

Technical Design Document

**Zephyr: A State-Based, Event-Driven, Domain-Specific Language for 2D, Top-Down, Action Role-Playing Games**

Version 1.0

|  |  |  |
| --- | --- | --- |
| **Position Title** | **Name** | **Signature** |
| Programmer | Tyler Wolverton |  |
| Advisor | Matt Butler |  |

* ***Delete all instructional information before turning in final version.***
* *Replace the image above with an appropriate image.*
* *Update the Header and Footer with the correct information.* 
  + *Lock the date to when the version is completed, not the date of its opening.*

Table of Contents

[Table of Figures 4](#_Toc63952770)

[Introduction 5](#_Toc63952771)

[Overview 6](#_Toc63952772)

[Scope 6](#_Toc63952773)

[End Product 6](#_Toc63952774)

[Tech 7](#_Toc63952775)

[Gameplay 7](#_Toc63952776)

[Development Schedule 9](#_Toc63952777)

[Schedule Detail 9](#_Toc63952778)

[Total Hours 9](#_Toc63952779)

[Technology Sources 10](#_Toc63952780)

[VSCode Plugin 10](#_Toc63952781)

[Doxygen 10](#_Toc63952782)

[Theory 11](#_Toc63952783)

[Genre 11](#_Toc63952784)

[Event System 11](#_Toc63952785)

[Domain-specific Languages 11](#_Toc63952786)

[Compiler 11](#_Toc63952787)

[Scanner 11](#_Toc63952788)

[Parser 12](#_Toc63952789)

[Virtual Machine 12](#_Toc63952790)

[Bytecode 12](#_Toc63952791)

[Finite State Machine AI 13](#_Toc63952792)

[Visual Studio Code Plugin 13](#_Toc63952793)

[Previous Work 13](#_Toc63952794)

[Naughty Dog’s Uncharted 2 Scripting Language 13](#_Toc63952795)

[Out of Scope 13](#_Toc63952796)

[Interactive Debugging 13](#_Toc63952797)

[Visual Scripting 14](#_Toc63952798)

[Artifact 15](#_Toc63952799)

[Implementation 15](#_Toc63952800)

[Compiler 15](#_Toc63952801)

[Scanner 15](#_Toc63952802)

[Parser 15](#_Toc63952803)

[Error Handling 15](#_Toc63952804)

[Hot Recompiling 15](#_Toc63952805)

[Virtual Machine 15](#_Toc63952806)

[Bytecode Generation 15](#_Toc63952807)

[Stack-based Interpretation 15](#_Toc63952808)

[ZephyrObject 15](#_Toc63952809)

[Architecture 15](#_Toc63952810)

[Diagram 15](#_Toc63952811)

[Syntax 16](#_Toc63952812)

[Results 17](#_Toc63952813)

[User Feedback 17](#_Toc63952814)

[LD Usage Feedback 17](#_Toc63952815)

[Profiling Results 17](#_Toc63952816)

[Example Images Section 18](#_Toc63952817)

[Example Image 1 18](#_Toc63952818)

[Example Image 2 19](#_Toc63952819)

[Example Image 3 20](#_Toc63952820)

[Bibliography 21](#_Toc63952821)

# Table of Figures

[Figure 1: Example Image 1 18](#_Toc61607850)

[Figure 2: Example Image 2 19](#_Toc61607851)

[Figure 3: Example Image 3 20](#_Toc61607852)

# Introduction

*Add your usual thesis introduction. Should be a short paragraph between 3 – 5 sentences in length, use common industry terms, and reference important existing algorithms or games.*

Zephyr is a domain-specific language that defines state machines that can listen to and fire events to interact with a custom C++ game engine. Source scripts are compiled into bytecode when the game loads on initial startup or via a hotkey. The bytecode is interpreted by a virtual machine (VM) at runtime whenever a script event is fired. The language is demonstrated with a 2D action role-playing game (RPG) with enemies to fight, keys to collect to unlock doors, and quests to complete, similar to games such as The Legend of Zelda and Dungeon Explorer.

# Overview

*Add a more detailed overview to this section explaining why this project is interesting to do and describe the level of detail needed in this document in order to assure the success of the project.*

## Scope

*Give a brief statement of the scope of the project from a technical point and view. Tie this to the design vision and introduce the overarching development plan for this project. Insert as much detail as necessary.*

* Compiler and VM
* Designer friendly syntax
* 2D top-down, action RPG demo game to show off scripts
* Map editor was not in scope

## End Product

*Describe the pieces that are going to be built in order to develop the final game as defined in the Game Design Document. This sets expectations for amount of work necessary.*

*Gameplay*

* *List separately all game mechanics that need to be implemented*

*Game Objects*

* *List separately all game objects that need to be implemented*
* *For example:*
* *Player Character(s)*
* *Enemies (list with description)*
* *Pick-ups*
* *Etc.*

*HUD & UI*

* *All GUI attributes implemented*
  + *List each aspect of the UI separately*

*Etc.*

### Tech

#### Syntax

#### Scanner

#### Parser

#### Interpreter

#### ZephyrScriptDefinition

#### ZephyrObject

#### Entity System

#### VS Code Plugin

#### Core Game Systems

#### Hot Recompile

#### GameAPI Methods

#### Doxygen Docs

### Gameplay

#### Controls

* WASD moves
* Arrows attack
* Space interacts with objects and NPCs

#### Game Objects

##### Enemies

* Blob
* Splitting blob

##### NPCs

* Old man

##### Pickups

* Keys
* Triple shot power up

##### Maps

###### XML-based maps

* Define tiles
* Define entities

##### User Interface

* Current quest shown in top left of screen

# Development Schedule

## Schedule Detail

*Use a table or other suitable method to list out the major milestones and approximately how must time was spent on each.*

## Total Hours

*Put the total hours here and any other summary information that is appropriate.*

# Technology Sources

*Describe the external tools and technology used during production. May be omitted if nothing applies.*

## VSCode Plugin

## Doxygen

# Theory

*Include as many sections as necessary to cover all theoretical background information that is considered a prerequisite to the thesis. Describe important math and algorithms.*

*This heading and the following three are interchangeable and optional. Choose the combination that best fits your specific thesis.*

## Genre

Action role-playing games (RPGs) are a good fit for a DSL since many of their game systems can be defined in data [1]. They contain varieties of enemies, dialogue with NPCs, and interactable level objects, such as locked doors and keys. To exercise the language, a small action RPG in the style of The Legend of Zelda was created [2]. Both The Legend of Zelda and Dungeon Explorer were used to identify common functionality within the genre in order to determine what functionality to expose to the Zephyr scripts [2] [3].

## Event System

Events are an effective way to communicate between states in a scripting system [4]. Events can also be sent between the game engine and script objects, allowing script code to respond to changes in the game and the game to be affected by script code [5]. An event system can accept events from both game engine code and script code by providing a common API with a generic parameter system.

## Domain-specific Languages

A domain-specific language (DSL) has a variety of benefits over a general-purpose language. The main reasons to use a DSL are to communicate with domain experts and focus on important concepts while hiding technical details [6]. Other benefits include errors that are specific to a particular domain, so they can be easily understood and reported, and less of a learning curve since DSLs take less time to learn and are easier to use for non-programmers [6]. Since DSLs only needs to serve a specific set of needs, they can be highly tuned for their domain and make bold design decisions that a general-purpose language could not [7]. DSLs can be designed to be very different than traditional programming languages if that would serve the task at hand, like the Whimsy language [8]. Zephyr still resembles traditional scripting languages but builds in game development concepts at the syntax level, like entities, 2-dimensional vectors, state machines, and event listening and firing.

## Compiler

Script source code must be compiled into bytecode so that a virtual machine (VM) can interpret the bytecode at runtime. Compilation is performed by first scanning source code into tokens and then parsing those tokens into bytecode.

### Scanner

After a file is read from disk, it is saved as a string and processed by the scanner. The scanner uses a single token lookahead approach to process each character in the source string and convert them into a list of tokens [9]. Each token holds the type of the token, the data the token represents, and the line number where the token appeared. The list of tokens is then passed on to the parser.

### Parser

A parser processes a list of tokens one by one, checks that the tokens form valid expressions, and then translates those expressions into bytecode. Some parsers convert tokens into an abstract syntax tree (AST) which is a tree that can be traversed to build each expression defined in the script code and then generate code from that AST in a second step [9]. The Zephyr compiler is a single-pass compiler, which converts tokens directly into bytecode. Zephyr compiles one source file into multiple chunks of bytecode to be managed as states and events, so any one bytecode chunk is small and does not need to know much information about surrounding code which makes a single-pass compiler feasible [9]. Single-pass compilers are also simpler to implement which is important for the scope of this thesis project [9]. The parser outputs a vector of bytes that correspond to each operation available in the language along with the data that is required for each operation.

#### Pratt Parser

A Pratt Parser is a parsing strategy in which a table is created that maps each token type to a set of functions to call when encountering that token in different expression types [9]. The table defines the function to call when the token starts a prefix expression, the function to call when the token is used in an infix expression, and the precedence level of the token [9]. A prefix operator comes at the beginning of an expression like **-**3. An infix operation comes in between two terms of an expression like 2 - 3. The precedence level defines the operator’s precedence when evaluating an expression to ensure in an expression like 2 \* 3 + 4 that 2 \* 3 is evaluated first. A benefit of using a Pratt Parser is that the parsing code can be simplified to flow through a small series of functions that operate in precedence order and call each function defined in the table, rather that writing specific functions for each precedence level and building a chain of functions that must be called for each expression. Zephyr’s compiler does not use a literal table, but a similar approach is taken when parsing expressions. Functions that switch on the token type to call the appropriate infix and prefix functions achieve the same outcome.

## Virtual Machine

Each chunk of bytecode is interpreted by a virtual machine at runtime. The VM processes the bytecode one byte at a time and executes each operation.

### Bytecode

Bytecode contains instructions and references to constant values used in operations [10]. Constant values are saved in a side vector and the bytecode saves the indices into that vector. A stack is used to store intermediate values during interpretation [10]. Each constant that will be used in an operation is loaded onto the stack before the op code for that operation is processed. When the op code is read, the most recent constants can be popped off the stack to be used in the operation. The stack can also be used to store the result of an operation which allows for expressions to chain together multiple operations. Another approach for VM interpretation is to use a register-based system which can take multiple parameters for each operation in the instruction. A stack-based VM’s instructions are smaller since they do not require any extra data, while a register-based VM will have longer instructions [10]. Stack-based VMs require multiple instructions to perform the same operations as a register-based VM since each parameter will need to be loaded onto the stack separately [10]. Zephyr uses a stack-based VM because code generation is simpler, the VM itself is easier to implement, and stack-based VMs are widely used in scripting systems. A register-based approach would not guarantee a more performant VM and would have extended the scope of the project to include register allocation during compilation.

## Finite State Machine AI

Finite state machines are a good fit for AI behavior and work very well in the context of a state-based DSL. The main features of an AI state are the update behavior logic and transitions between states [11]. Each state can define functions to call when transitioning out of or into the state.

## Visual Studio Code Plugin

Custom language extensions can be created for Visual Studio Code, allowing for an easier development experience for consumers of a language [12]. Static keywords can be defined easily, but highlighting variables and functions requires a mini-interpreter to be written inside the plugin to understand the language more deeply and know how to highlight. It is also possible to define an interactive programming environment and interactive debugger, but those would be too large of an undertaking for the scope of the thesis [13].

# Previous Work

*Include as many sections as necessary to describe existing implementations or games that relate to the thesis.*

## Naughty Dog’s Uncharted 2 Scripting Language

Uncharted 2’s scripting system takes a similar approach as Zephyr in defining a stateful scripting system. The core design is that each script defines a finite state machine and is associated with one game object [14]. Key functionality of the scripts includes the definition of attributes and states, an update function, the ability to respond to events, and state transitional actions [14]. The language uses a register-based VM and supports multi-threading using tracks [14]. Tracks can wait to execute until other tracks signal them which enables behavior to be synced across tracks [14].

# Out of Scope

## Interactive Debugging

Interactive debugging for script code was explored for this project, but ultimately determined to be too large of a project for the scope of the thesis. Visual Studio Code does allow for a debugger to be defined for a custom language, but it is not an easy task [13]. Entire research projects have been focused on how to create debuggers for DSLs, like a project to convert a DSL into the target language before debugging [15]. A roundtable discussion at GDC with developers who have implemented scripting systems in games confirmed that debugging tools are hard to develop, but also suggested that the types of errors typically encountered in script code could be debugged through variable inspection instead [16].

## Visual Scripting

Visual scripting was explored for the thesis but due to the complexity of implementing a UI node-based system in addition to the language features it was determined to be out of scope. Microsoft Visual Studio does provide a tool for generating graphical, node-based DSLs but that approach was complicated to use and required the DSL be defined in a specific format that would not fit Zephyr’s design [17]. The game 7 Billion Humans uses a graphical interface to solve programming puzzles and was used as an example of what could be done while investigating a visual scripting solution [18].

# Artifact

*Add sections under this heading as appropriate to describe the artifact.*

*This heading and the following three are interchangeable and optional. Choose the combination that best fits your specific thesis.*

# Implementation

*Add sections under this heading as appropriate to describe the artifact.*

## Compiler

### Scanner

* Text into tokens

### Parser

* Tokens into bytecode, saved in ZephyrDefinition
* Break down specific magic I do to make revised language work

### Error Handling

### Hot Recompiling

## Virtual Machine

### Bytecode Generation

### Stack-based Interpretation

## ZephyrObject

* State-based entity update
* How scripts use the event system

# Architecture

*Add sections under this heading as appropriate to describe the artifact.*

## Diagram

* Full compilation process
* VM process

# Syntax

* Types
* State, Function, Variable, Conversation definitions
* Screenshots of code

# Results

*Add sections under this heading as appropriate to describe any data, test results or conclusions.*

*This heading and the following three are interchangeable and optional. Choose the combination that best fits your specific thesis.*

# User Feedback

*Add sections under this heading as appropriate to describe any data, test results or conclusions.*

## LD Usage Feedback

# Profiling Results

*Add sections under this heading as appropriate to describe any data, test results or conclusions.*

* Discuss optimizations made to strings if I get time to do that

# Example Images Section

## Example Image 1

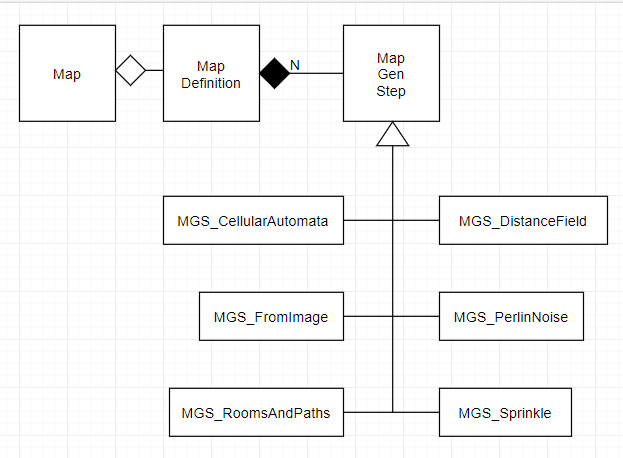


Figure 1: Example Image 1

## Example Image 2

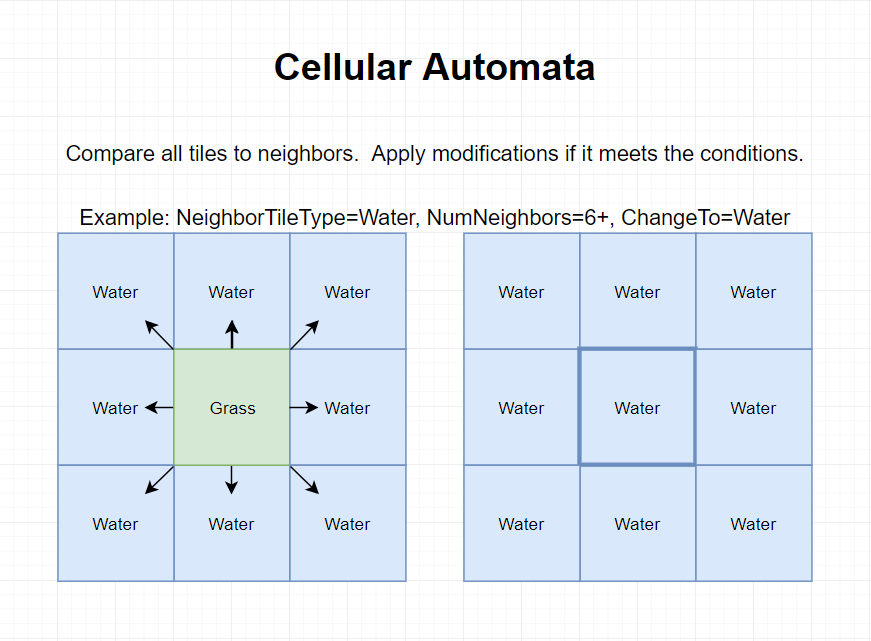


Figure 2: Example Image 2

## Example Image 3

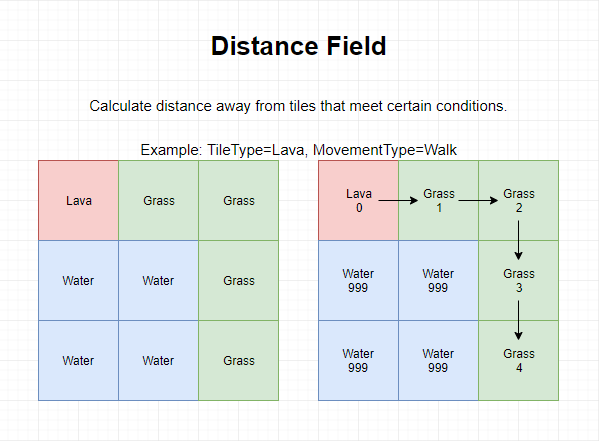


Figure 3: Example Image 3

# Bibliography

|  |  |
| --- | --- |
| [1] | M. Lewis, "Sculpting a Gameplay Scripting Solution," in *Game Developer's Conference*, San Francisco, 2016. |
| [2] | "The Legend of Zelda. [Nintendo Entertainment System]," Nintendo, Japan, 1985. |
| [3] | "Dungeon Explorer [TurboGrafx-16]," Atlus, Japan, 1989. |
| [4] | A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajama and D. Zufferey, "P: A Domain-Specific Language for Asynchronous Event-Driven Programming," July 2014. [Online]. Available: https://www.researchgate.net/publication/266660871\_P\_a\_domain-specific\_language\_for\_asynchronous\_event-driven\_programming\_invited\_talk\_abstract. [Accessed 9 July 2020]. |
| [5] | J. Gregory, "Scripting," in *Game Engine Architecture, 2nd Ed.*, Boca Raton, Taylor & Francis, 2014, pp. 954-978. |
| [6] | F. Tomasetti, "The complete guide to (external) Domain Specific Languages," 20 February 2017. [Online]. Available: https://tomassetti.me/domain-specific-languages/. [Accessed 11 June 2020]. |
| [7] | M. Mernik, J. Heering and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys,* vol. 37, no. 4, 2005. |
| [8] | M. West, "The Whimsy Of Domain-Specific Languages," Gamasutra, 3 September 2008. [Online]. Available: https://www.gamasutra.com/view/feature/130077/the\_whimsy\_of\_domainspecific\_.php. [Accessed 18 June 2020]. |
| [9] | R. Nystrom, Crafting Interpreters, 2020. |
| [10] | R. Nystrom, "Bytecode," in *Game Programming Patterns*, Coppell, TX, Genever Benning, 2014, pp. 155-180. |
| [11] | C. Berglund, "Designing a simple game AI using Finite State Machines," Gamasutra, 27 February 2017. [Online]. Available: https://www.gamasutra.com/blogs/CarlBerglund/20170227/292378/Designing\_a\_simple\_game\_AI\_using\_Finite\_State\_Machines.php. [Accessed 25 June 2020]. |
| [12] | B. Parizek, "Notes on how to create a Language Grammar and Custom Theme for a Textmate Bundle," [Online]. Available: https://benparizek.com/notebook/notes-on-how-to-create-a-language-grammar-and-custom-theme-for-a-textmate-bundle. [Accessed 10 2020]. |
| [13] | V. Kaplan, "Writing Your Own Debugger and Language Extensions with Visual Studio Code," *Code Magazine,* September/October 2018. |
| [14] | J. Gregory, "State-Based Scripting In Uncharted 2: Among Thieves," 2009. |
| [15] | H. Wu, J. Gray and M. Mernik, "Demonstration of a Domain-Specific Language," Elsevier Science B. V., Amsterdam, 2007. |
| [16] | M. Brockington, "Choosing, Designing and Implementing Scripting Languages: Tales from the Script," in *Game Developer's Conference*, San Francisco, 2001. |
| [17] | J. Partlow, "Overview of Domain-Specific Language Tools," Microsoft, 04 November 2016. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019. [Accessed 18 June 2020]. |
| [18] | "7 Billion Humans. [Nintendo Switch]," Tomorrow Corporation, United States, 2018. |