the compilation process. For instance, the names are all resolved to identifiers in the checked abstract syntax, and type data can be used as well.

Variables and functions that are statically defined are a problem in relation to the pipeline. Some types have variables that can be called by the user, which will have a presence in the pipeline. Throughout the pipeline starting from the checked abstract syntax, we think it is necessary to track these special variables separate from user defined ones, as the virtual machine needs to access these variables differently. This adds complexity to the code as the variables are defined multiple places in the code, and it would have been better if the types of variables were described at a single point in the code. This is not possible, however, as the variables are a very core part of the programming language, and simply adding them to the list of user defined variables is not possible, as some of them cannot be edited.

Functions like `instance_create` or `play_sound` suffer the same problem, but unlike with variables, there is no alternative here, as the defined functions have to exist separately in order to reference code written in the native language in the virtual machine. Also, due to the large number of functions that would have to be defined in order to be considered a complete language, only a small amount of these functions have been implemented in the system, to showcase the design.

Lexing and Parsing

Lexing

Lexing and Parsing uses the respective tools in the FSharp PowerPack, as described in the tools section. Therefore, the code written for these two areas are in the special languages used for these, and the corresponding `.fs` files are automatically generated code.

Lexing, being used for splitting a string into a list of tokens, is a simple process and contains little surprises in the design. However, there are still a couple of noteworthy aspects of the lexer.

A design decision made for numbers was not to take negative numbers into account. This was done because of issues in the parser, where a negative constant number (like -16) would not parse when trying to subtract two numbers (like 5-2). As a consequence, the two's complement nature of integers results in the minimum integer value being one above what is possible on the .NET platform, as the negative number is first lexed as a positive number, and then negated later.

Another decision is regarding context-dependant lexing. Due to the need for having keywords, it complicates naming in the editor, if the editor has to take the keywords into account, like throwing an error if someone tries to name an object after a used keyword. In order to avoid this, context dependant lexing is used in order to make sure that names defined in the editor can have the same name as a keyword. Using the F# however to create contexts proved difficult, however, and a workaround was required.

To make context-dependant lexing, each time a token is returned, it is passed through a method that saves that token, making it possible to check which token was returned previously. If the last token was "goto", "call" or a dot, it is guaranteed that the next token (if it can be lexed) should be a name. This does not cover all instances where names can be used, however. Functions calls does not use anything beforehand to help indicate that this is a name, and cast names are in parenthesis which does not dictate exclusion of keywords. To make these context dependant, we would need to change the language, like having a call keyword before function calls as well.

Parsing

The parser breaks down the lexed tokens into a tree structure of non-terminals, with each non-terminal describing a finite state machine. If the list of tokens does not match a possible state in the machine, the code is discarded, and an error is thrown. Otherwise, a tree structure in an abstract syntax is created. A full view of the parser structure in the Backus-Naur Form can be found in Appendix 7.

The basic structure of the programming language has three basic levels: The body, the statement and the expression.

The body

```

<jssection> ::= <n> <sectiontype> <n> <functionlist> EOF

<functionlist> ::= | <function> <functionlist> <n>
<function> ::= <typeempty> <name> ( <parametersdec> )
                                     <n> { <n> <body> <n> }
<sectiontype>: <eventfunc> | <functionfunc>
<eventfunc> ::= event ( <parametersdec> ) <n> { <n> <body> <n> }
<functionfunc> ::= <typeempty> function
                  ( <parametersdec> ) <n> { <n> <body> <n> }
<n> ::= | '/n' <n>
<typekind> ::= <boolean> | state | <value> | string | <object>
              | sound | speech | sprite | image | story | npc
              | item | equipment | questitem | accessory | class
<type> ::= <typekind> | <typekind> [ ] | <typekind> [ ] [ ]
<typeempty> ::= | <type>

<parametersdec> ::= | <n> <paramdec> <nextparamdec>
<nextparamdec> ::= | , <n> <paramdec> <nextparamdec>
<paramdec> ::= <type> <name>

<body> ::= | '/n' <n> <body> | <stmt> '/n' <n> <body>
          | <stmt> ; <n> <body> | <control> '/n' <n> <body>
          | <ctrlunbalanced> '/n' <n> <body>
          | <stmt> | <control> | <ctrlunbalanced>

```

The body is the main overall structure of the code, with the event and function declarations, something that Game Maker does not have. A code section contains either an event or a function, and a list of functions. Each of these has a list of arguments, and functions can have a return type as well.

The statement

```

<stmt> ::= <expr> | <vardec> | break | continue
        | call this . <name> ( <params> )
        | call Zone . <name> . <name> ( <params> )
        | call ( <access> ) . <name> ( <name> ) . <name> ( <params> )
        | return | return <expr>

<control> ::= if ( <expr> ) <n> <innerbody> <n> else <n> <innerbody>
            | while ( <expr> ) <n> <innerbody>
            | do <n> <innerbody> <n> while ( <expr> )
            | for ( <n> <inst> ; <n> <ltest> ; <n> <cexpr> <n> )
                <n> <innerbody>
            | switch ( <access> ) <n> { <n> <cases> <n> }

<ctrlunbalanced> ::= if ( <expression> ) <n>
                    <innerbody> <n> else <n> <ctrlunbalanced>
                    | if ( <expression> ) <n> <innerbody>

```

The statements are a single line of code in a block, like a return statement or variable declaration. Statements are divided into single lines, control structures, and unbalanced control structures. Statements are split between simple statements and control structures, for the simple reason that control structures should not terminate with a semicolon, and the unbalanced control structure is split to help avoid ambiguity.

The Expression

```

<expr> ::= <access> | <operationNeg>

<access> ::= this | this . <accessref> | other <accessoption>
          | global . <name> <accessoption>
          | global . <name> [ <expr> ] <accessoption>
          | global . <name> [ <expr> ] [ <expr> ] <accessoption>
          | <name> <accessoption>
          | Zone . <name> | Zone . <name> . <accessref>
          | <name> [ <expr> ] <accessoption>
          | <name> [ <expr> ] [ <expr> ] <accessoption>

<accessref> ::= <name> <accessoption>
             | <name> [ <expr> ] <accessoption>
             | <name> [ <expr> ] [ <expr> ] <accessoption>

<accessoption> ::= | ( <name> ) | . <accessref>
                 | ( <name> ) . <accessref>

<operatinNeg> ::= <operation> | - <expr>

```

```

<operation> ::= <access> <n> = <n> <expr>
| <access> <n> is <n> <name>
| <access> <n> = <n> <arr1assign>
| <access> <n> = <n> <arr2assign>
| <number> | <floatnum> | <cstring> | null
| <resource> | <expr> <n> ? <n> <expr> <n> : <n> <expr>
| ! <expr> | ( <n> <expr> <n> )
| <expr> + <expr> | <expr> - <expr> | <expr> * <expr>
| <expr> / <expr> | <expr> % <expr>
| <expr> <n> == <n> <expr>
| <expr> <n> < <n> <expr> | <expr> <n> <= <n> <expr>
| <expr> <n> != <n> <expr> | <expr> <n> >= <n> <expr>
| <expr> <n> && <n> <expr> | <expr> <n> || <n> <expr>
| <name> ( <n> <params> <n> )
| ++ <access> | -- <access> | <access> ++ | <access> --

```

Expressions always have a return-type, making it possible to link expressions together using different operands. Expressions are split into access and operation. The reason for this is, that variable access are considerably different from other operations, and are used in different circumstances. It wouldn't make sense to assign to a function call, or do a switch statement on a constant value. Statements are also split between simple statements and control structures, for the simple reason that control structures should not terminate with a semicolon.

Shift/reduce errors

A peculiarity with the parsing system has caused a few headaches during the course of the project. Due to the way the parsing system works, the implemented system doesn't attempt to predict multiple paths throughout the tree structure, which can cause confusion due to ambiguity, leading to the so-called shift/reduce errors. The previously mentioned example of a negation operation contra negative number is caused by this problem.

There are two general ways to avoid shift-reduce errors. The first is to declare precedence, which is used to remove ambiguity in expression ordering, like determining that multiplication is stronger, and should be determined before plus or minus. The other is to make the language more explicit to avoid ambiguity. For instance, the if-else control structure allows unbalanced structures, where there are no corresponding else clause, and in the case of two if statements and only one else statement, there is uncertainty to what if statement the else statement is used with. By creating a separate control non-terminable that disallows an unbalanced statement to be used in the if-statement and only in the else statement, this ambiguity is removed, as the other way around, with an unbalanced if structure in the if statement, simply cannot be parsed.

However, this also causes issues in regards to the new-line significance used in the program. While a single ambiguity like the above can be solved, it becomes a different issue when new-lines are used in many separate cases. The major issue with this is, that if a statement is allowed to end with a newline, then ambiguity arises for every use of new-lines. This has resulted in a design decision to move to a terminalist style with a semi-colon at the end of each statement, as the new-lines prevented correct parsing.

Code Checking

After parsing to abstract syntax, the tree structure could be considered ready for interpretation. However, as a design decision was to focus on static checking in order to raise as many of the problems in the code as possible at runtime, parsing is not enough. This is because the parser cannot catch all errors possible to write in the program.

One of the purposes of the static checking that is possible in the DSL is to check that all referred names actually exist at compile time, and all function calls has the correct amount of arguments. This is simple, as all elements at compile time has already been created in the editor, and the checker has to have access to these resources. Calling other functions and events is a different matter, however, as the list of arguments is defined in the code itself. Therefore, it is not possible to compile a single piece of code individually, it needs to know the general structure of other pieces of code. To accomplish this, the checking is split into two phases. The first phase loops through all the events and functions in the program and retrieves a header, containing the ID of the resource, the list of arguments with type, along with the possible return type if it is a function. All headers are then passed along as arguments when checking the code.

There are many areas in which the code checker has a responsibility to fail, if the code is not acceptable. A detailed list of issues that are dealt with in the checker:

- Event is declared in function, or visa versa,
- Using break and continue outside of a loop or switch statement,
- Using a switch statement over an array, or over the "this" keyword,
- Using a switch statement over a value (double) or a boolean,
- Checking a switch statement for multiple different enums,
- Checking a switch statement for the same thing multiple times,
- Using default or null checks multiple time in a switch statement,
- Checking a switch statement with the wrong type, or an array,
- Using a function that returns void in a situation where a return type was required,
- Using the "this" or "other" keywords in a function,
- Performing an event call on an object that has not been cast to a specific type,
- Writing a function with a return type, but never returning,
- Calling event or function with wrong amount or type of parameters,
- Attempting to put an array inside of an array,
- Assigning to "this", "other", or a globally accessed zone,
- Assigning to variables that are un-assignable, like "class" or "parent",
- Doing operations on types that do not support that operation,
- Comparing two types that cannot be compared in that context,
- Casting a variable of a type that isn't object or zone,
- Accessing a variable, event or resource that doesn't exist,
- Trying to access a local variable that does not exist or is not in scope.

An additional task for the static checker is to move the declaration of variables. In a compiled language that does not support dynamic allocation, it is important to define the size of a stack at compile time. To do this, a block starts by defining all the variables that are in scope before any other code is compiled. This moves only the declarations, not the assignments, in the case where a declaration is also an assignment, the code is split into two sections (for instance, from `bool x = false;` to `bool x;` at the start of the block, and `x = false;` at the previous position).

Though the static checker handles many of the problems that might later arise at runtime, it is not exhaustive. It cannot prevent, for instance, unavoidable null checks or search for infinite loops, and it doesn't check issues related to the editor, like defining an event callable by the engine with arguments that it shouldn't have. It also has a very significant flaw, it cannot determine at what position in the code a problem has occurred on, as this information disappears after the parsing stage. The parser has the opposite problem, however, as it can show a specific line, but it can only throw generic error messages. It should be possible to get line count after the parsing stage, by supplying meta-information when parsing terminals to the abstract syntax, and using that information in error messages. The only apparent solution to the generic error message problem, would be to implement a parser from scratch and include specific error messages for different structural checks.

Compiling

The compilation step is responsible for converting the parsed and checked code into a structure that can later be read in a virtual machine. As the virtual machine runs on top of another virtual machine, the Common Language Infrastructure, compilation does not need to take physical memory locations into account, as the virtual machine, being written and compiled on the .NET platform through C Sharp, further translates the code and handles physical memory. Therefore, compilation only needs to translate the code into a list of instructions that use a stack frame for calculations.

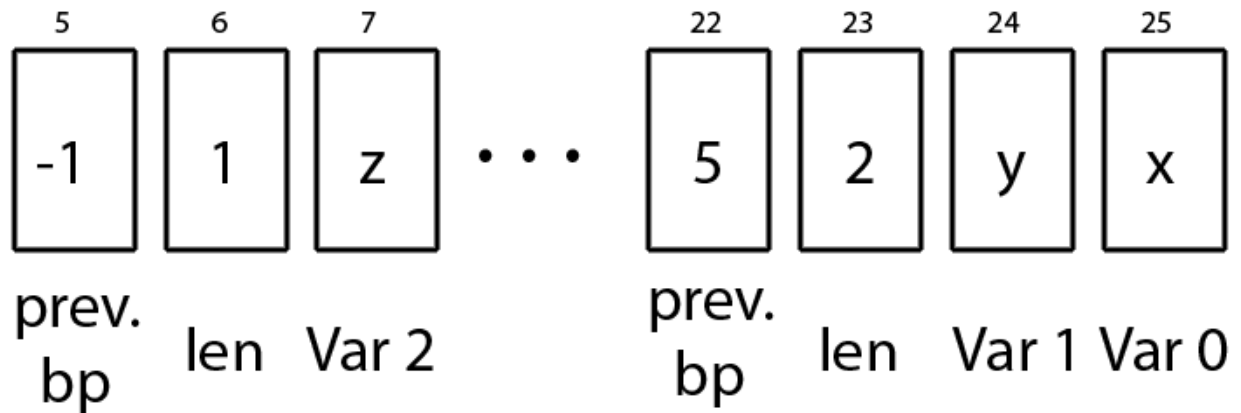
Instructions design

The instruction set consists of bytecode that describes the programming language in a way that makes it executable by the virtual machine, unlike the interpreter which uses the abstract syntax tree structure.

The basic aspects of the instructions is the two arrays that are maintained at runtime: the code and the stack. A description of how the instructions affects the code and stack, as well as the different pointers and counters, can be found in the comments for the virtual machine.

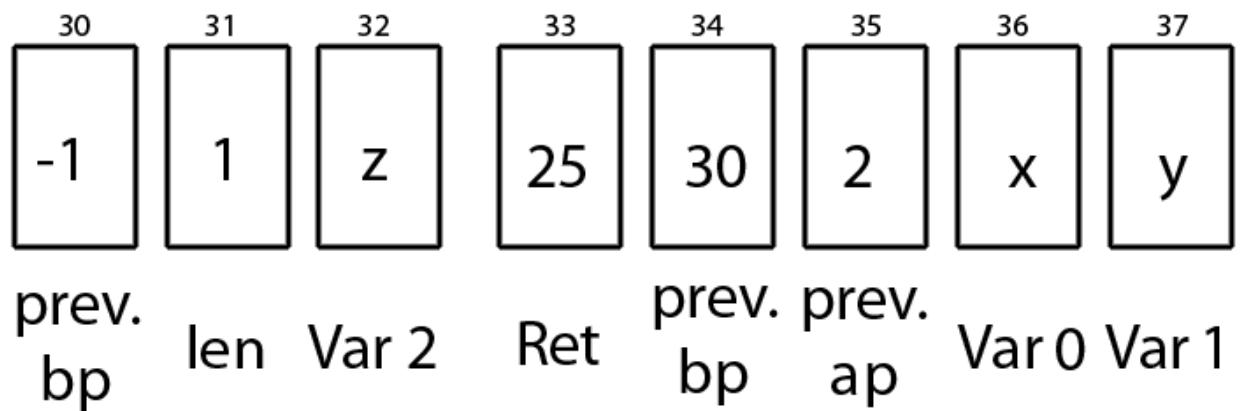
The instructions are based on the .NET instructions[34], and a shorthand versions of some input values are used. This is a way so that, in situations where very short values are frequent, it is possible to use a byte representation of a number, instead of an integer, which would be four times that length. This is not the reason why the instructions are based on .NET, however.

In the current design, the virtual machine is created on top of .NET, meaning that the virtual machine runs on top of another virtual machine. This could potentially be made more efficient by simply having the compiler compile directly to .NET compatible code, and have the .NET platform execute the code. The instructions are designed so that transitioning to such a system would be as easy as possible.



The variable frame

An important aspect of the instructions at runtime is the different kind of frames on the stack. There are two types of stack frames, the variable frame and the call frame. The variable frame consists of a reference to the previous variable frame, the amount of variables in the frame, followed by the variables. In the above example, the current base pointer is at 22.



The call frame

In order to have multiple code executions on the same stack, a call frame is used to identify each individual function. The call frame has a reference to the position in the bytecode to return to after a call has finished, as well as to the base pointer (to the active variable frame at call time) and argument pointer (to the active method at call time) when calling, followed by the arguments for the call. In the above example, the current argument pointer is 36, and position 25 in the bytecode contains the statement after the call.

In this language the arguments are separate from the variables. This is because variables are moved to the earliest in the block when declared, meaning that each block of code has a block of variables on the stack frame. Judging by the looks of the Java bytecode, this is different to how Java does it, where the variables are not declared at the start of the block, but of the start of the method, meaning that only a single stack pointer is required. As this is a more complicated process, the variable blocks are used here as a simpler alternative that produces the same results.

The compiler

The compiler is constructed to write code backwards, carrying the rest of the code as an input for each function call, similar to the continuation passing style. The advantage for this is, that it avoids the use of the expensive append operation. As lists in F Sharp are constructed similar to a linked list, appending two lists would be an $O(n)$ operation, due to the need of having to copy the lists. As this operation would have to be done after every single function call, should they return a list of instructions, this will cause the program to become significantly less efficient. However, when passing the remainder of the code, it will be possible to creating the front of the list in the method using the simple cons operator. This is much more efficient, with no need to copy the lists into a new one, as the cons operator simply needs to move a single pointer. However, there are still parts of the compiler that uses the append operand, when constructing the blocks for the variable stack frames. This is due to how the block needs information on the amount of variables declared in the block, and in other words need information that is gathered in a previous method, which requires a list to be constructed. Redesign of the code could fix this issue, and remains an area of improvement for the compiler.

The other use of the backwards compilation is for optimizations of the code through pattern matching. Using pattern matching, it is possible to check the next few instructions in the list, making it possible to optimize based on later instructions. For instance, when adding two constants, the resulting instructions would be the two values followed by an add instruction. It would then be possible, when making the constant value instruction, to check if the later two instructions are another value and an add instruction, in which case it should just return the added value instead, reducing the amount of instructions in the final code. While these optimizations are possible in the code, they are not implemented as they are only optimizations. Backwards compilation would make it possible and very simple to write optimizations for the compiler at a later date.

Throughout the compiled instructions, the code maintains a reference to the various arguments and variables. As the instructions works with the variables top-down, it is not possible to simply use the ID of the variables, as the static checker uses, as it would not be very efficient to go all the way back to the first base pointer, and then count upwards, especially considering that it makes most sense that the last declared variables are checked first, as they are statistically more likely to be the ones used, as most variables should be declared in the same block that actually uses them. Instead, the references to variables are converted so that the integer used to refer to a variable matches its position in the block of declared variables, with new variables being declared at the top. This means that two different operands can compare to variable 0, and actually work on different variables.

Labels are heavily used in the compiler as well. Labels are a special instruction, that makes it easier to make control structures by maintaining a list of labels, and then using these labels as possible jump-points later in the compilation. The advantage of this is, that the position in the code that the label represents, is not actually known, not only due to the instructions being written backwards, but due to how one would define a "position". Only when the code is fully assembled can the jumps be created.

Actually saving the instructions to bytecode is another issue, as many of the instructions have parameters. While the instruction itself can be identified by a single byte, the remaining instructions contains additional information. As the instructions will always have the same amount of parameters, simply containing the parameter data as bytes after the instruction byte would make it possible to identify it at runtime. Thus, it simply becomes a question of storing the data as bytes, which can be done easily through the bit converter in the .NET framework.

The length of the individual values may not be the same, though. In the case of strings, which has no set length, storage may vary greatly. These are stored by having an integer count the length of the string, followed by the two bytes for each individual character. The alternative would be to use the encoding tools supplied by the .NET framework, but it has to be possible to know the amount of bytes in the string, as the array of bytes contains more than just the string. This is simple for 32-bit integers and doubles, each of which use 4 and 8 byte respectfully, but not so for strings.

Also, the length of integers may vary as well, due to the shorthand representations that reduces them to one byte. To do this, an integer that may have a shorthand has an additional byte in front of it, determining its shorthand status. If it is in fact a shorthand, the length of the variable is one, and if not, the length is five (the identifying byte along with the actual byte values). While this technique does raise the size of integers by 25%, it should still be an optimization overall when shorthands are used in cases where low values are expected, like variable positions. The technique should obviously not, and is not, used in situations where low values are not expected, like resource identifiers or positions in the code.

Having parameters for instructions does mean that labels does not have to point at the position of the instruction, but the position of the first byte of it. Therefore, the compiler has to calculate the length of the instructions when building up the labels. When a label instruction is then encountered, the correct byte position can then be assigned for the label identifier, with the label instruction itself counting for zero bytes. When the instructions are then saved, the label instructions can be removed completely, while the jumps refer to the correct byte position in the code.

Virtual Machine

The virtual machine utilizes an infinite loop that continues to loop over the code, using a program pointer that is continuously increased by at least one. It also uses a base pointer and argument pointer in order to locate the start of the topmost set of variables and arguments, and a stack pointer which points to the top of the stack used by the virtual machine.

The stack consists of a struct which may contain either a byte, an integer, a double, a string or an object. The first implementation of this system used unions, by using the explicit `StructLayout` attribute, and having the contents all have a field offset of the same amount. However, this is not actually possible, as in C# it is not actually possible to refer to numerical values like integers in the same union as object references, despite the reference being numbers.

In order to fix this issue, it was necessary to use a wrapper class in order to include numerical values inside of an object, meaning that all contents of a stack object is an object. This is incredibly unoptimized, as this means that new objects are created constantly, completely unnecessarily. The other option, to break values down into bytes and having those stored on the stack, doesn't work either, as the amount of conversions would be just as unoptimized. This is a problem that appears unsolvable without going into the use of unsafe code or simply moving away from the .NET platform entirely.

An issue that arose was how to refer to variables. As they are numerical values, it is not possible to refer to them from multiple locations, which would be necessary when loading and saving references on the stack, away from the actual location in the Resource Manager. Therefore, the wrapper class again becomes usable, as if all values are stored within a wrapper object, then it will be possible to simply refer to the wrapper, and update the value through that, as the wrapper is referred to through call by reference instead of call by value. This also creates unoptimized code, as every instance of an object now contains a large amount of sub-objects that only contains a value, wasting resources when the amount of instances increase significantly.

A notable issue that can cause problems in very specific circumstances is, that if a shorthand use of a variable is used at the very end of a code array, then it would possibly exceed that array, causing a crash. For instance, if an integer has a shorthand value of zero, it would only contain one byte, instead of the five bytes that needs to be checked. For these situations, the length of the code is checked contra position. If the position is close to the end of the code, it instead inserts zeros at the remaining positions, as the code guarantees that these won't be used in the case of a shorthand value.

Generally, throwing exception during runtime is somewhat rare, as the static checker guarantees that the the code itself is valid. However, that doesn't exclude the few situations that would indeed allow for problems that cannot be checked at compile time, stack memory exceeded, null variable accessed, division by zero, badly cast objects and attempts to access arrays outside of their actual length. These are the only situations that can cause an exception to be thrown, should the code be valid.

There is no dynamic checking to make sure that the code is valid at run-time, though, as all code is assumed to origin from the compilation using the static checker. As someone would have to essentially hack into the program in order to fault this assumption, we regard this as being solid. However, another problem is related to how variables are set, by setting the content of a wrapper in the virtual machine.

For instance in the case of a sound, pan and volume has a range of states that can be set. However, the constraints are set when setting the wrapper, not the actual value. Therefore, it is possible to set a volume to a negative value, and receive a contract error when retrieving the value at a later stage. The only solution to this, without changing the variable system, would be to perform integrity checks, which would result in a performance drop.

Testing

Unit testing is performed for the programming language, with over one thousand two hundred tests for the entire program, covering testing the parser, the static checker, the compiler, byte compiler and virtual machine.

Testing the parser is an important part, despite the parsing happening in external code. This is because of how the shift/reduce errors can easily make some written code illegal, despite what should be possible to write. Negative tests are not tested for this part, however, and neither is the result of the test. What is important if it can even parse at all, and not throw an unexpected exception based on an unforeseen combination of tokens. The testing scheme for the parser utilizes the fact that the possibilities of the language is already written beforehand, making it possible to write a program that generates test cases that gives code coverage.

The way the code is performed is, that each token that is a non-terminal has a method, where each option results in duplicating the content of a code section into multiple files, branching out the example code to make a list of code files that grants code coverage to the entire language. In order to make sure as few tests are run in order to achieve this, each time a non-terminal has been checked, a boolean is set to true. Next time that non-terminal is checked, it doesn't split up the code, but only returns the most minimal case of that non-terminal. This technique grants code coverage of the parser in 158 automatically generated test cases.

For the static checker, both positive and negative checks are used. Of course, as a Resource Manager object is required in order to check code, the Resource Manager for the editor is filled with resources. It should be noted, however, that exceptions thrown because of bad code is not checked. The difference between 'bad code' exceptions and regular exceptions is, that it should not be possible to get to a 'bad code' exception through calling the starting function with an abstract syntax retrieved from parsed code.

There are less tests for the compiler and byte compiler, as there are only positive tests for these, as exception should not be thrown. The virtual machine tester does test for both positive and negative tests, and utilized the in-game Resource Manager, similar to the static checker using the editor Resource Manager. The runtime tests are only possible due to the different instructions are split up into separate methods, calling the method initiating a for loop wouldn't give very good test cases as it would not be possible to test the state in between each individual instruction call.

Conclusion

Domain Specific Language

In order to provide scripting capabilities for the editor, a domain specific language called JourneyScript was created. This language uses a compilation pipeline, including lexing, parsing, static checking, and compiling.

For Domain Specific Languages to be used in a game editor, it is important to focus on static checking, in order to minimize the amount of errors at run-time as much as possible. It is also necessary to pay notice to what error messages are supplied to the user, so that they can be fixed as easily as possible. For the static checker used in this project, location information (line, column) was neglected, which is a significant problem, on top of the lack of information found in exceptions called by the parser. Despite locating as many problems as possible, there is still the issue that run-time, like detecting infinite loops. This reduces the reliability of the language.

Compilation for JourneyScript is handled through the use of a continuation-styled backwards compiler, which improves compilation time greatly, by avoiding list duplications. Backwards compilation was however not possible for variable declarations, as the variables has to be passed on in a forward fashion. Also, the compiler generates code that can be reduced to fewer instructions at compile time, like adding two constants instead of simply having a single constant with the added values. This is something that backwards compilation makes easy, but has not been made for the project. This should, however, be taken into account when creating a compiler.

After creating the virtual machine, it was found that creating it on top of the .NET platform incurs some optimization issues. This is due to it being implemented on a higher level of abstraction, with aspects like object references being used together with numeric values causing problems. This is a significant issue for domain specific languages, if they're implemented on the .NET platform, and lends credence to the use of a simple interpreter instead of a compilation scheme.

Editor

By far the most important factor we found for the editor is how drawing is performed. Due to the focus of using XNA for drawing in WPF, a huge efficiency loss was found. An important aspect is to properly dispose of loaded resources, along with keeping most of the resources on the hard drive when it was not actually displayed. However, using XNA in WPF causes significant problems with correctly disposing of images. This comes on top of issues with XNA, which prevents large images from being drawn. The solution to these problems seem to be to not contain a list of images to draw, and instead only have a list of references of what to draw.

Another important aspect of the editor to take into account is how windows open each other. If one wants to have multiple active windows that can all manipulate data at the same time, threading becomes a significant issue to deal with. Making sure which windows are acceptable to have active at the same time becomes a significant question that should be answered early in the design process to avoid confusion later on. As this was not taken into account in this project, only one window can be active at any time.

User created content

In regards to transferring user created content from the editor into the game, it is important to take the saved and compiled project files into account. In this project, we save and compile to a single file, where we make use of ZIP in order to create an archive. This also opens up for the use of creating content packages, where the user can export and import specific assets between projects. Such a feature would be useful for editors, though we have not implemented it for this project.

In order to save data persistently, we focus on serialization to XML, and having two separate data structures describing the data, the editor data and the game data. In order to save, the editor data is serialized, and in order to compile, the editor data is transferred into game data and serialized. In order to control the order at which objects are loaded, we focus on simplifying architecture by using ID references. ID references can then be linked to the actual objects through the Resource Manager using a binary search algorithm for efficiency. This currently break code contracts regarding the existence of objects in the Resource Manager, as the automatic serialization sets IDs that does not currently exist in the Resource Manager when loading. This can be solved by forcing deserialization in a specific manner.

Reflection

This project proved to be very time-consuming, due to the sheer complexity of a JRPG system. It was originally planned that the the game client itself should have been made, but it became very clear that this would not be possible in the given time-frame.

A major time sink in the program was the use of systematic design, especially considering that we have written 200 pages of design documentation for BON (available on the CD), task descriptions and language description alone. The large focus on systematic design is likely to have saved a lot of time early on, but further in the project redesigns became frequent, which made updating the existing documentation a significant process that simply wasted time.

While there are still many small areas that needs to be worked on if this editor was to be used with an actual game, overall it works about as well as other professional editors. Because of this we consider the project a success, as we have not only created an editor for a JRPG system with a domain specific language, but we have also found a series of design issues that can be circumvented in future designs. Despite of our misconceptions about the size of the project, and the fact that a large game would not be possible due to the memory limits in the editor, we completed what we consider a solid product within the project timeframe.

Appendix

1 - Glossary

Journey

The actual content of a JRPG, with the storyline, world, towns and dungeons. This term is used to describe the user-generated content.

Game

The executable which can open and interpret individual journeys. It can be considered to be similar to a client, except the retrieved data is not from another computer, as that is not currently possible.

Editor

The development environment where game logic is made. This includes the area where the user writes programming, as well as defining all aspects of a journey

Paradigm

A specific world-view, with a focus on tacit-knowledge in a group. It is the underlying model for how theories and practices are generally understood in scientific communities.

Sprite

An image to be used in a game, which may contain animation (multiple underlying images).

Tileset

A tileset is an image divided into a series of smaller images, where the images are used separate from each other.

Stat

A stat, or status, is the current description of a character. This includes current health and mana, along with the current value of any attributes the character has, like strength.

Mana

Mana is an expression used in role playing games to express the life force a character has needed to use magic spells. Each magic spell takes up a certain amount of mana, and if the character does not have enough of this life force, he cannot cast the spell.

NPC

NPC stands for Non Player Character. As the name implies, it's a character that is not a player.

Domain Specific Language

A Language that is not a general purpose language, as it can only be used in a specific domain. It is written specifically for one domain, and multiple DSLs may exist in a program for different sub-domains.

Abstract Syntax Tree

A programming language can be seen as a tree structure, with the first node of the tree consisting of the main execution and the end of file, and the leaf nodes representing operants, constants, or variables.

LALR parser

Look-Ahead Left-Right Parser, scans input left-to-right in a single scan without guessing.

Terminal / Non-Terminal

In parser specifications, terminals are translated into parts of an abstract syntax tree, while non-terminals do not, instead offering options for parsing. In BNF, a non-terminal always appears at least once on the left side of the translation, while terminals are never on the left side.

Event-Driven Programming

A language paradigm, in which the execution flow is determined by events, like mouse clicks.

Compilation Pipeline

The process of translating a string to executable bytecode.

Virtual Windows

Virtual windows is a series of windows created in order to present data in a system, and for designing what goes in what windows, how the data is graphically displayed, and similar.

Single & Multi-page dialogues

A single page dialogue is a system where one window is active at any given time, and navigation is provided for activating different windows. Multi-page dialogues, on the other hand, allows for multiple windows (though possibly independent elements inside of one window) to be active at the same time.

State Diagram

A state diagram is a way of describing any finite state machine, like the parser or window navigation. The state diagram shows the different states of a system, as well as the different ways of moving between the different states.

Shift/Reduce Error

When navigating a state diagram in the parser, shift/reduce errors can arise when ambiguity arises due to multiple paths being possible from a given state. More specifically, a shift/reduce error occurs when it is both possible to reduce the stack to a terminal and to continue with the state-diagram by shifting the active token to the next position in the list of lexer tokens. For instance, if one has a non-terminal with "IF expr THEN stmt" and "IF expr THEN stmt ELSE stmt", then there will be a shift/reduce error at ELSE due to both these routes in the state diagram may be used.

Lexer Token

The job of the lexer is to consolidate string input into a series of tokens. Tokens may be just a name that may be referred to, or it may contain data. For instance, a "NAME" token would likely contain a string.

Continuation Passing Style

CPS is a style of functional programming, where a continuation function is supplied as a monad to each function, and the function ends by calling the monad with the result of the function. Therefore, calling a function using continuation passing style means not only calling a method, but also supplying what to do with the result after completion.

Singleton / Multiton Design Pattern

The singleton design pattern focuses on making sure that only one instance of a class exists, and that this class can be accessed globally. Multiton is similar, except that instead of guaranteeing that only one instance exists, it guarantees that a set number of elements exists, where access to each element can be found by using a key.

Monads

A Monad is a structure used in functional programming to represent programming logic. This can be used, for instance, to describe continuations, or to contain a state.

Discrete Event Simulation

Discrete Event Simulation, DES, is a way of describing a system by a sequence of events in a chronological order.

2 - Bibliography

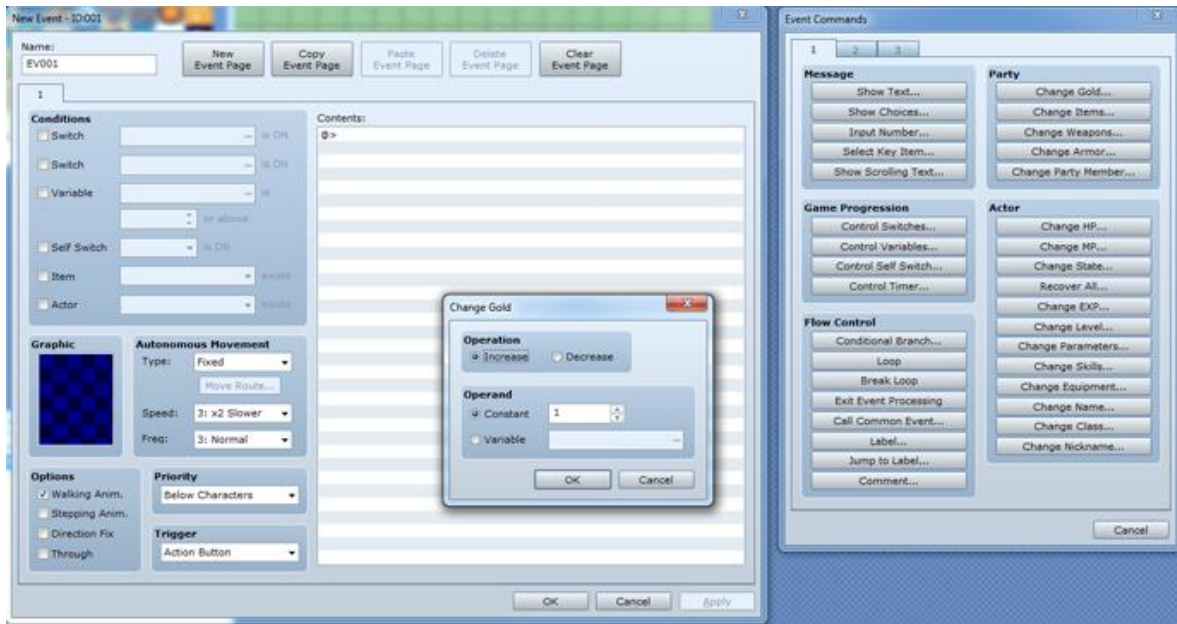
- [1] - [http://msdn.microsoft.com/library/dd233181\(VS.100\).aspx](http://msdn.microsoft.com/library/dd233181(VS.100).aspx)
- [2] - <http://fsharpowerpack.codeplex.com/>
- [3] - <http://www.microsoft.com/visualstudio/>
- [4] - <http://www.nunit.org/>
- [5] - <http://fsunit.codeplex.com/>
- [6] - <http://www.w3.org/XML/>
- [7] - http://en.wikipedia.org/wiki/Microsoft_XNA
- [8] - User Interface Design - A Software Engineering Perspective - Soren Lauesen, Pearson Education Limited, 2005. ISBN: 0-321-18143-3
- [9] - <http://msdn.microsoft.com/en-us/library/ms754130.aspx>
- [10] - http://www.bon-method.com/index_normal.htm
- [11] - <http://www.fmod.org/>
- [12] - <http://research.microsoft.com/en-us/projects/contracts/>
- [13] - <http://www.codeproject.com/Articles/42490/Using-AvalonEdit-WPF-Text-Editor>
- [14] - <http://www.codeproject.com/Articles/42849/Making-a-Drop-Down-Style-Custom-Color-Picker-in-WP>
- [15] -
Code Metrics Power tool:
<http://www.microsoft.com/en-us/download/details.aspx?id=9422>
Code Metrics Viewer: <http://visualstudiogallery.msdn.microsoft.com/9f35524b-a784-4dbc-bd7b-6babd7a5a3b3>
- [16] - <http://www.garshol.priv.no/download/text/bnf.html>
- [17] - http://en.wikipedia.org/wiki/Entity-relationship_model
- [18] - [http://en.wikipedia.org/wiki/Zip_\(file_format\)](http://en.wikipedia.org/wiki/Zip_(file_format))
- [19] - <http://www.square-enix.com/na/title/finalfantasy/>
- [20] - <http://www.square-enix.com/na/title/dragonquest/>
- [21] - http://en.wikipedia.org/wiki/Breath_of_Fire
- [22] - <http://tvtropes.org/pmwiki/pmwiki.php/Main/BrokenBridge>
- [23] - http://wiki.yoyogames.com/index.php/Game_Maker
- [24] - http://wiki.yoyogames.com/index.php/Documentation:The_Game_Maker_Language
- [25] - <http://www.gmtoolbox.com/>
- [26] - <http://www.rpgmakerweb.com/>
- [27] - <http://www.ruby-lang.org/en/>
- [28] - <http://www.embarcadero.com/products/delphi>
- [29] - <http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt>
- [30] - <http://swierl.tudelft.nl/twiki/pub/Main/PastAndCurrentMScProjects/JeroenDobbe.pdf>
- [31] - <http://blogs.msdn.com/b/nicgrave/archive/2010/07/25/rendering-with-xna-framework-4-0-inside-of-a-wpf-application.aspx>
- [32] - http://en.wikipedia.org/wiki/Model_View_ViewModel
- [33] - <http://en.wikipedia.org/wiki/Simula>
- [33] - <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>

3 - Editor Images

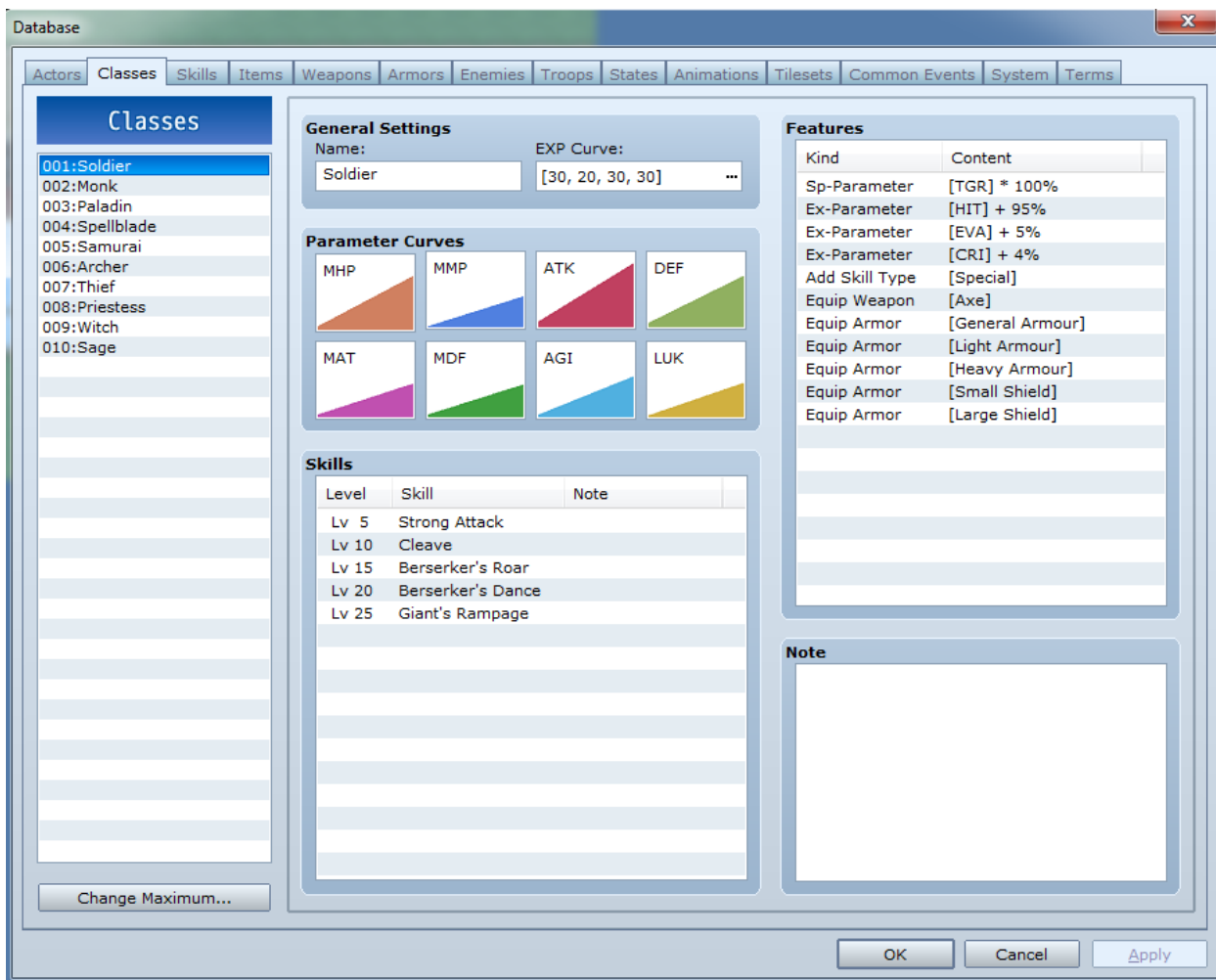
RPG Maker

```
Contents:
@>Control Variables: [0001] = 0
@>Loop
  @>Conditional Branch: Variable [0001] == 10
    @>Break Loop
    @>
  : Else
    @>Change Gold: - 1
    @>Control Variables: [0001] += 1
    @>
  : Branch End
  @>
: Repeat Above
@>
```

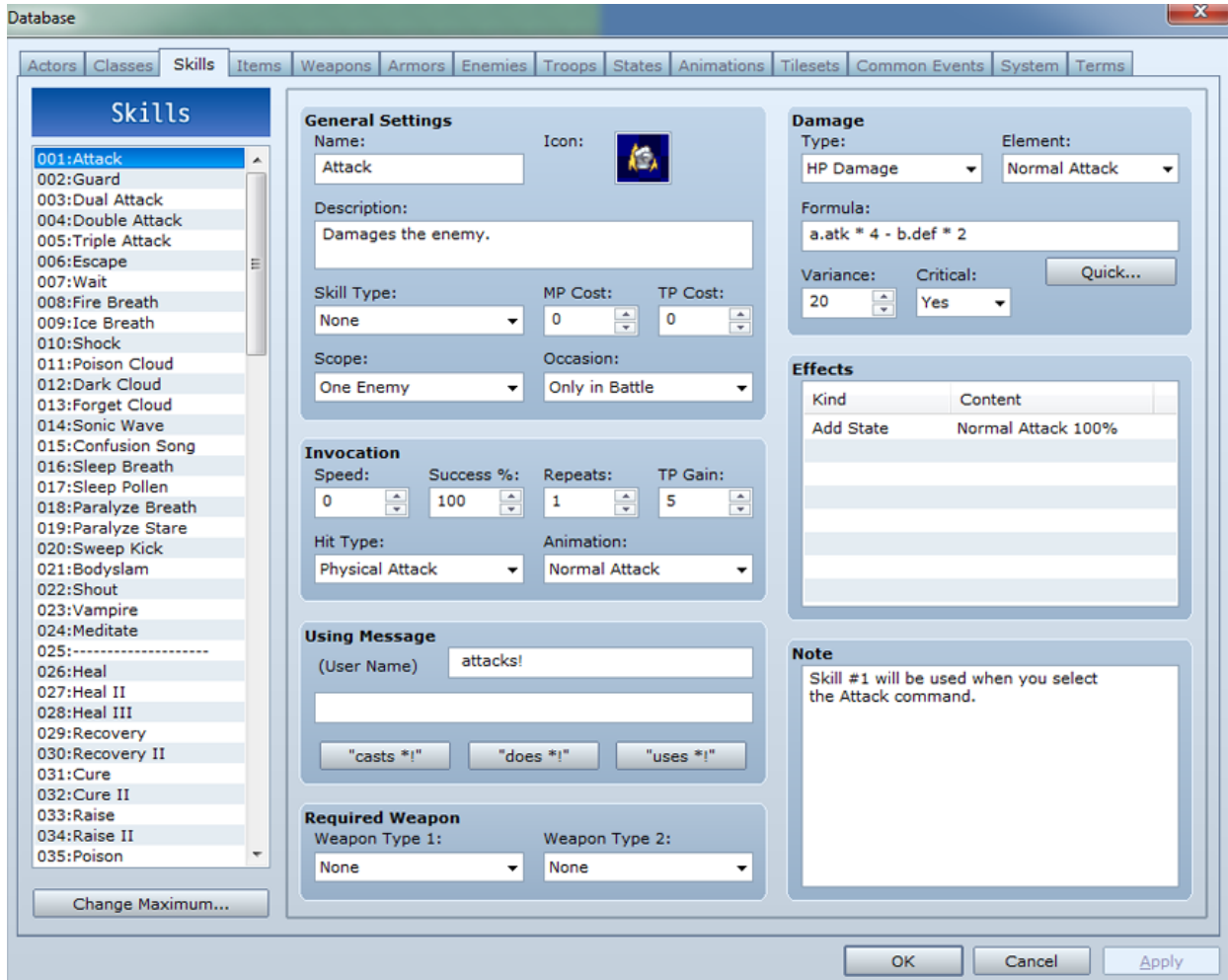
(The programming language of RPG Maker)



(The event system in RPG Maker, with commands)



(The class system in RPG Maker)



(The skill system used in RPG Maker)

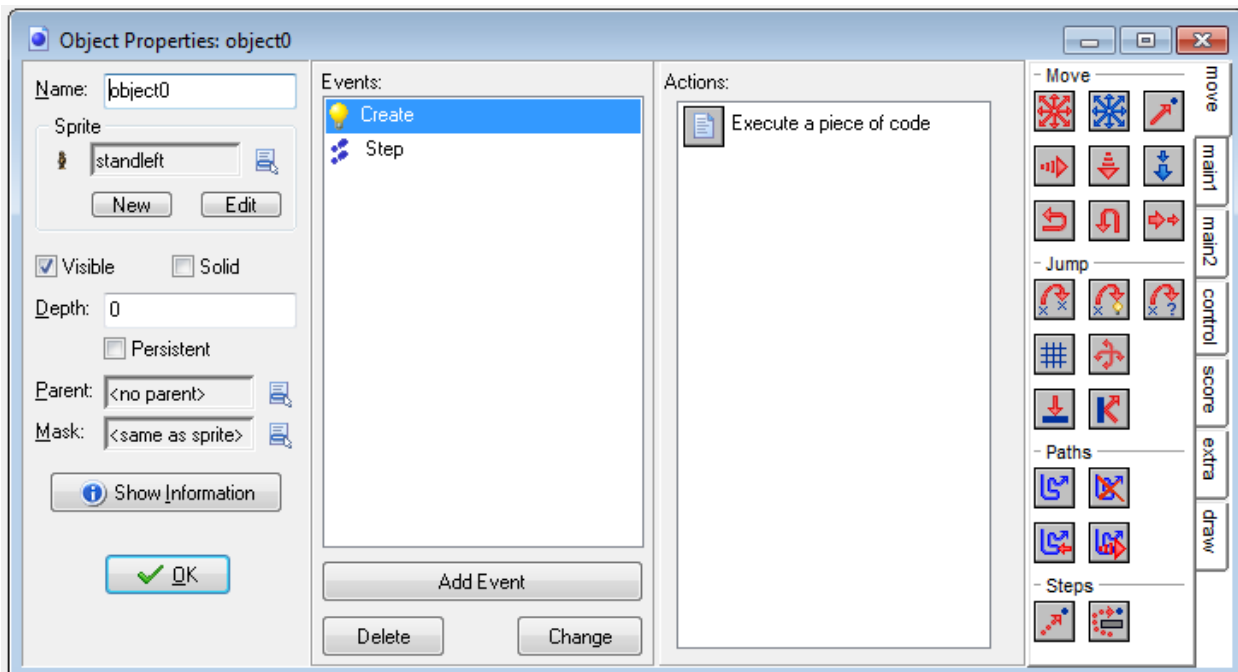
Game Maker

The image shows a screenshot of the Game Maker code editor. At the top, there is a toolbar with various icons for editing and a text input field labeled 'Name: script0'. Below the toolbar is a code editor window containing the following code:

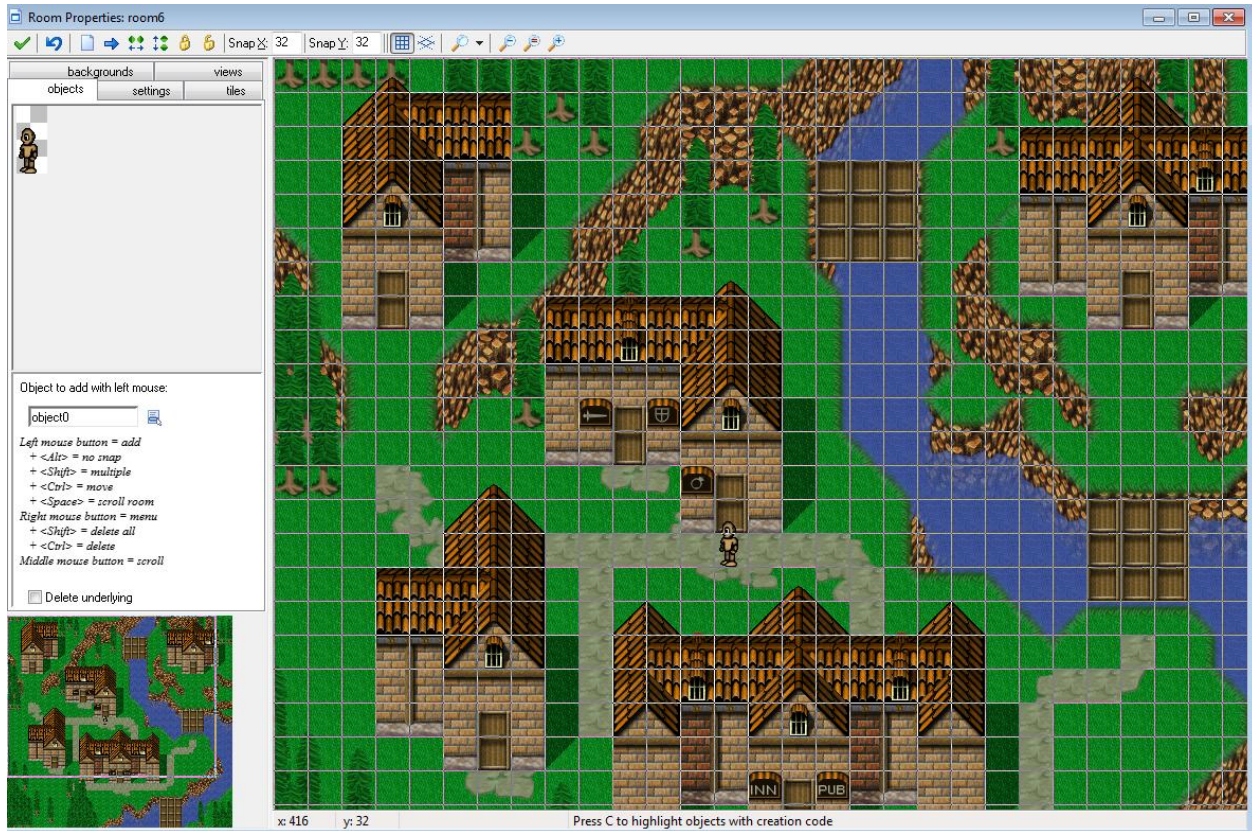
```
1 var gold,i;  
2 i = 0;  
3 gold = arg0;  
4 while (i < 10)  
5 {  
6 gold -= 1;  
7 i += 1;  
8 }
```

The code is highlighted in yellow.

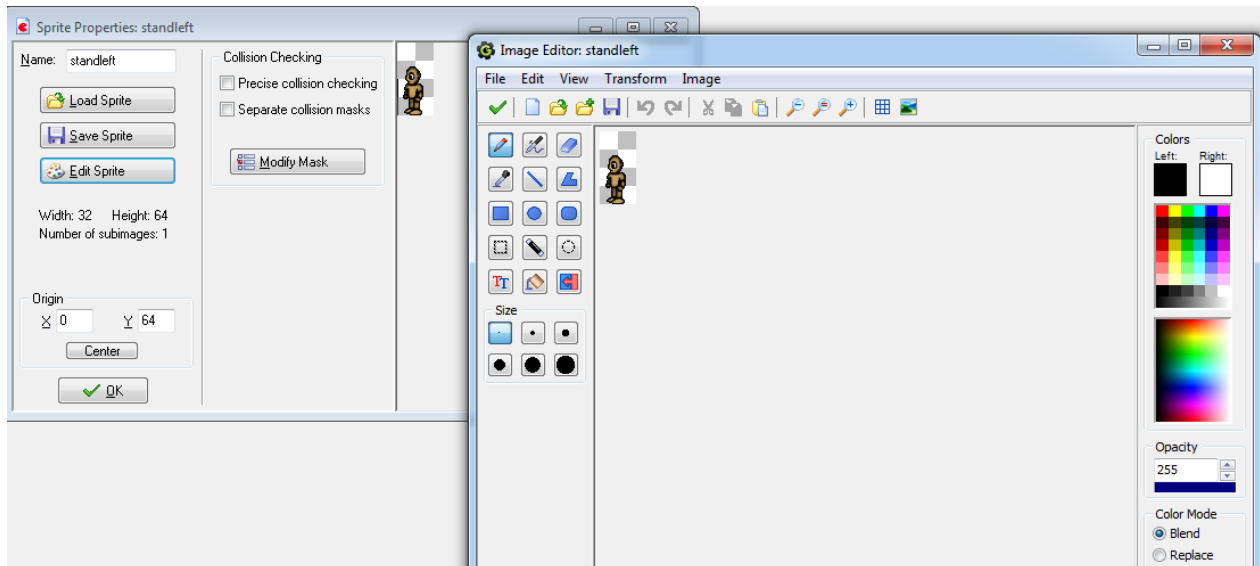
(Example of a code section in Game Maker)



(Object screen in Game Maker with events and Drag & Drop)



(The Room (Area) screen in Game Maker)



(The included image editor in Game Maker)

4 - Task Descriptions

1. Journey Editor

T1.1: Manage Journey
T1.1: Manage Resources
T1.2: Manage Image
T1.3: Manage Sprite
T1.4: Manage Sound
T1.5: Manage Function
T1.6: Manage Enum

2. Profession Editor

T2.1: Manage Profession
T2.2: Manage Equipment

3. Companion Editor

T3.1: Manage Companion

4. Ability Editor

T4.1: Manage Ability
T4.2: Manage Effect
T4.3: Manage Condition
T4.4: Manage Ability Animation

5. Item Editor

T5.1: Manage Item
T5.2: Manage accessory

6. Troop Editor

T6.1: Manage Troops
T6.2: Manage Monster
T6.3: Manage Drops

7. Area Editor

T7.1: Manage Area
T7.2: Manage Background

8. Object Editor

T8.1: Manage Class
T8.2: Manage Zone
T8.3: Manage Combat Background
T8.4: Manage Event
T8.5: Manage Variable

9. Storyline editor

T9.1: Manage Storyline
T9.2: Manage Quest Item

10. NPC editor

T10.1: Manage NPC
T10.2: Manage Speech
T10.3: Manage Shop

T1.1: Manage Journey

Start: User opens editor

End: User closes editor

Frequency: Very often

Subtasks:	Example Solution:
1. Open journey (<i>optional</i>) a. Open new empty journey b. Open previous journey	Either opens a new empty journey, or if previously journey has been saved, opens the latest worked on journey
12. Save journey (<i>optional</i>)	Menu option with possible shortcut on main toolbar that saves the journey
13. Load journey (<i>optional</i>)	Menu option with possible shortcut on main toolbar that opens the file system to find a journey to open and replace current journey
13. Compile journey (<i>optional</i>) a. Compile journey successfully b. Compile journey with errors	Menu option with possible shortcut on main toolbar. Any errors during compilation should be gathered in a list of all outstanding issues
14. Manage errors and warnings (<i>optional</i>)	Subpage with list of all found problems.

T1.2: Manage Resources

Start: User opens editor

End: User closes editor

Frequency: Very often

Subtasks:	Example Solution:
1. Select tab <i>(optional)</i> a. Select Object tab b. Select Resource tab c. Select Profession tab d. Select Ability tab e. Select Story tab f. Select Troop tab . h. Select item tab	7 tabs, each with a subset of tree lists - the object tab has classes, zones, functions and enums, the resources tab has sprites, images and sounds, the profession tab has professions and companions, the ability tab has abilities, the story tab has story, npc, shop and speeches, the troop tab has troops and monsters, and the item tab has equipment, accessories, items and quest items
2. Add new tree list element <i>(optional)</i> a. Add new resource b. Add new folder	Right click add on tree list. Adds resource on selected tree with a standardized name
3. Remove tree list element <i>(optional)</i> a. Remove resource b. Remove folder	Right click remove on tree list. Removes resource or folder - if folder is selected, removes all inside of the folder
4. Rename tree list element <i>(optional)</i> a. Rename resource b. Rename folder	Double click to rename element.
5. Manage tree list element <i>(optional)</i>	Double click on element in tree list to open the respective editor
6. Move tree list element <i>(optional)</i>	Click and drag element in tree list to move the element

T1.3: Manage Image

Start: User opens image editor

End: User closes image editor

Frequency: Often

Subtasks:	Example Solution:
1. Find image (<i>optional</i>)	Button opens file system search for valid image file
2. Set image tileset status (<i>optional</i>)	Checkbox if image is tileset, and text fields for horizontal and vertical tile offset, spacing and tile sizes
3. Close image editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current image

T1.4: Manage Sprite

Start: User opens sprite editor

End: User closes sprite editor

Frequency: Often

Subtasks:	Example Solution:
1. Find image (<i>optional</i>)	Button opens file system search for valid image file, and adds it to image list
2. Remove image (<i>optional</i>)	Button removes selected image
3. Move image (<i>optional</i>)	Buttons for moving image in list of images
4. Set image center position (<i>optional</i>)	Normal text fields, with coordinates for the center of the selected image
5. Set collision box (<i>optional</i>)	Normal text fields for x position, y position, width and height of the collision box
6. Close sprite editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current sprite

T1.5: Manage Sound

Start: User opens sound editor

End: User closes sound editor

Frequency: Often

Subtasks:	Example Solution:
1. Find sound (<i>optional</i>)	Button opens file system search for valid sound file
2. Play sound (<i>optional</i>)	Button, visible if sound is not playing
3. Pause sound (<i>optional</i>)	Button, visible if sound is playing
4. Stop Sound (<i>optional</i>)	Button
5. Set loops (<i>optional</i>)	Checkbox
6. Set volume (<i>optional</i>)	Adjustable bar, with text indicator
7. Set panning (<i>optional</i>)	Adjustable bar, with text indicator
8. Close sound editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current sound

T1.6: Manage Function

Start: User opens function editor

End: User closes function editor

Frequency: Often

Subtasks:	Example Solution:
1. Set code (<i>optional</i>)	Large text field with keyword highlighting
2. Close function editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current function

T1.7: Manage Enum

Start: User opens enum editor

End: User closes enum editor

Frequency: Often

Subtasks:	Example Solution:
1. Create new enum instance <i>(optional)</i>	Button that adds a new entry into list of enum instances, with a number increased by one, and with a temporary name
2. Set enum instance name <i>(optional)</i>	Double clicking on the list of enum instances opens up a prompt where you can change the name of the enum instance
3. Remove enum instance <i>(optional)</i>	Button that removed currently selected enum instance on a list, updating the numbers of the later enum instances on the list
4. Close enum editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current enum

T2.1: Manage Profession

Start: User opens profession Editor

End: User closes profession Editor

Frequency: In the region of 1-6 times per Journey

Subtasks:	Example Solution:
1. Set profession name <i>(optional)</i>	Normal text field
2. Set health-mana scaling levels <i>(optional)</i>	Two adjustable bars for health and mana that adjusts each other to end on set overall value
3. Set power scaling levels <i>(optional)</i>	Multiple adjustable bars for strength, intelligence, stamina and agility, that adjust each other to end on set overall value
4. Add ability <i>(optional)</i>	Button that opens up a hierarchical view of the sprites for selection, and adds it to the ability list
5. Edit ability <i>(optional)</i>	Double click on list to edit selected ability in ability editor
6. Remove ability <i>(optional)</i>	Button to remove selected ability from list
7. Add equipment <i>(optional)</i>	Button that opens up a hierarchical view of the equipment for selection, along with a "none" option
8. Edit equipment <i>(optional)</i>	Double click on list to edit selected equipment in equipment editor
9. Remove equipment <i>(optional)</i>	Button that removes selected item from list
10. Add sprites <i>(optional)</i>	A drop down of required character animations, and their associated sprites shown when animation is selected. Button that opens up a hierarchical view of the sprites for selection, along with a "none" option.
11. Edit sprite <i>(optional)</i>	Double click on list to edit selected sprite in sprite editor
12. Remove sprite <i>(optional)</i>	Button that removes selected sprite from list
13. Close profession editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current profession

T2.2: Manage Equipment

Start: User opens equipment editor

End: User closes equipment editor

Frequency: Often

Subtasks:	Example Solution:
1. Set equipment type <i>(optional)</i>	Radio buttons with the 3 equipment types
2. Set power balancing levels <i>(optional)</i>	Multiple adjustable bars that adjust each other to end on set overall value
3. Create new equipment instance <i>(optional)</i>	Button that adds a new entry into the equipment name list with temporary content
4. Set equipment instance level <i>(optional)</i>	Normal text field for selected instance on list
5. Set equipment instance name <i>(optional)</i>	Normal text field for selected instance on list
6. Set instance description <i>(optional)</i>	Normal text field for selected instance on list
7. Remove equipment instance <i>(optional)</i>	Button that removes currently selected instance on list.
8. Close equipment editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current equipment

T3.1: Manage Companion

Start: User opens companion Editor

End: User closes companion Editor

Frequency: In the region of 1-5 times per Journey

Subtasks:	Example Solution:
1. Set name <i>(optional)</i>	Normal text field
2. Set profession <i>(optional)</i>	Button that opens up a hierarchical view of the professions for selection, along with a "none" option
3. Edit profession <i>(optional)</i>	Double click on profession to edit it in profession editor
4. Add ability <i>(optional)</i>	Button that opens up a hierarchical view of the ability for selection, and adds it to the ability list
5. Edit ability <i>(optional)</i>	Double click on ability on list to edit it in ability editor
6. Remove ability <i>(optional)</i>	Button that removes selected ability from list
7. Set ability learning level <i>(optional)</i>	Normal text field for selected ability on list
8. Close companion editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current companion

T4.1: Manage Ability

Start: User opens ability Editor

End: User closes ability Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set ability name (<i>optional</i>)	Normal text field
2. Set damage/heal amount (<i>optional</i>)	Normal text field
3. Set damage type (<i>optional</i>)	Drop down menu with limited types, including physical, pure magic, fire damage and heal.
4. Set ability mana cost (<i>optional</i>)	Normal text field
5. Set hit rate (<i>optional</i>)	Normal text field
6. Set strikes (<i>optional</i>)	Normal text field
7. Set target types (<i>optional</i>)	Drop down menu with limited target types
8. Set ability cast time (<i>optional</i>)	Normal text field
9. Add effect (<i>optional</i>)	Button that adds a new empty effect to the list
10. Edit effect (<i>optional</i>)	Double click on effect on list to edit it in effect editor
11. Remove effect (<i>optional</i>)	Button that removes selected effect from list
12. Add condition (<i>optional</i>)	Button that adds a new empty condition to the list
13. Edit condition (<i>optional</i>)	Double click on condition on list to edit it in condition editor
14. Remove condition (<i>optional</i>)	Button that removes condition from list
15. Add ability animation (<i>optional</i>)	Button that adds a new empty animation to the list
16. Edit ability animation (<i>optional</i>)	Double click on ability animation to edit it in ability animation editor
17. Remove ability animation (<i>optional</i>)	Button that removes animation from list
18. Close ability editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current ability

T4.2: Manage Effect

Start: User opens effect editor

End: User closes effect editor

Frequency: Often

Subtasks:	Example Solution:
1. Set effect type <i>(optional)</i>	Drop down menu with limited effect types
1. Set effect value <i>(optional)</i> a. Choose desired numerical value b. Choose no value	If the given effect type requires a numerical value, a normal text field will be used
4. Set effect duration <i>(optional)</i>	Drop down menu with limited round numbers
5. Set target types <i>(optional)</i>	Drop down menu with limited target types
6. Add applied effect <i>(optional)</i>	Button that adds a new empty effect to the list
7. Edit applied effect <i>(optional)</i>	Double click on applied effect to edit it in a new effect editor
8. Remove applied effect <i>(optional)</i>	Button that removes the selected effect
9. Add required condition <i>(optional)</i>	Button that adds a new empty condition to the list
10. Edit required condition <i>(optional)</i>	Double click on condition to edit it in condition editor
11. Remove required condition <i>(optional)</i>	Button that removes the selected condition
12. Add expires condition <i>(optional)</i>	Button that adds a new empty condition to the list
13. Edit expires condition <i>(optional)</i>	Double click on condition to edit it in condition editor
14. Remove expires condition <i>(optional)</i>	Button that removes the selected condition
17. Close effect editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current effect

T4.3: Manage Condition

Start: User opens condition editor
 End: User closes condition editor
 Frequency: Often
 Difficult: None

Subtasks:	Example Solution:
1. Set condition type <i>(optional)</i>	Drop down menu with limited types
2. Set condition value <i>(optional)</i> a. Choose selected ability to have used b. Choose desired numerical value c. Choose no value	Different setup for each condition type. For conditions requiring no value, no additional work required. For conditions with a number value, a normal text field is used. For checking ability used, selection is done in a hierarchical view
3. Close effect editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current condition

T4.4: Manage Ability Animation

Start: User opens ability editor
 End: User closes ability editor
 Frequency: Often
 Difficult: None

Subtasks:	Example Solution:
1. Set start and end frames <i>(optional)</i>	Normal text fields with scroll wheel adaptor
2. Set sprite <i>(optional)</i>	Button that opens up a hierarchical view of the sprites for selection, along with a "none" option
3. Set movement type <i>(optional)</i>	Drop down menu of limited types
4. Set animation speed <i>(optional)</i>	Normal text field
5. Set animation from position <i>(optional)</i>	Normal text fields and drop down with coordinate type
6. Set animation to position <i>(optional)</i>	Normal text fields and drop down with coordinate type
7. Close ability animation editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current ability animation

T5.1: Manage Item

Start: User opens item Editor

End: User closes item Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set item name (<i>optional</i>)	Normal text field
2. Set item description (<i>optional</i>)	Normal text field
3. Set chosen ability (<i>optional</i>)	Button that opens up a hierarchical view of the abilities for selection, along with a "none" option
4. Edit ability (<i>optional</i>)	Double click on ability to edit it in ability editor
5. Close item editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current item

T5.2: Manage Accessory

Start: User opens accessory Editor

End: User closes accessory Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set accessory name <i>(optional)</i>	Normal text field
2. Set accessory description <i>(optional)</i>	Normal text field
3. Add ability <i>(optional)</i>	Button that opens up a hierarchical view of the abilities for selection, and adds it to the ability list
4. Edit ability <i>(optional)</i>	Double click on ability to edit it in ability editor
5. Remove ability <i>(optional)</i>	Button to remove selected ability on list
6. Set accessory type <i>(optional)</i>	Drop down with limited accessory types
7. Set accessory value <i>(optional)</i>	Normal text field used by certain types
8. Set buff type <i>(optional)</i>	Drop down with limited buff types, visible if accessory type is buff
9. Set status effect type <i>(optional)</i>	Drop down with limited status effect types, visible if accessory type is status effect resistance or immunity.
10. Set magic element type <i>(optional)</i>	Drop down with limited magic types, visible if accessory type is magic resistance
6. Close accessory editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current accessory

T6.1: Manage Troop

Start: User opens troop Editor

End: User closes troop Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set monster <i>(optional)</i>	6 radio buttons positioned as they would in battle, with the name or image of the selected monster next to it. Button that opens up a hierarchical view of the monster for selection, along with a "none" option
2. Edit monster <i>(optional)</i>	Double click on monster name to edit monster in the monster editor
3. Set troop rate <i>(optional)</i>	Slider
4. Set background music <i>(optional)</i>	Button that opens up a hierarchical view of the sounds for selection, along with a "none" option
5. Close troop editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current troop

T6.2: Manage Monster

Start: User opens monster Editor

End: User closes monster Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set monster name <i>(optional)</i>	Normal text field
2. Set monster type <i>(optional)</i>	Radio buttons with the 4 monster types
3. Set power levels <i>(optional)</i>	Multiple adjustable bars for health, attack, defense and magic values that updates the suggested monster level based on the values
4. Set status effect immunities <i>(optional)</i>	List of status effects to apply with checkboxes for each effect
5. Set magic type affinity <i>(optional)</i>	A list of all magic types, each with radio buttons for immune, absorb, resistant, neutral and weakness
6. Add ability <i>(optional)</i>	Button that opens up a hierarchical view of the abilities for selection, and adds it to the ability list
7. Edit ability <i>(optional)</i>	Double click on ability to edit it in ability editor
8. Remove ability <i>(optional)</i>	Removes selected ability from list
9. Add drop <i>(optional)</i>	Button that adds an empty drop to the list
10. Edit drop <i>(optional)</i>	Double click on drop to edit it in drop editor
11. Remove drop <i>(optional)</i>	Removes selected drop from list
12. Close monster editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current monster

T6.3: Manage Drop

Start: User opens drop Editor

End: User closes drop Editor

Frequency: Often

Subtasks:	Example Solution:
1. Set drop or steal (<i>optional</i>)	Radio buttons with drop and steal
2. Set amount (<i>optional</i>)	Normal text field
2. Set dropped item type (<i>optional</i>)	Drop down with limited item types
3. Set drop chance (<i>optional</i>)	Normal text field
4. Set dropped item (<i>optional</i>)	Button that opens up a hierarchical view of the items for selection, if item is selected
5. Set dropped accessory (<i>optional</i>)	Button that opens up a hierarchical view of the accessories for selection, if accessory is selected
6. Add dropped quest item (<i>optional</i>)	Button that opens up a hierarchical view of the quest items for selection, if quest item is selected
7. Close drop editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current drop

T7.1: Manage Area

Start: User opens editor

End: User closes editor

Frequency: Very Often

Subtasks:	Example Solution:
1. Set area name <i>(optional)</i>	Normal text field
2. Set area size <i>(optional)</i>	Normal text field
3. Set current depth value <i>(optional)</i>	Normal text field, and drop down of used ones
4. Set grid size <i>(optional)</i>	Normal text fields
5. Set remove below <i>(optional)</i>	Checkbox
6. Set displayed options <i>(optional)</i>	Checkboxes for zones, objects, and others
7. Set background music <i>(optional)</i>	Button that opens up a hierarchical view of the song folder setup for selection, along with a "none" option
8. Add background <i>(optional)</i>	Button that adds a new background to the list
9. Edit background <i>(optional)</i>	Double click on background to edit it in background editor
10. Remove background <i>(optional)</i>	Button that removes selected background
11. Select tileset <i>(optional)</i>	Button that opens up a hierarchical view of the image folder setup for selection, along with a "none" option
12. Select tile <i>(optional)</i>	Clicking on displayed tileset selects specific tile below mouse cursor
13. Set tile <i>(optional)</i>	Left mouse click adds selected tile on to area at depth, and removes any tiles with right click
14. Select class <i>(optional)</i>	Button that opens up a hierarchical view of the class folder setup for selection
15. Set instance <i>(optional)</i>	Left mouse click adds selected tile on to area, and removes any instaces with right click
16. Select zone <i>(optional)</i>	Button that opens up a hierarchical view of the zone folder setup for selection
17. Set zone position <i>(optional)</i>	Left mouse down drags a zone to area, and removes any zone instances with right click

T7.2: Manage Background

Start: User opens background editor

End: User closes background editor

Frequency: Often

Subtasks:	Example Solution:
1. Set image <i>(optional)</i>	Button that opens up a hierarchical view of the image folder setup for selection, along with a "none" option
2. Set background depth <i>(optional)</i>	Normal text field
3. Set repeating background <i>(optional)</i>	Two checkboxes, one for horizontal repeating, one for vertical
4. Set background speed <i>(optional)</i>	Normal text fields
5. Set position <i>(optional)</i>	Normal text fields
6. Close background editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current background

T8.1: Manage Class

Start: User opens class editor

End: User closes class editor

Frequency: Often

Subtasks:	Example Solution:
1. Set start sprite <i>(optional)</i>	Button that opens up a hierarchical view of the sprite folder setup for selection, along with a "none" option
2. Set parent class <i>(optional)</i>	Button that opens up a hierarchical view of the class folder setup for selection, along with a "none" option
3. Set start depth <i>(optional)</i>	Normal text field
4. Set if solid <i>(optional)</i>	Checkbox
4. Create new object variable <i>(optional)</i>	Button that adds new variable with temporary info into list
5. Edit object variable <i>(optional)</i>	Double click variable on list to edit it in variable editor
6. Remove object variable <i>(optional)</i>	Button that removes selected object variable from list
7. Set NPC <i>(optional)</i>	Button that opens up a hierarchical view of the NPC folder setup for selection, along with a "none" option
8. Edit NPC <i>(optional)</i>	Double click on NPC to edit it in NPC editor
9. Add event <i>(optional)</i>	Button that opens a prompt of event type. Selecting custom event prompts event name, and selecting collision event prompts a hierarchical view of the class folder setup for selection.
10. Edit event <i>(optional)</i>	Double click on event to edit it in event editor
11. Remove event <i>(optional)</i>	Button for removing selected event from list
12. Close class editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current class

T8.2: Manage Zone
 Start: User opens zone editor
 End: User closes zone editor
 Frequency: Often

Subtasks:	Example Solution:
1. Set zone having depth <i>(optional)</i>	Checkbox
2. Set troop <i>(optional)</i>	Button that adds an empty troop to the list
3. Edit troop <i>(optional)</i>	Double click troop on list to edit in troop editor
4. Remove troop <i>(optional)</i>	Button that removes selected troop from list
5. Set combat background <i>(optional)</i>	Button that adds an empty combat background to the list
6. Edit combat background <i>(optional)</i>	Double click background on list to edit it in combat background editor
7. Remove combat background <i>(optional)</i>	Button that removes selected background from list
8. Set terrain <i>(optional)</i>	Drop down menu with limited terrain types
9. Set combat rate <i>(optional)</i>	Slider
10. Set color <i>(optional)</i>	Color picking tool
11. Create new zone variable <i>(optional)</i>	Button that adds new variable with temporary info into list
12. Edit zone variable <i>(optional)</i>	Double click variable on list to edit it in variable editor
13. Remove zone variable <i>(optional)</i>	Button that removes selected zone variable from list
14. Add event <i>(optional)</i>	Button that opens a prompt of event type. Selecting custom event prompts event name, and selecting collision event prompts a hierarchical view of the class folder setup for selection.
15. Edit event <i>(optional)</i>	Double click on event to edit it in event editor
16. Remove event <i>(optional)</i>	Button for removing selected event from list
17. Close zone editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current zone

T8.3: Manage Combat Background

Start: User opens combat background editor
 End: User closes combat background editor
 Frequency: Often

Subtasks:	Example Solution:
1. Set image <i>(optional)</i>	Button that opens up a hierarchical view of the image folder setup for selection, along with a "none" option
2. Set background depth <i>(optional)</i>	Normal text field
4. Set repeating background <i>(optional)</i>	Two checkboxes, one for horizontal repeating, one for vertical
5. Set background speed <i>(optional)</i>	Normal text fields
6. Set position <i>(optional)</i>	Normal text fields
7. Close combat background editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current combat background

T8.4: Manage Event

Start: User opens event editor
 End: User closes event editor
 Frequency: Very Often

Subtasks:	Example Solution:
1. Set code <i>(optional)</i>	Large text field with keyword highlighting
5. Close event editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current event

T8.5: Manage Variable

Start: User opens variable editor

End: User closes variable editor

Frequency: Often

Subtasks:	Example Solution:
10. Set variable type (<i>optional</i>)	Drop down box
11. Set variable name (<i>optional</i>)	Normal text field
12. Set array type and length (<i>optional</i>)	Check boxes and normal text fields
13. Set default value (<i>optional</i>)	Normal text field
17. Close variable editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current variable

T9.1: Manage Storyline

Start: User opens storyline editor

End: User closes storyline editor

Frequency: Often

Subtasks:	Example Solution:
1. Add state <i>(optional)</i>	Button that adds a new state to the storyline, increasing the state value by one
2. Add state progress <i>(optional)</i>	Button that adds a new progress to the selected state
2. Add progressed storyline state <i>(optional)</i>	Button that opens up a prompt, where storyline and new state can be selected
3, Remove progressed story state <i>(optional)</i>	Button that removes selected progressed storyline state from the progresses list
3. Set progresses NPC <i>(optional)</i>	Button that opens up a hierarchical view of the NPC folder setup for selection, along with a "none" option
4. Edit progresses NPC <i>(optional)</i>	Double click on NPC to edit it in NPC editor
5. Set progresses required amount <i>(optional)</i>	Normal text field
6. Set progresses removes items <i>(optional)</i>	Checkbox
7. Set progresses Quest Item <i>(optional)</i>	Button that opens up a hierarchical view of the quest item folder setup for selection, along with a "none" option
8. Edit progresses Quest Item <i>(optional)</i>	Double click on Quest Item to edit it in Quest Item editor
9. Set progresses Speech <i>(optional)</i>	Button that opens up a hierarchical view of the speech folder setup for selection
4. Edit progresses Speech <i>(optional)</i>	Double click on NPC to edit it in Speech editor
9. Remove progress <i>(optional)</i>	Button that removes selected progress from selected state
9. Remove state <i>(optional)</i>	Removes state selected on list, changing the values of the following states on the list
10. Close storyline editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the current storyline

T9.2: Manage Quest Item

Start: User opens quest item editor

End: User closes quest item editor

Frequency: Often

Subtasks:	Example Solution:
1. Set quest item name (<i>optional</i>)	Normal text field
2. Set quest item description (<i>optional</i>)	Normal text field
3. Close quest item editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the quest item

T10.1: Manage NPC

Start: User opens NPC editor

End: User closes NPC editor

Frequency: Often

Subtasks:	Example Solution:
1. Set NPC name (<i>optional</i>)	Normal text field
2. Set NPC role (<i>optional</i>)	Drop down of limited NPC role
3. Set healing price (<i>optional</i>)	Normal text field - only available if NPC type is healer
4. Set storyline (<i>optional</i>)	Button that opens up a hierarchical view of the storyline folder setup for selection, along with a "none" option
5. Set shop (<i>optional</i>)	Button that opens up a hierarchical view of the shop folder setup for selection, along with a "none" option - only available if NPC type is shop
6. Close quest NPC editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the NPC

T10.2: Manage Speech

Start: User opens speech editor
End: User closes speech editor
Frequency: Often

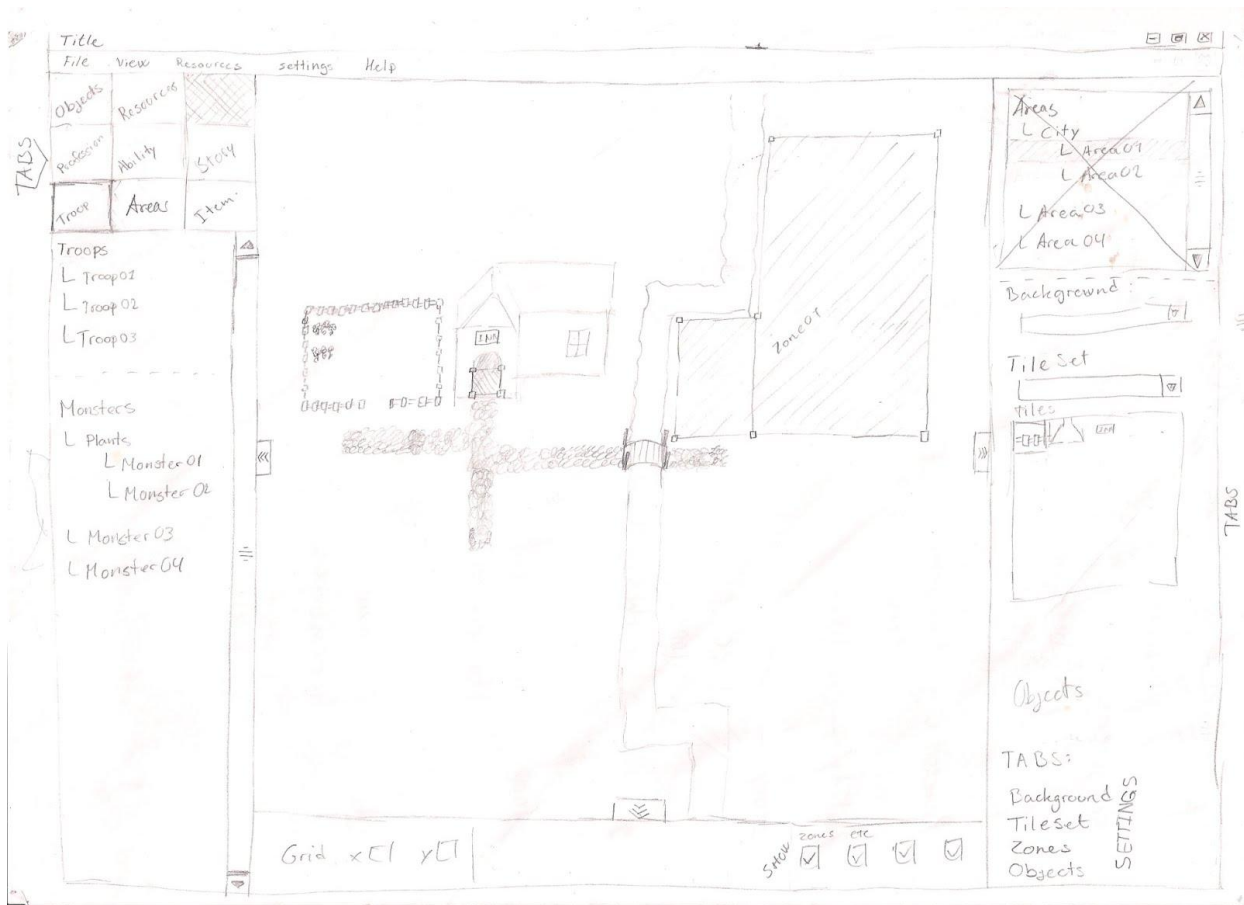
Subtasks:	Example Solution:
1. Set text <i>(optional)</i>	Large text field with keyword highlighting
2. Close speech editor a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the speech

T10.3: Manage Shop

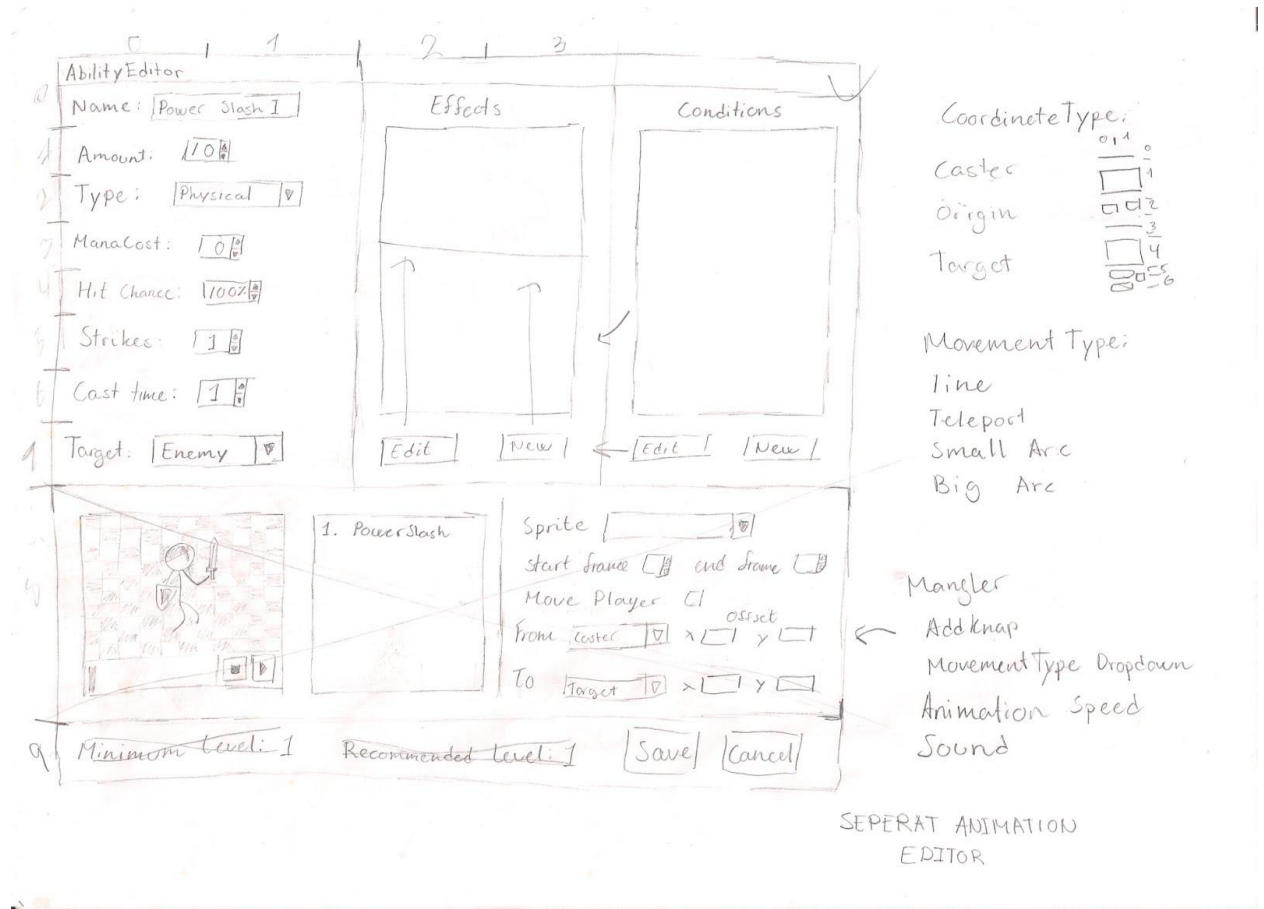
Start: User opens shop editor
End: User closes shop editor
Frequency: Often

Subtasks:	Example Solution:
1. Set equipment level <i>(optional)</i>	Normal text field
2. Add item <i>(optional)</i>	Button that opens up a hierarchical view of the items for selection, and adds it to the list
3. Edit item <i>(optional)</i>	Double click on item in the list to edit it in item editor
4. Remove item <i>(optional)</i>	Button that removes selected item from list
5. Add accessory <i>(optional)</i>	Button that opens up a hierarchical view of the accessory for selection, and adds it to list
6. Edit accessory <i>(optional)</i>	Double click on accessory in the list to edit it in the accessory editor
7. Remove accessory <i>(optional)</i>	Button that removes accessory from list
8. Set companions to stock for <i>(optional)</i>	List of all available companions, with a check box for each entry
9. Close shop editor <i>(optional)</i> a. Accept changes b. Cancel changes	Buttons that saves and cancels changes made to the shop

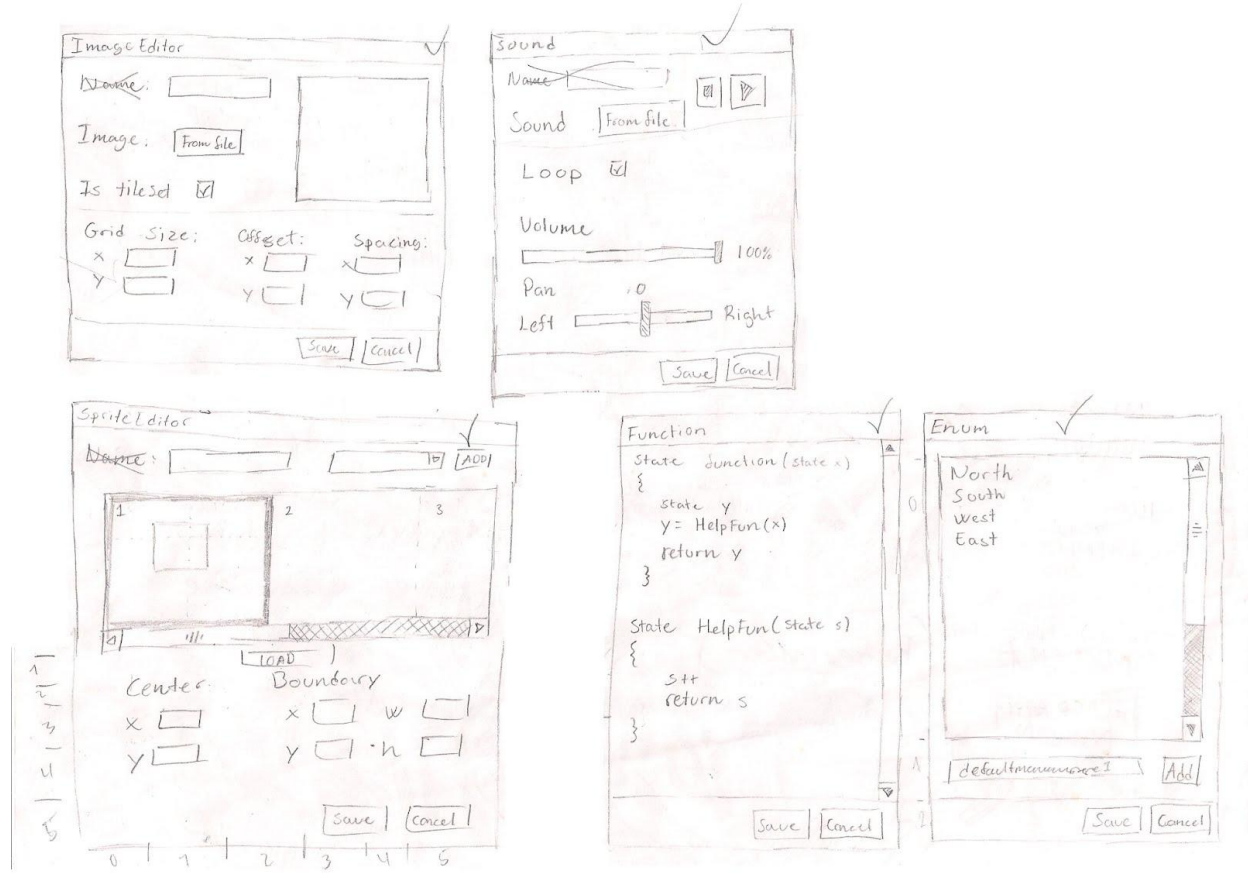
5 - Virtual Windows



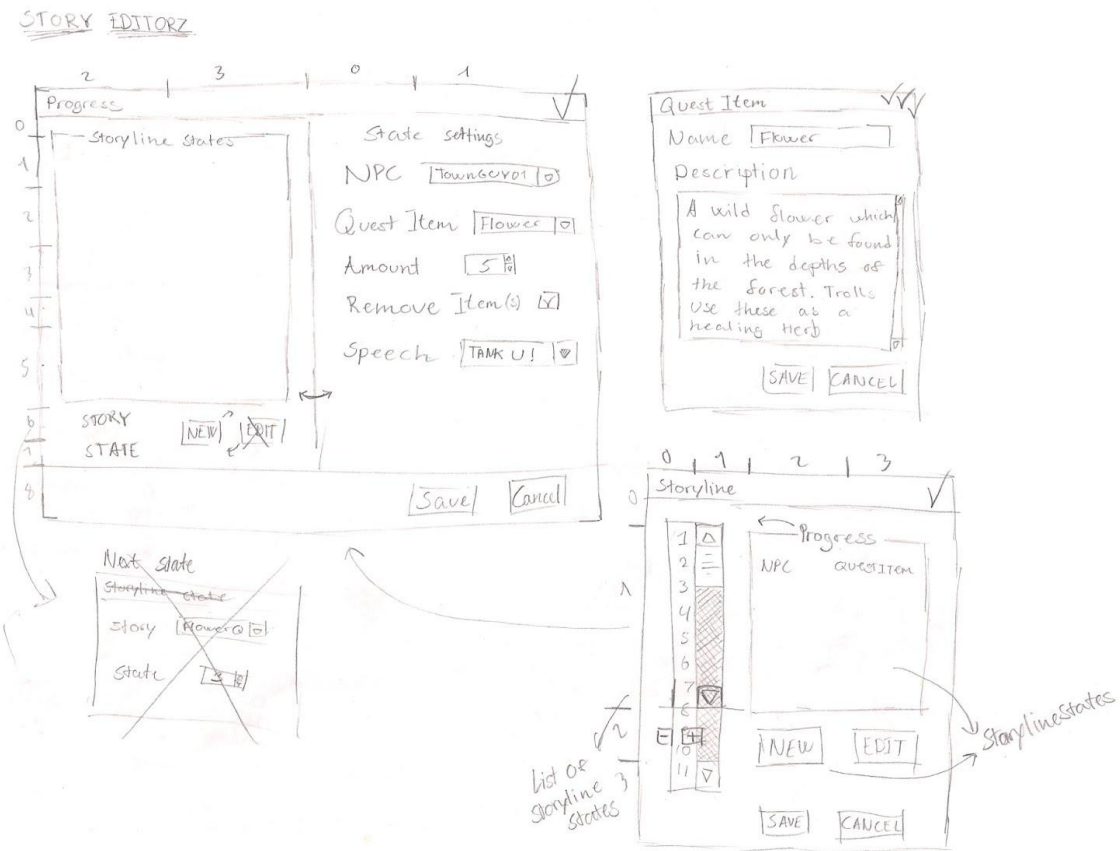
(Main window, with resource views and area)



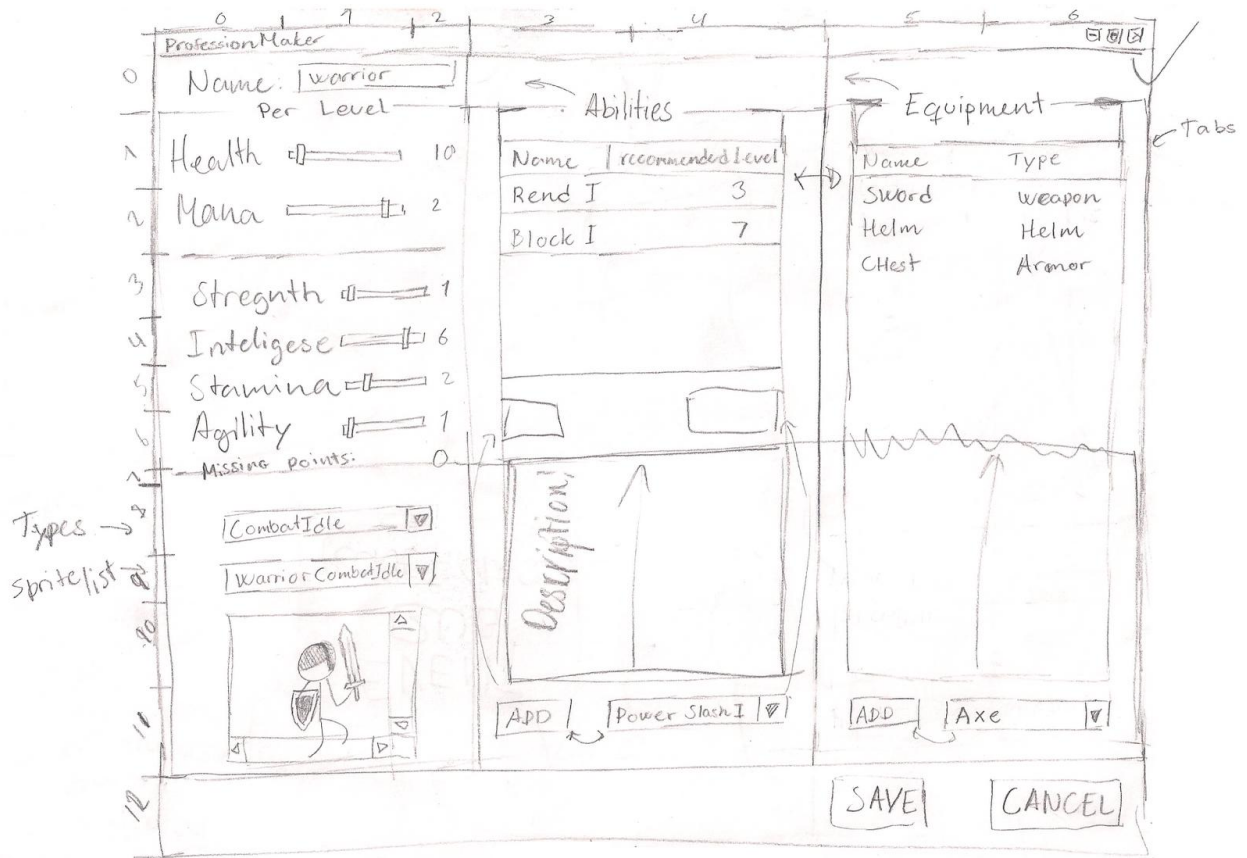
(Ability Editor)



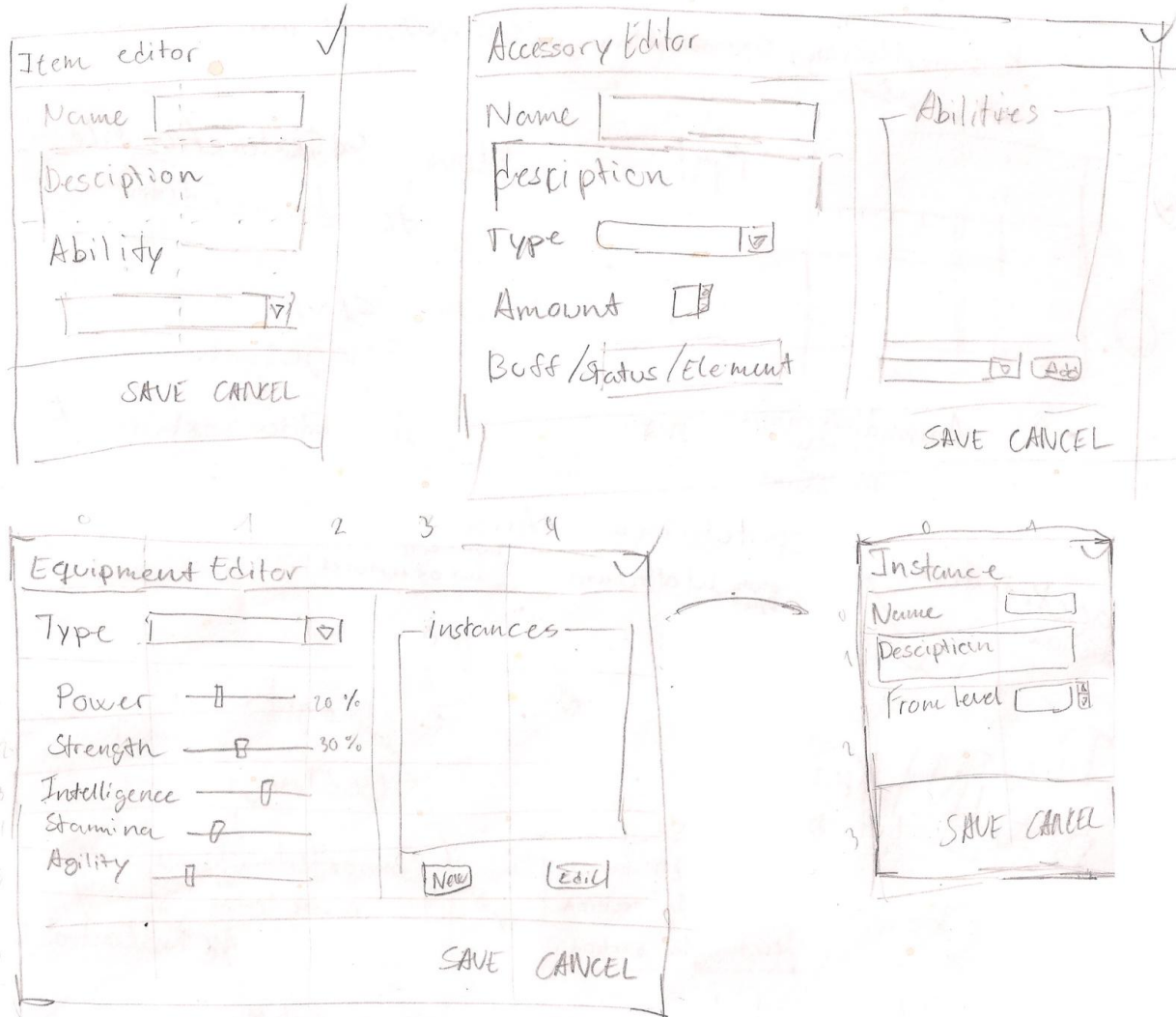
(Resource Editor)



(Story Editor)



(Profession Editor)



(Item Editors)

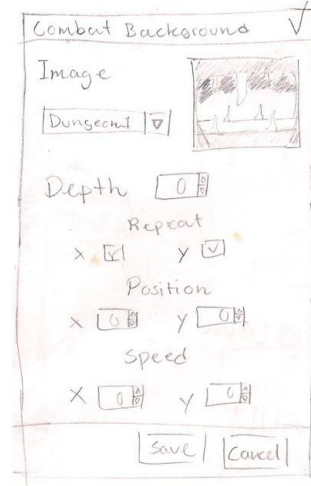
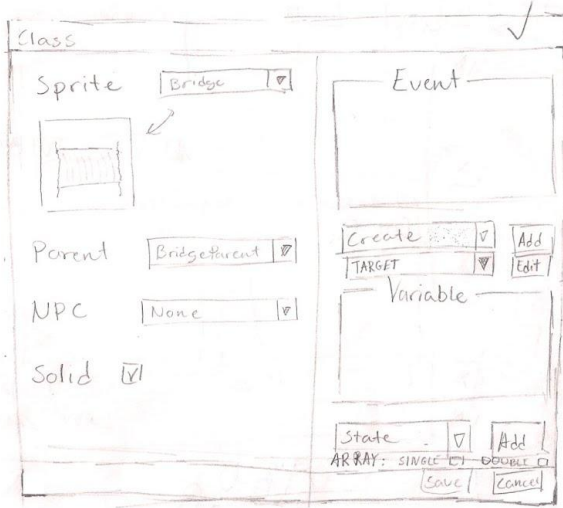

```

Event
event()
{
  count ++
  if (count > 10)
  {
    count = 0
    call this.forcedCombat()
  }
}
    
```

Save Cancel

Mangler Double Array

(Object Editor, Part 1)



<BUTTON>
Button UP
↳ Action HIDE

(Object Editor, Part 2)

6 - JourneyScript

Variable types

JourneyScript uses several types:

- booleans, which are either true or false
- states, which corresponds to integers
- values, which corresponds to doubles
- objects, which are references to other objects
- strings, which are simple text elements
- sounds, which are sound resource elements with some additional information
- sprites, which are advanced image resource elements with some additional information
- images, which are simple image resource elements with some additional information
- speeches, which are text resource elements with some additional information
- storylines, which are named state resource elements
- NPCs, which are characters that can perform speeches
- items, which is a collection of all in-game items

Booleans

JourneyScript uses booleans for truth and false. In order to define a new boolean, the following code is used:

```
boolean x = false;  
x = 1;
```

Booleans can only be assigned the values 0 and 1. The keywords `true` and `false` can be used instead of these for use in readability, but are simply translated to 1 and 0 respectively. Setting a boolean to a value above zero will result in the value being true, otherwise zero. Booleans can be used in assignments with states and values, but not in arithmetic operations. `bool` can be used as a short version of `boolean`.

States

Booleans will often not be the natural choice for cases where there may be more than two options for a variable. In this case, states can be used. States correspond to integers in other languages. In order to define a new state, the following code is used:

```
state x = 10;  
x = -5;
```

States are very useful in switch statements, and enumerations are a finite set of states.

Values

In the case of calculations, states are not usable if a decimal value is desired. For this, values can be used. Values translates to floating point numbers in other programming languages.

```
value x = 0;  
x = 3.124;
```

The keyword `value` is used to define values, with `val` can be used as a shorthand of `value`. Values can noticeably not be used in switch statements, but can be used easily in both assignments and arithmetic operations with states.

Objects

Object references cannot be used in arithmetic operations apart from equality comparisons. In order to define an object reference, the following code is used:

```
object o = null;  
o = this;  
o.x = 240;
```

Notice that an object reference cannot use the same name as a boolean or value, and that `null` is a special keyword indicating no object. `obj` can be used as a shorthand of `object`. Objects has several object variables that can be called on them. Notice that it is not possible to call custom defined object variables through object references besides `this` without using a casting to the desired class.

- `x`, a value - the x-position of the object
- `y`, a value - the y-position of the object
- `speed`, a value - the current overall position moving along a direction per game loop step
- `direction`, a value - the direction moved by the object, from 0 to 360
- `hspeed`, a value - the horizontal movement by the object per step
- `vspeed`, a value - the vertical movement by the object per step
- `acceleration`, a value - the increase of speed by step
- `accdirection`, a value - the direction in which acceleration works
- `sprite`, a sprite - the sprite assigned to the object - notice that this may return `null`
- `animation`, a value - the current animation number of the shown sprite
- `animspeed`, a value - the increase of the animation numbers by step
- `depth`, a state - the current depth of the object
- `solid`, a boolean - if the object is traversable (cannot be changed)
- `class`, a class - the class of the current object (cannot be changed)
- `parent`, an object - the parent class of the current object (cannot be changed)

Strings

Strings are text elements that are inserted straight into the code. Like objects, arithmetic operations cannot be used, besides comparisons (lesser and greater using alphanumeric sorting), and addition. A string is defined by using the `string` keyword:

```
string x = "abc ";  
x = x + "val: " + 10;
```

Strings can be any size, including empty and even null. Strings are assigned to using text inside quotation marks. Newlines inside of the quotation marks are translated to newlines in the string. Likewise, written escape sequences like `\n` and `\r` are kept in the string. It is also possible to add booleans, states and values together with strings, as long as they are added to the right side of a string - either a reference or a new string.

It should be noted, that string objects that are created in an event and assigned to a local variable will be re-created every time the event is called. This may cause significant resource problems, if a string is recreated in an often repeated event. Consider assigning the string to an object variable in order to prevent unnecessary string creations. Also, string concatenations are done two at a time - a long line of strings will not run optimized. JourneyScript does not have a string builder. Lastly, all strings in JourneyScript are encoded in UTF-8.

Sounds

`sound` elements are references to individual sound effects and music numbers that are saved in the sound resource structure for the project. The `play_sound` function is used to create sound elements from the list of sound resources:

```
sound s = play_sound(Sound.attack02);  
s.volume = 50;
```

It should also be noted the sound is a specific instance of a sound playing, and where comparing two sounds created separately, would be false. Each sound have different variables:

- `loop`, a boolean - true/false if song loops
- `volume`, a state - how loud the sound is played, from 0 to 100
- `pan`, a state - the sounds position, left/right speaker wise, from -100 to 100

Sprites

Each object has a `sprite` element assigned to it at all times - a sprite being the image that is drawn at the object's location. Each sprite consists of not just one image, but a series of images which are then animated. Besides the image, additional information is required to correctly position a sprite, with the origin of the sprite being drawn at the origin of the object. It is not possible to change any of this information at runtime.

To access the sprites, either get the sprite directly through the object, or retrieve it from the global sprite resource list:

```
sprite spr = Sprite.soldierWalkUp;  
this.sprite = spr;
```

Each sprite have different variables:

- width, a state - the width of the sprite (cannot be changed)
- height, a state - the height of the sprite (cannot be changed)
- frames, a state - the number of frames the sprite has (cannot be changed)

Images

`image` elements are similar to sprites, but serves a different purpose. While sprites involves animations and are tightly bound with objects and their execution, images are not animated, has a fixed center of origin, and cannot be assigned to an object. The purpose of images are for the background, and can be split into sub-images to be used as tiles. It is not possible to change any of this information at runtime. Images can be loaded from the global image resource list:

```
image im = Image.Cloud03;
```

Each image have different variables:

- width, a state - the width of the image (cannot be changed)
- height, a state - the height of the image (cannot be changed)
- tileset, a boolean - determines if the image is a tileset (cannot be changed)

Speeches

`speech` elements are in essence strings. However, they are stored in an external resource file, and are all loaded at the beginning of the game. The speech resource is made purposely for use in creating speeches for NPCs, signs, and other similar text throughout a Journey. It is not possible to create speeches at runtime - that is the purpose of using strings.

The main advantage of the speeches is, that it is possible to have words be changed depending on the player's choice of character:

- `{{gender}}` - is replaced by the player characters choice of gender - 'he', 'she' or 'it'...
- `{{genderp}}` - is replaced by a possessive gender word - 'his', 'her', or 'its'...
- `{{name}}` - is replaced by the player characters name
- `{{title}}` - is replaced by the player characters title - 'hero', 'warror', 'princess'...
- `{{class}}` - is replaced by the player characters class - 'white mage', 'soldier', 'bard'...
- `{{weapon}}` - is replaced by the player characters weapon - 'sword', 'bow', 'whip'...

In the resource structure, each speech has an id, which is referred to by a name in the global resource list `Speech`. Speeches can be retrieved from this resource list.

```
speech sp = Speech.townNPC03;
```

Storylines

A game is constructed over a series of storylines. A `storyline` element is in and of itself little more than a state - an integer value representing the current progress through the storyline. Each storyline has a name, and can be accessed through the global storyline list:

```
storyline sto = Storyline.mainquest4;  
sto.current = 6;
```

Each storyline has some variables:

- `current`, a state - the current state of the storyline

NPCs

In the game, NPCs are characters that, while not necessarily important to the plot, gives speeches when talked to based on the current progress of a given storyline. The existing NPCs can be retrieved through the global NPC list:

```
npc cha = NPC.villeger4;
```

Items

Items are the weapons, armor, accessories, quest necessary objects and consumables that can be stored in the player's inventory. An `item` can be retrieved through multiple items lists:

```
equipment it1 = Equipment.Sword(56);
accessory it2 = Accessory.BloodRing;
item it3 = Item.HighPotion;
questitem it4 = QuestItem.OddCrystal;
```

Note, that the names of the items are not the in-game name, as these may contain spaces. Also note, that equipment need the level of the equipment in order to retrieve it.

Enums

Enumerations can be accessed through the global enum collection. Each enum has a list of individual states. For instance:

```
state s = Enum.keys.BossKey;
```

Classes

A `class` is the type of an object, and what defines an object. While there may be multiple object instances of a class, there can be only one class of a given name. Classes can be referenced either through the object through the class variable, or through the class global resource. The special `is` relation are used to see if an object can be cast to a specific class - either the class or its parent.

```
class c = this.class;
if (other is c)
    other.x = 2;
```

Areas

Areas are the sections of the game, one of which is the current one. Areas cannot be referenced by variable, but the names of areas can be used with the `goto` statement, where the current area is set to the area with the given name:

```
goto OverworldMap;
```


Scope of variables

JourneyScript uses three scope layers for variables. Global, Object, and Local. Global and Object are in the heap - global being a global repository all objects can access, and Object being an instance of a given object, which has its own variables that can be set. Local is on the stack, and is only visible inside of a given block, much as local variables in other programming languages.

Global variables uses the `global` keyword to refer to a list of variables in a special singleton object that all other objects can access. All global variables, both boolean and values, must be defined at compile time.

Global variables can be accessed with:

```
global.x = 10;  
value x = global.x;
```

Object variables can be referred through known object - the current using the `this` keyword. The event caller is referred to using the keyword `other`. It is possible to modify the other object's variables during the event - and generally possible to modify the variables of any referenced object.

```
object x = other;  
x.y = this.y;
```

Notice that it possible for there to be a variable of the same name internally, in the object, and in the global space without them ever conflicting - due to the need of using `this` and `global` to refer to the object and global variables.

It should be noted that all global and object variables are initialized to either false in case of booleans, zero in case of states and values, and null in case of anything else, including arrays.

Arrays

Arrays are created similar to arrays in other programming languages. Brackets indicates a variable being an array - for values, the `value` keyword is mandatory when creating value arrays:

```
value[] x = val[10];  
x[9] = 5;  
x[3] = 2.00;
```

Notice, that in the value array, different entries may store different types - some doubles, some integers. Also, shorthands may be used interchangeably.

Besides simple arrays, it is also possible to create two-dimensional arrays. In order to do so, another set of brackets are included.

```
val[][] x = value[10][20];  
x[3][2] = 5.0;
```

It is possible to redefine arrays, by setting it to another set of arrays to create a new 1 or 2 dimensional array with a new length, as long as the amount of dimensions match the array:

```
val[][] x = value[10][20];  
x[3][2] = 5.0;  
x = value[4][3];  
x[3][2] = 5.0;
```

It is also possible to set the content of the array instead of the length:

```
val[] x = {1.0, 2.0, 3.0, 4.0};  
val[][] y = { {1.0, 2.0}, {2.0, 3.0}, {3.0}, {4.0} };
```

It should be noted that the two-dimensional arrays are not jagged, and the dimensions of all internal arrays in the set content has to be of the same length.

Conditionals and Loops

Loops and conditionals are similar to other languages. Simple conditions can be constructed with `if` and `else`. The keyword `if` is followed by a condition in parentheses. If the condition is true, the following block of code will be executed. If not, it will be skipped, and an optional block of code after an `else` keyword will be executed instead. A block of code is either lines of code in brackets, or a single line of code. Conditionals for values and states are true if the value is larger than zero, and for all other references, it is true if the reference is not null.

An example of a conditional:

```
if (x == 0)
  x++
else if (x > 10)
{
  x--
}
else
  x = 0
```

Besides the if-else conditional, the in-line ternary conditional is also usable. A variable can be assigned to an expression that starts with a conditional. If this evaluates to true, the expression after a question mark will be used, otherwise the expression next to the following colon will be used. An example of a ternary conditional:

```
x = 1 > 0 ? true : false
```

For loops are constructed using the keyword `for`, a parenthesis section, and a code block. In the parenthesis, three sub-sections are included, divided by semi-colons - the initiator, which is run before the loop starts, the conditional, which checks if another loop is to be performed, and the incrementor, which is performed at the end of each loop. Each of these may be empty - which is how an infinite loop is normally defined. It is also possible to use the `break` statement to prematurely end the loop, and the `continue` statement to jump directly to the incrementor. Examples of for loops:

```
for(val x = 0; x < 10; x++)
  x++

for(;;)
{
  x++
}
```

While loops are similar to for loops, except it does not have an initiator, or an incrementor. It is constructed using the `while` keyword, a conditional in parenthesis, and a code block. Like with for-loops, breaks and continues can be used to prematurely end or restart the loop. An example of a while loop:

```
while (true)
  x++
```

A variant of the while loop is the do-while loop. It is constructed using the `do` keyword, followed by a code block, followed by the `while` keyword and a conditional in parenthesis. The difference between this and a standard while loop is, that the code block will be executed once before the conditional is checked - even if it is false. An example of a do-while loop:

```
do
{
  x++
}
while (true)
```

Switches are an optimized alternative to an if-else structure. In the switch, a variable is checked, and if it is equal to one of the cases, the code for that case is performed until the next case is reached (there is no fall through) or a `break` keyword is used. A switch is constructed with a `switch` keyword, followed by a variable in parenthesis. Followed is a block of code inside brackets, with separate cases. It is possible to have multiple cases use the same code by separating them with commas. The code is performed if any of the cases it is attached to are true. Notice that only one instance of a case may be used in a switch. An optional default case can be inserted at the end of the switch, which will be performed if no other case is met.

For booleans, only true and false are usable, so they are not really ideal for switches. For states, all integers can be used. In the case of objects, the class of the object is used, along with null. For sprites, sounds, speeches and images, using resource accesses will be possible, also along with null, and no resource reference is required, as the type is gathered by the switched variable. It is not possible to use switches with values. Examples of switches:

```
switch (x)
{
  case 1,2: { x++; break }
  case 3: x--
  case 4: x = 8
  default: x = 0
}

switch (spr)
{
  case SoldierMoveUp: x = 0
  case SoldierMoveDown: x = 1
  case SoldierMoveLeft: x = 2
  case SoldierMoveRight: x = 3
}
```

Events

JourneyScript is an event based system. An event can be considered a method that can be invoked, either by the engine or by other events. Events are attached to a class, and is the primary area where the user can write code.

An event is assigned for a class or zone, which contains a code segment. This code segment includes an event, with a possible set of parameters, and local functions that can be called from the event.

An example of an event with a local function:

```
event(state x,  
      val y) {  
    extraLocalFunktion();  
}  
  
val extraLocalFunktion ( ) { return 0; }
```

Each event has an event method, which is invoked when the event is called. When creating the event, a list of parameters can be set, comma-separated in parenthesis. When calling the event, the correct parameters must be supplied. The type of the parameter must be written before the name of the parameter.

To call an event, one needs to have an object reference, and know what class the object is. Only by casting an object to a given class, or if calling it on the this reference, is it possible to call it's events. The code in order to call an event is:

```
call oref(ObjectClass) EventName(x,y)
```

Notice that casting an object reference to a class that it is not, and is not a child of, will trigger an exception. To prevent this, it is possible to check the castability of an object:

```
if (oref is ObjectClass)  
    call oref(ObjectClass) EventName(x,y)
```

In events, a special keyword `other` is usable in order to reference the object that called the current event. However, if the event call is from a zone or from the system, `other` will be null. If using an event that may be triggered by the system or by user defined zones, checking if `other` is null before using it is essential.

7 - JourneyScript, Backus-Naur Form

```

<char> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
          p | q | r | s | t | u | v | w | x | y | z | A | B | C | D |
          E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
          T | U | V | W | X | Y | Z | _ | -
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= | <digit> <digits>
<charordigit> ::= | <char> <charordigit> | <digit> <charordigit>
<escape> ::= \" | \\ | \t | \n | \r
<special> ::= | ! | @ | # | £ | ¤ | $ | % | & | / | { | ( | [ | ) | ]
              | = | } | ? | + | ´ | | | , | ; | . | : | ^ | " | ~ | * | '
<any> ::= | <char> <any> | <digit> <any>
          | <escape> <any> | <special> <any>

<number> ::= <digit> <digits>
<floatnum> ::= <digit> <digits> . <digits> | . <digit> <digits>
<name> ::= <char> <charordigit>
<cstring> ::= " <any> "

<boolean> ::= boolean | bool
<value> ::= value | val
<object> ::= object | obj

<jssection> ::= <n> <sectiontype> <n> <functionlist> EOF

<functionlist> ::= | <function> <functionlist> <n>
<function> ::= <typeempty> <name> ( <parametersdec> ) <n>
                                     { <n> <body> <n> }

<sectiontype>: <eventfunc> | <functionfunc>
<eventfunc> ::= event ( <parametersdec> ) <n> { <n> <body> <n> }
<functionfunc> ::= <typeempty> function
                  ( <parametersdec> ) <n> { <n> <body> <n> }

<n> ::= | '/n' <n>

<typekind> ::= <boolean> | state | <value> | string | <object>
              | sound | speech | sprite | image | story | npc
              | item | equipment | questitem | accessory | class
<type> ::= <typekind> | <typekind> [ ] | <typekind> [ ] [ ]
<typeempty> ::= | <type>

<parametersdec> ::= | <n> <paramdec> <nextparamdec>
<nextparamdec> ::= | , <n> <paramdec> <nextparamdec>
<paramdec> ::= <type> <name>

```

```

<body> ::= | '/'n' <n> <body> | <stmt> '/'n' <n> <body>
        | <stmt> ; <n> <body> | <control> '/'n' <n> <body>
        | <ctrlunbalanced> '/'n' <n> <body>
        | <stmt> | <control> | <ctrlunbalanced>

<innerbody> ::= <n> <singleinnerbody> <n> | <n> { <n> <body> <n> } <n>

<singleinnerbody> ::= ; | <stmt> ; | <control> | <ctrlunbalanced>

<expr> ::= <access> | <operationNeg>

<access> ::= this | this . <accessref> | other <accessoption>
        | global . <name> <accessoption>
        | global . <name> [ <expr> ] <accessoption>
        | global . <name> [ <expr> ] [ <expr> ] <accessoption>
        | <name> <accessoption>
        | Zone . <name> | Zone . <name> . <accessref>
        | <name> [ <expr> ] <accessoption>
        | <name> [ <expr> ] [ <expr> ] <accessoption>
<accessref> ::= <name> <accessoption> | <name> [ <expr> ]
<accessoption>
        | <name> [ <expr> ] [ <expr> ] <accessoption>
<accessoption> ::= | ( <name> ) | . <accessref> | ( <name> ) .
<accessref>

<operatinNeg> ::= <operation> | - <expr>

<operation> ::= <access> <n> = <n> <expr> | <access> <n> is <n> <name>
        | <access> <n> = <n> <arrlassign>
        | <access> <n> = <n> <arr2assign>
        | <number> | <floatnum> | <cstring> | null
        | <resource> | <expr> <n> ? <n> <expr> <n> : <n> <expr>
        | ! <expr> | ( <n> <expr> <n> )
        | <expr> + <expr> | <expr> - <expr> | <expr> * <expr>
        | <expr> / <expr> | <expr> % <expr>
        | <expr> <n> == <n> <expr>
        | <expr> <n> < <n> <expr> | <expr> <n> <= <n> <expr>
        | <expr> <n> != <n> <expr> | <expr> <n> >= <n> <expr>
        | <expr> <n> && <n> <expr> | <expr> <n> || <n> <expr>
        | <name> ( <n> <params> <n> )
        | ++ <access> | -- <access> | <access> ++ | <access> --

```

```

<resource> ::= | Sprite . <name> | Image . <name>
            | Sound . <name> | Speech . <name>
            | Storyline . <name> | NPC . <name>
            | Accessory . <name> | Item . <name>
            | QuestItem . <name> | Class . <name>
            | Zone . <name> | Equipment . <name> ( <number> )
            | Enum . <name> . <name>

<arr1assign> ::= <type> [ <number> ] | { <varlelement> }
<arr2assign> ::= <type> [ <number> ] [ <number> ] | { <var2element> }
<varlelement> ::= <expr> <varlnext>
<varlnext> ::= | , <n> <expr> <varlnext>
<var2element> ::= { <varlelement> } <var2next>
<var2next> ::= | , <n> { <varlelement> } <var2next>
<params> ::= | <expr> <nextparams>
<nextparams> ::= | , <n> <expr> <nextparams>

<stmt> ::= <expr> | <vardec> | break | continue
        | call this . <name> ( <params> )
        | call Zone . <name> . <name> ( <params> )
        | call ( <access> ) . <name> ( <name> ) . <name> ( <params> )
        | return | return <expr>
<control> ::= if ( <expr> ) <n> <innerbody> <n> else <n> <innerbody>
            | while ( <expr> ) <n> <innerbody>
            | do <n> <innerbody> <n> while ( <expr> )
            | for ( <n> <inst> ; <n> <ltest> ; <n> <cexpr> <n> )
                <n> <innerbody>
            | switch ( <access> ) <n> { <n> <cases> <n> }
<ctrlunbalanced> ::= if ( <expression> ) <n>
                    <innerbody> <n> else <n> <ctrlunbalanced>
                    | if ( <expression> ) <n> <innerbody>

<vardec> ::= <typekind> <name> <nextvardec>
        | <typekind> <name> <n> = <n> <expr> <nextvardec>
        | <typekind> [ ] <name> <nextarr1dec>
        | <typekind> [ ] <name> <n> = <n> <arr1assign> <nextarr1dec>
        | <typekind> [ ] [ ] <name> <nextarr2dec>
        | <typekind> [ ] [ ] <name> <n> = <n> <arr2assign> <nextarr2dec>

<nextvardec> ::= <n> | , <n> <name> <nextvardec>
              | , <n> <name> <n> = <n> <expr> <nextvardec>

```



```
<nextarr1dec> ::= <n> | , <n> <name> <nextarr1dec>  
                | , <n> <name> <n> = <n> <arr1assign> <nextarr1dec>
```

```
<nextarr2dec> ::= <n> | , <n> <name> <nextarr2dec>  
                | , <n> <name> <n> = <n> <arr2assign> <nextarr2dec>
```

```
<inst> ::= | <access> <n> = <n> <expr>
```

```
<lttest> ::= | <expr>
```

```
<cexpr> ::= | <operation>
```

```
<cases> ::= case <case> <orcase> : <n> <innerbody> <n> <casen>
```

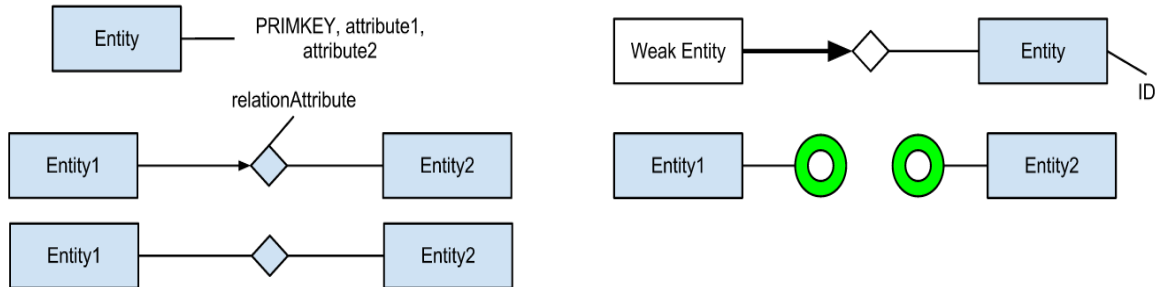
```
<casen> ::= | case <case> <orcase> : <n> <innerbody> <n> <casen>  
            | default : <n> <innerbody>
```

```
<case> ::= <number> | <cstring> | <name> . <name> | <name> | null
```

```
<orcase> ::= | , <n> <case> <orcase>
```

8 - E/R Model Explanation

Due to the amount of different types of E/R models, it is necessary to describe the specific syntax used for the project.



Entities contains a series of attributes, where entities are translated to classes in the implementation. Primary keys that uniquely identify the entities (used for uniqueness in the unique lists through override of the equals methods) are written with full uppercase letters. All attributes are simple types or enums. References to other classes are not written as attributes, but instead using relations, which are written as a diamond shape. The relations may or may not be named.

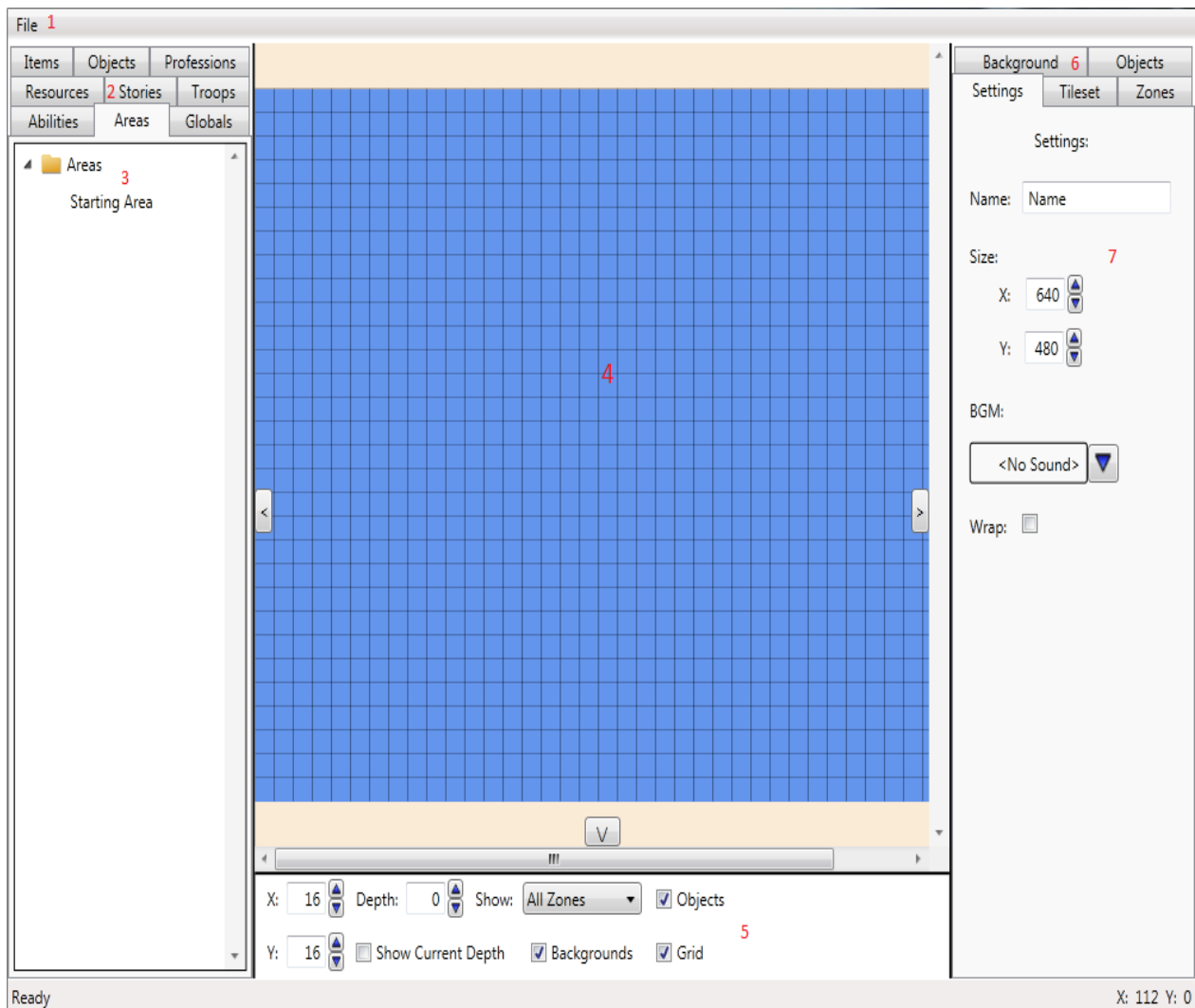
There are different types of relations between entities. Each relation is connected to two entities (more is possible, but that is not used in this case). The type of connector between the relation and the entity matters, and shows in what amount the entity is involved with in the relation. A simple lines indicates that many relations between the two are possible, while an arrow indicates that only one relation is possible. If a line is used on both ends, there is a many-to-many relationship, and the relation has to be expanded to a class itself. If there is an arrow at one end, then the first entity may simply have a variable relating to it. It may still have to be expanded to a new class in the case it has an attribute of its own - which a simple variable reference would not be able to contain.

The width of the connector also matters - thin shows that there may be no relations, and if used with an arrow, that means that a reference may be null. A special case of this being used are weak entities, which are shown with a white background instead of blue. Weak entities cannot describe themselves fully without the use of another class, and have a relation of exactly one to another entity - the relations that are part of this master-slave relation is also shown with a white background. In a database that would be shown by having the weak entity contain the primary keys of the master entity, but in our implementation, weak entities are simply part of the master entities composition, and the weak entities cannot be referenced outside of it. That does mean that all non-weak entities can be accessed globally, which is done through the resource manager.

Finally, in the case where two entities are too far apart to create a simple link, portal relation nodes are used specifically for this project, because we are thinking with portals. Portal nodes are connected by color, and otherwise works as a normal relation would.

9 - User Manual

Main Window



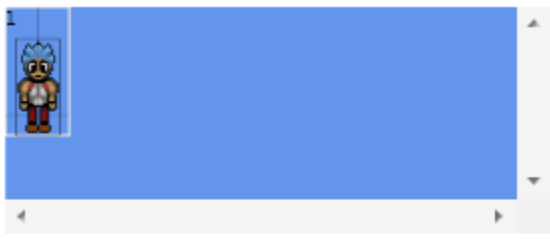
1. The file menu where you save, load and compile projects.
2. The resource tab, this gives access to all project resources.
3. The resource view, here resources are displayed.
4. The area map where tiles, zones and objects can be placed.
5. The area view settings.
6. The area settings tab, giving access to tiles, zones etc.
7. The setting for each tab, this is where you set size, choose tiles etc.

Tutorial - Creating the Warrior:

Name:	<input type="text" value="Power Slash I"/>
Amount:	<input type="text" value="10"/>
Type:	<input type="text" value="Physical Damage"/>
Mana Cost:	<input type="text" value="0"/>
Hit Rate:	<input type="text" value="100"/>
Strikes:	<input type="text" value="1"/>
Cast Time:	<input type="text" value="1"/>
Target:	<input type="text" value="Enemy"/>

Creating the Ability

1. Create new ability by right clicking the *Abilities* folder, followed by New Ability.
2. Rename the new ability by right clicking and choosing *Rename*.
3. Double click the ability to open the Ability Editor (or by right click > *Edit*).
4. Set the following values in the fields and leave all other things unchanged.
5. Click *Save* to finalize the changes.

					
<input type="button" value="Load Image"/>					
Center:		Boundary:			
X:	<input type="text" value="16"/>	X:	<input type="text" value="5"/>	Width:	<input type="text" value="22"/>
Y:	<input type="text" value="54"/>	Y:	<input type="text" value="16"/>	Height:	<input type="text" value="48"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>					

Load the Sprites

1. Create new sprite by right clicking the *Sprites* folder, followed by New Sprite.
2. Rename the new sprite by right clicking and choosing *Rename*.
3. Double click the sprite to open the Sprite Editor (or by right click > *Edit*).
4. Use the Load Image button to load each individual part of the sprite.
5. Set the following values in the fields.
6. Press Save to save the changes.
7. Rinse and repeat until all sprites have been created.

Creating Equipment

1. Create new equipment by right clicking the *Equipment* folder, followed by New Equipment.
2. Rename the new equipment by right clicking and choosing *Rename*.
3. Double click the equipment to open the Equipment Editor (or by right click > *Edit*).
4. Set the values as shown in the picture below.

The screenshot shows the Equipment Editor dialog box. On the left, there are sliders for various stats: Defence (60%), Strength (25%), Intelligence (0%), Stamina (15%), and Agility (0%). The 'Type' dropdown is set to 'Armor'. On the right, the 'Equipment Instances' list contains one item: 'Trainee's Leather Vest - 1'. At the bottom, there are buttons for 'New', 'Edit', 'Save', and 'Cancel'.

5. To add the leather vest. Press the new button and double click the item.
6. Set the values as shown and press save.

The screenshot shows the Equipment Editor dialog box for 'Trainee's Leather Vest'. The 'Name' field contains 'Trainee's Leather Vest'. The 'From Level' dropdown is set to 1. The description text area contains 'Made from bunny hides.'. At the bottom, there are buttons for 'Save' and 'Cancel'.

7. Finalize it by pressing the save button.

Creating the Profession

Name:	Warrior
Health:	20
Mana:	1.0
Strength:	5.5
Intelligence:	1.0
Stamina:	2.5
Agility:	1.0
Missing Points:	0

Equipments:	
WarriorArmor	
WarriorArmor	Add

1. Create new profession by right clicking the *Professions* folder, followed by New Profession.
2. Rename the new profession by right clicking and choosing *Rename*.
3. Double click the profession to open the Profession Editor (or by right click > *Edit*).
4. Set the values as shown in the picture below.
5. Add the created equipment list by choosing the equipment and pressing add.
6. Add the created ability list by choosing the ability and pressing add.
7. Write a fitting description.
8. In the sprite menu choose the type of sprite animation you want along with the correct sprite.
9. Do that for all sprites for this profession.
10. Press the Save button to finalize the changes.

Abilities:	
PowerSlashI	
<No Ability>	Add