

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 |  |

Table of Contents

[Table of Figures 4](#_Toc68734781)

[Introduction 5](#_Toc68734782)

[Overview 6](#_Toc68734783)

[Scope 6](#_Toc68734784)

[End Product 6](#_Toc68734785)

[Language Features 6](#_Toc68734786)

[Game Features 6](#_Toc68734787)

[External Tools 7](#_Toc68734788)

[Development Schedule 8](#_Toc68734789)

[Schedule Detail 8](#_Toc68734790)

[Total Hours 8](#_Toc68734791)

[Technology Sources 9](#_Toc68734792)

[Visual Studio Code Plugin 9](#_Toc68734793)

[Doxygen 9](#_Toc68734794)

[Theory 10](#_Toc68734795)

[Genre 10](#_Toc68734796)

[Event System 10](#_Toc68734797)

[Domain-specific Languages 10](#_Toc68734798)

[Compiler 10](#_Toc68734799)

[Scanner 10](#_Toc68734800)

[Parser 10](#_Toc68734801)

[Virtual Machine 11](#_Toc68734802)

[Bytecode 11](#_Toc68734803)

[Finite State Machine AI 12](#_Toc68734804)

[Visual Studio Code Plugin 12](#_Toc68734805)

[Previous Work 12](#_Toc68734806)

[Naughty Dog’s Uncharted 2 Scripting Language 12](#_Toc68734807)

[Out of Scope 12](#_Toc68734808)

[Interactive Debugging 12](#_Toc68734809)

[Visual Scripting 12](#_Toc68734810)

[Artifact 13](#_Toc68734811)

[Implementation 13](#_Toc68734812)

[Compiler 13](#_Toc68734813)

[Scanner 13](#_Toc68734814)

[Parser 13](#_Toc68734815)

[Error Handling 13](#_Toc68734816)

[Hot Reloading 13](#_Toc68734817)

[Virtual Machine 14](#_Toc68734818)

[Stack-based Interpretation 14](#_Toc68734819)

[Error Handling 15](#_Toc68734820)

[ZephyrScript 15](#_Toc68734821)

[Architecture 16](#_Toc68734822)

[Syntax 17](#_Toc68734823)

[Types 17](#_Toc68734824)

[Functions 18](#_Toc68734825)

[States 18](#_Toc68734826)

[Results 19](#_Toc68734827)

[User Feedback 19](#_Toc68734828)

[Level Designer Feedback 19](#_Toc68734829)

[Profiling Results 19](#_Toc68734830)

[Bibliography 20](#_Toc68734831)

# Table of Figures

[Figure 1: Parser Error 13](#_Toc68735207)

[Figure 2: Runtime Error 15](#_Toc68735208)

[Figure 3: Architecture Overview 16](#_Toc68735209)

[Figure 4: Sample Zephyr Code 17](#_Toc68735210)

[Figure 5: Function Example 18](#_Toc68735211)

[Figure 6: State Example 18](#_Toc68735212)

# Introduction

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

Scripting languages are a common component of commercial game engines. By defining gameplay interactions in a simpler, more intuitive language, both designers and programmers can rapidly add new features to a game. Support for a scripting language requires considerable architectural work in the C++ game code in order to update entities with script behavior and expose engine functionality to the scripts. Creating Zephyr and the accompanying game was a great opportunity to explore the systems required to support a scripting language as well as gain insight into what designers need to be productive in real usage scenarios.

## Scope

The major components required for the project include: a compiler and virtual machine, designer friendly syntax, hot reloading, and a 2D, top-down, action RPG demo game. In order to use scripts in the game, the source files must be compiled into bytecode chunks and then executed by a VM. Each entity will own an object that manages the bytecode chunks for that entity and will execute the bytecode on entity update, state change or events. The entity system will be one component of the demo game, which will also provide a basic framework for an action RPG along with C++ events that are exposed to the scripts. To ensure the scripts are useful, feedback from designers will be incorporated into the syntax design process. Hot reloading for scripts will be supported via a hotkey at runtime. By allowing the scripts to be reloaded without restarting the application, iteration time of feature development can be improved.

## End Product

The final product will be composed of three major components: the Zephyr compiler and VM, a game that utilizes Zephyr scripts in its entity model, and external tools to make authoring scripts easier. Together, these features provide a framework for the rapid development of 2D, top-down, action RPGs.

### Language Features

* Designer friendly syntax
* Scanner to convert source files into a list of tokens
* Parser to convert a list of tokens into chunks of bytecode
* VM to interpret chunks of bytecode at an entity’s request

### Game Features

* ZephyrScriptDefinition objects to manage the bytecode chunks for each type of enemy
* ZephyrScript object owned by an entity to manage script states and events
* Entity system that supports updating and entity’s current state bytecode chunk as well as responding to events
* Hot recompile the scripts at runtime via a hotkey
* GameAPI events that are exposed to scripts

#### Gameplay

#### Controls

* WASD moves
* Arrow keys attack
* Space interacts with NPCs

#### Game Objects

##### Enemies

* Blob
* Splitting blob
* Golem
* Trap statues

##### NPCs

* Old man

##### Pickups

* Keys
* Triple shot power up

##### Maps

###### XML-based maps

* Define tile types
* Define tiles based on tile types
* Define maps composed of tiles and allow placement of entities in the map
* Define entity types with physics, animation, and script information

##### User Interface

* Current quest shown in top left of screen

### External Tools

* Doxygen docs to document all GameAPI events
* Visual Studio Code plugin with syntax highlighting

# Development Schedule

## Schedule Detail

|  |  |  |
| --- | --- | --- |
| Milestone | End Date | Task Hours |
| 1 | 09/09/20 | 20 |
| 2 | 09/23/20 | 18 |
| 3 | 10/07/20 | 18 |
| 4 | 10/21/20 | 20 |
| 5 | 11/04/20 | 18 |
| 6 | 11/18/20 | 8 |
| 7 | 12/01/20 | 6 |
| Winter Break | 01/28/20 | 30 |
| 8 | 02/25/20 | 13 |
| 9 | 03/04/20 | 14 |
| 10 | 03/18/20 | 11 |
| 11 | 04/01/20 | 13 |
| 12 | 04/15/20 | 4 (+5 for work on DFS II) |
| 13 | 04/29/20 |  |
| 14 | 05/04/20 |  |
| Total |  | **198** |

## Total Hours

In total 250 hours were spent on development tasks, research, and documentation.

# Technology Sources

## Visual Studio Code Plugin

Visual Studio Code was used to generate a plugin that allowed syntax highlighting for Zephyr scripts in the VS Code editor. A language template was used as the basis for the plugin which allowed highlighting for key words that are defined in Zephyr.

## Doxygen

Doxygen was used to generate html documentation for the engine methods exposed to script developers via the GameAPI interface.

# Theory

## 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

## 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

# Implementation

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

## Compiler

### Scanner

The scanner reads in a source file as a string and processes each character using a single token lookahead approach. Only the current token and the next token need to be known in order to classify and save each token type. Tokens hold an enum for their type, a string containing the data in the source code that corresponds to that token, and the line number of the source file it was found on.

### Parser

The parser iterates through the list of tokens generated by the scanner and generates bytecode chunks for each state and function definition. Each Zephyr script is converted into a ZephyrScriptDefinition object which holds the global state bytecode chunk along with each state bytecode chunk. Each state chunk contains a map of the event bytecode chunks that are defined in that state.

### Error Handling

Parser errors are reported to the dev console with the line number and error message. If an entity’s script contains an error, the script is marked as invalid and that entity will have “Script Error” displayed on it in the world and no update or event bytecode chunks will be executed for that entity.

A picture containing graphical user interface

Description automatically generated

Figure : Parser Error

### Hot Reloading

All data files including Zephyr scripts, XML maps, XML entity definitions, audio, sprites, and the XML game configuration file can be reloaded without closing the application by pressing F5. Hot reloading deletes all data definition objects and generates them again in order to respect any dependencies amongst the different files. F6 will reload just the Zephyr scripts, which is useful for modifying an entity’s AI without resetting the level. Using the script only reload can break functionality that depends on entity spawning, since the entities are not reloaded with this operation, so it must be used with caution while the safer reloading method is to reload all data files instead.

## Virtual Machine

### Stack-based Interpretation

When the VM interprets a bytecode chunk, all temporary constant values are pushed into a stack that later operations can pop to retrieve the constants. A variant data type containing the variable type as well as a union of all supported Zephyr types, ZephyrValue, is used for the constant stack data. Any code that needs to save information for future operations like variable assignment expressions, function calls, and if statements push that data onto the stack as well. As an example, each parameter in a function call as well as the total number of parameters is pushed onto the stack when interpreting the parameter definitions, and then the function call operation can use that data to know exactly how many parameters to pass into a function.

### Error Handling

Types are checked by the interpreter when performing operations. If an invalid operation is requested, an error will be printed to the dev console and that entity’s script will be placed into an invalid state. Since each entity errors individually, two entities of the same type with the same error will only display the error at the point that each of them runs into the problematic code.

Chart

Description automatically generated

Figure : Runtime Error

## ZephyrScript

Each entity can own zero or one ZephyrScript object. The ZephyrObject handles all script updates and event calls. Upon entering a state, any functions that were declared in the old state will be unregistered for the entity in the event system. Then, any functions that are declared in the new state will have a subscription added for the entity. Whenever that event is fired by another ZephyrScript object or the game engine, the event’s bytecode chunk will be executed by the VM.

# Architecture

The diagram below shows how Zephyr scripts can interact with each other and with the game engine.

1. A Zephyr script is compiled into bytecode chunks.
2. When the entity updates or receives an event, the corresponding bytecode is interpreted by the VM.
   1. Function call – fires an event either in another ZephyrObject or one of the events exposed to the scripts from the engine in the GameAPI file.
   2. ChangeState – changes the entity’s current state, calling any OnEnter and OnExit functions defined for that state change.

Diagram

Description automatically generated

Figure : Architecture Overview

# Syntax

## Types

The syntax of Zephyr was designed to resemble existing scripting languages in order to be easier for programmers and designers to transition to Zephyr from other languages. The language contains the types Number, String, Bool, Vec2, and Entity. Any type can be converted to Bool or String during operations involving those types, but valid operations between other types are defined on a case-by-case basis. The language is dynamically typed since the types are checked at runtime. The main reason for this is to support Entity objects. Entity variables are stored as the id of the entity and any accessor looks up the requested member at runtime to see if it exists for the entity. There is no way to define subclasses in Zephyr, so the compiler cannot know if a defined Entity contains the members at compile time and must instead wait until runtime when the entity has been initialized to tell which expressions are valid.

Text

Description automatically generated

Figure : Sample Zephyr Code

## Functions

Functions can be defined either inside a state or in the global state. Functions are implemented using the event system. A function definition registers that function name with the event system, and a function call fires an event with the provided parameters as the event arguments. Upon receiving a subscribed event, an entity will execute the bytecode chunk associated with the event. Variables are passed into functions by reference, allowing for data to be returned from a function call. The one exception to this is member variables for Vec2 and Entity, which are passed by value instead.

Text

Description automatically generated

Figure : Function Example

## States

A state can only be defined in the global state. It is not valid to define substates inside a state definition. Each state has built in functions OnEnter, OnExit, OnUpdate which are called when entering, exiting, or when the parent entity updates. ChangeState is used to transition to a new state in a Zephyr script.

Text

Description automatically generated

Figure : State Example

# Results

# User Feedback

## Level Designer Feedback

Level designers were consulted on the syntax of Zephyr scripts and provided valuable insights that were integrated into the final artifact. The largest change was made after multiple designers agreed that communication between scripts was one of the largest pain points they had while using scripting engines. The point about communication led to the addition of an Entity built-in type and the ability to directly access variables and call methods on entity variables. Designers were later given the scripts and an .exe of the demo game to modify. The feedback sessions brought attention to a collection of bugs found by exercising the code in new ways as well as some small annoyances that were improved upon. The overall takeaway from the feedback sessions was that the language was intuitive to use and could be picked up from skimming the code examples present in the demo game.

## Diablo’s Gate

Zephyr scripts were used in the development of another project, Diablo’s Gate, which is a Diablo style action RPG. The game uses the mouse to control player movement, which enemies to attack and items to collect. The player also has special moves that can be activated by clicking on a button in a UI HUD. By adding new functionality to GameAPI and sending click events to the scripts, it was possible to develop entity AI and the combat system in Zephyr scripts. The ability to use scripts along with hot reloading made development of game features much faster.

# Profiling Results

A test level was used to record the time required to update all entities in the map containing the player and various amounts of enemy blobs. Each enemy and the player were firing events on their update steps, while the blobs were colliding with each other and the level walls and changing states when seeing the player approach or leave their chase areas. This test exercised every main feature of the language.

|  |  |
| --- | --- |
| # of Entities | Update Time (ms) |
| 1 | 0 |
| 11 | 1.2 |
| 21 | 3.3 |
| 31 | 6.3 |
| 41 | 10 |
| 51 | 14.7 |
| 101 | 62.6 |

# 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. |