A Visual Programming Environment for Authoring ASD Therapy Tools

by

Mwawî Fred Msiska

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Stellenbosch University

Supervisor: Prof. L van Zijl

November 2011

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: .....

M.F. Msiska

Date: 2011/11/21 .....

Abstract

3D virtual environments can be used as therapy tools in patients with autism spectrum disorders (ASDs); however, the development of such tools is time-consuming. A 3D virtual environment development platform for such tools has been developed specifically for the South African context, because of the language and culture sensitivity of these therapy tools.

The 3D virtual environment development platform has a Lua scripting interface for specifying logic in the virtual environments. Lua is a textual programming language, and presents a challenge to ASDs therapists' ability to create therapy tools without engaging an expert programmer.

The aim of this research was to investigate the design and implementation of a visual programming environment to support non-expert programmers in scripting within the 3D virtual environment development platform.

Various visual program representation techniques, reported in the literature, were examined to determine their appropriateness for adoption in our design. A visual programming language based on the "building-block" approach was considered the most suitable. The research resulted in the development of a visual script editor (VSE), based on an open source framework called the OpenBlocks library.

The VSE successfully alleviated the syntax burden that textual programming languages place on non-expert programmers. The fitness of purpose of our VSE was exemplified in a sample 3D virtual environment that was scripted using the VSE. Despite the success, we argue that the applicability of the "building-block" approach is limited to domain-specific programming languages due to the absence of visual expressions for defining user-defined types, and for specifying hierarchy.

Acknowledgements

I owe my deepest gratitude to my supervisor, Prof. Lynette van Zijl, for her support, mentorship and patience throughout the research work that culminated into this thesis.

I would also like to thank Daniel Wendel for providing useful information on the OpenBlocks library. This information was crucial to the understanding of the OpenBlocks API.

Finally, it is also an honour for me to thank the following people: my parents for their encouragement and their undying faith in me; my wife, Grace, for her understanding and encouragement during the research; and my daughter, Sangwani, for sacrificing daddy's attention.

Contents

List of Figures

Chapter 1

Introduction

This thesis investigates the design and implementation of a visual programming environment for authoring 3D virtual environments that can be used as learning aids; and as ASDs therapy tools. The visual programming environment is necessitated by the desire to enable ASDs therapists to build therapy tools, since they are endowed with domain knowledge. The research has led to the development of the visual script editor (VSE), a prototype system for graphical programming of virtual environments.

1.1 Background

Newschaffer *et al* [50] define autism spectrum disorders (ASDs) as "a group of neurodevelopmental disorders characterized by core deficits in three domains: social interaction, communication, and repetitive or stereotypic behaviour." Prevalence rates as high as 110 per 10000 have been reported in the literature [18, 42].

The use of computerised tools has been shown to have great potential in mitigating the learning difficulties associated with people with ASDs [30, 48]. Virtual environments (VEs), in particular, are among the most promising in this regard [51, 52, 57]. Chamberlain [62] argues that computerised ASDs therapy tools have language and cultural dependencies; and consequently, motivates the need for a tool to support the development of therapy tools specific to the South African context.

The work documented in this thesis is part of the broader ASD Assist project at

the Computer Science department at Stellenbosch University. The main aim of the ASD Assist project is to build software which can be used as ASDs therapy tools. The therapy tools in the project are centred on a 3D game engine, called Myoushu, which was developed by Chamberlain [62].

Myoushu is essentially a free and open-source 3D game engine, based on the Object-oriented Graphics Rendering Engine (OGRE) [60], that can be used to develop 3D VEs which can be used as teaching aids and ASDs therapy tools [62]. The Myoushu game engine provides a Lua [37] scripting interface which offers a layer of abstraction from the engine's low level C/C++ implementation. It also includes a library for importing content from scene files exported from Blender [10]. This allows for rapid development of VEs by importing prebuilt scenes and then using Lua to add logic to the scenes.

1.2 Research Objective

The inclusion of a Lua scripting interface in the Myoushu game engine is unarguably a step towards lowering barriers to the development of VEs based on the game engine. Lua, being a textual programming language, still presents a challenge to people with limited programming experience. The main aim of this thesis is to investigate methods to lower the VEs development barrier further, in order to accommodate novice programmers. We focus on the addition of a simpler programming interface, the VSE, on top of the Lua scripting interface.

The Myoushu game engine includes a library of Lua functions, which can be used through the Lua scripting interface. Since functions can be added to and removed from the library as the game engine evolves, the VSE has to be flexible enough to accommodate such changes to the library.

The other aspects of the game generation process, namely the design of a storyline and the construction of scene artefacts like 3D models and animations, are beyond the scope of our investigation. The task of creating a storyline is better left to the ASDs therapist, the domain expert. There already exist tools for visual creation of 3D models and scenes. One such tool is the open-source Blender [10]. We encourage

its use because the Myoushu game engine is capable of importing scenes exported from Blender.

1.3 Motivation

The development of VEs is time consuming [62] and expensive; and furthermore, if the VEs are to be used for ASDs therapy, then there is need for the VEs to be adapted to the specific needs of each individual patient. To cut the development time and budget for the ASDs therapy tools, we propose that a repository of VE artefacts[1] be made available; and then give the responsibility of authoring the final therapy tools to the therapist. Since the therapist may not be a programmer (and should not be forced to become one), there is need to simplify the scripting interface.

We do not replace the Lua programming interface in the Myoushu game engine with the VSE; rather, the VSE translates the graphical programs into Lua. This setup not only takes the burden of the final object code optimisation from the VSE, but also affords the possibility for expert Lua programmers to textually edit programs, created by non-Lua programmers, in order to add advanced features to the VEs.

1.4 Approach

In the pursuit of the research objective, various possibilities are considered; from fully-fledged visual programming languages to simple game creation tools. This investigation guides the formulation of constraints on the VSE, which include: the type of visual programming language to use; the extent to which the visual programming paradigm is used; and the appropriate level of abstraction.

Roque [55] states that the development of a graphical programming environment is time consuming. Because of this, and the limited time frame in which the thesis had to be completed, we explore the prospect of extending an existing open-source framework in constructing our prototype graphical programming environment. The OpenBlocks [55] framework is adopted, and is discussed further in Chapter 3.

[1] Examples of which include 3D models and prebuilt scenes (without logic).

We build on the OpenBlocks framework and realize our visual programming language, which forms the basis of the VSE. The visual programming language is designed to mirror traditional textual programming languages in the sense of having a fixed core; and being extendable through the addition of third party libraries.

The use of OpenBlocks also enables us to quickly evaluate different visual element designs using traditional user interface evaluation techniques.

1.5 Terminological Note

In this thesis, we use the words *program* and *script* interchangeably to mean a simple program spanning no more than a thousand lines of source instructions. In the same vein, the words *programming* and *scripting* are used interchangeably.

We also use the words *user*, *programmer* and the phrase *"novice programmer"* to mean a person with limited or no programming experience, for whom the VSE is intended.

1.6 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents an overview of visual programming; and its applicability to the problem investigated in this thesis. Several approaches to visual programming are explored; and suitability for adoption in our VSE is examined. We then look at existing visual VE authoring tools that inspired our work. The design and implementation of the VSE is described in Chapter 3. Chapter 4 discusses the creation of scripts, for an educational game, using the VSE to demonstrate its capabilities. Chapter 5 discusses the extent to which the research objective, outlined in Section 1.2, is met. We also discuss the shortfalls of the VSE and suggest remedies. The chapter ends with a summary of general conclusions drawn from the research.

Chapter 2

Literature Survey

There is a plethora of literature on visual programming languages and systems. Our investigation focused on the identification of the available options regarding visual representation of programs. We therefore do not expend much effort in describing individual visual programming languages or systems in detail; rather, we identify and discuss categories of visual program representation techniques.

The chapter begins with a description of the visual programming paradigm; and the relationship between a visual programming language (VPL) and a visual programming environment (VPE), in Section 2.1. The section also discusses the goals of the visual programming concept; and the strategies that can be used to realize these goals. Section 2.2 presents a summary of visual program representation techniques that have been reported in the literature. We conclude Section 2.2 with a brief discussion on the suitability of each technique as a solution to our problem. Section 2.3 discusses three examples of VPEs, for authoring games, whose visual representation technique is similar to the one adopted in the VSE. We end the chapter with a chapter summary in Section 2.4.

2.1 Overview of Visual Programming

Visual programming is programming in which semantics is expressed using more than one dimension [22, 49]. Textual programming is considered one-dimensional in the sense that a program is composed of only linear strings. Typing the pro-

gram statements or expressions on multiple lines does not introduce an additional dimension since no additional semantics are implied by the line breaks [22].

A visual programming language (VPL) does not necessarily exclude the use of text. Text is permitted, but in a multidimensional context [22]. The only constraint on a VPL is that a significant portion of the semantics of the language should be expressed in other dimensions. Text is still the most natural medium for expressing entities such as names of objects and numeric values.

Examples of the additional dimensions available to a visual programming language include: 2D or 3D images, which can be used to represent an entity, an expression or a command; relative positions among objects can be used to represent relationships among entities, expressions and commands; and the dimension of time can be used to specify the chronological order of events in the programming by demonstration approach.

The term VPL is reserved for multidimensional programming languages in which the visual expressions can be directly compiled (or interpreted) into executable binary code, without first converting the visual program into a human-readable (and editable) textual programming language [22]. If the visual expressions are first converted into a traditional textual programming language before final compilation (or interpretation) into an executable binary, then the configuration is only referred to as a visual programming environment (VPE).

Graphical user interface (GUI) builders like the ones used in Microsoft Visual Studio [11] and NetBeans' Swing GUI Builder [15] cannot be classified as VPEs. In such systems, the use of visual expressions is limited to the specification of static GUIs. Thus only the compositional aspect of programming is used. The other two aspects, namely selection and iteration, are not used; and consequently, these GUI builders are not programming languages.

Burnett [22] identifies the three major goals of VPL research, namely: "to make programming more understandable to some particular audience"; to reduce human effort in ensuring syntactic correctness of programs; and to reduce program development time. Burnett [22] states that VPLs use the following four strategies in order to achieve these goals:

1. *Concreteness.* Concreteness is the opposite of abstractness. An expression is concrete if it specifies some aspect of the program using a specific instance, rather than an abstraction. Green and Petre [31] state that learning to think in abstract terms is a significant cognitive accomplishment; and consequently, programming systems that demand a lot of abstractions do not motivate potential users. In addition, such systems force users to first define the required abstractions, which can take a significant portion of the total solution time, before they can start thinking about the solution; thus the users suffer from delayed fulfilment.

2. *Directness.* Directness is the distance between a user's perception of a real world problem and its representation in a program. The smaller the gap, the easier it should be for the real world problem's domain expert to solve the corresponding programming problem by drawing insight from the real world problem [31].

3. *Explicitness.* An explicit expression directly states its semantics without the need for the reader to infer it. For example, promotion of lower precision values to higher precision in an algebraic expression in Java is not explicit; on the other hand, directed edges explicitly depict control flow in flowcharts.

4. *Immediate visual feedback.* Burnett [22] defines immediate visual feedback as "the automatic display of effects of program edits." Tanimoto [61] conceived the term *liveness* to describe the immediacy of feedback in visual programming systems. Tanimoto proposes four levels of liveness. At liveness level 1, there is no feedback. The visual representation of the program[1] merely documents a program. At level 2, the visual program displays feedback, but on request. At level 3, program edits that change the semantics of the program automatically trigger computation; thus the feedback is immediate. At level 4, the system is continually live; and thus there is immediate feedback due to user edits, as well as other system events.

[1]Note the use of the phrase "visual representation of the program." This is not a visual program in the sense that the visual representation is not interpretable into executable binary form.

The four strategies listed above are not always desirable in their entirety. Concreteness and directness can sometimes be in conflict; and as such tradeoffs need to be adopted. For example, lowering the level of concreteness (increasing abstraction) can result into improved directness [31]. Although liveness levels 3 and 4 seem attractive, they are not critical unless the VPL is intended to be used in the context of a live system[2] that is processing time-varying data [61]. Liveness level 2 is therefore sufficient for our VSE.

2.2 Visual Program Representation Techniques

Our categorization of visual program representation techniques is primarily based on the visual programming language classification system devised by Burnett and Baker [23]; and a classification of programming systems by specification style by Myers [49]. Burnett and Baker identify three broad subcategories under "visual representations" namely: diagrammatic languages, iconic languages, and languages based on static pictorial sequences. We found this classification of visual program representation techniques insufficient since form-based, spreadsheet-based and demonstrational techniques would not fit under any subcategory. On the other hand, certain categories in Myers' classification could be collapsed into one; for example, data-flow graphs and directed graphs are special cases of dataflow VPLs.

Our categorization takes the subcategories under "paradigms" in [23] and filters out the subcategories which also apply to textual programming languages. Iconic and jigsaw puzzle pieces [49] languages are added to the remaining subcategories. We verified the new categorization by ensuring that all categories (or subcategories), based on visual program representation, presented in both [23] and [49] are assigned to a category under our new categorization. The resulting five categories are:

1. Form-based and Spreadsheet Languages;
2. Dataflow Languages;
3. Iconic Languages;

[2] A system that is running – its logic has to be updated without restarting it.

4. Programming by Demonstration; and

5. Building-block Programming.

These categories are discussed in Sections 2.2.1 to 2.2.5.

2.2.1 Form-based and Spreadsheet Languages

Form-based VPLs express a program using a set of forms. A form is a uniquely identifiable set of cells. A cell is the smallest program unit; and can hold a value or a formula. A form-based computer language qualifies as a programming language if it supports composition[3], selection[4] and iteration. This requirement excludes the *property sheets* used in simple user interface builders (like the ones used in Microsoft Visual Studio and NetBeans GUI Builder) from the category.

Form-based VPLs are the most basic form of visual programming languages [25]. As the name suggests, form-based VPLs use the metaphor of a paper-based form. It is thus not surprising that all the examples that we found in the literature express their semantics declaratively.

Form-based languages have been grouped together with spreadsheet languages because a spreadsheet, which is a tabular structure, can be considered a restricted type of a form [25]. In pure spreadsheet languages, all the cells on the form have the same geometry; are arranged in a rectangular grid; and nested cells are not allowed. In addition, Rothermel *et al* [56] and Burnett *et al* [21, 24] report that pure spreadsheets are constrained by Alan Kay's *value rule*, which states that a cell's value is solely determined by the value or formula explicitly given to it by the user [39].

One of the earliest examples of a form-based VPL is Query-By-Example [64]. The VPL supports data definition and manipulation in a relational database. A query is built by creating a skeleton table and filling it with an example solution. Example solutions are constructed using example elements[5] and constant elements[6]. The

[3]Combining several functions or computation steps into a more complex one.
[4]Conditional branching; for example, an *if* statement.
[5]A value that is not necessarily in the database.
[6]A value that must be in the database.

two types of elements are visually distinguished by underlining the latter. The language is capable of visually expressing complex SQL queries involving multiple tables.

Financial spreadsheet applications are also an example of form-based languages; however, most of them are not pure spreadsheet languages because of the use of tools such as macros which violate the value rule [24]. In financial spreadsheets, the typical content of a cell is textual (including values and formulas that reference other cells), although graphics are used for data visualisation. Some spreadsheet VPLs accept graphics as cell contents. Examples of such languages are Forms/3 [21, 24] and Penguims [34].

2.2.2 Dataflow Languages

Dataflow VPLs use a graphical representation of a dataflow execution model to express the semantics of a program. The graphical representation typically takes the form of a directed graph in which nodes represent functions, and the edges represent the flow of data [32, 38].

Johnston [38] describes a popular pure dataflow execution model called the *token-based dataflow model*. In this model, a node represents a function; and an edge represents the flow of a data token. The model does not specify any order of execution; instead, a node becomes active when it receives a token from each incoming edge. Once a node is activated, it consumes the tokens; uses the tokens in its computation; and finally generates a set of new tokens which are placed on its outgoing edges. Each edge transfers at most one token at any given time. A node becomes inactive as soon as it places output tokens on its outgoing edges. An edge may fork into several edges; and in such a case, a token flowing in the original edge is duplicated such that each child edge carries a single copy of the original token. Several edges cannot, however, combine into one edge.

A pure token-based dataflow execution model explicitly supports (parallel) composition; and iteration through cyclic directed graphs. The model does not explicitly support selection.

Johnston [38] describes an extension, the *gates* concept, to the pure dataflow model

that can be used to explicitly express selection. There are two kinds of gates, the *merge* gate and the *switch* gate. The two gates are shown in Figure 2.1.

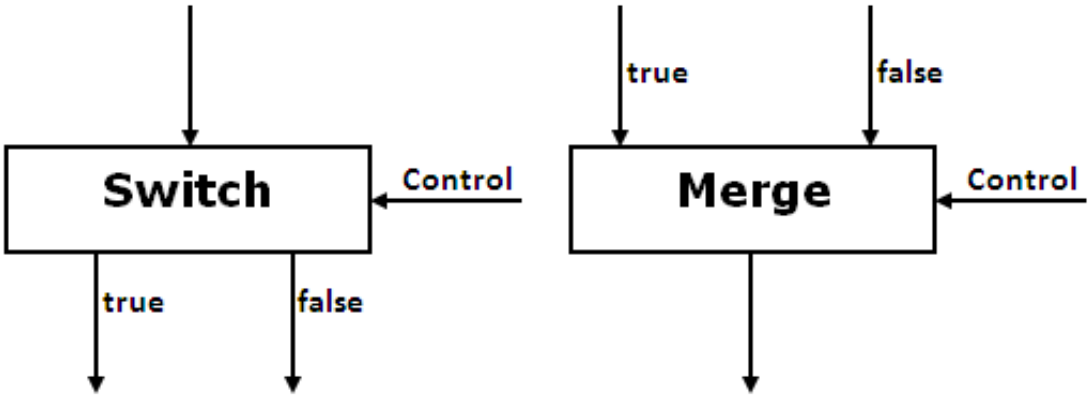

Figure 2.1: Switch and Merge nodes (adapted from [38], page 4)

The switch gate accepts a single token and a *control* token. The control token is a Boolean and can assume either of the values *true* or *false*. When the control token is *true*, the data token is propagated to the "true" outgoing edge; and when the control token is *false*, the data token is propagated to the "false" outgoing edge.

The merge gate has a control incoming edge and two other incoming edges labelled "true" and "false." When the control token is *true*, the data token on the "true" incoming edge is propagated to the outgoing edge; and when the control token is *false*, the data token on the "false" incoming edge is propagated to the outgoing edge.

Hils [32] presents a survey of dataflow VPLs. The VPLs surveyed range from domain-specific VPLs to general purpose VPLs.

2.2.3 Iconic Languages

An icon is a sign that resembles some world object. The icon and the world object share some common visual properties such shape and colour; however, the icon may be an abstracted version of the real world object and some properties like size are usually different [40].

Not all visual languages that use icons are iconic. A visual language is iconic if it expresses semantics using *iconic sentences*. Myers [49] defines an iconic sentence as a visual sentence that is composed of icons in which relative positions of icons convey additional semantics regardless of whether the icons are connected.

We identified two subcategories of iconic languages namely: *graphical rewrite rules* and *comic strip programming*; which are briefly discussed in Sections 2.2.3.1 and 2.2.3.2 respectively.

2.2.3.1 Graphical Rewrite Rules

Graphical rewrite rule systems typically have a grid of rectangular cells on which icons can be placed. A statement in such a system takes the form of a *before-after* rule. The *before* expression is an iconic sentence representing a situation (placement of runtime icons) that the processor can recognize at runtime. The *after* expression is also an iconic sentence involving, among others, a subset of the icons in the *before* expression. New icons can be introduced in the *after* expression. The *after* expression is a runtime placement of icons that replaces the *before* placement when the processor recognizes the latter placement.

In graphical rewrite rule systems, an icon forming part of an iconic sentence refers to all instances of the same icon at runtime. This implies a high level of *directness*. It also implies a high level of *directness* since runtime icons are represented by similar icon instances in iconic sentences, rather than some abstraction.

An example of a visual programming system that uses graphical rewrite rules is Stagecast Creator [8].[7] Figure 2.2 shows an example of Stagecast Creator's graphical rewrite rules. The rule states that when the humanoid character is in a cell whose right-hand side adjacent cell contains an ice cube, the humanoid character must step on top of the cube. In some cases, characters can have multiple appearances[8]. In such cases, both relative position and appearance are meaningful. For example, if the humanoid character had multiple appearances, then the rule in Figure 2.2 would be interpreted as, "when the humanoid character is in a cell whose

[7]Stagecast Creator also uses programming by demonstration and form-based programming in addition to the graphical rewrite rules.

[8]Each appearance is actually a different icon.

right-hand side adjacent cell contains an ice cube; and the humanoid character is facing right; then the humanoid character must step on top of the cube."

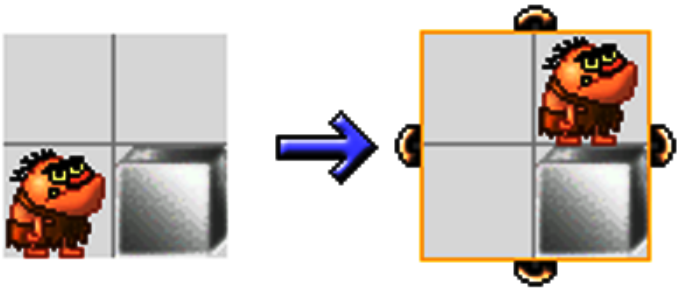

Figure 2.2: A screenshot from Stagecast Creator [8] showing a graphical rewrite rule

Kindborg and McGee [40] name three limitations of graphical rewrite rules; and propose a new approach to iconic programming called *comic strip programming*. These limitations include:

1. *Limited graphical programming expressiveness.* The weakness stems partly from the high level of concreteness and directness inherent in graphical rewrite rules. Although the concept of classes is implicit in iconic sentences, there is no concept of inheritance; and as such the specification of general rules becomes verbose. For example, we need a rule for each icon representing a different kind of animal to state that an animal drops to the ground when it is in the air.

   In general, the before and after pictures of an iconic sentence are restricted to refer to the same location of the world. In addition, the icons appearing in the before and after pictures must be the same, except in the cases of creation or deletion of icons. These restrictions make it difficult to express rules relating two icons in isolated locations. For example, rules such as, "when the alarm goes off, every person should vacate the building," are almost inconceivable in graphical rewrite rule systems;

2. *Counter-intuitive programming representation.* Rader *et al* [53] recognize the inadequacy of before-after rules in conveying semantics. The before and after

pictures obfuscate the event that occurred in between. For example, no single iconic sentence can represent a rule that says a dog standing next to a table should jump and land in the same location, and facing the same direction as before, since the before and after pictures would be exactly the same.

In some cases, it may be difficult to notice the difference between the before and after pictures [53]. For instance, consider the move-left rule shown in Figure 2.3. Rader *et al* [53] suggest that the change is not obvious because most people are accustomed to reading from left to right and the icon motion goes against the arrow.

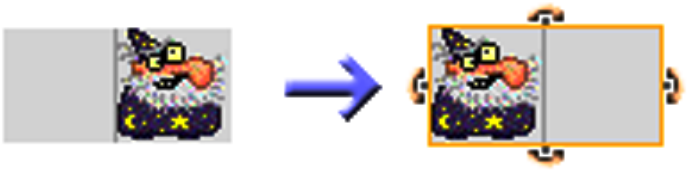

Figure 2.3: A screenshot from Stagecast Creator [8] showing the subtleness of a "move left" rule

Seals *et al* [58] observe a problem emanating from the sensitivity of graphical rewrite rules to the context. Figure 2.4 illustrates attempts to make a rule that moves a character diagonally. Although the relative positions of the humanoid character is the same in the before and after pictures in both rules, the semantics of the rules are different. The first rule says that the character must move to the top of the block if the character is immediately to the left of, and facing the block. The second rule says that the character should move to the top-right adjacent cell, if that cell contains no other object;

3. *Rigid-world appearance and dynamics.* Most graphical rewrite rule systems are grid-based. Although gridlines are typically not displayed at runtime, rule matching is performed at grid-cell level. The processor checks for the presence or absence of a particular icon in a particular cell. There is only one possible placement of an icon within a cell. This implies that each scene has only a discrete number of locations in which icons can be placed; and consequently, icons cannot overlap (except in cases when two icons, with

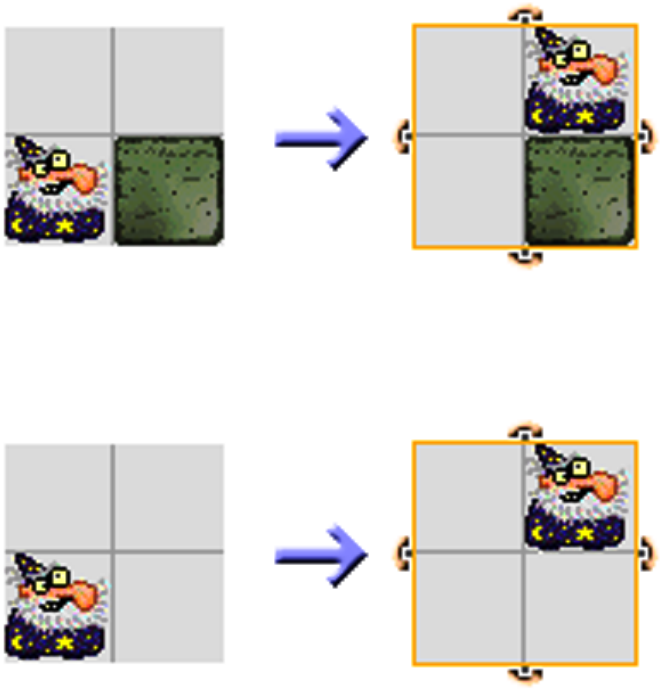

Figure 2.4: Screenshots from Stagecast Creator [8] showing context sensitivity of graphical rewrite rules

different geometries, are placed in the same cell) and object motion is not smooth.

2.2.3.2 Comic Strip Programming

Kindborg and McGee [40] proposed a new iconic programming technique that addresses the limitations of graphical rewrite rules. Their technique is called *comic strip programming* and is inspired by the comics' ability to represent a dynamic world using static pictures or drawings.

Comic strip programming relaxes the graphical rewrite rule system's requirement that all icons appearing in the before picture must also appear in the after picture unless creation or deletion is implied. This relaxation simplifies the specification of rules such as, "when a humanoid character touches a ghost character, the ghost disappears." Figure 2.5 shows a comic strip iconic sentence representing the rule, "when a humanoid character touches a ghost character, the ghost disappears." This

is not a graphical rewrite rule because of four reasons:

1. The grid-based nature of graphical rewrite rule systems cannot express the *touch* action since each icon is constrained to occupy only the centre of a cell;
2. There is an explicit expression for deletion – the × icon superimposed on the icon to be deleted;
3. The absence of the humanoid icon in the "this happens" picture does not imply the deletion of the icon; rather the icon has been omitted for brevity since the deletion of the ghost character does not affect the humanoid character in the "this happens" picture.
4. The *touch* condition is general in comic strip programming – the relative position of the two icons does not matter as long as the two icons touch each other. If the touch condition was expressible in a graphical rewrite rule system, then multiple rules would be required to capture all the possible relative position of the touching icons.

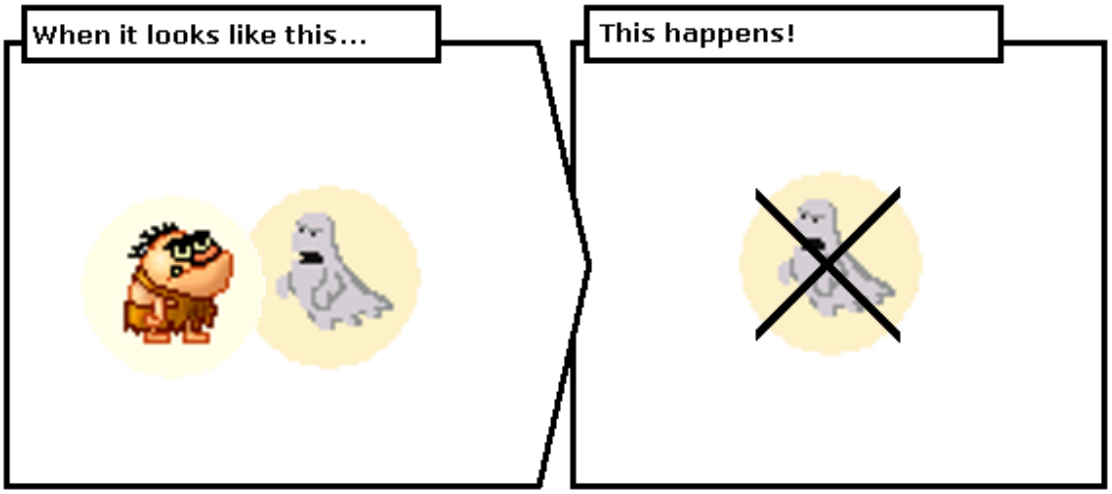

Figure 2.5: An example of a comic strip iconic sentence (adapted from [40], page 113)

The graphical rewrite rule systems' tenet that the before and after pictures must show the same location in the world is also relaxed in comic strip programming. This enables the expression of rules involving icons at different locations in the

world. Such expressions are problematic in graphical rewrite rule systems. Figure 2.6 is an example of such rules given by Kindborg and McGee [40]. The sentence states that when there is a shining sun anywhere in the world, then any instance of the character smiles[9]. Figure 2.7 shows an example of a rule which has more than

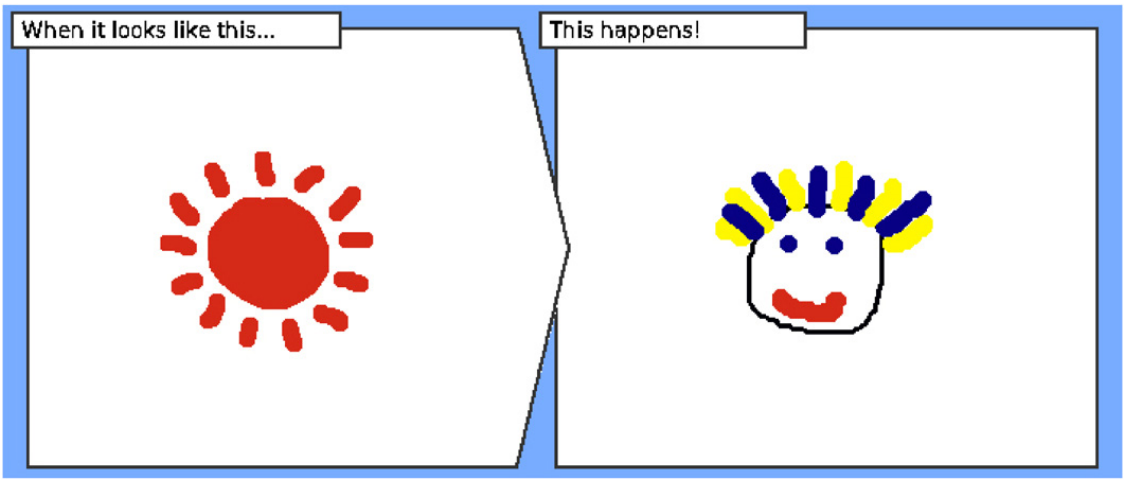

Figure 2.6: Rule involving objects at different locations (reprinted from [40], page 114)

one icon in the preconditions which may not be in the same location. The isolated icon, the sun, is placed in an inner panel.

Comic strip programming uses contextual signs to express actions or aspects of the world that do not have (and should not have) iconic representation at runtime. For instance, a rule such as "when the user presses the left arrow keyboard key, the character must move left". Such a rule is inexpressible in a graphical rewrite rule system because although we can use an iconic representation of a keyboard key when specifying the rule, the rule cannot match any situation at runtime since there should not be a keyboard key icon in the world at runtime. Figure 2.8 shows an example of a keyboard rule in comic strip programming. The rule states that when the left arrow keyboard key is pressed, the dog must move to the left. Note the use of a "ghost image" contextual sign to signify that the dog is moving to the left.

[9]Note that the character in this case has multiple appearances that are depicted using different icons.

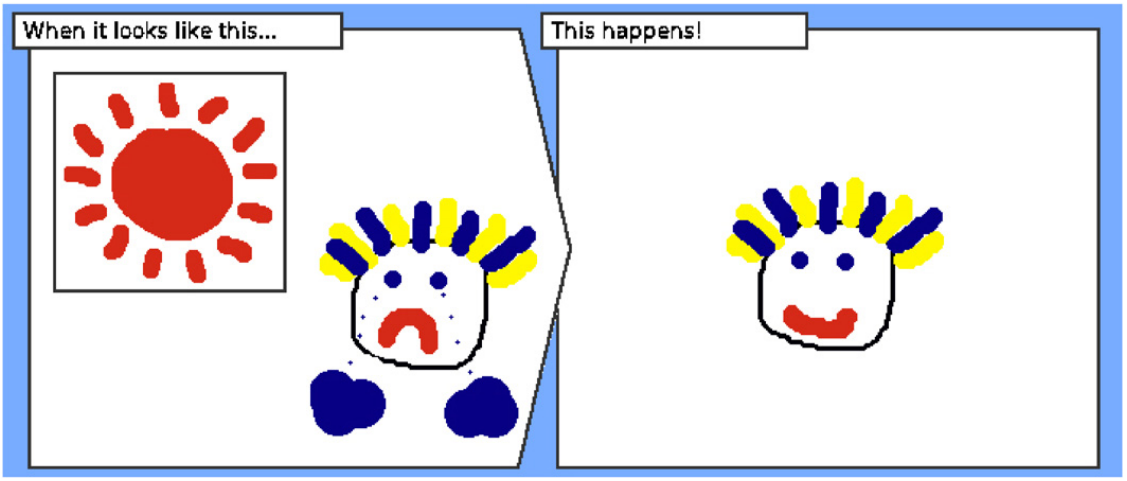

Figure 2.7: The use of an inner panel to express spatial separation (reprinted from [40], page 114)

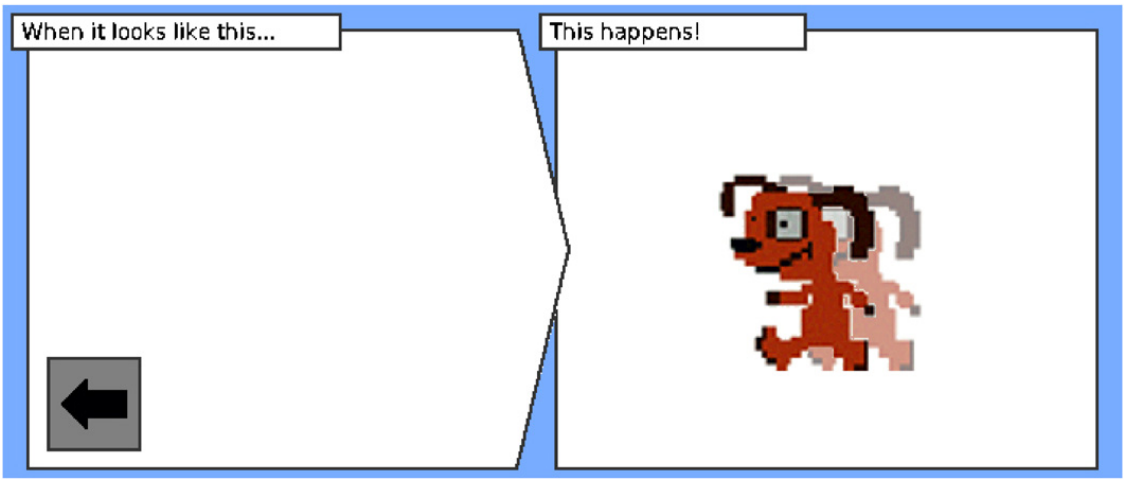

Figure 2.8: A keyboard rule (reprinted from [40], page 115)

Contextual signs can also be used to ease the "rigid-world appearance and dynamics" problem mentioned in Section 2.2.3.1. For example, Figure 2.9 shows a rule that says that if the dog is facing left and 0.5 seconds have elapsed since the last time the rule was evaluated, the dog must move to the left. The system timer icon in this case can be used to control the animation speed more flexibly than is possible in a graphical rewrite rule system.

Finally, comic strip programming is not grid-based; and consequently allows ar-

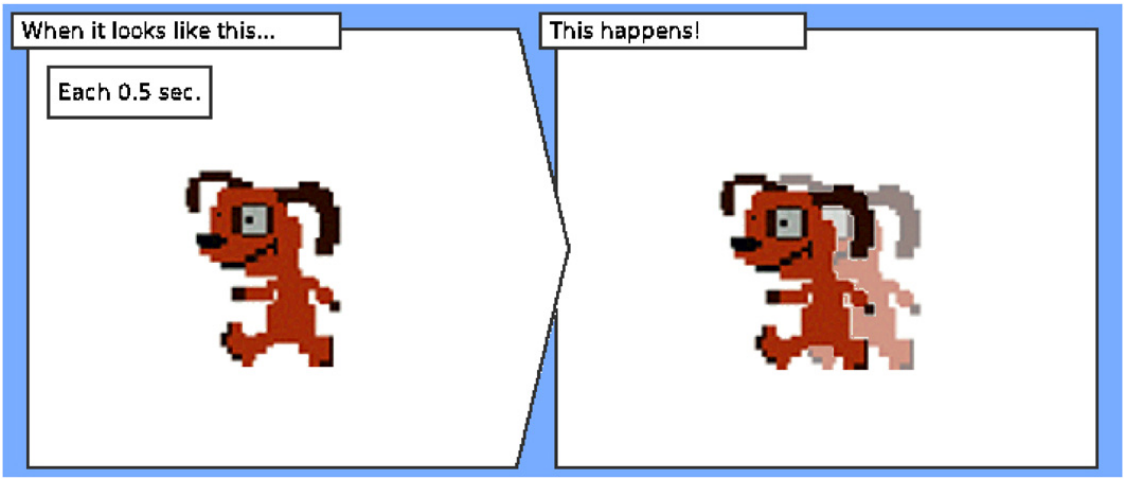

Figure 2.9: A rule involving system clock (reprinted from [40], page 115)

bitrary placement of icons in the world. Icons can be resized, and are allowed to overlap to any extent.

2.2.4 Programming by Demonstration

Programming by demonstration VPLs facilitate programming by using examples [49]. By-demonstration VPLs are typically *selection-action* VPLs in the sense that an example specifies a situation that the programming system should recognize; and an action that should be taken thereafter [17].

There are two approaches to programming by demonstration. The first approach is referred to as *programming with examples* in [49] and *strict recording* in [17]. In this approach, the programming environment simply records user actions and the context at program editing time; and replays the exact user actions when the context arises at runtime. The programming system does not attempt to discover and generalise the underlying algorithm.

The second approach is referred to as *programming by example* in [49]. In this approach, the programming system tries to infer a general algorithm from an example or a sample execution trace.

The by-demonstration approach is always used in combination with another representation approach, in which the examples are expressed. Macros in financial

spreadsheet applications are an example of this approach.

Stagecast Creator[8] allows users to create graphical rewrite rules using the by-demonstration approach. The *before* and *after* pictures are created by using examples of icons that are extracted from the runtime environment.

2.2.5 Building-block Programming

Building-block programming, also called *jigsaw puzzle pieces* programming in [49], uses tiles to compose a visual program. The tiles are designed such that they only fit together in ways that express legal expressions in the visual language.

Each tile represents a program building block. The granularity of the block's representation ranges from a simple value to a complete subroutine. Each tile typically has two sets of connectors, which we call *plugs* and *sockets* in this text. The basic visual connection rule for the tiles is that a tile's plug can only connect to another tile's socket if the plug's shape can completely fill the socket[10]. Figure 2.10 shows two tiles from Scratch[11] [7] connecting to form a sequence of two commands.

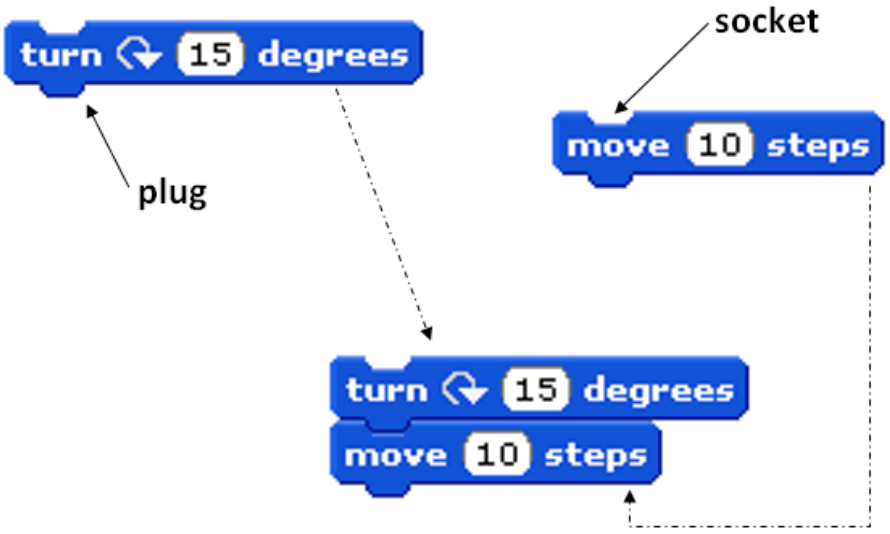

Figure 2.10: Tiles from Scratch [7]

[10] A socket is typically carved into a tile, whereas a plug protrudes from the tile.
[11] An example of a building-block VPE.

The building-block approach originated from the Glinert's BLOX methodology [28, 29], which is described by Kopache and Glinert in [43]. Some of the earliest systems based on this approach are C^2 [43] and VITPE [59]. More recent building-block programming systems are discussed in Section 2.3.

2.2.6 Discussion

This section discusses the factors that led to the adoption of the building-block approach for our VSE. We evaluate each of the representation technique described in the preceding sections using the following criteria:

1. *Fitness for purpose.* The goal of the VSE system is to simplify the programming aspect of educational games development for our in-house game engine. We assume the existence of prebuilt 3D models and static 3D scenes. The VPE that we seek must therefore support specification of the game logic without requiring the user to create 3D artefacts from scratch. There should also be a loose coupling between the 3D artefacts and the VPE, so that new artefacts can be imported into the VPE without requiring reprogramming the VSE;

2. *Effort required to develop a testable prototype.* The time in which the thesis was to be completed was limited; and as such, opportunities to cut the prototype development time were considered. One such opportunity was the availability of an open source framework on which to base the VSE; and the other was the closeness of mapping between the representation technique and the target textual programming language, so that less effort would be expended on developing the VSE's code generator. The target textual programming language was Lua, which is the scripting interface in the Myoushu game engine;

3. *Expressiveness of the approach.* Since the target audience of the VSE are not expert programmers, we sought a representation technique that was simple enough to learn; but, at the same time, expressive enough to express all the game logic without having to use an additional representation technique.

Form-based VPLs

The form-based VPL approach is limited to the declarative programming paradigm. However, the specification of 3D game logic entails controlling the order of executing various animations, among other things. This entails that a language to support the specification of game logic should support the expression of sequences. Although it may be argued that order of events can be expressed in a declarative language, this usually requires a greater cognitive effort than would be the case if an imperative language was used [27].

Another difficulty with using the form-based approach is extensibility. Form-based VPLs, such as Forms/3 [21], that include graphics as first class types, use other forms to define user-editable attributes of the graphical types. We concluded that it would not be feasible in our case to use the form-based paradigm to express the low level instructions necessary to render the "graphical values." This idea is exemplified in Forms/3, where the low-level instructions for drawing the graphics are contained in hidden cells that are not user-editable. The low level instructions, however, can refer to the user-editable attributes controlling the size of the image, for example.

If the form-based VPE is to be extensible through the addition of third-party graphical types, then there would be a need for a simple mechanism for linking user-editable attributes of the graphical types to the low level instructions responsible for rendering the graphics. Forms/3, for example, supports user-defined graphical types in a restricted way. First, the user-defined graphical type can only be composed from the built-in graphical types; and, secondly, composition is specified textually in a cell. The textual input in this case is non-trivial since it embeds the logic of rendering the graphics. This detracts from the intention of lowering barriers to programming.

Furthermore, we intend to use pre-existing 3D models; and we do not want to burden the user with understanding the composition of the models. The user cannot create forms to fully represent the 3D models' attributes and behaviours without this knowledge.

Lua, the target textual programming language for the VSE's code generator, is imperative whereas form-based VPLs are inherently declarative. This implies a

wide gap between the textual representation of the visual program and the target programming language; and thus a lot of effort would be required to develop the code generator in our situation.

Dataflow VPLs

Dataflow languages offer an improvement over the lack of explicit expression of sequence observed in form-based languages. The order of firing of nodes on a specific path is predictable; although the order is indeterminate for two nodes appearing on different paths. The dataflow model is inherently a parallel execution model [38]; and therefore simplifies the specification of parallel threads.

The dataflow approach has limited scalability. As the number on nodes and edges increases, the logic becomes increasingly difficult to follow. Furthermore, there is not much room on the screen to render the dataflow graph such that no edges cross. The criss-crossing edges detract from readability of the program.

The parallel nature of the dataflow execution model presents a challenge when translating the visual program into the target textual language, Lua. This problem derives from the lack of constraints on the order of firing of nodes appearing on parallel paths. A single dataflow visual program has the potential to map into several sequential programs with different semantics. A substantial amount of effort would therefore be required to develop a code generator for such a visual programming language.

Iconic VPLs

Iconic VPLs normally use *direct manipulation* when creating the *before* and *after* pictures. For instance, icons can be dragged around in order to change their positions in the runtime environment. The VSE's runtime environment is a 3D virtual world as opposed to the 2D world in iconic VPEs like Stagecast Creator and ComiKit [40]; and as such, the use of direct manipulation to create the rules becomes less intuitive when manipulating a 3D scene that is displayed on a 2D screen.

The 3D environment introduces additional degrees of freedom with respect to object placement; and this increases the difficulty that iconic languages have in generaliz-

ing situations. For example, whereas eight graphical rewrite rules are required in a 2D grid-based world to state that when a dog is sitting next to a ghost, the ghost disappears; twenty six rules would be required in a 3D grid-based world to express the same rule.

Creating games goes beyond making characters react to various situations. Some information, which cannot have a visual representation in the runtime environment, needs to be maintained in a game. The execution model of rule-based iconic VPLs is that when a situation in the runtime environment matches the *before* picture of a rule, then the action implied from the *after* picture is executed. Since a variable, such as one used to track the player's score, does not have a visual presence; a rule involving the variable cannot be formulated or executed. This weakness is demonstrated in Stagecast Creator, where the graphical rewrite rules are augmented with a form-based approach in order to support variables and user interaction.

Rule-based iconic VPLs are declarative; and thus suffer from the same sequencing problem discussed above under form-based and dataflow VPLs.

By-demonstration VPLs

Programming by demonstration is not a stand-alone paradigm. It relies on other representation techniques to represent the example; thus the limitations of the underlying representation technique are inherited. Moreover, each of the two approaches to programming by demonstration has its own weakness.

The strict recording approach suffers from a lack of generalisation. The exact sequence of recorded actions is repeated when a runtime context that resembles a program editing time context arises.

The programming by example approach suffers from another form of the generalization problem. The problem stems from the attempt to guess the algorithm underlying the examples demonstrated by the user. Landauer and Hirakawa [44] suggest that users may provide insufficient examples, missing out important special cases; and consequently the system may guess an invalid generalisation.

Building-block VPLs

The building-block approach can be considered a visualization of an underlying textual language. This means that a building-block VPL can be given as much expressive power as a textual language without requiring the use of an additional representation technique. This is advantageous to the users since they only need to learn one program expression technique. At the same time, the use of the building-block technique eliminates the need for the user to worry about the syntactic correctness of the program. Syntactic correctness is enforced automatically by taking the burden of typing syntactic tokens from the user; and allowing only syntactically correct composition of building blocks representing tokens.

The other attractive aspect of the building-block approach is that the closeness of mapping between the VPL and the target textual programming language, Lua in our case, can be increased intentionally. This affords us the opportunity to reduce the amount of effort required to develop the VPL-to-textual-programming-language translator. We were also able to find open source frameworks that support rapid development of building-block VPLs.

Despite the advantages discussed above, building-block VPLs, like all VPLs, still suffer from the *Deutsch limit* [9], which states that the size of problems that can be solved by a visual program is limited since no more than fifty graphical programming icons can be legibly displayed on the screen. The extent to which this limitation is addressed is discussed in Chapter 3.

In the following section, we look at three examples of game authoring tools that use the building-block visual program representation technique.

2.3 Related Visual Game Generators

This section discusses three examples of game authoring VPEs based on the building-block representation techniques. The VPEs are Scratch, Alice and StarLogo TNG. The VPEs were chosen because they are open source; and therefore presented an opportunity to reuse their visual expression subsystems. Other building-block-based VPEs include LogoBlocks [19], Pet Park Blocks [26] and App Inventor [3].

2.3.1 Scratch

Scratch [7] is the simplest of the three building-block VPEs. Scratch supports the creation of interactive stories, animations, games, music and art [6]. Figure 2.11 shows the Scratch main window. The main window is divided into five sections which we call *drawer switcher*, *blocks drawer*, *script window*, *stage* and *stage/sprite window*. The building blocks in Scratch, which we will now call blocks for brevity,

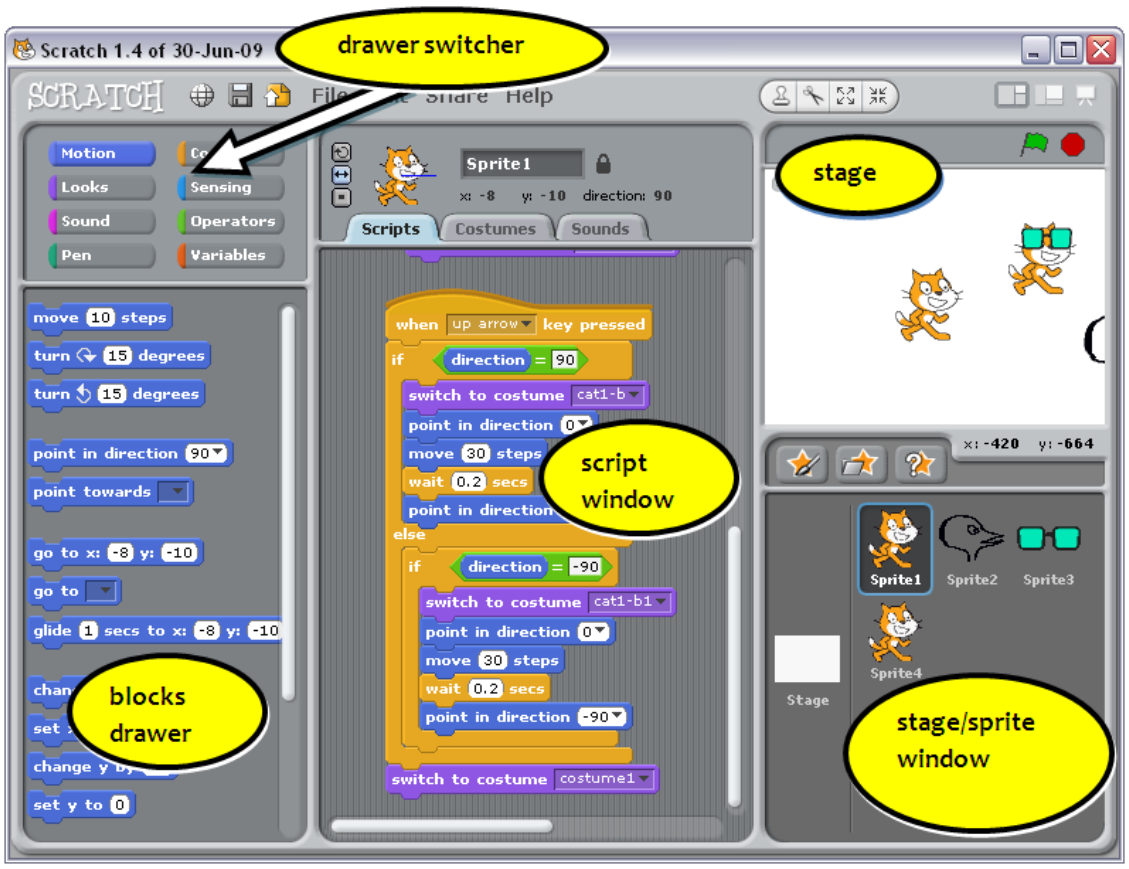

Figure 2.11: A screenshot from Scratch [7] showing the main window

are grouped into eight categories. Blocks belonging to the same category share the same colour. Each block category is visually represented by a button labelled with the category's name. Clicking a category button from the drawer switcher causes the blocks drawer to display blocks from the corresponding category. This setup is useful because it facilitates faster searching for the blocks; and it also makes the screen tidier by limiting the number of blocks appearing on the screen.

The blocks drawer contains instances of blocks that can be used to build programs. Each block instance in the blocks drawer is capable of generating an infinite number of replicas of itself, in theory. When a user drags a block from the blocks drawer, the block is duplicated; and the original block remains in the blocks drawer whereas the new copy can be dropped into the script window. When a block is dragged and dropped back into the blocks drawer, it is deleted.

The script window is where the script is crafted. Scratch maintains as many script windows as the numbers of sprites on the stage; however, only one script window is displayed at a time. A script window for a particular sprite can be displayed by selecting the corresponding sprite from the stage/sprite window.

Scratch supports event-driven programming through the four *when* blocks shown in Figure 2.12. A stack of blocks, whose top block is connected to a *when* block, is executed when the event stated in the *when* block occurs at runtime.

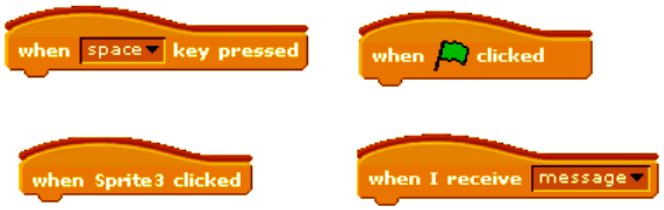

Figure 2.12: Scratch's event blocks

These four *when* blocks also facilitate the specification of concurrent scripts. Figure 2.13 shows an example of using *when* blocks to express concurrency. Since the *play note* and the *play drum* blocks are attached to separate instances of the *when...key pressed* block (each instance having the same parameter, namely, `space`), the *play note* and *play drum* blocks are executed concurrently when the user presses a space keyboard key at runtime.

Figure 2.13: Concurrency in Scratch

Scratch also supports the definition and manipulation of variables and lists. Both variables and lists can be defined as global, in which case they are available in all script windows for the project. Variables and lists can also be defined as local, in which case they are only available in the script window for the sprite for which they are defined.

We observed the following weaknesses in the Scratch VPE:

1. *Lack of support for modularity.* Scratch does not implement any mechanism to fully support modular abstraction. There is, however, limited modularity which results from the isolation of scripts when the script uses more than one event block (*when...* block). This limited modularity cannot be used to implement abstraction. The lack of abstraction means the script for controlling animation is tightly coupled with the script controlling the high level logic of the games; thus the user is forced to reason about animation and game logic simultaneously;

2. *Lack of a visual representation of constants.* Scratch does not have a set of blocks representing various types of constants[12]. This means that constant values have to be typed directly into the expressions in which they are required. Although blocks representing algebraic operators screen out non-numeric data, syntax errors[13] are still possible as demonstrated in Figure 2.14. In the figure, the *add...to...* block inserts the string, "scratch", into the empty list, `list2`. The *item...of...* block returns the first item of `list2`. The "multiply" block multiplies 2 and the string "scratch"[14]. The *say...* block displays the result of the multiplication in a sprite's callout. Figure 2.15 shows the result of running the script;

3. *Misleading implementation of lists.* The *add...to...* block accepts variables representing lists in the first socket, meaning that a list can be inserted into another list. There is, however, no expression for extracting elements of a nested list;

[12]Scratch has only one block representing the two types of supported variables, namely numeric and text.

[13]Scratch conveniently treats syntax errors as runtime errors, affecting only the computed result.

[14]This connection should have been rejected.

Figure 2.14: A syntax error in Scratch: an attempt to multiply a number and a string

Figure 2.15: The syntax error in Figure 2.14 treated as a runtime error by Scratch

4. *Representation technique shifts.* Scratch lacks blocks for defining variables and lists. Variables and lists are defined by using forms, which is a deviation from the building-block representation technique.

2.3.2 Alice

Alice [2] is a VPE that simplifies creation of 3D virtual environments. Alice can be used to create interactive stories, games and videos. At the time of writing this thesis, the latest release version was Alice 2.2; however, there was also an Alice 3.0 beta version. There are no significant changes between Alice 2.0 and Alice 2.2; and as such, our analysis is based primarily on version 2.0. In Alice 3.0, the visual program representation is almost the same as that in Alice 2.0. The major change that we noted, concerns the screen layout.

The Alice 2.0 main window, shown in Figure 2.16, is divided into five sections, namely:

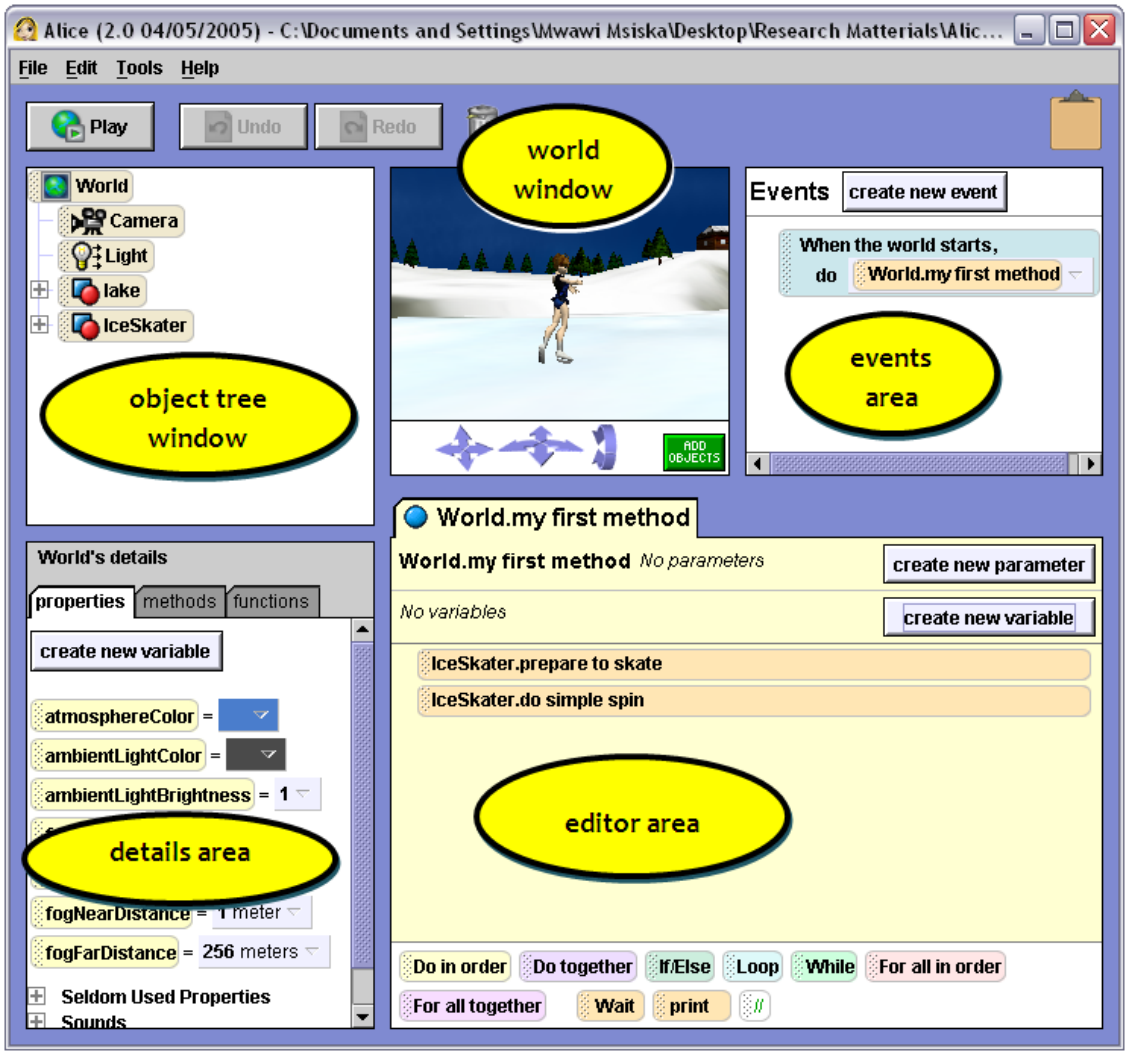

Figure 2.16: A screenshot from Alice [2] showing the main window

1. *World window.* This window offers a preview of a 3D scene that is being created. The window allows a user to specify the initial placement of objects in the scene, in a direct manipulation manner. The objects can be moved around, resized and rotated by using the mouse and keyboard.

2. *Object tree window.* This window displays all the programmable objects appearing in the world window. In Alice 2.0, objects are hierarchical; for example, a "man" object has a hand which in turn has a finger. Animations can be specified on the whole object; or on its individual parts. The object window thus facilitates selection of objects and their parts;

3. *Details area.* This area displays all the properties, methods and functions defined on an object selected from the object tree window.

4. *Editor area.* This is where scripts are composed. Below the editor area, there is a panel containing control flow blocks.

5. *Events area.* Alice supports event-driven programming; and this area is used for connecting events with their events handlers.

The blocks in the details area, and in the panel below the editor area, generate an infinite number of instances of program building blocks, like the blocks in a *blocks drawer* in Scratch; but, unlike in Scratch, these blocks have the same shape. In Alice, the blocks only take on their final differentiated shapes when they are dropped in either the editor area or the events window. The blocks in the details area, and in the panel below the editor area, can thus be considered icons representing the final program building blocks.

Concurrent programming is supported in Alice through the use of *do together* and *for all do together* blocks. The *do together* block is a container that holds methods and functions that must be executed in parallel. The *for all do together* is also a container block that is used to repeatedly execute several functions and methods in parallel.

Alice also supports the declaration and manipulation of variables and lists. Unlike Scratch, Alice is a strongly typed VPE. All user-defined variables and lists/arrays have a global scope in Alice.

The Alice VPE has the following shortfalls:

1. *Similarity of blocks.* All the program building blocks in Alice have a similar outer outline, the box outline. The logical structure is thus not easy to spot at a glance. The problem is worsened by the subtle difference in colour among the different types of blocks and the background. Consequently, the appearance of scripts is more textual than graphical. This problem is illustrated in Figure 2.17.

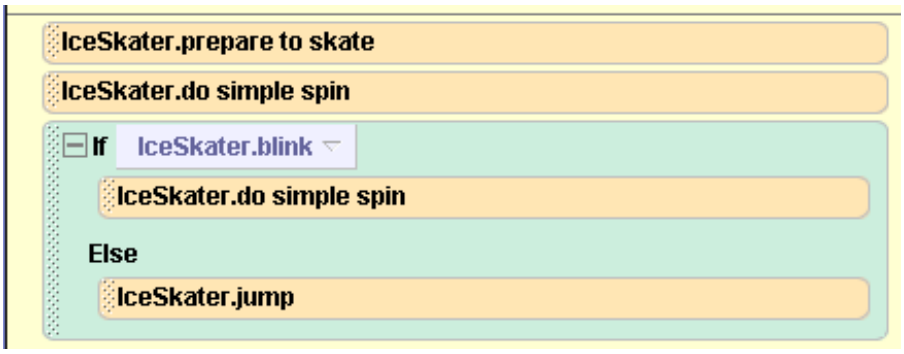

Figure 2.17: Alice blocks' lack of differentiated shape outlines

2. *The absence of a connector metaphor.* Connections among the blocks in a program are not explicit. There are no traditional matching *sockets* and *plugs* to visually suggest compatibility of blocks. This problem has been partially addressed in Alice 3 beta – all functional blocks have compatibility suggestive connectors; however the problem still remains in command and flow control blocks.

3. *Representation technique shifts.* The problem of representation technique changes when handling advanced concepts is not limited to variable/list definitions as in Scratch. A combination of forms and menus is also used when entering constant values, and when building algebraic, relational and logical expressions.

4. *Breakdown of abstraction in Alice 3.0 beta.* We observed a mismatch of object abstraction between object references in the object selector and the editor area. For instance, a cat's head can be selected by first selecting the cat, and then its head from two drop-down lists; but, in the editor area, the cat's head is represented by a function call which returns a head, whose argument is another function call which returns a cat. This functional representation can be confusing since it represents a shift from a higher level of concreteness to a lower level.

5. *Fine grain animation.* The hierarchical structure of objects in Alice affords the users the possibility of fine grain animations, such as moving an arm relative to the torso of a person. The correct specification of such animations

requires significant skill in 3D geometry to prevent, say, the arm disconnecting from the torso.

6. *Strict insistence on connectedness.* Alice does not permit the placing of a block in the editor area if it does not form part of the script in the editor area. This is inconvenient since, for example, a complex algebraic expression cannot be rearranged without first deleting some components and redrawing them from the details area. In an ideal situation, the system should have allowed the components to be temporarily unconnected to facilitate restructuring of the algebraic, relational and logical expression without the delete-redraw sequence.

7. *Inefficient use of the screen real estate.* Although Alice supports modular abstraction through functions and methods, all the user-defined methods and functions are displayed in the script area at all times. Alice 3 beta exacerbates the problem by eliminating the events area, which means that all event handlers have to be defined and linked in the editor area.

2.3.3 StarLogo TNG

StarLogo TNG[15] was designed for the creation of simulations of complex systems [4]. It uses blocks similar to those used in Scratch; but unlike Scratch, StarLogo TNG's runtime environment is a 3D virtual world.

StarLogo TNG's VPL addresses all the problems we identified in Scratch and Alice. We defer the detailed description of the VPE in StarLogo TNG to Chapter 3 to avoid repetition since the design of our VSE is largely based on StarLogo's front-end subsystem.

2.4 Chapter Summary

This chapter began by providing an informal definition of visual programming. It introduces the two concepts of a visual programming language (VPL) and a visual programming environment (VPE); and states the difference, which is based on

[15]TNG stands for The Next Generation.

the presence or absence of an underlying textual programming language. It also describes goals of visual programming; and the four approaches to achieving the goals, namely: concreteness, directness, explicitness and immediate visual feedback.

The chapter then discusses five visual program representation techniques, namely: form-based, dataflow, iconic, by-demonstration and building-block. We argued that the building-block approach is the most suitable for our problem because of its high degree of expressiveness; its abstraction of low-level syntax; its closeness of mapping to the target textual programming language; and the availability of an open source framework on which to build our prototype.

The chapter ends with a discussion on three open-source examples of VPEs that use the building-block representation technique. We compare the three VPEs with the aim of identifying the best framework on which to base our prototype. StarLogo TNG's VPE was identified as the most suitable for our purposes.

In the next chapter we discuss the design of our VSE.

Chapter 3

The VSE Programming Environment

This chapter discusses the design of a prototype visual programming environment for creating scripts to run on the Myoushu game engine. The prototype is called the visual script editor (VSE). We discuss how the VSE prototype is realized by extending the OpenBlocks system. One such extension is the inclusion of extra connector shapes that can be used to create additional data types.

We also discuss the design of additional modules that further simplify the creation of games that run on the Myoushu game engine. Examples of such additional modules include: a scene object manager, which inspects XML scene files and creates blocks representing scene objects; and a game GUI manager, which scans GUI layout files and creates blocks representing GUI widgets.

Section 3.4.5 discusses the code generator module. Despite the syntax error avoidance property of the building-block visual program representation technique, compile-time errors and warnings can still occur. We discuss the source of these errors and warnings; and the measures taken to tackle the errors and warnings, in the code generator module.

OpenBlocks-based visual programs are stored in text files, in XML format. These XML files are quite large compared to the corresponding Lua script files generated from the visual programs, and the file size ratio typically exceeds 10:1. When reading in the visual programs from the XML files, the XML content is first parsed

into DOM [5] trees; and then the DOM trees are processed to generate directed graphs, which are OpenBlocks' (and consequently VSE's) internal representation of visual programs. These graphs are memory intensive. In Section 3.4.6 we discuss measures that we have taken to reduce this memory footprint.

We begin this chapter by providing an overview of the game production process, in the context of the Myoushu game engine.

3.1 The Game Production Process

The process of creating a 3D game involves several activities, including: the design of a storyline; the design and development of 3D game objects, or the acquisition of prebuilt 3D game objects; building scenes using the 3D game objects; creating animations for some of the 3D game objects; specifying when to execute the various animations; and specifying mechanisms to keep track of the state of the game.

As mentioned in Chapter 1, our work does not include the development of storylines. We assume the existence of a storyline for which a 3D virtual world is sought. We also assume the existence of prebuilt 3D game objects and scenes. We briefly discuss 3D game objects and scenes because they are part of the direct input to the game engine; and they also are a source of information that is critical to the operations of the VSE.

3.1.1 The Game Engine

The design of our VSE is constrained by the Myoushu game engine, which was developed by Chamberlain [62]. Henceforth, we use the phrase *"the game engine"* to refer to the phrase *"the Myoushu game engine"*. The game engine's major inputs include: Lua script files; 3D scene definition files; 3D game object definition files; graphical user interface (GUI) layout definition files; sound files, educational content files; and user input through the mouse and keyboard.

The Lua scripts specify the game logic which manipulates 3D scenes; controls any GUI that appear in the virtual environment; keeps track of the state of the game; keeps track of the player's performance measure; and handles user input.

Before we discuss the challenges posed by the Lua scripting interface to both experienced and inexperienced programmers, we describe the relationships between the game engine and 3D scenes; and between the game engine and the virtual environments' GUIs.

3.1.2 3D Models and Scenes

3D scenes are passed to the game engine through text files, in XML format, called Dotscene files. Dotscene files contain only the scene layout and animations defined for the 3D objects in the scene. The layout information includes the 3D coordinates at which an object is placed, and the orientation of the object. The actual 3D game object geometry definitions are stored in separate files, called mesh files.

When the game engine renders a scene in a 3D virtual environment, it begins by reading the contents of a Dotscene file; and then locating all the mesh files named in the Dotscene files. The game engine then renders each game object in the scene; and uses shading and texture details specified in a materials file to colour and texture the game objects.

Once a scene has been rendered, animations defined for some of the 3D game objects can be executed. Instructions to load a scene and execute a particular animation defined in the scene can be specified in a Lua script.

3.1.3 Graphical User Interface

GUI definitions are stored in text files, called GUI layout files. GUI layout files are in XML format. The layout file defines the various widgets appearing on a particular GUI. This file specifies the 2D coordinates of each widget, the widgets' names and other attributes that control the appearance of the widgets. The file also associates a widget's event name with the name of an event handler for each widget whose events must be handled.

When the game engine receives a request to display a particular GUI, it reads the GUI layout file and creates all the widgets described in the file. Each widget's name becomes available to the Lua script controlling the GUI. The widgets can thus be manipulated by passing their names to special Lua library functions. A

Lua script controlling a GUI needs to provide the definition of an event handler for each widget's event that was associated with an event handler name in the GUI layout file. The name of the event handler in the Lua script must match the name of the event handler name in the GUI layout file.

3.1.4 The Lua Scripting Challenges

The game engine has a Lua scripting interface. Lua scripts are used to specify game logic which manipulates scenes and GUIs. The Lua scripting interface poses challenges to the specification of game logic to both experienced and inexperienced programmers.

Before the advent of the VSE, there was no integrated development environment (IDE), be it textual or visual, to simplify the management of the enormous amount of information that must be available when creating the Lua scripts. The author of a Lua script needs to be aware of the programmable content of a scene or an in-game GUI. Although this information is contained in the Dotscene and the GUI layout XML files, it is rather inconvenient for a programmer, whether experienced or inexperienced, to be manually searching these files for programmable elements. Consequently, there is need for an IDE capable of browsing the XML files and extracting script related information; and displaying such information to the user.

Lua is a textual programming language whose mastery requires a significant amount of effort to be expended in learning the syntax. If the person learning Lua is absolutely new to programming, he/she must also learn computational thinking in addition to learning the syntax. Computational thinking skills are indispensable in game logic specification; however, the syntax burden adds no value to the game creation process, and hence our desire to eliminate it.

The target audience of our VSE is ASDs therapy specialists. These are domain experts, but are not necessarily programmers. We seek to empower the therapists with a tool to enable them to rapidly create therapeutic games to match their patients' diverse specific needs. We do not, however, want to force them to learn the syntax of Lua.

The next section discusses the design objectives of the VSE.

3.2 Design Objectives

To lower the barriers to scripting discussed in Section 3.1.4, we considered developing an IDE, the VSE, with the following objectives:

1. to represent the majority of scripting commands and expressions by graphical elements, so that scripting can be done using a predominantly drag-and-drop interaction style;
2. to represent programmable scene and GUI artefacts by graphical elements that are compatible with the graphical elements in 1;
3. to present the user with a simple-to-use interface for managing scenes and their content;
4. to present the user with a simple-to-use interface for managing GUIs and their content;
5. to automate the process of exporting scripts and other resources to the game engine for execution; and
6. to generate Lua scripts from the graphical programs mentioned in 1. The Lua code should be human-readable to the extent that it can be modified by a skilled Lua programmer.

Objectives 1 and 2 allow the virtual environment (VE) author to specify the VE logic without the need to learn the syntax of Lua, the target language. The author will, however, still need to master the art of computational thinking[1]; which, coincidentally, is a by-product of using our VSE due to the nature of the visual programming representation technique that we have adopted [33].

Objectives 3 and 4 allow the author to discover the abstract contents of scene files and GUI layout files without having to manually inspect the XML files representing scenes and GUIs. This is necessary since the Dotscene files are quite large, even for modest scenes.

[1]Wing [63] describes computational thinking as "solving problems, designing systems and understanding human behaviour, by drawing on the concepts fundamental to computer science"

Objective 5 takes the drudgery of declaring the location of various resources, required by the game engine in order to execute the script for the VE, from the author.

Objective 6 is included in case the intended VE authors eventually learn and master Lua. In this case, the author may find the use of the VSE tedious, as suggested by Koegel and Heines [41]; however, the author might want to modify VEs previously developed using the VSE. In this case, all the author needs to do is to modify the Lua code generated from the visually composed scripts.

In the next section we describe the OpenBlocks library, which forms the basis of our VSE, and discuss how it satisfies objectives 1 and 2. We also discuss how the OpenBlocks library can be used to overcome the program representation weaknesses in Scratch and Alice, which we discussed in Section 2.3.1 (on page 26) and Section 2.3.2 (on page 29), respectively.

3.3 The OpenBlocks Library

The OpenBlocks library was developed by Ricarose Roque [55]. The library was developed to speed up the time consuming process of creating a VPE [55]. The OpenBlocks library supports the development of VPEs that use the building-block representation technique. A complete set of program building block types can be created by writing specifications in a single XML file. The XML file also specifies the grouping of blocks into drawers. This XML file is discussed further in Section 3.4.5.5 (on page 61).

The OpenBlocks library was created from StarLogo TNG's VPE called StarLogoBlocks; and thus the library supports the development of VPEs that look and function like StarLogo TNG's StarLogoBlocks. For that reason, we use StarLogoBlocks to concretise the capabilities exposed by the OpenBlocks library.

Figure 3.1 shows a screenshot of the StarLogoBlocks VPE. The main features of StarLogoBlocks, which the OpenBlocks library inherited, include:

1. *The page concept.* The program editing area in StarLogoBlocks is divided into sections called pages. Each page is delineated with two parallel vertical

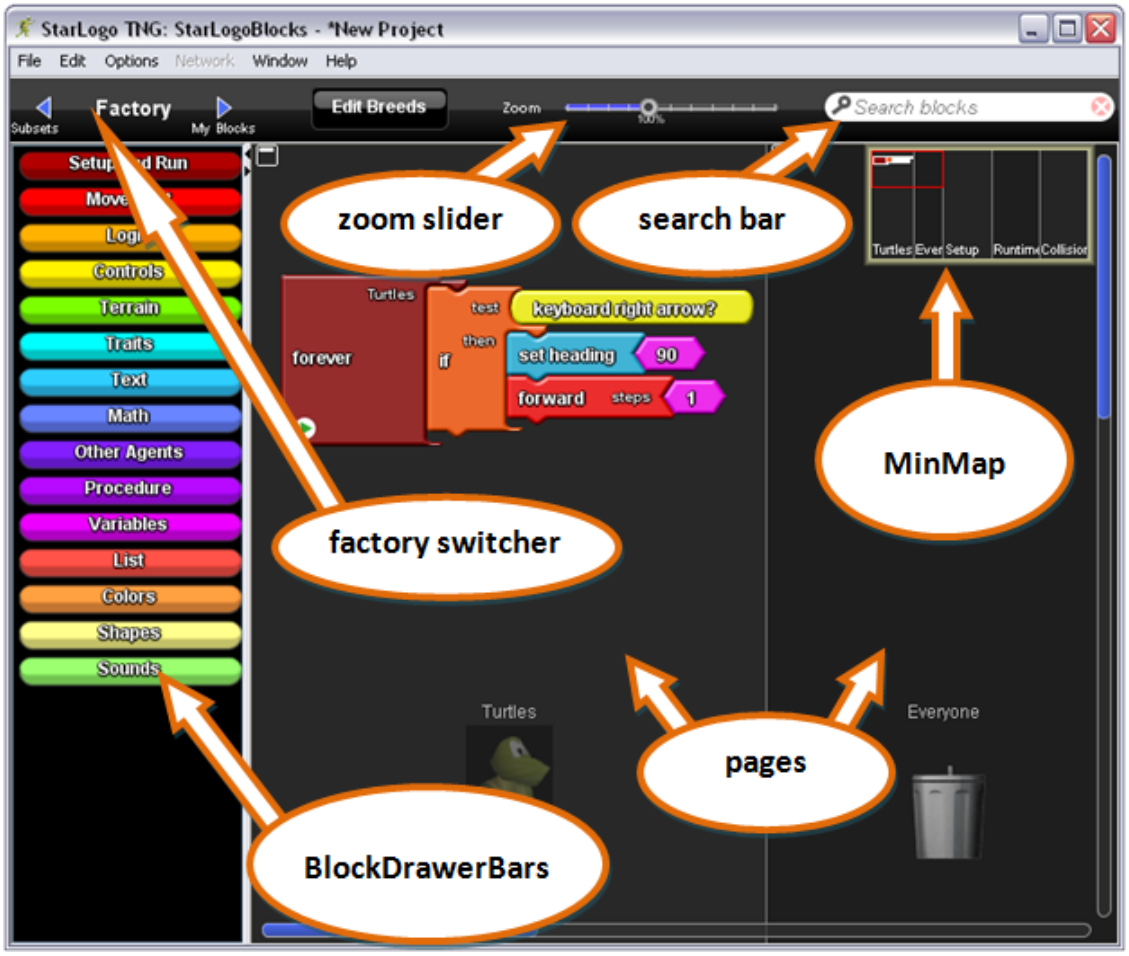

Figure 3.1: A screenshot from StarLogo TNG [4] showing the StarLogoBlocks VPE

lines, one to the left side and the other to the right side. Each page has a name that refers to either a game object or some abstract concept, such as the program's runtime phase. All the code in a page implicitly refers to the game object or the abstract concept referred to by the page's name, except when the code explicitly refers to another game object. A StarLogoBlocks page is similar to a script window in Scratch; except that in StarLogoBlocks, all pages are visible in the program editing area at all times. Each page can be minimized and restored; and as such, the concept of pages does not constitute an inefficient use of the screen real estate, a problem that we observed in Alice. The concept of pages allows code for a specific game object to be composed, without the need to mention the game object in each statement that refers

to the game object;

2. *The MinMap.* The MinMap is a miniature representation of the entire program editing area. It facilitates rapid navigation within the program editing area. A user can navigate to a remote section of the program by selecting the section from the MinMap;

3. *The factory switcher.* StarLogoBlocks supports two types of block drawers[2], namely *static* and *dynamic* drawers. Static drawers are fixed and cannot be changed by user action in a typical programming session. On the other hand, dynamic drawers are not fixed during a typical programming session. Dynamic drawers are automatically added and deleted as *breeds*[3] are added and removed from a program. Dynamic drawers hold, among others, blocks for manipulating user-defined variables, procedures, functions and actual parameters. The factory switcher is used for switching between static and dynamic drawer sets;

4. *BlockDrawerBar.* A BlockDrawerBar is used for opening a drawer. Unlike in Scratch, block drawers in StarLogoBlocks remain closed until a user opens a drawer to use its contents. Once a block has been dragged out of a block drawer, the drawer returns to its closed state. This arrangement allows StarLogoBlocks to handle more block categories without cluttering the screen;

5. *The zoom slider.* The zoom slider controls the level of magnification of the program editing area, thus controlling the number of blocks viewable per screen area. This feature accommodates users with diverse visual acuities;

6. *The search bar.* The search bar facilitates searching for program building blocks. This feature becomes handy when a user knows the name of the block he/she seeks to drag out of a drawer, but has forgotten the name of the drawer holding the block. The user is not required to type the complete name of the block since StarLogoBlocks searches for all blocks whose labels contain the text entered by the user.

[2]Containers holding instances of program building blocks.

[3]A *breed* in StarLogo TNG refers to a collection of scene objects of the same kind. Each breed is assigned a page on which to specify logic which controls every member of the breed.

StarLogoBlocks uses blocks similar to Scratch's blocks. These blocks do not suffer from the lack of differentiated outer shape and the lack of a connector metaphor, a problem that we observed in Alice. The StarLogoBlocks library has a constraint that all programming entities, including constants, be represented by blocks. This constraint eliminates syntax errors stemming from the lack of blocks representing constant values, which were observed in Scratch.

The lack of modular abstraction in Scratch is addressed by StarLogoBlocks in two ways. First, StarLogoBlocks has a block for defining procedures and functions. Once a procedure or a function has been defined, StarLogoBlocks automatically creates blocks that can be used to call the procedure/function. Secondly, the concept of pages in StarLogoBlocks implements a modular abstraction similar to that of a class in an object oriented programming language, but in a limited way. Only procedures/functions, defined on one page, are available to other pages. Any code that is outside a procedure/function is not available in the other pages, except for global variables.

The problem of representation technique shifts, in both Scratch and Alice, when defining variables and lists is nonexistent in StarLogoBlocks. StarLogoBlocks has blocks for defining variables and lists. Once a variable or a list has been defined, StarLogoBlocks automatically provides blocks for manipulating the variable/list. No forms[4] are used when defining variables.

The breakdown of abstraction observed in Alice 3.0 beta is absent in StarLogoBlocks. The problem in Alice arises due to the use of icons to represent block types, and only allowing the block instances to take their final form when they are dropped into the editor area. On the other hand, blocks in StarLogoBlocks are never iconified when they are in the block drawer; and therefore a block's appearance is the same regardless of its location.

StarLogoBlocks does not support breaking down a game object into its parts; and thus the problems stemming from attempts to write code to implement fine grain animations do not arise. The task of implementing fine grain animations is left to the author of the 3D model and not the user of the VPE.

[4]Forms, as a visual program representation technique, as discussed in Section 2.2.1 (on page 9).

Unlike Alice, StarLogoBlocks does not insist on strict connectedness of blocks appearing in the program editing area. This means that programs can be reorganised in a more flexible manner.

StarLogoBlocks affords the opportunity to manage the screen real estate, unlike Alice. The screen can be managed in three ways:

1. by minimizing and restoring pages;
2. by using the zoom slider to control the number of blocks visible per unit area; and
3. by collapsing and expanding procedure/function definition code. Figure 3.2 shows an example using this feature. Part (a) shows an expanded version of a procedure definition stack. Part (b) shows the same procedure definition stack with its body collapsed.

Figure 3.2: Screenshots from StarLogo TNG showing code folding

Lane *et al* [45] report that graphical user interfaces comprising menus, icons and toolbars detract from the potential productivity of experienced users. Lane *et al* [45] motivate the provision of keyboard shortcuts to enhance the productivity of experienced users. McCaffrey [46] extended the StarLogoBlocks by adding a feature called *TypeBlocking*, which allows blocks to be drawn on a page using only the keyboard.

TypeBlocking does not break the building-block representation technique; rather, it is a keyboard shortcut for a sequence of actions that includes: selecting the appropriate BlockDrawerBar panel; clicking the desired BlockDrawerBar; scrolling to the desired block; and dragging and dropping the block onto a page. This sequence of actions can be replaced by typing the first few letter of a block label and pressing "enter" when a unique match is found.

3.4 The Design of the VSE

Figure 3.3 shows the major modules of our VSE system. The arrows in the figure point from the client side of the relationship between two modules. The scene object

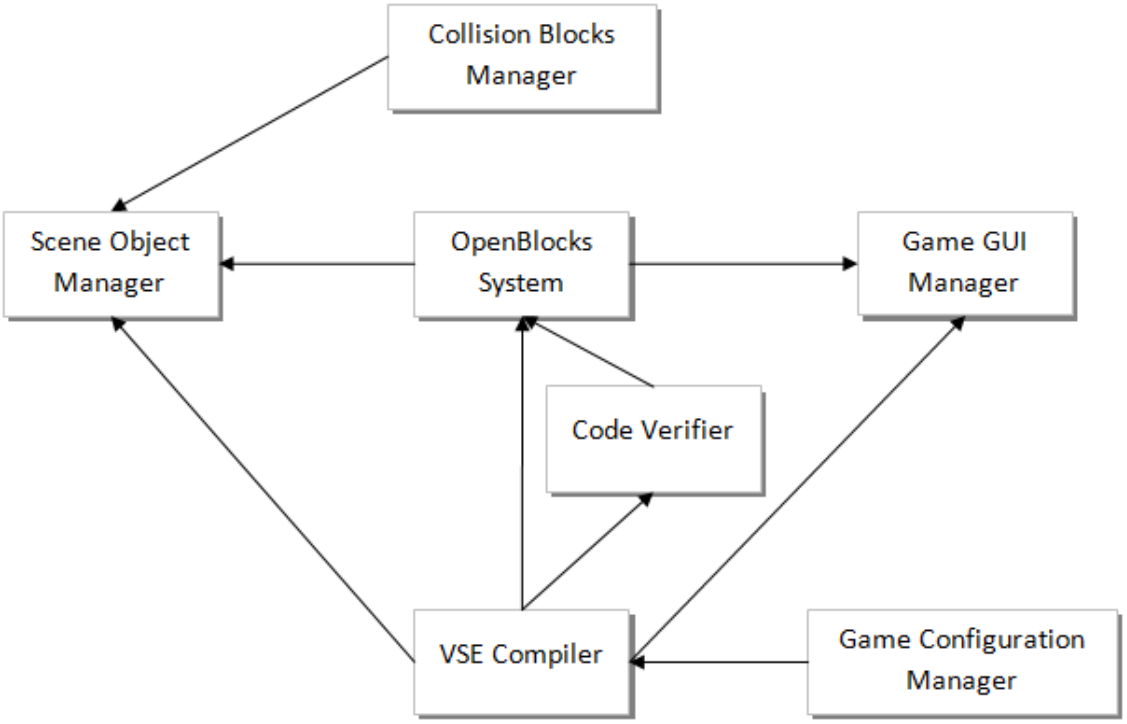

Figure 3.3: The VSE system overview

manager is responsible for informing the OpenBlocks module of blocks representing scene artefacts that the OpenBlocks module needs to create and put in appropriate block drawers. The collision blocks manager keeps tracks of information on possible collisions among game objects and informs the OpenBlocks module of the collision

handler blocks that are required in the collisions drawer. The game GUI manager maintains information about game GUIs that need to be programmed, and informs the OpenBlocks module about the required blocks representing GUIs and widgets. The blocks verifier is used by the code generator module to check for potential compile-time errors and warnings. The code generator generates Lua code from a visual program in the OpenBlocks module. The game configuration manager sets up a complete game configuration that can be exported to the game engine for execution.

These modules are discussed further in Sections 3.4.1 through 3.4.5.

3.4.1 Extensions to the OpenBlocks Library

3.4.1.1 Connector Shapes

The latest version of the OpenBlocks library, at the time of writing this thesis, had only fourteen connector shapes. This meant that for a non-trivial VPE such as StarLogoBlocks, the connector shapes had to be reused in several dissimilar contexts.

Roque [55] discusses a problem, emanating from overloaded block connector shapes, that was exposed through user tests. Block connector shape overloading refers to the use of one block connector shape to represent more than one block connection rule. An example of an overloaded connector shape in StarLogoBlocks is the *command* connector shape. Algorithm 3.1 shows the *command* connector shape's compound connection rule, which embeds two basic connection rules in an **if-then-else** structure. In the algorithm, the two blocks that are being considered for connection are `block1` and `block2`; and one of the blocks has the socket, `socket`, while the other block has the plug, `plug`.

Figure 3.4 shows an example, from StarLogo TNG, of the breakdown of the plug-socket connector metaphor due an overloaded *command* connector shape. In parts (a) and (b) connection is rejected, even though the socket and plug shapes seem compatible, because exactly one of the blocks is a *setup* block. In part (c) the connection is accepted since both blocks are *setup* blocks; and in part (d) the connection is accepted since neither of the blocks is a *setup* block. This is a departure

1: **if** `typeOf(socket)` = `typeOf(plug)` = `command` **then**
2: **if** (`typeOf(block1)` = `setup`) XOR (`typeOf(block2)` = `setup`) **then**
3: reject the connection
4: **else**
5: accept the connection
6: **end if**
7: **end if**

Algorithm 3.1: Connection rule for the overloaded *command* connector shape

Figure 3.4: Confusion emanating from overloaded connector shapes (adapted from [55], page 30)

from the affordance offered by the complementary plug and socket shapes; and can confuse the user if he/she finds outs that some complementary shapes do not connect.

Another problem emanating from the limited number of connector shapes in StarLogo TNG is the representation of objects such as shapes and sound using strings. While it may be argued that this design is technically sufficient (since, for example, a string can refer to the file path for a resource), the use of such strings introduces the possibility of otherwise avoidable runtime errors, such as a user setting a game character's shape to an obviously impossible shape, as shown in Figure 3.5.

Figure 3.5: An obvious error caused by allowing strings to refer to objects

As a solution to the aforementioned problems, we extended the OpenBlocks library by adding seven extra connector shapes. The resulting twenty one connector shapes were sufficient to visually specify all the block connection rules in our VSE, without resorting to overloading, or using strings to refer to different types of objects.

All the additional seven connectors can be reused in other building-block VPEs; but the connector-name-to-shape mapping would have to be adapted because most of the names we used were designed specifically for the VSE system. We note that one connector, namely *proc-var*, has the potential of being reused in other building-block VPEs, in the same context as it was used in our VSE.

3.4.1.2 The proc-var Connector

The *proc-var* connector has been designed to be used on block stubs associated with procedures and functions. A block stub is a special kind of a block which depends on another block kind for its full functionality. Block stubs are used, for example, to automatically provide *getter* and *setter* blocks for user-defined variables.

The original OpenBlocks library provides only one block stub associated with procedures/functions, called *caller*. This stub is used to automatically provide blocks that can be used to call user-defined procedures/functions.

We noted that in a programming language that supports first-class functions (such as Lua, our target textual programming language), functions/procedures can be assigned to variables or passed to other functions/procedures. We thus required a feature in the VSE to parallel this concept, hence the addition of a *proc-var* block stub to the procedure block in OpenBlocks.

3.4.1.3 Floating-Point Numbers versus Integers

Out of the fourteen connector shapes in the original OpenBlocks library, only one connector shape is used for connecting blocks that handle numeric data. This shape is called the *number* connector shape in StarLogo TNG. It can be argued that one numeric type can adequately represent both floating-point numbers and integers [35]; however, this detracts from the explicitness of a VPL. Consider the code fragment depicted in Figure 3.6. The code fragment attempts to extract a name from a list of names. Since a floating-point number has been used to refer to

a position (an index) in the list, the user cannot know whether this will return the third or fourth name without consulting the VPEs documentation on how fractional list indexes are handled.

Figure 3.6: A floating-point number used to index a list

A seemingly plausible solution would be to create a separate connector shape for connecting integers. This presents a challenge since most of the operators defined for floating-point numbers are also defined for integers; and, therefore, would need to be defined for integers as well. Allowing integers and floating-point numbers to have non-intersecting sets of operators would clutter the *math* block drawer with an unnecessarily large number of similar operators. If we assign a special connector shape to integers, then we are making a visual statement that integers and floating-point numbers are different entities.

We adopted a compromise in which we treat floating-point numbers as a super class of integers; that is, all integers are also floating-point numbers, but floating-point numbers are not integers. Both integers and floating-point numbers share a common connector shape, but blocks representing integer and floating-point number constants have different colours. Note the difference between the concepts of *connector kind* and *connector shape*, namely: a connector kind determines whether two blocks can connect, whereas a connector shape visually suggests, to the user, whether two blocks can connect. Integer and floating-point number blocks share only the connector shape.

A floating-point number socket accepts connections from both floating-point number and integer plugs; on the other hand, an integer socket only accepts connections from integer plugs. In our design, a binary algebraic operator always returns a floating-point number regardless of the type of its operands.

It is sometimes desirable to convert a floating-point value to an integer; and for this purpose, we provide a special block, called *int*, that has a number socket on its right side, and an integer plug on its left side.

3.4.1.4 Copy and Paste

The OpenBlocks library supports copying and pasting of a group of connected blocks. The library also supports context sensitive menus that are activated when a user presses the right-hand mouse button while the cursor is within a page.

We noted that the context sensitive menus only included commands for adding and removing comments, and rearranging blocks in all pages. In StarLogo TNG, for example, the copy and paste functionality is only available from the edit menu and through the `CTRL+C` and `CTRL+V` keyboard key combinations.

Accessing the copy and paste functionality in the manner described above is inefficient, since the user has to either switch between the mouse and the keyboard; or move the mouse pointer from a page to the edit menu and click on the appropriate command. We decided to speed up the copy and paste operations by including the options of copy and paste in the context sensitive menus.

3.4.2 Managing Scenes

The main aim of the scene object manager is to manage all the elements of a given scene on behalf of a user, such that the user can access all the programmable content from the scene without having to manually inspect the verbose Dotscene file representing the scene.

The user interface to the scene object manager allows the user to perform the following four operations:

1. *Load a scene.* This operation allows a user to navigate through the file system and select a Dotscene file to be included in the game. Once a Dotscene file is selected, the scene manager searches the file for scene object declarations and animation definitions; and records this information. The scene object manager then prepares the pages for the newly loaded scene.

2. *Change a scene.* This operation allows a user to switch between previously loaded scenes.

3. *Edit a scene.* This operation allows the user to mark game objects, from a particular scene, that he/she wishes to manipulate in the program. Once a game object has been marked, the scene object manager instructs the OpenBlocks module to create new blocks for manipulating the marked game object. These new blocks include: one block for executing animations defined for the game object, blocks representing all the animations defined for the game object, and blocks representing all cameras defined in the scene.

4. *Unload a scene.* Unloading a scene deletes the scene from the current game. All the code that was written for the scene is lost; but the Dotscene file from which the scene content was extracted remains intact.

In the next two sections, we discuss why we mark scene objects as programmable and non-programmable; and also discuss how animations are handled by the VSE.

3.4.2.1 Programmable versus Non-programmable Scene Objects

A scene in a 3D game typically contains dozens of game objects; however, most of these objects are included for cosmetic purpose, and the game logic does not make any reference to them. For example, a room might have several lights that are never referred to in the game logic. The Dotscene file includes all the objects in the scene irrespective of their importance to the game logic. If all the game objects were to be represented in the VSE, there would be a large number of pages, most of which would be blank. Recall that, as mentioned in Section 3.3 (on page 40), each game object's code is specified on a separate page.

One of the solutions to this problem would be to adopt Alice's way of representing scene objects, in which all scene objects are listed in an object tree window; and having the details area list all the properties and operations (including animations) of a scene object selected from the object tree window. One problem with this approach is that each block that manipulates a game object has to explicitly state the name of the game object that it refers to, resulting in blocks cluttered with

text. Another problem would be that the listing of game objects in the object tree window would be quite long, making searching for a particular object difficult.

Another solution would be to automatically minimize all the pages that do not contain any code. There are two problems associated with this approach. Firstly, how do we distinguish a page that has been minimized because it is empty and a page that has been minimized because we are currently not interested in its contents? The other problem is that a minimized page is represented by a narrow vertical strip that runs the entire height of the drawing canvas. The vertical strip is labelled with the name of the page. This may amount to poor management of the screen real estate if there are many blank pages compared to non-blank pages.

The solution that we finally arrived at, displays pages only for the scene objects that a user explicitly marks as programmable. This approach also affords the opportunity for the collision blocks manager to limit the number of collisions that need to be generated.

A collision block represents an event handler that must be executed when two scene objects collide at runtime. If there are n game objects, then the total number of possible collisions is given by $n\,(n-1)\,/2$, since each game object can collide with any of the other $n-1$ game objects.

It is thus desirable to make n as small as possible; otherwise, the *collisions* drawer can easily have an unnecessarily large number of collision blocks, making it difficult to search for a particular collision block. In our approach to managing scene objects, only scene objects marked as programmable are considered when the collision blocks manager generates collision blocks.

3.4.2.2 High-level versus Low-level Animations

Animating an object in a 3D virtual environment involves 3D operations, on an underlying mesh or skeleton model, including: scaling, rotation, translation and deformation. Of these operations, scaling, rotation and translation are the easiest to specify, when applied to a 3D model without an underlying skeleton.

Some 3D models have an underlying skeleton model. Skeleton models allow the specification of animations of parts of the 3D model relative to the whole. Skeleton

models, for example, allow a humanoid character to be able to flex its legs and arms as it moves.

We consider the specification of animations involving deformation and skeletons too complex for our target audience. The specification of a visually realistic animation, such as a human walking, would require mastery of non-trivial 3D geometry.

Our VSE provides operators for specifying only high-level animations involving scaling, translation and rotation. By "high-level" we mean that an operation is carried out on the object as a whole, disregarding any underlying skeleton and deformation models that the 3D model may have. The VSE, however, also includes a mechanism for executing more detailed animations that are specified in a Dotscene file. Animations involving deformation models and skeletons can thus be specified by an experienced 3D modeller using applications such as Blender; and the animations can be made available to our target audience for "playing."

3.4.3 Managing Game GUI

The function of the game GUI manager is to read GUI layout XML files and display blocks representing all the widgets specified in the layout file. Also recall that a GUI layout file lists all the user events that must be handled for the GUI. The game GUI manager has a user interface for adding and removing a GUI to/from a game.

Each GUI is a collection of widgets. One of these widgets is designated the top-level widget and acts as a container for the other widgets; however, programmatically, all widgets are treated in the same manner. For example, the top-level widget can be displayed, hidden, or its caption can be set, just like any other widget.

The game GUI manager is also responsible for organizing GUI code into pages. We were faced with three possibilities for organizing GUI code into pages, namely:

1. *Treat each widget as a separate object.* This entails having a separate page for each widget. This arrangement seems to provide the advantage of being able to write code referring to a widget without having to repeatedly make the reference explicit. This advantage is inconsequential since widget code

typically refers to other objects, other than the same widget. This approach would also suffer from inefficient use of the screen real estate since most of the widgets would typically have a minimal amount of code.

2. *Lump all GUI code on one page.* This resolves the inefficient use of the screen real estate problem, but lumps code belonging to different GUIs on one page. If this approach were adopted, then each statement would be required to explicitly display the name of the GUI to which it refers; thus muddling the blocks with text.

3. *Treat each GUI as an object.* This option represents a trade-off between the two extremes discussed above. Each GUI is treated as an object with respect to page allocation. This drastically reduces the number of pages, compared to that required in the first approach; and, at the same time, eliminates the need for each statement to explicitly display the name of the GUI to which it refers.

We adopted the third option since it minimizes wastage of the screen space and, at the same time, it prevents cluttering the blocks with GUI name references.

3.4.4 Managing Game Configuration

The game configuration manager module was included to facilitate easy exportation of completed games to the game engine.

A typical game configuration is composed of several Lua files, one for each scene and one for each GUI. In addition to the Lua script files, the game configuration also includes several resource files, namely: Dotscene files, describing scene layout and animations; mesh files, which hold polygonal mesh models for 3D scene objects; skeleton files, which hold skeleton models for some of the 3D game objects; materials files, which contain shading information for scenes; texture files, which are image files used for texturing scene objects; dialogue and educational content files containing textual information presented in the scenes; sound files; and GUI layout files containing GUI layout definitions.

Each game configuration has to have a configuration file which the game engine uses to discover the location of Lua script files and other resource files on a computer's file system. The game configuration manager has two responsibilities, namely:

1. Making sure that all the script and resource files, relating to a specific game, are placed in one folder. Placing all the script and resource files in one folder permits file paths in configuration files to be relative, thus allowing the game folder to be moved around the file system without the need to change the configuration files. The location of the root folder of the game engine, in which all game folders must be placed, can thus be changed without affecting game configurations.

2. Automatically generating game configuration files. We included this feature so the user should not worry about keeping track of all the resource files used in a game. All the user needs to be able to do is load (or unload) the required scenes and GUI layouts, create the necessary visual scripts using the building-block paradigm, and click on a single menu option to run the game.

3.4.5 The Code Generator

This section discusses the translation of a visual program into a Lua program. This translation is the responsibility of the code generator module. The translation process omits some steps that are used in traditional compilers; for example, a scanner and a parser are not required because the internal representation (a directed graph) of a visual program in our VSE can be considered as a forest of abstract syntax trees, if we ignore some edges.

Although the building-block representation technique effectively prevents syntax errors in a visual program, there is still potential for compile-time errors. Such errors, and measures taken to address them, are also discussed in this section.

We begin by discussing a code generation process that assumes that the visual program has no errors.

3.4.5.1 The Code Generation Process

The code generation process does not include the traditional scanning and parsing stages. This is because a stack of blocks representing a program fragment is internally represented as a graph on which an abstract syntax tree can be superimposed. For example, the stack of blocks in Figure 3.7 is represent by the graph

Figure 3.7: A stack of blocks

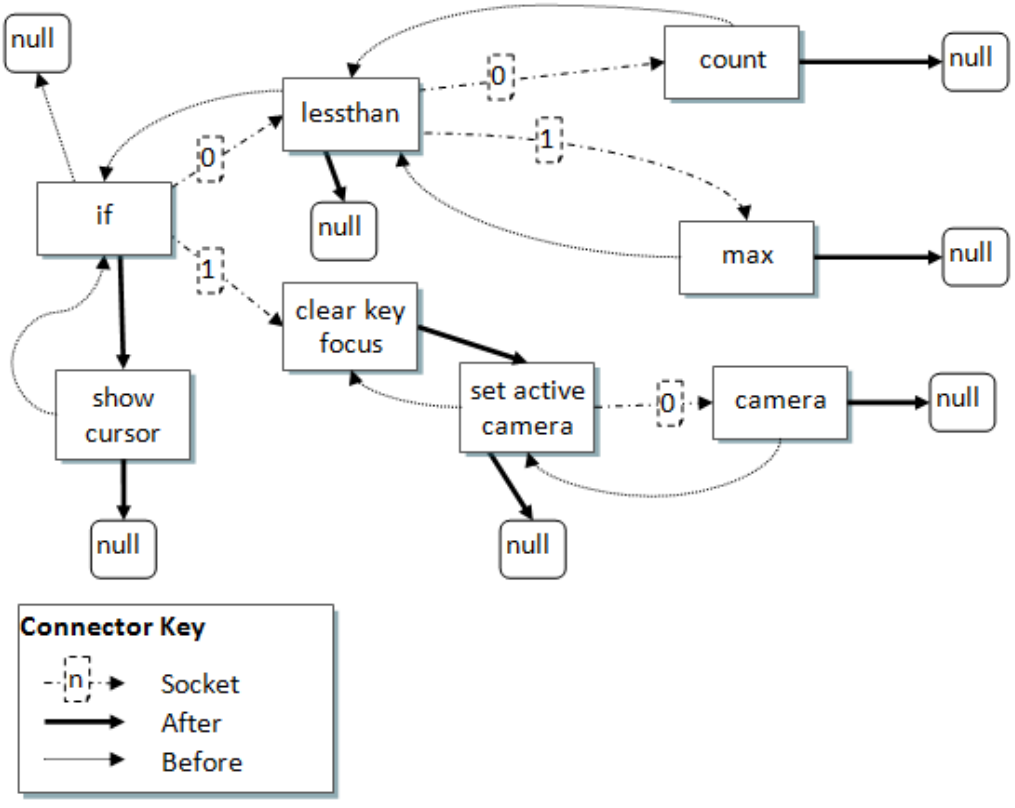

Figure 3.8: An internal representation of the stack of blocks in Figure 3.7

in Figure 3.8, from which the abstract syntax tree in Figure 3.9 is constructed by ignoring the "before" edges and all edges that lead to vertices labelled "null."

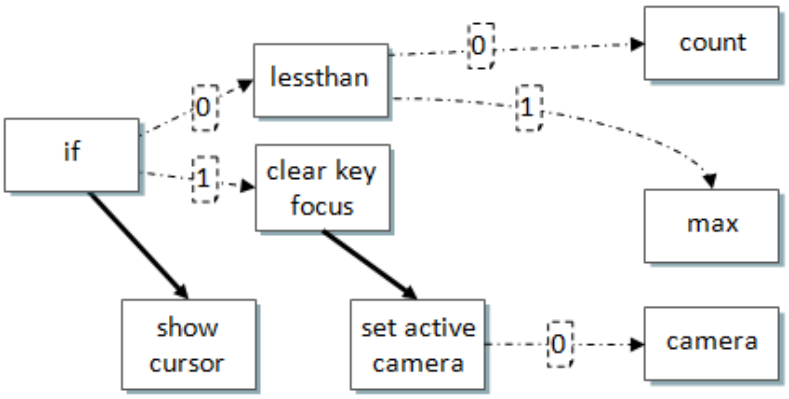

Figure 3.9: An abstract syntax tree extracted from Figure 3.8

In the abstract syntax tree shown in Figure 3.9, the "after" edges connect statements in sequence, whereas the "socket" edges connect internal parts of a single statement.

Algorithm 3.2 traverses the abstract syntax tree while printing out the corresponding Lua code. In the algorithm, `fixedCodeList` is an ordered list of Lua code fragments; `socketList` is an ordered list of socket edges appearing on a node; `isEmpty(list)` returns *true* if the list is empty and *false* otherwise; and `next(list)` returns the next element in the list – the first call to `next(list)` returns the first list element, and the subsequent calls return the subsequent list elements. Note that line 10 is a recursive call.

The first call to Algorithm 3.2 normally passes the top-level block in a stack of blocks. The algorithm completely generates Lua code for the current block before considering a block that is connected to the *after* connector. While generating Lua code for the current block, recursive calls to the algorithm are used to generate Lua code for any blocks connected to sockets appearing on the current block.

The VSE inherits the flexibility of OpenBlocks with respect to the connectedness of blocks appearing on a page. Consequently, a page can have more than one top-level block. In such a case, the top-level block appearing in the topmost part of the page

```
1: while b ≠ NULL do
2:     fixedCodeList ← getFixedCodeList(b)
3:     socketList ← getSockets(b)
4:
5:     if (isEmpty(socketList)) then
6:         print next(fixedCodeList)
7:     else
8:         for all socket in socketList do
9:             print next(fixedCodeList)
10:            generateCode(getBlockAt(socket))
11:            print next(fixedCodeList)
12:        end for
13:    end if
14:
15:    b ← getBlockAfter(b)
16: end while
```

Algorithm 3.2: `generateCode(Block b)`

is processed first. If there is more than one such block, the leftmost of these blocks is processed first. If both the vertical and horizontal placement of top-level blocks coincide, the order of processing such top-level blocks is random.

The other problem emanating from non-insistence on connectedness is that some sockets can be left without plugs when the code generator module is evoked. The next section discusses how this problem is resolved.

3.4.5.2 Code Generator Warnings

The OpenBlocks system guards against the connection of code fragments into a syntactically invalid program. It does not, however, control how a user disconnects blocks. One consequence of this is that some socket connectors might not be connected to any blocks when the code generator module is evoked.

One of the functions of the block verifier module in Figure 3.3 (on page 45) is to look for blocks with empty sockets. If such blocks are found, and have not already been highlighted in red to signify compile-time errors, they are highlighted in yellow in order to warn the user of potential logical errors emanating from the empty sockets.

A program with these warnings still compiles. The missing blocks in the sockets are replaced by default values, shown in Table 3.1, in the generated Lua code.

Type of missing block	Default Lua value
number	0
integer	0
string	`""` (empty string)
boolean	`false`
list	`{}` (empty list)
the rest	`nil`

Table 3.1: Default values for missing blocks

This design prevents forcing the user to connect blocks in the empty sockets, in case the user omits the blocks to imply the default values. The user is however warned, in case the omission is accidental.

The next section discusses problems that may arise due to the flexibility in naming variables and procedures/functions.

3.4.5.3 Resolving Potential Naming Conflicts

The OpenBlocks system does not impose any syntactic rules on identifiers; and thus the user is free to compose variable and procedure names using any combination of characters available from his/her keyboard. Furthermore, variables of different types are represented by blocks having different connector kinds and shapes in order to visually suggest their incompatibility. Since a Boolean and a number variable, for example, are visually represented by incompatible blocks, there is no need for preventing a user from giving the same name to both variables. Unique visual identification is derived from the combination of the textual label and the shape of the block.

The OpenBlocks system only prevents a user from declaring more than one variable of the same type using the same name. Variables of different types may use the same name. This behaviour is inherited by our VSE partly because it is expensive to check against the reuse of an identifier name across multiple types; and the reuse of identifier names across different types is less restrictive to the user.

The code generator transforms the free-styled identifiers to syntactically correct Lua identifiers by replacing any illegal character with the sequence `_charcode`, where `charcode` is the ASCII character code for the illegal character. An exception is the space character, which is replaced by an underscore character only, instead of the sequence `_32`. This scheme was adopted in order to partially preserve the meaning of identifiers in the generated Lua code, in case an expert Lua programmer will want to modify the Lua code.

The potential name conflicts among variables of different types are resolved by prefixing the variable names with a code that identifies the type of the variable. This also makes the Lua code more readable since the intended type of a variable is made explicit by the prefix of the variable name.

The next section discusses errors caused by deletion of blocks representing variable, procedure and function definitions.

3.4.5.4 Identifier Definition Errors

The OpenBlocks system supports the declaration of variables, lists, procedures and functions. There are blocks that are used to declare variables, lists, procedures and functions. Once a declaration has been made, by dragging a declaration block onto a page, the system automatically creates blocks for generating other blocks for manipulating the declared entity. These new blocks are used, for example, for setting the value of a variable and calling a procedure or a function.

A declaration block can be deleted from a page when its corresponding manipulation blocks are still appearing on a page. This causes a situation similar to an attempt to reference a variable or a procedure that has not been declared, in statically typed textual programming languages.

One approach to resolving this problem would be to automatically delete the manipulation blocks appearing on a page whenever a corresponding declaration block has been deleted. This would be inconvenient if, for example, the deletion was accidental. If this approach is adopted, the accidental deletion cannot be rectified by simply replacing the deleted declaration block. All the instances of the manipulation blocks would also need to be redrawn, on the page, and reconnected to the rest of the program.

We adopted the default behaviour of OpenBlocks in our VSE. While deletion of a declaration block removes the corresponding manipulation block generators from the appropriate drawers, the instances of manipulation blocks that were already on a page are left intact. This means that an accidental deletion of a declaration block can be undone by simply placing another declaration block on a page that has the same label as the labels appearing on the corresponding manipulation blocks.

The lack of propagation of deletion of a declaration block to its corresponding manipulation blocks can lead to the "undefined identifier" compile-time error if the target textual programming language is statically typed. Furthermore, a call to an undefined function/procedure constitutes a runtime error in Lua. Although our target textual programming language, Lua, is dynamically typed [36], our VSE is statically typed in order to enhance the simplicity of ensuring correctness in visual programs [47]. Since the building-block representation technique already ensures type consistency, the only potential problem in our design with regard to type checking is the attempt to manipulate undefined entities.

The block verifier module is responsible for searching for variable, list, procedure and function manipulation blocks that do not have corresponding declaration blocks. When such blocks are found, they are highlighted in red and the code generation process is halted. At this point, it is up to the user to resolve the problem by either providing the missing declaration blocks, or removing references to the undeclared entities.

The next section discusses how the VSE can be extended by adding function libraries.

3.4.5.5 Extensibility

The visual programming language in the OpenBlocks library is defined in an XML file. The file defines all the program building block types that a specific implementation has access to. A block type definition includes the following information: the name of the block type; the default label on an instance of the block type; the number and types of connectors on the block; and a classification of the block, which controls the geometry of the block. In addition, the XML file contains the

following: a specification of how to group blocks into families[5] and drawers; and the initial division of the scripting area into pages. Henceforth, we refer to this XML file as *the language defining XML file*. A document type definition (DTD) for the language defining XML file is presented in Appendix A (on page 97).

The OpenBlocks library does not include a code generator; and consequently, the XML file does not include any specification on how to generate a target textual program. The OpenBlocks library, however, has the capability to extract language specific information from the language defining XML file. This language specific information is declared using a special tag in the XML file.

The design of the VSE code generator is divided into two parts. The first part, which we call the core, handles all the blocks that are standard in our visual language. These blocks include: blocks for building algebraic expressions; blocks for specifying flow control; blocks for declaring and manipulating variables, lists, procedures and functions; and blocks representing constants. The other part of the code generator processes blocks representing calls to the game engine's Lua library functions.

We designed the core of the code generator to be inextensible. The specifications of blocks in the core are still defined in XML file; however, there is a tight coupling between the block definitions and the code generator core. This was done deliberately to prevent ad hoc changes to the semantics of our visual language, since such changes would detract from the learnabilty of the VSE's visual language.

The other part of the code generator processes blocks representing Lua library functions. This part needs to be extensible because we anticipate changes to the game engine's Lua library as the game engine gains popularity. We thus decided to place the Lua code fragments, discussed in Algorithm 3.2 (on page 58), in the language defining XML file, for all blocks representing calls to the game engine's Lua library functions. The `getFixedCodeList(b)` function in Algorithm 3.2 (on page 58) extracts the Lua code fragments from the language specific property elements of XML nodes representing blocks representing calls to the game engine's

[5] A block family is a set of blocks in which any instance appearing on a page can be switched with any other member in the set without having to delete the block and dragging a new instance from a block drawer.

Lua library functions. New blocks representing calls to the library functions can thus be added to the XML file without the need to modify the code generator; similarly, blocks representing calls to library functions can be deleted from the XML file without the need to modify the code generator.

3.4.6 Persistent Storage of Visual Programs

The OpenBlocks library has functionality for generating XML text representing the state of the entire script editing area. This XML text contains information on the state of block drawers and the state of pages. This XML text can be saved in a text file; and later on used to reload a visual program.

When loading a visual program, the OpenBlocks system parses XML text representing a visual program into a document object model (DOM) [5] tree. The DOM tree is then processed further to obtain graphs representing stacks of blocks, such as the one shown in Figure 3.8 (on page 56).

StarLogo TNG version 1.2, which is based on the OpenBlocks library, saves each complete visual program in one XML file. This means that whenever a visual program is loaded, the entire DOM tree has to be loaded and kept in main memory until the DOM tree is fully processed to generate the graphs representing a visual program. This arrangement is sufficient in StarLogo TNG 1.2 since each visual program manipulates only one scene.

Our VSE, on the other hand, supports the development of visual programs that manipulate multiple scenes. We extended the OpenBlocks system to allow the generation of XML text representing scene objects. This XML text is added to the XML text representing blocks. We also include XML text representing GUIs, their widgets, and event handler names, in the XML text that needs to be saved. If all this information was saved in a single XML file, and reloaded all at once, the resulting DOM tree would too large to fit in main memory[6] if each scene has a large amount of visual code.

Figure 3.10 shows the division of a single project's XML text into multiple files. Each project is stored in at least three files. The project file is an XML file that

[6]The upper limit was set at 256 MB during testing.

stores file paths to all the other XML files in the project. The project file also records the files holding the section of visual program that was displayed in the script editing area at the time the visual program was saved. The main file holds the part of the visual program that is displayed in the GUI pages for the current project. A scene file holds part of the program pertaining to a single scene.

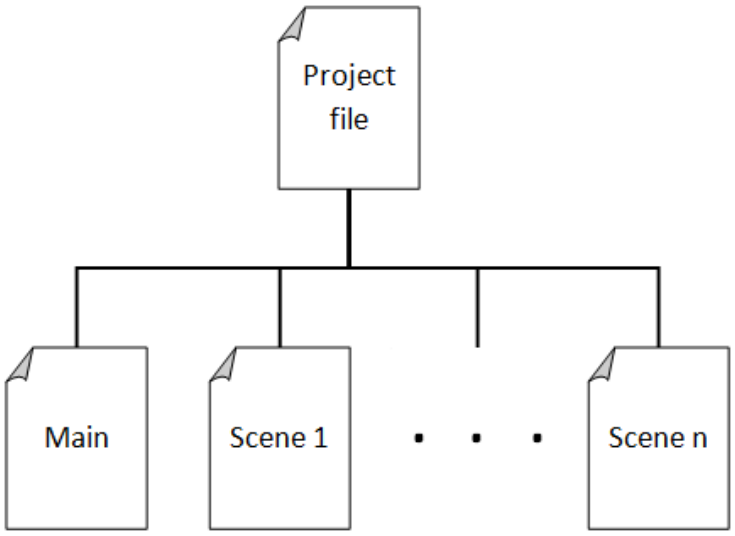

Figure 3.10: Files of a VSE program

When loading a visual program, the project file is loaded first. A project object is created from the contents of the project file. The project object then locates the main file; opens it; and parses it to create a DOM tree. The project object then locates the scene that was open when the program was last saved; and parses its contents into another DOM tree. The two DOM trees are then merged and passed to the OpenBlocks subsystem for the final loading of the visual program represented by this merged DOM tree.

Using this technique optimises memory usage by making sure that only the scene that the user is currently working on is loaded; and the other scenes are purged from memory. GUI pages are always displayed because any scene can refer to any GUI in the program.

The VSE should therefore be able to handle larger visual programs than StarLogo TNG, at least in theory.

3.5 Chapter Summary

In this chapter, we discussed an overview of the development process of games that are executable on the Myoushu game engine. The lack of an IDE and the syntax burden imposed by the textual Lua programming interface motivated the design and development of an IDE, with a visual scripting language.

The IDE, which we call the VSE, is based on the OpenBlocks library. The VSE purports to lower the barriers to programming by representing programs using building blocks that connect only in ways that guarantee syntactically correct programs. The VSE also inspects various resource files, such as scene files and GUI layout files, and automatically generates program building blocks for manipulating their content; thus taking the drudgery of manually inspecting these files from the user.

The OpenBlocks library was extended to, among other things, provide: extra connector shapes that enhance the expressiveness of the visual language without having to reuse connectors in different conflicting contexts; copy and paste functionality through context sensitive pop-up menus; and a mechanism to support the treatment of procedures and functions as first class types – a functional programming concept.

It was observed that the freedom of placing blocks on a page without connecting them to any other blocks is useful because it allows programs to be built in a more flexible manner. This can, however, lead to some blocks having empty sockets, representing missing data, at compile-time. Default values are used in such cases; however, the user is warned in case the default behaviour was not intended.

We also observed that deleting blocks can introduce compile-time errors if a declaration block is deleted while the page still has blocks that refer to the entity whose declaration block has been deleted. Such blocks, whose corresponding declaration blocks have been deleted, are highlighted in red at compile-time; and cause the code generation process to be halted prematurely, to avoid generating incorrect Lua code.

Unlike StarLogo TNG 1.2, which is also based on the OpenBlocks library, the VSE supports programs with multiple scenes; and thus the VSE should be able to

cope with much larger visual programs than those that StarLogo TNG can handle. Consequently, VSE programs are saved in multiple files which are selectively loaded to reduce the main memory footprint.

To demonstrate the fitness of purpose of the VSE, the next chapter discusses the development of an educational game that was previously scripted directly in Lua.

Chapter 4

Example of Usage: Journey to the Moon

This chapter discusses the testing of the VSE through the implementation of an educational game, called *Journey to the Moon*. At the time of testing, there was already an implementation of this game, whose scripts were created directly in Lua. The goal of the testing was to verify that we could reproduce the logic of these scripts using the VSE's visual programming language. We successfully reproduced the scripts, even though we encountered some issues that prompted changes to the original design of the VSE.

4.1 Introduction

Journey to the Moon is an educational game that was developed by Chamberlain [62]. The target audience of the game is grade one pupils. A player of the game does not directly control the main character in the game. The main character, nevertheless, interacts with the player through text and audio. The main character presents the player with problems, which are displayed textually. The problems involve simple arithmetic and language. The player provides answers by typing them using the keyboard. If the player answers five questions correctly, an animation is played and the game proceeds to the next state. If a wrong answer is provided, the same question is repeated for up to three times before a different question is asked.

The storyline of the game is as follows: A scientist receives a message to go to the moon and investigate some mystery. The story is divided into two scenes. In the first scene, the scientist is in his laboratory when he receives the message. His goal for the first scene is to get into a spaceship and take off. He discovers that most of his equipment is malfunctioning, and needs to correct the malfunction by solving some simple arithmetic and language problems. The player's task is to solve these problems on behalf of the scientist. Figure 4.1 is a screenshot from the game showing one of the problems that the player has to solve to unlock a garage door so that the spaceship can take off.

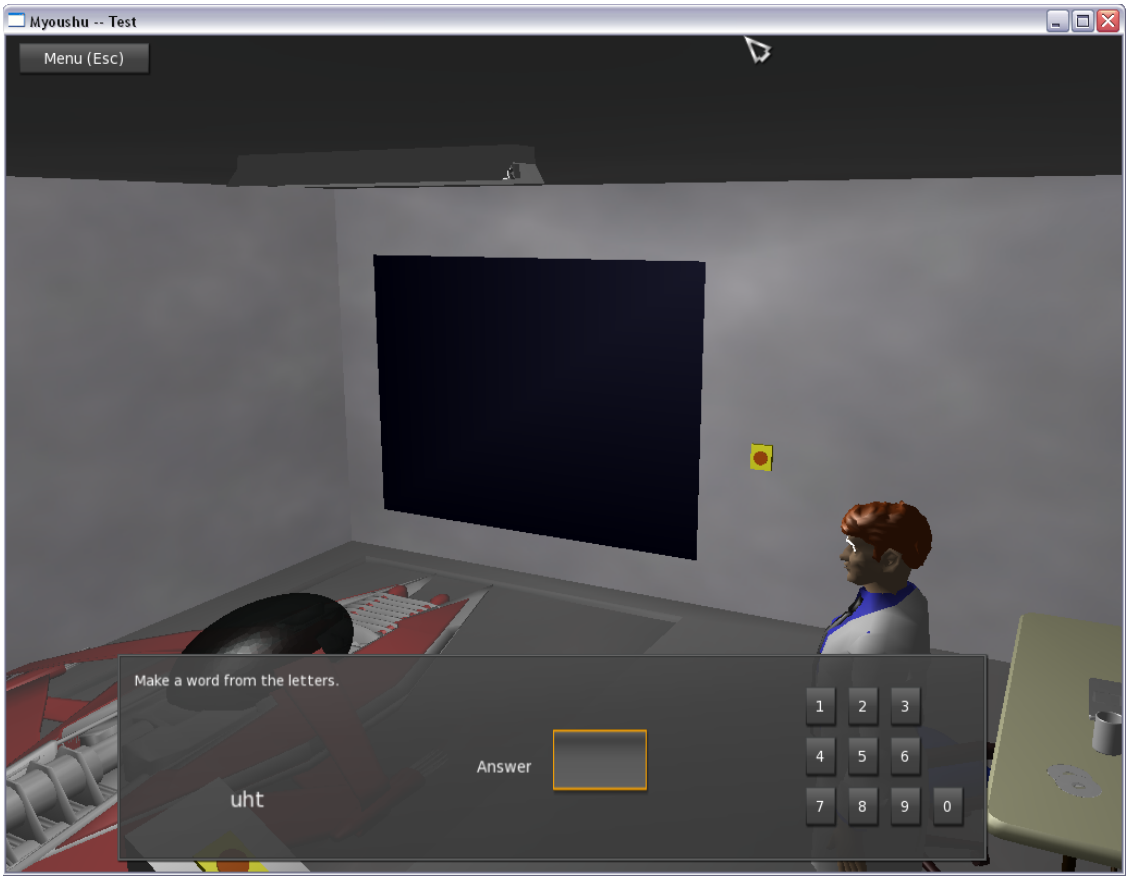

Figure 4.1: A screenshot from *Journey to the Moon* (adapted from [62], page 90)

The first scene ends with the spaceship taking off. In the second scene, the spaceship is flying through space. The spaceship encounters several asteroids in its flight path. The scientist has to dodge or destroy the asteroids to avoid damaging the

spaceship's protective shield. The player has to solve sets of problems to enable the scientist to dodge or destroy the asteroids. Each problem now has a time limit. If the time limit expires before the player provides the correct answer, the asteroid hits the spaceship. If the spaceship is hit by at least three asteroids, the shield is completely destroyed, the mission is aborted, and the second scene restarts.

Journey to the Moon is one of the games that were developed to test the Myoushu game engine. It was selected as our test case because it makes use of more than 80% of the game engine's capabilities [62]. The remaining capabilities, namely networking and replays can easily be added to the VSE by adding appropriate blocks and drawers specifications to the language defining XML file that we discussed in Section 3.3 (on page 40).

4.2 The Goals of Testing

The main aim of scripting *Journey to the Moon* in the VSE was to confirm that the VSE is capable of supporting the specification of logic in VEs developed for the Myoushu engine. The objective is not to test the Myoushu game engine. Rather, we check whether the VSE can support visual expression of game logic, and whether the visual expressions can be converted to Lua code that can be executed by the game engine.

This research does not include a user study. Such a study will be conducted in a follow-up research by others members of the ASD Assist research group. Nevertheless, the success of similar VPEs has been reported in [16, 20, 54].

Our testing was focused on verifying that the design objectives set out in Section 3.2 (on page 39) are met. In particular, we sought to verify that:

1. the majority of scripting commands and expressions are represented by graphical elements, so that scripting can be done using a predominantly drag-and-drop interaction style;

2. programmable scene and GUI artefacts are represented by graphical elements that are compatible with the graphical elements in 1. These visual

elements, representing programmable artefacts, should be automatically generated upon the user loading the corresponding resource files.

3. the user is presented with a simple-to-use interface for managing scenes and their content;

4. the user is presented with a simple-to-use interface for managing GUIs and their content;

5. the process of exporting scripts and other resources to the game engine for execution is automated; and

6. the VSE is able to generate Lua scripts from the graphical programs mentioned in 1. The Lua code should be human-readable to the extent that it can be modified by a skilled Lua programmer.

In addition to these objectives, we also sought to verify that the visual language is extendable without needing to reprogram the code generator. The extensions are mainly in the form of addition of new blocks representing new library functions. These extensions need to be completely expressible in the language defining XML file.

The next section discusses how the scripting of *Journey to the Moon* was tackled to ensure that the goals of testing, discussed above, are met.

4.3 Testing Strategy

There were two aspects to the testing of the VSE. The first concerns the composition of visual programs. We sought to verify that the semantics that was previously expressed directly in Lua could be also be expressed by the VSE's blocks.

The second aspect relates to the semantic equivalence of Lua scripts generated from the VSE's visual programs and those which were previously created directly. The scripts that were previous expressed directly in Lua had no syntax errors, since the Myoushu engine's Lua interpreter was able to parse and execute the scripts. Thus, checking for absence of syntax errors in the generated Lua code was one of the steps towards verifying the equivalence with the original Lua code.

The generated Lua code was tested for syntax errors using a third-party pretty-printer, called LuaFormat [1]. We integrated LuaFormat into the VSE. The VSE's code generator runs LuaFormat on any Lua file that it generates. LuaFormat's primary function is to transform Lua code that is not correctly formatted (indented) to nicely formatted Lua code. LuaFormat includes a Lua parser, and prematurely stops outputting formatted Lua code if the input Lua file contains any syntax error.

We exploited the behaviour of LuaFormat when processing Lua code with syntax errors to check the Lua files, generated by the VSE, for syntax errors. Each group of blocks, whose generated Lua code was under test, was linked to a *procedure* block[1]. Figure 4.2 shows an example of connecting blocks under test to a *procedure* block in order to detect syntax errors. The resulting formatted code is shown in Listing 4.1. Note that a function definition in Lua is terminated by the **end** keyword. If any of the three blocks below the *procedure* block, labelled `Test blocks`, resulted into the generation of Lua code that had syntax errors, LuaFormat would have stopped printing the formatted code before the **end** keyword.

Figure 4.2: Wrapping blocks under test in a procedure to detect syntax errors

```
1  function proc_Test_blocks()
2          guiGetWidget("StaticImageDash"):setCaption("Question");
3  end
```

Listing 4.1: Formatted Lua code generated from the blocks in Figure 4.2

We did not create the entire game from scratch. We reused artefacts such as scene files, GUI layout files, sound files and educational content files. Recall that the

[1] A *procedure* block is used for defining functions and procedures.

VSE is not tasked with creating a game's visual artefacts like game characters and GUIs. There are graphical editing tools for creating these, such as Blender [10] for creating scenes and its objects and characters, and MyGui's Layout Editor [13] for creating GUIs that are used for dialog inside a game's VE. We refer to the collection of scene files, GUI layout files, sound files and educational content files as resource files.

The game scripting process involved creating scripts controlling GUIs used in the game's VE. GUI controlling scripts mainly specify code to be executed when a user interacts with a widget on the GUI; for example, what should happen when a user clicks on a particular button.

We tested the GUI scripting capability of the VSE by first loading the five GUI layout files defined for *Journey to the Moon*. The original Lua files for the GUIs were retrieved, and we tried to recreate the logic using the VSE's block language. The goal was to verify that each referenced widget was represented by a block in the VSE. We also sought to check that the VSE automatically generated event handler definition blocks for all the event handlers that were specified in the GUI layout files.

The scene scripting capability of the VSE was tested in a manner similar to that for testing GUI scripting capability. Scene files (Dotscene files) were first loaded, and then we tried to imitate the code in the scene's corresponding Lua script file. The process was slightly different from the GUI's. Loading a scene does not result in automatic generation of blocks for manipulating the scene content. Blocks for manipulating scene objects are generated upon marking the objects as programmable in the form shown in Figure 4.3. We sought to verify that all objects referenced in the original Lua script file for a particular scene are available from the form in Figure 4.3; and that upon marking an object as programmable, all of its defined animations are automatically represented by appropriate blocks.

Sound files are handled differently. There is no file that lists all sound files used in a game. However, all the sound files used in a game are typically placed in one folder. The testing in this case sought to verify that the VSE automatically searches for sound files, and creates representative blocks, when a user specifies a folder containing the sound files.

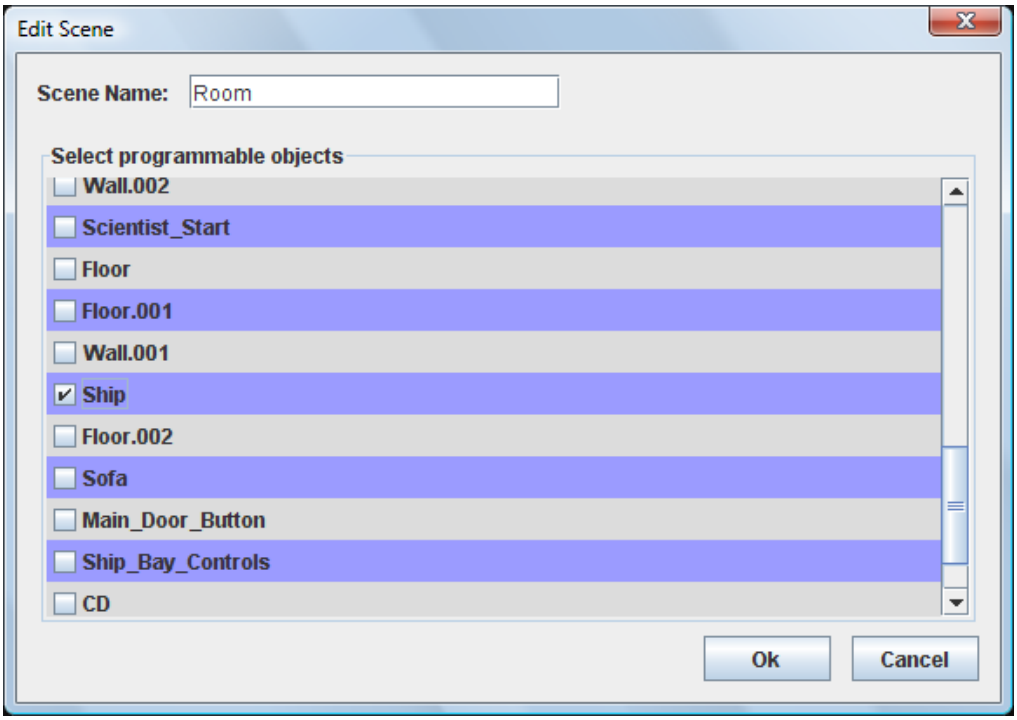

Figure 4.3: Screenshot from the VSE showing a form for marking scene objects as programmable or non-programmable

Educational content is also handled differently. This content is stored at a central location inside the game engine's root folder. The user is thus not responsible for locating and loading the educational content. The VSE provides three blocks for controlling loading of the educational content. Figure 4.4 shows these blocks. The top block loads the content in a specified language[2]; the middle block is for translating the content to Afrikaans; and the block at the bottom is for reverting the content's language to English. The goal of testing this feature was to verify that the VSE generates Lua code that correctly loads and translates the educational content. This was verified by observing the game at runtime to check for language consistency and that only valid (not null) problems were presented in the game.

The development of *Journey to the Moon* was deliberately started when not all the Lua library functions were represented by blocks in the VSE. This was done in order to verify that blocks representing Lua library functions could be added through

[2]The VSE currently supports two languages, namely Afrikaans and English, English being the default language.

Figure 4.4: Blocks for loading and translating educational content

definitions in the language defining XML file only, without having to modify the VSE's code generator.

The next section discusses observations that we made while implementing *Journey to the Moon*'s scripts in the VSE.

4.4 Observations

4.4.1 Literal Constants

The first objective of testing concerned the extent to which visual expressions are used in the VSE. The only time we had to type text was when specifying literal values in blocks representing constant values. There are only three types of literal constants that can be typed from the keyboard, namely number, integer and string. Boolean type has built-in blocks representing the constants *true* and *false*.

The user cannot label blocks representing integers and numbers with invalid integers and numbers, respectively. Whenever a value is being entered for these blocks, the VSE listens to keyboard events and nullifies any keystroke that would invalidate a number or an integer.

When entering string literal constants, we did not have to worry about Lua syntactic constraints on string literals. We simply typed in the characters constituting the string, and the VSE's code generator made sure that the string is converted to a syntactically correct Lua string, while preserving the content of the string. The code generator delimited the string literal with appropriate characters and replaced any special characters with the appropriate escape sequences.

4.4.2 Block-to-Function Mapping

Block-to-function mapping is the assignment of Lua function calls to VSE blocks. We noted that Myoushu's Lua function library had numerous functions. Furthermore, the original Lua scripts for *Journey to the Moon* had direct calls to Myoushu's C function library. If each library function was to be represented by a block in the VSE, the resulting number of building blocks would be overwhelming. The large number of blocks would detract from the learnability of the VSE, thus defeating our goal of lowering the barriers to scripting.

We observed that a significant portion of the library functions, both Lua and C, could be considered *helper* functions, whose call fulfil preconditions for another function's call. We created a new library, which encapsulates a sequence of calls to helper functions and the main function call into a single function call.

Listing 4.2 is an example of a situation that required a higher level library function that encapsulates lower level calls. Listing 4.2 is an excerpt from an original Lua script for controlling the main menu in *Journey to the Moon*. The code fragment's task is to add an entry to a game log entity, `gSessionsEntity`. A log entity's entry is a one-dimensional Lua table. The elements of the table are strings.

```
1  -- Create the new session
2  local values = Myoushu.ListValue()
3  local val = Myoushu.Value()
4  gSessionTimestamp = os.time()
5  val:set(gSessionTimestamp)
6  values:push_back(val);
7  val:setString(gPlayerName)
8  values:push_back(val)
9
10 gSessionsEntity:addEntry(values)
```

Listing 4.2: An excerpt from an original Lua script file controlling the main menu (a GUI) in *Journey to the Moon*

We created a single high-level library function to implement the task being performed by Listing 4.2. The new library function is shown in Listing 4.3. The function accepts two parameters, namely `entity` and `itemList`. Here, `entity` is the log entity to which a new entry is being added, and `itemList` is the new entry, that is, a one-dimensional Lua table of strings.

```
1  function _add_log_entry(entity, itemList)
2          local values = Myoushu.ListValue()
3          local val = Myoushu.Value()
4          local n = _list_length(itemList)
5
6          for i = 1, n do
7                  val:setString(tostring(itemList[i]))
8                  values:push_back(val)
9          end
10
11         entity:addEntry(values)
12 end
```

Listing 4.3: A more task-oriented library function implementing the task in Listing 4.2

After implementing this new library function, the task in Listing 4.2 could then be visually expressed using six blocks, as shown in Figure 4.5.

Figure 4.5: Adding an entry to log entity

4.4.3 Variables and Connector Shapes

In Section 3.4.5.5 (on page 61) we mentioned that blocks for defining and manipulating variables are part of the core of the VSE's block language. However, during the development of the scripts of *Journey to the Moon*, it became apparent that the VSE should support the addition of blocks for defining and manipulating variables through the language defining XML file. If a library function accepts an actual parameter of a new type, then there is a need to add blocks for declaring and manipulating variables of the new type.

As an example, reconsider the code fragment in Figure 4.5. In the figure, the block labelled `gSessionsEntity` reports the value of a log entity variable. If we did not provide blocks for defining and manipulating log entity variables, the only way to access a log entity would be through the `get log entity` block in Figure 4.6. The

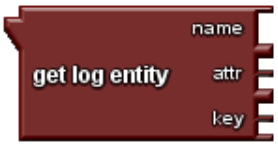

Figure 4.6: Block for retrieving or creating a log entity

`get log entity` block represents a function call that retrieves a log entity having the properties represented by the three blocks connected to it. If such a log entity does not exist, the function creates a new log entity and returns the new log entity. Although it is sufficient to use this function call block every time we reference a log entity, the resulting code would be unnecessarily verbose. The code in Figure 4.5, for example, degrades to the code in Figure 4.7.

Based on this observation, we implemented two changes to the VSE. First, we modified the code generator to allow complete specification of variable declaration and manipulation blocks in the language defining XML file. By complete specification, we mean the specification of the visual appearance and behaviour of the block, as well as the translation into Lua code. Secondly, we created nine additional block connector shapes, bringing the total to thirty. Of these connectors, twenty five have already been used in the current implementation of the visual programming language. The remaining five are reserved for future use.

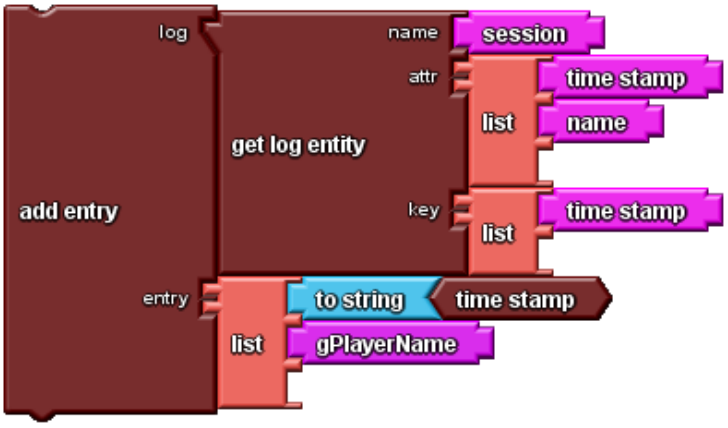

Figure 4.7: The equivalent of the code in Figure 4.6 in the absence of log entity variables

We included unused connector shapes because adding a new connector shape involves two steps, the first of which requires recompilation of the VSE's source code. In the first step, the geometry of the shape is defined. Once the geometry has been defined, the VSE's source code has to be recompiled. The second step assigns a name to the new connector shape. This is done in the language defining XML file, and does not require recompilation of the VSE's source code.

4.4.4 GUIs and Memory

Editing visual scripts for controlling GUIs was found to be memory intensive. When a user loads a GUI layout file into the VSE, the VSE automatically creates blocks representing all widgets on the GUI, and blocks for defining and evoking an event handler for each declared event handler. These blocks are placed in a drawer; and, except for the event handler definition blocks, these blocks act as factories for generating an infinite number of copies. Thus loading a GUI layout file results in the automatic creation of a significant number of blocks, even before the user starts composing the visual script for the GUI. We do not allow the user to mark widgets as programmable and non-programmable, as is the case with scene objects, because a typical GUI controlling script makes reference to the majority of the GUI's widgets.

Figure 4.8 shows part of the visual script controlling the main menu GUI in *Journey to the Moon*. The left side shows some of the blocks representing widgets on

the main menu GUI. These blocks are from the *MainMenu* drawer, which was automatically created when we loaded the GUI layout file for the main menu. The memory footprint of this script was 33.4 MB[3].

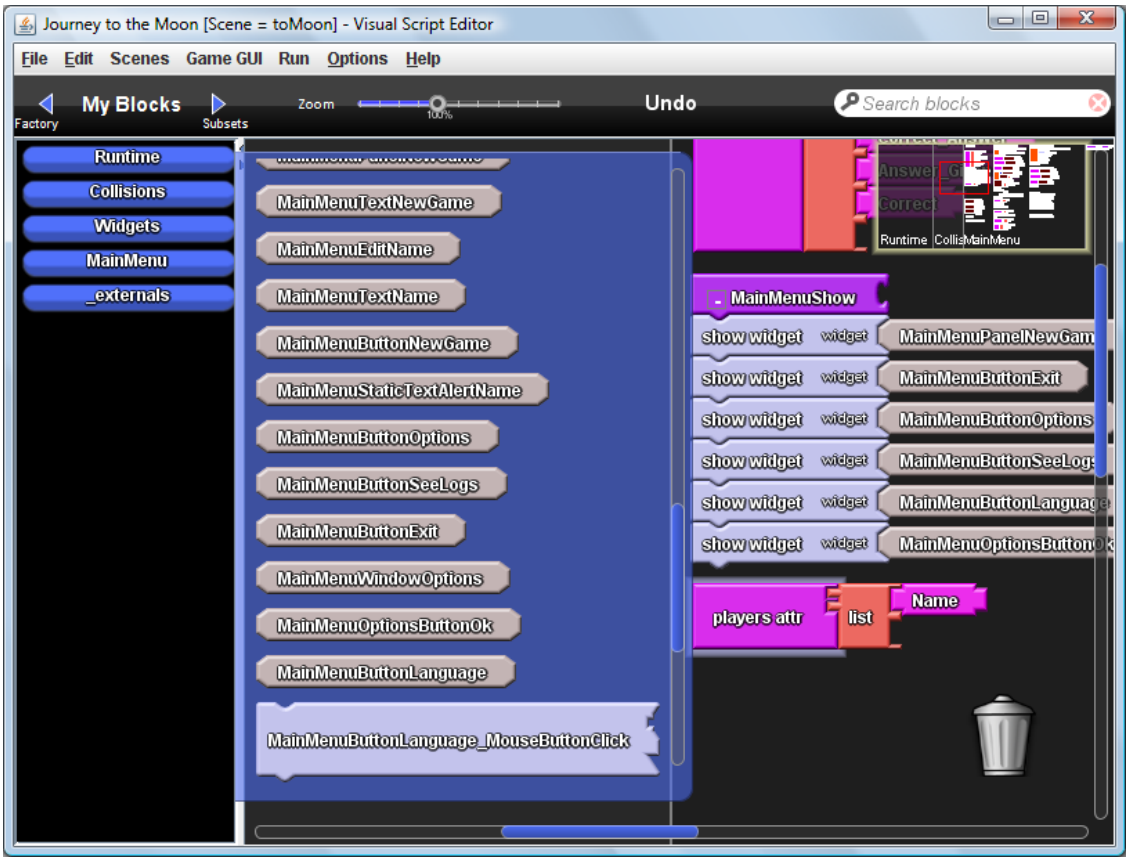

Figure 4.8: GUI widget blocks in a drawer and part of a GUI script

This observation suggested that the project organization model discussed in Section 3.4.6 (on page 63) would lead to excessive memory usage. Consequently, we modified the model, as shown in Figure 4.9. The actual GUI code is no longer contained in the main file. The main file only maintains GUI metadata. Each GUI's script is individually stored in a separate file. With this new arrangement, the user can choose which of the project's GUIs to display in the workspace, thus

[3]We measured this using NetBean's Profiler. This memory estimate is for the GUI code only, and excludes the memory consumed by the VSE (with the block language loaded), which was 22.1 MB.

controlling the amount of memory used by the VSE at any given time. We recommend displaying at most one GUI script at a time. Functions, procedures and global variables and lists defined in the hidden GUI pages are accessible from a special drawer, labelled `_externals`.

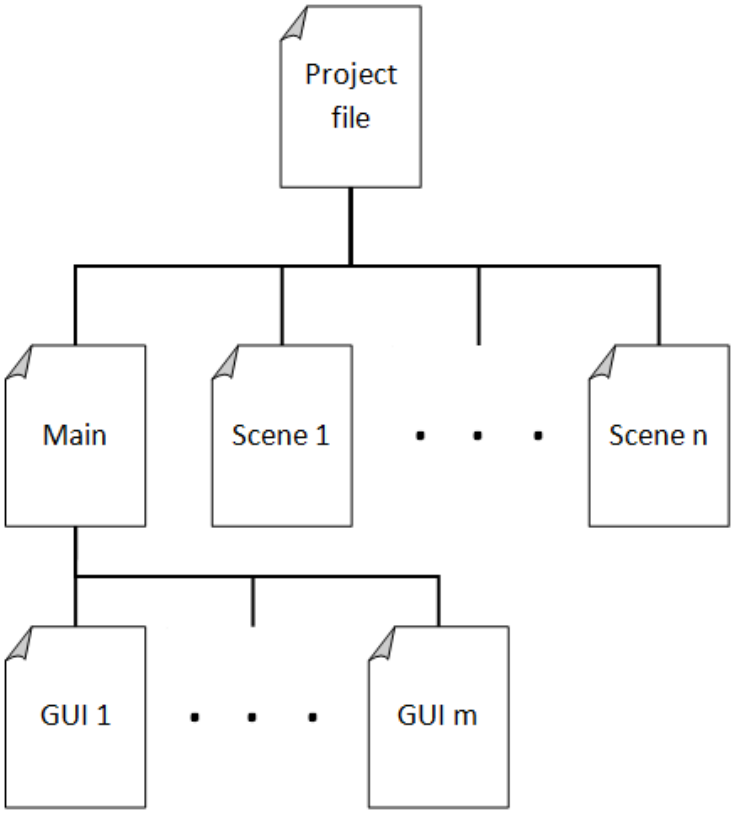

Figure 4.9: Modified project organization

4.4.5 Routine Tasks

We observed that some tasks were routine in specific types of scripts. For example, a GUI script that includes event handlers has to register all the event handlers with the game engine. We excluded such tasks from the programming interface, and let the code generator automatically generate code for such routine tasks. This translates to both lowering the barriers to scripting, as well as enhancing the performance of the VSE by excluding the extra blocks associated with routine code.

4.5 Chapter Summary

In this chapter, we discussed the testing of the VSE through the implementation of an educational game, called *Journey to the Moon*. The implementation of the game was successful; however, we encountered some problems which prompted us to change the original design of the VSE.

We discovered that we could not map the game engine's library functions to VSE blocks in a one-to-one fashion, lest we end up with too many blocks representing library function calls, which would compromise learnability of the system.

We also learnt that addition of blocks representing library function calls can trigger the need for new data types, and consequently new variable types. We modified the code generator to allow specification of new variable types in the language defining XML file. Another direct consequence of this was the need for extra block connector shapes, which we created.

We also noted that GUI visual scripts were memory intensive. We thus modified the structure of projects to make it possible for GUI script pages to be hidden (and thus unloaded from memory) to control the memory footprint of the VSE application.

Finally, we observed that some tasks in the scripts were routine. We excluded such tasks from the visual programming interface, and let the code generator automatically generate code for such tasks.

Chapter 5

Results and Conclusions

5.1 The VSE and the Research Objective

The main objective of our research was to investigate methods to lower the barrier to the development of VEs based on the Myoushu game engine. We focused on the addition of a VPE layer on top of the pre-existing Lua scripting interface. We discuss how the VSE meets this objective in Section 5.1.1 through Section 5.1.6. Section 5.1.7 discusses how the question of extensibility has been addressed.

5.1.1 Elimination of the Syntax Burden

The VSE provides the programmer with all the necessary program building blocks, so that the programmer does not have to type any syntactic structure. A program is composed, mostly, by dragging, dropping and connecting blocks. Blocks can only be connected in ways that assure syntactically correct programs. The connective-compatibility of blocks is visually suggested to the programmer by using a plug-socket connector metaphor. A plug and a socket can connect only if they have complementary shapes.

It may be argued that the plug-socket connector metaphor plays the same role as syntax error highlighting in IDEs such as Eclipse [12] and NetBeans [14]. We consider the plug-socket connector metaphor to be superior, since it prevents syntax errors from occurring; rather than allowing an error to occur, and then inform the

programmer of the syntax error. This idea is in line with the ancient wisdom that prevention is better than cure.

The VSE blocks hide the syntactic structure of the programming constructs they represent. By using these blocks, a programmer cannot, for example, forget to terminate the body of an *if* statement, in Lua, by an **end** keyword.

The VSE helps the programmer prevent syntax and logical errors caused by typographical errors. Listing 5.1 shows a Lua code fragment that was supposed to generate ten random integers and store them in the table, `nums`. The *while* loop in the code fragment cannot terminate because the value of `num` is never changed within the body of the loop. On line 5, the programmer wanted to write `num = num + 1`, but accidentally typed `nun = num + 1`. This statement does not result in an "unknown identifier" error because variable definition is implicit in Lua, thus the first run through the loop defines a new global variable, `nun`. This logical error cannot

```
1  nums = {}
2  num = 0;
3  while (num < 10) do
4          nums[num] = math.random(0, 10)
5          nun = num + 1
6  end
```

Listing 5.1: A non-terminating loop caused by a typographical error

occur in the VSE, unless the variable `nun` was already explicitly defined at the time of composing the loop. A variable name is typed only once, when defining the variable. Subsequent references to the variable are made through variable getter blocks that are automatically generated by the VSE.

Blocks representing algebraic, relational and logical operators have sockets which explicitly indicate the relative positions of their operands. Each operand can be an expression of arbitrary complexity. There is no possibility for ambiguity regarding which sub-expressions any given operator refers to. This eliminates the need for the notion of operator precedence and parenthesis. The programmer can thus build

algebraic, relational and logical expressions without having to remember operator precedence rules of the operators involved, or having to worry about balancing parentheses in a complex expression.

The VSE does not enforce any syntactic constraints on identifiers. Programmers can create identifiers using any combination of characters available from the keyboard. The only restriction is that no two variables of the same type may be defined using the same identifier. The novice programmer does not have to worry about using only syntactically correct identifiers.

Finally, blocks representing string constants allow the string constants to be entered in a direct manner. The programmer is never concerned about syntactic rules for strings, such as: delimiting string literals with the appropriate special character, and using escape sequences in place of special characters.

5.1.2 Hidden Technical Jargon

The VSE's visual programming language hides technical jargon associated with the Myoushu game engine, and 3D graphics in general. Consider the Lua statement in Listing 5.2, for example. The statement moves a scene object, referred to by the name `box01`, forward by `z` units. Before the advent of the VSE, the programmer had to understand how the Myoushu game engine represents scenes before writing code to manipulate scene objects. In this example, the user would need to know that any scene object is contained in an object of type `SceneNode`; that the 3D geometry technical term for movement is *translation*; that translation is specified by a 3D vector and a transformation space; and that self-relative motion uses the transformation space `Myoushu.Node_TS_LOCAL`. This would be too much material to learn for someone new to programming and 3D VEs.

```
1 scene:getSceneNode("box01"):translate(0,0, z,
2                           Myoushu.Node_TS_LOCAL)
```

Listing 5.2: A move forward command

The VSE replaces this statement with the blocks shown in Figure 5.1. The object being moved in this statement is implied from the name of the page on which the `forward` block appears, and the scene is implied from the current scene which the script is controlling. Note how the `Myoushu.Scene` object, `Myoushu.SceneNode` ob-

Figure 5.1: VSE's version of the move forward statement

ject, the `Myoushu.Node_TS_LOCAL` transformation space, the vector and the *translate* jargon are hidden from the programmer.

5.1.3 Provision of Task-Oriented Library Functions

The mapping between blocks representing Myoushu's library function calls and the actual library functions is not one-to-one. It was noted that in several cases, a group of functions implement a single logical task. Each of such group of library functions was encapsulated into one higher-level, and more task-oriented, function which was represented by a single block. Refer to Section 4.4.2 (on page 75) for a more detailed discussion and an example.

This encapsulation of low-level functions into more task-oriented functions lowers the programming barrier by increasing the *terseness* of the VSE's visual programming language. Green and Petre [31] define the diffuseness/terseness of a programming language as a measure of the number of primitives required to implement a task. Green and Petre [31] argue that terse programming languages, requiring fewer primitives to implement a given task, are generally more usable because less material needs to be held in the programmer's limited short-term memory when implementing each of the program's logical tasks.

5.1.4 Static Typing

Although Lua is a dynamically typed programming language, the VSE has a static type system. Variable definition is explicit, and once defined, type consistency is enforced through the plug-socket connector metaphor.

```
num = 0
nums = {}
while num ~= maximum do
        nums[num] = num
        num = num + 1;
end
```

Listing 5.3: A subtle logical error in Lua

Dynamic typing and implicit variable declaration in Lua can cause subtle logical errors, which a novice programmer might find hard to debug. Consider the program fragment in Listing 5.3, in which the author was trying to initialize a Lua table with values 1, 2, 3, $\cdots$, `maximum`. At this point `maximum` is not defined. In Lua, any undefined variable has a default **nil** value. The value **nil** is the only value in Lua's *Nil* data type. Furthermore, Lua allows equality operations on arguments of different types; the result of such operations being *true* in the case of `~=`, and *false* in the case of `==`. The *while* loop in Listing 5.3 thus runs forever until the system runs out of memory and the program crashes.

Such a logical error is irreproducible in the VSE because each operator (including functions and procedures) enforces type consistency, and variable definition is explicit.

5.1.5 Automatic Discovery of Programmable Elements

Lua scripts for controlling scenes and GUIs typically make reference to scene objects and their associated animations, and GUI widgets and event handlers, respectively. When creating these Lua scripts directly (textually), the programmer has to manually examine XML files for scenes and GUIs to discover programmable content. XML files representing scenes (Dotscene files) typically contain hundreds of XML nodes, thus searching for programmable element is an extra burden, and more so to a novice programmer.

The VSE has eliminated this burden by searching these files, on behalf of the programmer, for programmable content, and automatically creating blocks repre-

senting the programmable content in these files

5.1.6 Resource File Management

A Myoushu 3D VE typically comprises several resource files, including: Dotscene files, polygonal mesh model files, skeleton model files, collision mesh files, GUI layout files, sound files, files containing shading and texture information, dialog translation files, educational content files and Lua script files. Maintaining these resource files, and configuration files that point to the locations of these files, can be a daunting task to a novice programmer.

The VSE has taken this drudgery away from the programmer. The VSE keeps track of the resource files on behalf of the user; and is responsible for creating the configuration files when the programmer wants to test a VE on the game engine.

5.1.7 Extensibility

A secondary research objective concerned the extensibility of the VSE through the addition (and removal) of blocks representing library function calls. We sought to simplify the process of adding and removing blocks representing library function calls.

We have successfully decoupled XML specifications of the VSE's visual programming language from the VSE's code generator. New blocks representing new types of variable and library functions can now be fully specified in an XML file that stores the specifications of the VSE's block language. The specifications of the new blocks embed instructions for the code generator module, which inform the code generator how to transform the blocks into the target Lua. Block specifications, representing variable types and library function calls, can thus be added to and removed from the language defining XML file without requiring any changes to the code generator.

The next section discusses the general limitations of the building-block visual program representation technique. The discussion is primarily based on our experience with the OpenBlocks framework.

5.2 Limitations of the Building Block Approach

The OpenBlocks framework has been successfully used to implement our VSE, which has lowered the barriers to the task of creating scripts for VEs based on the Myoushu game engine. Despite this success, the building-block visual program representation technique, which the OpenBlocks framework uses, has limitations that affect the scalability of visual programming languages that are purely based on this representation technique. In this section, we discuss three of these limitations, and argue that the OpenBlocks framework cannot be used to implement an object-oriented general-purpose visual programming language.

5.2.1 Lack of Hierarchy

The plug-socket connector metaphor used on connector shapes makes it impossible for blocks to visually express *is-a-kind-of* relationships. An example of this problem, concerning the representation of floating-point numbers and integers, has been discussed in Section 3.4.1.3 (on page 48). Both floating-point numbers and integers are kinds of numbers. These two kinds of numbers have additional properties that are unique to each kind, which makes some situations demand the use of precisely one kind. For example, counting is normally done using positive integers, whereas the result of real number division is a floating-point value. It would seem most natural to represent the two kinds of numbers using blocks with dissimilar connector shapes to visually emphasize their distinct special properties.

If integer blocks use connector shapes that are different from those on blocks representing floating-point numbers, then all operators defined for floating-point numbers cannot be applied to integers. The reason for this is that an operator for floating-point numbers (for instance, the addition operator) would have sockets that accept only other floating-point numbers. A different version of the operator would need to be defined for integers; and another that accepts an integer as its left side operand and a floating-point number as its right side operand; and yet another version with a floating-point number as its left side operand and an integer as its right side operand. Thus each binary algebraic operator would need to have four different versions. Each function that accepts a generalised number parameter would also need to have two versions, one for integer parameters and the other for

floating-point parameters. This approach would result in an unnecessarily large number of blocks representing algebraic operators.

The solution that we adopted in our VSE was to use the same connector shapes for integers and floating-point numbers. The blocks representing the two kinds of numbers are differentiated using colour. Connection of shapes involving integers and floating-point numbers no longer follows the default plug-socket connector metaphor. We added a rule that overrides the metaphor, namely: floating-point number sockets accept both kinds of number plugs, whereas integer sockets only accept integer plugs.[1] This amounts to connector shape overloading, discussed by Roque in [55], which detracts from the learnability of the visual language.

The problem stems from the rigidity of the plug-socket connector metaphor. A socket should not accept plugs of any type other than the type of the socket. An exception to this rule are the polymorphic sockets, which generally accept plugs of any kind; and polymorphic plugs, which generally fit in any socket kind.

This difficulty in expressing hierarchical relationships is not limited to the representation of numbers. In general, the plug-socket connector metaphor makes it impossible for a visual programming language that is purely based on the building-block approach to visually express the object-oriented programming concept of inheritance.

5.2.2 Connector Shapes

The idea behind using differentiated connector shapes in the plug-socket connector metaphor is to visually suggest, to the programmer, the compatibility of blocks. The compatibility of complementary plugs and sockets should thus be readily discernible at a glance, without undue effort. We believe that a building-block VPE has a limit on the number of different connector shapes, beyond which some of the connector shapes will start to appear too similar to be distinguishable at a glance.

The implication of this observation is that there is a stringent limit on the number of data types that a building-block VPE can visually discriminate using the plug-socket connector metaphor. Thus the concept of user defined types is unlikely to be

[1] Note that connector shapes and connector kinds are different concepts. Although integers and floating-point numbers share a connector shape, their underlying connector kind is different.

feasible in VPEs that are purely based on the building-block visual representation technique and employ the plug-socket connector metaphor. It is thus not surprising that none of the three related VPEs that we surveyed has support for user defined types.[2]

5.2.3 Variable Scoping

The current design of the OpenBlocks framework's GUI does not offer opportunity to implement variable scoping. This problem emanates from the way block drawers and pages are laid out on the GUI. A typical OpenBlocks' GUI has several pages which are placed in a contiguous manner across the width of the screen. A block from any drawer is accessible to any page. When a variable is declared, by dragging a variable declaration block onto a page, the OpenBlocks framework automatically creates blocks for manipulating the variable, and places these blocks in a dynamic drawer that has the same name as the page on which the declaration block appears. Since a block from any drawer can be dragged and dropped onto any page on the GUI, the scope of any declared variable is global.

For small programs, the lack of local variables is not a major concern. However, the problem is worrisome in the context of function/procedure formal parameters. The OpenBlocks framework treats formal parameters as constant value parameters, and does not provide blocks for setting or changing the value of a formal parameter. These formal parameters are considered local to the function/procedure that declares them, at least in theory. In practice, however, declaration of a formal parameter (by connecting a *parameter* block to a *procedure* block), results in the OpenBlocks framework creating a block for retrieving the value of the parameter. This retrieval block is placed in a dynamic drawer that is labelled with the same name as the page on which the *parameter* declaration block appears. Consequently, the *parameter* block can be dragged and dropped onto any page, and any location within the page.

Figure 5.2 shows a formal parameter, `test par`, which is declared in the function

[2]Recall that Alice does not use the plug-socket connector metaphor. We suspect that Alice's lack of support for user-defined types is due the use of built-in expression builders that ensure the syntactic correctness of expressions. It would be impossible for user-defined types to be created without updating the rules that the expression builders use.

`parameter demo`, and has been used outside the function body in the `load edu content` statement. The identifier `test par` is undefined in the context of the `load edu content` statement. However, this does not constitute a syntax error in the Lua code fragment generated from this statement; rather, it is a logical error since the value **nil** will eventually be used to index a Lua table represented by the `language list` block.

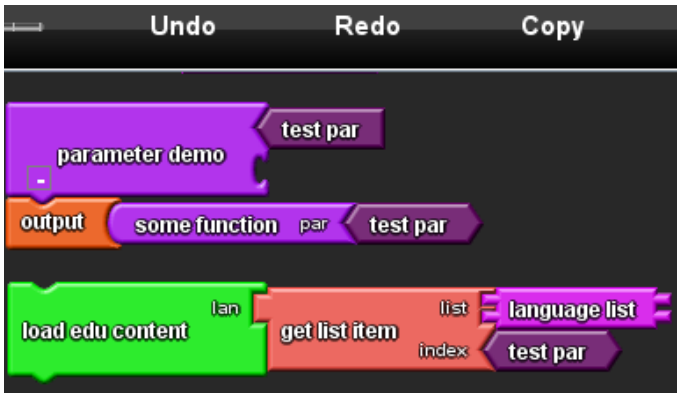

Figure 5.2: A screenshot from the VSE showing a logical error emanating from the lack of mechanisms to control the scope of formal parameters

The lack of variable scoping mechanism in the OpenBlocks framework also has the potential to limit the size of visual programs that can be created, before access control to the shared state becomes an issue.

The next section discusses a proposal on how the variable scoping problem can be addressed in future.

5.3 Future Work

The previous section, Section 5.2.3, explains the difficulty in enforcing variable scoping in the OpenBlocks framework. In summary, variable scoping cannot be enforced in a straightforward manner due to the centrality of block drawers with respect to pages and the page contents. The lack of variable scoping has the potential to limit the size of programs that can be developed using the OpenBlocks

framework; and in the worst case scenario, the lack a visual expression for limiting variable scope can result in logical errors (if a dynamically typed underlying language is assumed, otherwise syntax errors) due to misplaced formal parameters.

We propose visually limiting the scope of variables at three levels, namely: global level, page level and local level. These are discussed further in the next three sections.

5.3.1 Global Variables

Global variables represent the status quo. These will be accessible to any page, and any section within the page. When global variables are defined, their manipulation blocks shall be accessible from centralized block drawers, as is the case in the current implementation of the OpenBlocks framework.

5.3.2 Page Variables

The scope of a page variable shall be limited to the page on which the variable is defined. A page variable shall represent the same concept as a private static

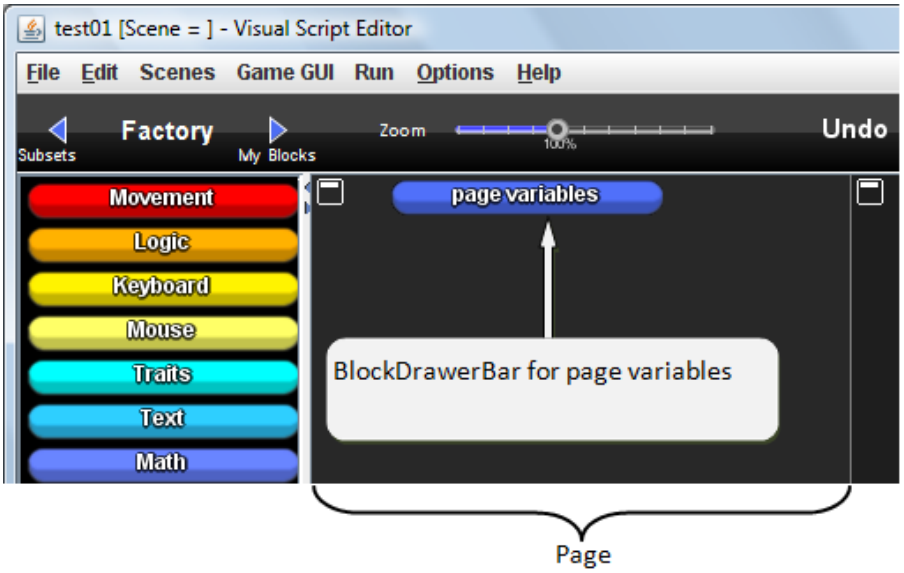

Figure 5.3: Page variables

variable in an object-oriented programming language. Each page will have its own

local drawer for holding blocks for manipulating page variables. Figure 5.3 shows a page with a *BlockDrawerBar* for accessing its local drawer. All blocks accessed from a page's local drawer can only be dragged and dropped within the page containing the local drawer.

5.3.3 Local Variables

Local variables shall have the most limited scope. A local scope can be limited by two blocks, namely: a general-purpose scope delimiter (shown in Figure 5.4) and a procedure definition block (shown in Figure 5.5). Each scope delimiting block will have an inner panel (sub-page) for holding other blocks. A variable defined inside an inner panel can only be dragged and dropped only within the panel. The scope delimiting block will have its own local drawer for holding blocks for manipulating local variables.

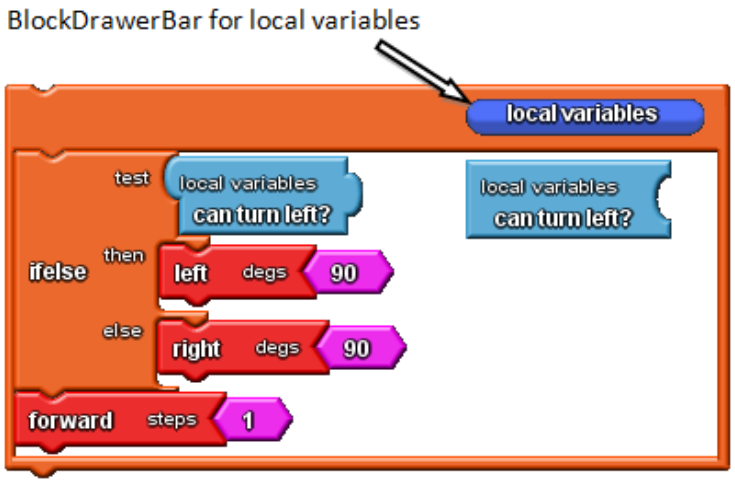

Figure 5.4: A local variable scope-delimiting block

Formal parameter blocks that are connected to the new procedure definition block will be treated like local variables, and thus getter blocks for the formal parameters will be accessible only from the local drawer of the procedure definition block.

General-purpose local scope-delimiting blocks can be nested as shown in Figure 5.6. In this case, variables defined in the outer scope-delimiting block shall be accessible in the inner scope-delimiting block.

Figure 5.5: A local variable scope-delimiting procedure block

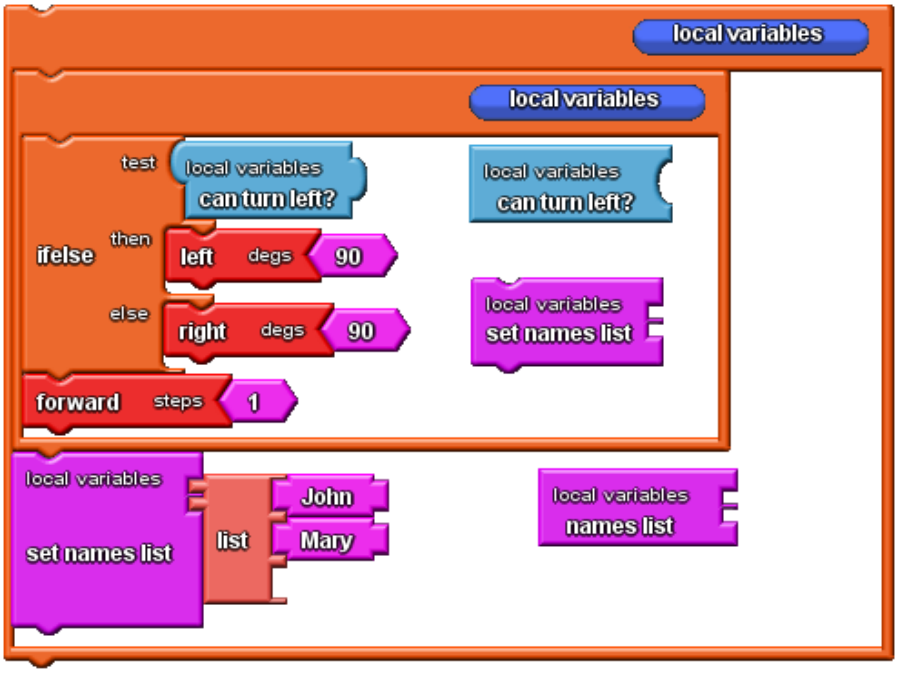

Figure 5.6: Nested scope-delimiting blocks

5.4 Conclusion

The main aim of our research was to investigate methods to lower the barriers to the development of VEs based on the Myoushu game engine.

The outcome of the research was a prototype VPE, called the visual script editor (VSE). The VSE significantly lowered the barriers to the creation of Lua scripts, for

controlling VEs' logic, by allowing scripts to be constructed from jigsaw puzzle-like building blocks that connect only in ways that guarantee syntactic correctness of the scripts; thus the novice programmer can concentrate on the logic rather than the syntax.

The VSE further lowers the scripting barrier by:

1. Hiding technical jargon associated with 3D graphics in the context of the Myoushu game engine;
2. Reducing the number of primitive operations by providing more task-oriented library functions;
3. Superimposing static typing over the dynamically typed Lua, thus making it impossible for scripts to have logical errors emanating from type inconsistency;
4. Automatically scanning resource files and providing the user with blocks representing programmable content; and
5. Managing game configurations on behalf of the programmer.

The VSE is based on the OpenBlocks framework, which exhibits several weaknesses that might affect the size of programs that can be developed in the VSE. These weaknesses include: lack of visual expressions for hierarchy; a stringent upper limit on the number possible connector shapes (and thus an upper limit on the possible number of data types); and variable scoping. These limitations are not significant when the application domain is constrained, as was the case in our investigation. Furthermore, we suspect that if a novice programmer starts worrying about issues like inheritance, user-defined types and variable scoping, then it may be that the *novice* assumption is no longer valid; thus the programmer needs to graduate to the more powerful textual programming languages.

Appendices

Appendix A

A DTD for the VSE's Language Definition XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
    Adapted from StarLogo TNG's lang_def.dtd
-->

<!ELEMENT BlockLangDef ( BlockConnectorShapes, BlockGenuses,
    BlockFamilies?, BlockDrawerSets?, Pages?, TrashCan?, MiniMap?)>

<!--This defines a mapping from block connector shape type to
    number
-->
<!ELEMENT BlockConnectorShapes (BlockConnectorShape*)>
<!ELEMENT BlockConnectorShape EMPTY>
<!ATTLIST BlockConnectorShape shape-type CDATA #REQUIRED>
<!ATTLIST BlockConnectorShape shape-number CDATA #REQUIRED>

<!ELEMENT BlockGenuses (BlockGenus*)>

<!--This defines a single block genus-->
<!ELEMENT BlockGenus (description?, BlockConnectors?,Stubs?, Images?,
    LangSpecProperties?)>
<!ATTLIST BlockGenus name CDATA #REQUIRED>
<!ATTLIST BlockGenus initlabel CDATA #REQUIRED>

<!-- the kind of a genus can affect the rendering of a block.
```

```
     relevant kinds are:
    - command: performs an operation and may take in more than one
               input
    - data: returns primitive values such as number, string, boolean
    - function: takes in an input and performs an operation to produce
                an ouput
-->
<!ATTLIST BlockGenus kind CDATA #REQUIRED>
<!ATTLIST BlockGenus color CDATA #REQUIRED>
<!ATTLIST BlockGenus editable-label (yes|no) "no">
<!ATTLIST BlockGenus label-unique (yes|no) "no">
<!ATTLIST BlockGenus is-label-value (yes|no) "no">
<!ATTLIST BlockGenus label-prefix CDATA #IMPLIED>
<!ATTLIST BlockGenus label-suffix CDATA #IMPLIED>
<!ATTLIST BlockGenus page-label-enabled (yes|no) "no">

<!--is-starter and is-terminator only apply to blocks of kind: command
-->
<!ATTLIST BlockGenus is-starter (yes|no) "no">
<!ATTLIST BlockGenus is-terminator (yes|no) "no">

<!--This defines a block description and the description of its block
    arguments
-->
<!ELEMENT arg EMPTY>
<!ATTLIST arg n CDATA #REQUIRED name CDATA #IMPLIED>

<!ELEMENT description (text, arg-description*)>
<!ELEMENT text (#PCDATA|note|em|i|br|arg)*>
<!ELEMENT arg-description (#PCDATA|em|i|arg)*>
<!ATTLIST arg-description n CDATA #IMPLIED name CDATA #IMPLIED
    doc-name CDATA #IMPLIED>
<!ELEMENT note (#PCDATA|arg|i)*>
<!ELEMENT em (#PCDATA)>
<!ELEMENT i (#PCDATA)>
<!ELEMENT br (#PCDATA)>

<!--BlockConnectors are where blocks get connected-->
<!ELEMENT BlockConnectors (BlockConnector*)>
<!ELEMENT BlockConnector (DefaultArg?)>
<!ATTLIST BlockConnector label CDATA #IMPLIED>
<!ATTLIST BlockConnector label-editable (yes|no) "no">

<!-- Order matters with socket connectors and at most one plug is
```

```
     allowed (no multiple return types)
-->
<!ATTLIST BlockConnector connector-kind (plug|socket) #REQUIRED>

<!-- for connector-type use the shape-type values specified in block
    connectors
-->
<!ATTLIST BlockConnector connector-type CDATA #REQUIRED>
<!ATTLIST BlockConnector position-type (single|mirror|bottom) "single">
<!ATTLIST BlockConnector is-expandable (yes|no) "no">

<!ELEMENT DefaultArg EMPTY>
<!ATTLIST DefaultArg genus-name CDATA #REQUIRED>
<!ATTLIST DefaultArg label CDATA #IMPLIED>

<!ELEMENT Stubs (Stub*)>

<!--This defines a stub of a block, so that the block can exist as a
    single entity and have mini-references to it-->
<!ELEMENT Stub  (LangSpecProperties)>
<!ATTLIST Stub scope CDATA #IMPLIED>
<!ATTLIST Stub stub-genus (getter|setter|caller|agent|inc) #REQUIRED>

<!-- Defines the images that are drawn on the block itself.
     Note: For now, only one image is enabled and wrap-text and
     image-editable have no effect.
     Note: make sure FileLocation specified is relative to workspace
     directory -->
<!ELEMENT Images (Image)>
<!ELEMENT Image (FileLocation)>
<!ATTLIST Image wrap-text (yes|no) "no">
<!ATTLIST Image image-editable (yes|no) "no">
<!ATTLIST Image block-location (center|east|west|north|south|
    southeast|southwest|northeast|northwest) "center">
<!ATTLIST Image width CDATA #IMPLIED>
<!ATTLIST Image height CDATA #IMPLIED>
<!ELEMENT FileLocation (#PCDATA)>

<!ELEMENT LangSpecProperties (LangSpecProperty*)>
<!ELEMENT LangSpecProperty (#PCDATA)>
<!ATTLIST LangSpecProperty key CDATA #REQUIRED>
<!ATTLIST LangSpecProperty value CDATA #REQUIRED>

<!--This defines a BlockGenus Family-->
```

```
<!ELEMENT BlockFamilies (BlockFamily*)>
<!ELEMENT BlockFamily (FamilyMember*)>
<!ELEMENT FamilyMember (#PCDATA)>

<!-- Defines BlockDrawerSets and their Block Drawer content-->
<!ELEMENT BlockDrawerSets (BlockDrawerSet*)>
<!ELEMENT BlockDrawerSet (BlockDrawer*)>
<!ATTLIST BlockDrawerSet type (bar|stack) "bar">
<!ATTLIST BlockDrawerSet name CDATA #REQUIRED>
<!ATTLIST BlockDrawerSet location (east|west|north|south|northeast|
    southeast|southwest|northwest) "west">

<!-- window-per-drawer specifies if each drawer should be its own
     draggable window.  otherwise, all the drawers are contained
     within one draggable window and only one drawer can be opened at
     a time. Whether the window is draggable depends on whether
     drawer-draggable is set to "yes."
-->
<!ATTLIST BlockDrawerSet window-per-drawer (yes|no) "yes">
<!ATTLIST BlockDrawerSet drawer-draggable (yes|no) "yes">

<!--This defines BlockDrawers and their content-->
<!ELEMENT BlockDrawer ( (BlockGenusMember | Separator | NextLine)* )>
<!ATTLIST BlockDrawer name CDATA #REQUIRED>
<!ATTLIST BlockDrawer type (default|factory|page|custom) "default">
<!ATTLIST BlockDrawer is-open (yes|no) "no">
<!ATTLIST BlockDrawer button-color CDATA #REQUIRED>
<!ELEMENT BlockGenusMember (#PCDATA)>
<!ELEMENT Separator EMPTY>
<!ELEMENT NextLine EMPTY>

<!-- Defines Pages dividing the Block Canvas and the optional
     PageDrawers associated with them. Each Page can have only
     one PageDrawer. For now, either every page must have a drawer,
     or no pages can have drawers. The block canvas need not
     contain any pages.  You may choose to have a blank canvas
     instead of a canvas of pages.
-->
<!ELEMENT Pages (Page*)>

<!-- drawer-with-page auto generates a new drawer for each new page
     created by a user and creates an empty drawer for each page that
     does not specify a page drawer
-->
```

```
<!ATTLIST Pages drawer-with-page (yes|no) "no">
<!ELEMENT Page (PageDrawer?)>
<!ATTLIST Page page-name CDATA #REQUIRED>
<!ATTLIST Page page-width CDATA #REQUIRED>
<!ATTLIST Page page-drawer CDATA #IMPLIED>
<!ATTLIST Page page-color CDATA #IMPLIED>
<!ATTLIST Page page-shape CDATA #IMPLIED>

<!ELEMENT PageDrawer (BlockGenusMember*)>

<!-- If specified a trashcan will appear on the workspace.
           For both of its child elements, a location for the images
     should be specified relative to the working directory. The
     open trash image appears when a user drags a block over the
     trashcan.  The closed trash image is the default image during
     steady state.
-->
<!ELEMENT TrashCan (OpenTrashImage, ClosedTrashImage)>
<!ELEMENT OpenTrashImage (#PCDATA)>
<!ELEMENT ClosedTrashImage (#PCDATA)>

<!-- By default, a MiniMap will always appear in the upper right
     corner of the block canvas, unless enabled is set to "no."
-->
<!ELEMENT MiniMap EMPTY>
<!ATTLIST MiniMap enabled (yes|no) "yes">

<!-- By default, typeblocking will be enabled, such that when the user
     types onto the canvas blocks will fly out that match the entered
     text.
-->
<!ELEMENT Typeblocking EMPTY>
<!ATTLIST Typeblocking enabled (yes|no) "yes">
```

Appendix B

A DTD for the VSE's Project XML Files

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Root element -->
<!ELEMENT Project (SceneFiles, LastScene, MainPath)>

<!ELEMENT SceneFiles (SceneFile*)>
<!ELEMENT SceneFile EMPTY>
<!ATTLIST SceneFile name CDATA #REQUIRED>
<!ATTLIST SceneFile path CDATA #REQUIRED>
<!ELEMENT LastScene EMPTY>
<!ATTLIST LastScene name CDATA #REQUIRED>
<!ATTLIST LastScene path CDATA #REQUIRED>
<!ELEMENT MainPath EMPTY>
<!ATTLIST MainPath path CDATA #REQUIRED>
```

Appendix C

A DTD for the VSE's Main XML Files

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Root element -->
<!ELEMENT GUIs (MyGUI*)>

<!ELEMENT MyGUI (Widget*)>
<!ATTLIST MyGUI type CDATA #REQUIRED>
<!ATTLIST MyGUI name CDATA #REQUIRED>
<!-- load determines whether the GUI's page should be displayed in the
     workspace.
-->
<!ATTLIST MyGUI load (true|false)>

<!ELEMENT Widget (UserString*)>
<!ATTLIST Widget type (StaticText|Edit|Button|Window|Widget|
    StaticImage|MultiList|List) #REQUIRED>
<!ATTLIST Widget name CDATA #REQUIRED>

<!ELEMENT UserString EMPTY>
<!ATTLIST UserString key (eventButtonClick|eventKeyButtonReleased|
    eventListSelectAccept|eventUnknown) #REQUIRED>
<!ATTLIST UserString value CDATA #REQUIRED>
```

Appendix D

A DTD for the VSE's XML Files for Saving Scene Blocks

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Root element -->
<!ELEMENT CODEBLOCKS (Pages, Scene)>

<!ELEMENT Pages (Page+)>
<!-- NOTE: Exclusive saving (where only changed information or
     information that differs from the language definition file is
     saved) is not enabled for pages YET
-->
<!ATTLIST Pages is-blank-page (yes|no) "no">
<!ELEMENT Page (PageBlocks?)>
<!ATTLIST Page page-name CDATA #REQUIRED>
<!ATTLIST Page page-width CDATA #REQUIRED>
<!ATTLIST Page page-color CDATA #REQUIRED>
<!ATTLIST Page page-drawer CDATA #IMPLIED>

<!ELEMENT PageBlocks (Blocks*, BlockStub*)>

<!-- If the Block is a BlockStub, then specify the following -->
<!ELEMENT BlockStub (StubParentName, StubParentGenus, Block)>
<!ELEMENT StubParentName (#PCDATA)>
<!ELEMENT StubParentGenus (#PCDATA)>

<!ELEMENT Block (Label?, PageLabel?, Location?, BoxSize?, Collapsed?,
    Comment?, BeforeBlockId?, AfterBlockId?, CompilerErrorMsg?, Plug?,
```

```
    Sockets*, BlockStubInfo?, LangSpecProperties?)>
<!ATTLIST Block id CDATA #REQUIRED>
<!ATTLIST Block genus-name CDATA #REQUIRED>
<!ATTLIST Block has-focus (yes|no) "no">

<!ELEMENT Label (#CDATA)>
<!ELEMENT PageLabel (#CDATA)>
<!ELEMENT CompilerErrorMsg (#CDATA)>

<!-- x, y  location within the block's page-->
<!ELEMENT Location (X, Y)>
<!ELEMENT X (#CDATA)>
<!ELEMENT Y (#CDATA)>

<!-- width, height box size -->
<!ELEMENT BoxSize (Width, Height)>
<!ELEMENT Width (#CDATA)>
<!ELEMENT Height (#CDATA)>

<!-- existence of this element implies the block is collapsed -->
<!ELEMENT Collapsed EMPTY>

<!-- Comment widget associated with a Block -->
<!ELEMENT Comment (Text, Location, Collapsed?)>
<!ELEMENT Text (#PCDATA)>

<!--  Only the Block ID of the block connected at these connectors are
      specified.   -->
<!ELEMENT BeforeBlockId (#CDATA)>
<!ELEMENT AfterBlockId (#CDATA)>

<!ELEMENT Plug (BlockConnector)>
<!ELEMENT Sockets (BlockConnector*)>

<!-- NOTE: Exclusive saving is not enabled for Block Connectors YET -->
<!ELEMENT BlockConnector (EMPTY)>
<!ATTLIST BlockConnector label CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-kind (plug|socket) "socket">
<!ATTLIST BlockConnector init-type CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-type CDATA #IMPLIED>
<!ATTLIST BlockConnector con-block-id CDATA #IMPLIED>
<!ATTLIST BlockConnector position-type (single|mirror|bottom) #IMPLIED>
<!ATTLIST BlockConnector is-expandable (yes|no) #IMPLIED>
```

```
<!ELEMENT LangSpecProperties (LangSpecProperty*)>
<!ELEMENT LangSpecProperty (#PCDATA)>
<!ATTLIST LangSpecProperty key CDATA #REQUIRED>
<!ATTLIST LangSpecProperty value CDATA #REQUIRED>

<!-- Scene Content -->
<!ELEMENT Scene (SceneObj*, camera*)>
<!ATTLIST Scene name CDATA #REQUIRED>
<!ELEMENT SceneObj (Name,Programmable,Animations)>
<!ELEMENT Name (CDATA)>
<!ELEMENT Programmable (true|false|yes|no)>
<!ELEMENT Animations (Animation*)>
<!ELEMENT Animation (CDATA)>
<!ELEMENT camera EMPTY>
<!ATTLIST camera name CDATA>
```

Appendix E

A DTD for the VSE's XML Files for Saving GUI Blocks

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Adapted from StarLogo TNG's save_format.dtd
-->

<!-- Root element -->
<!ELEMENT Page (PageBlocks?)>

<!ATTLIST Page page-name CDATA #REQUIRED>
<!ATTLIST Page page-width CDATA #REQUIRED>
<!ATTLIST Page page-color CDATA #REQUIRED>
<!ATTLIST Page page-drawer CDATA #IMPLIED>

<!ELEMENT PageBlocks (Blocks*, BlockStub*)>

<!-- If the Block is a BlockStub, then specify the following -->
<!ELEMENT BlockStub (StubParentName, StubParentGenus, Block)>
<!ELEMENT StubParentName (#PCDATA)>
<!ELEMENT StubParentGenus (#PCDATA)>

<!ELEMENT Block (Label?, PageLabel?, Location?, BoxSize?, Collapsed?,
    Comment?, BeforeBlockId?, AfterBlockId?, CompilerErrorMsg?, Plug?,
    Sockets*, BlockStubInfo?, LangSpecProperties?)>
<!ATTLIST Block id CDATA #REQUIRED>
<!ATTLIST Block genus-name CDATA #REQUIRED>
<!ATTLIST Block has-focus (yes|no) "no">
```

```
<!ELEMENT Label (#CDATA)>
<!ELEMENT PageLabel (#CDATA)>
<!ELEMENT CompilerErrorMsg (#CDATA)>

<!-- x, y  location within the block's page-->
<!ELEMENT Location (X, Y)>
<!ELEMENT X (#CDATA)>
<!ELEMENT Y (#CDATA)>

<!-- width, height box size -->
<!ELEMENT BoxSize (Width, Height)>
<!ELEMENT Width (#CDATA)>
<!ELEMENT Height (#CDATA)>

<!-- existence of this element implies the block is collapsed -->
<!ELEMENT Collapsed>

<!-- Comment widget associated with a Block -->
<!ELEMENT Comment (Text, Location, Collapsed?)>
<!ELEMENT Text (#PCDATA)>

<!--  Only the Block ID of the block connected at these connectors are
      specified.   -->
<!ELEMENT BeforeBlockId (#CDATA)>
<!ELEMENT AfterBlockId (#CDATA)>

<!ELEMENT Plug (BlockConnector)>
<!ELEMENT Sockets (BlockConnector*)>

<!-- NOTE: Exclusive saving is not enabled for Block Connectors YET -->
<!ELEMENT BlockConnector (EMPTY)>
<!ATTLIST BlockConnector label CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-kind (plug|socket) "socket">
<!ATTLIST BlockConnector init-type CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-type CDATA #IMPLIED>
<!ATTLIST BlockConnector con-block-id CDATA #IMPLIED>
<!ATTLIST BlockConnector position-type (single|mirror|bottom) #IMPLIED>
<!ATTLIST BlockConnector is-expandable (yes|no) #IMPLIED>

<!ELEMENT LangSpecProperties (LangSpecProperty*)>
<!ELEMENT LangSpecProperty (#PCDATA)>
<!ATTLIST LangSpecProperty key CDATA #REQUIRED>
<!ATTLIST LangSpecProperty value CDATA #REQUIRED>
```

Bibliography

[1] Lua 5.1 Prettyprinter, last accessed in August 2011. http://sites.google.com/site/wowsaiket/Other/LuaFormat-5-1.

[2] Alice: educational software that teaches students programming in a 3D environment, last accessed in July 2011. http://www.alice.org.

[3] App Inventor, last accessed in July 2011. http://appinventor.googlelabs.com/about/.

[4] StarLogo TNG, last accessed in July 2011. http://education.mit.edu/projects/starlogo-tng.

[5] XML DOM tutorial, last accessed in July 2011. http://www.w3schools.com/dom/default.asp.

[6] Scratch: documentation site, last accessed in June 2011. http://info.scratch.mit.edu/About_Scratch.

[7] Scratch: imagine, program, share, last accessed in June 2011. http://scratch.mit.edu/.

[8] Stagecast, last accessed in June 2011. http://www.stagecast.com.

[9] Visual languages, last accessed in June 2011. http://www.hypernews.org/HyperNews/get/computing/visual.html?inline=-1.

[10] Blender, last accessed in March 2011. http://www.blender.org.

[11] Development for biginners – tier one: Windows development. MSDN, last accessed in September 2011. http://msdn.microsoft.com/en-us/beginner/bb308891.aspx.

[12] Eclipse - The Eclipse Foundation open source community website, last accessed in September 2011. `http://www.eclipse.org/`.

[13] MyGUI – fast, flexible and simple GUI, last accessed in September 2011. `http://mygui.info/`.

[14] NetBeans IDE, last accessed in September 2011. `http://netbeans.org/features/index.html`.

[15] NetBeans IDE - Swing GUI Builder (Matisse) features, last accessed in September 2011. `http://netbeans.org/features/java/swing.html`.

[16] Ahern, T. C. The effectiveness of visual programming for model building in middle school. In *Proceedings of the 38th ASEE/IEEE Conference on Frontiers in Education*, pages S3D–8–S3D–13. IEEE, 2008.

[17] Ambler, A. L. and Hsia, Y.-T. Generalizing selection in by-demonstration programming. *Journal of Visual Languages & Computing*, 4(3):283–300, 1993.

[18] Baird, G., Simonoff, E., Pickles, A., Chandler, S., Loucas, T., Meldrum, D., and Charman, T. Prevalence of disorders of the autism spectrum in a population cohort of children in South Thames: the special needs and autism project (SNAP). *The Lancet*, 368(9531):210–215, 2006.

[19] Begel, A. LogoBlocks: a graphical programming language for interacting with the world, 1996. `http://llk.media.mit.edu/papers/archive/begel-aup.pdf`.

[20] Begel, A. and Klopfer, E. StarLogo TNG: an introduction to game development. *Journal of E-Learning*, 2005.

[21] Burnett, M., Atwood, J., Djang, R. W., Reichwein, J., Gottfried, H., and Yang, S. Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.

[22] Burnett, M. M. Visual programming. In J. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*, pages 275–283. John Wiley & Sons, Inc., 1999.

[23] Burnett, M. M. and Baker, M. J. A classification system for visual programming languages. *Journal of Visual Languages & Computing*, 5(3):287–300, 1994.

[24] Burnett, M. M. and Gottfried, H. J. Graphical definitions: expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5:1–33, 1998.

[25] Catarci, T., Costabile, M. F., Stefano, L., and Carlo, B. Visual query systems for databases: a survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[26] Cheng, A. C. *A graphical programming interface for a children's constructionist learning environment.* Master's thesis, Massachusetts Institute of Technology, 1998. `http://mit.dspace.org/bitstream/handle/1721.1/17468/41869196.pdf?sequence=1`.

[27] Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., and Zugal, S. Declarative versus imperative process modeling languages: the issue of understandability. In W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, C. Szyperski, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 353–366. Springer Berlin Heidelberg, 2009.

[28] Glinert, E. P. Towards second generation interactive, graphical programming environments. In *Proceedings of the Second IEEE Computer Society Workshop on Visual Languages*, pages 61–70. June 1986.

[29] Glinert, E. P. Out of Flatland: towards 3D visual programming. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*, pages 292–299. IEEE Computer Society Press, 1987.

[30] Goldsmith, T. R. and LeBlanc, L. A. Use of technology in interventions for children with autism. *Journal of Early and Intensive Behavior Intervention*, 1(2):166–178, 2004.

[31] Green, T. R. G. and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[32] Hils, D. D. Visual languages and computing survey: data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[33] Howland, K., Good, J., and Nicholson, K. Language-based support for computational thinking. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 147–150. IEEE Computer Society, 2009.

[34] Hudson, S. E. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics*, 13:209–239, July 1994.

[35] Ierusalimschy, R. Programming in Lua: 2.3, last accessed in July 2011. `http://www.lua.org/pil/2.3.html`.

[36] Ierusalimschy, R. Programming in Lua: 2.3, last accessed in July 2011. `http://www.lua.org/pil/2.html`.

[37] Ierusalimschy, R. Programming in Lua, last accessed in March 2011. `http://www.lua.org/pil/`.

[38] Johnston, W. M., Hanna, J. R. P., and Millar, R. J. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, March 2004.

[39] Kay, A. Computer software. *Scientific American*, pages 53–59, September 1984.

[40] Kindborg, M. and McGee, K. Visual programming with analogical representations: inspirations from a semiotic analysis of comics. *Journal of Visual Languages & Computing*, 18(2):99–125, 2007.

[41] Koegel, J. F. and Heines, J. M. Improving visual programming languages for multimedia authoring. In *Proceedings of the World Conference on Educational Multimedia and Hypermedia*, pages 286–293. Association for the Advancement of Computing in Education, 1993.

[42] Kogan, M. D., Blumberg, S. J., Schieve, L. A., Boyle, C. A., Perrin, J. M., Ghandour, R. M., Singh, G. K., Strickland, B. B., Trevathan, E., and Van Dyck, P. C. Prevalence of parent-reported diagnosis of autism spectrum disorder among children in the US, 2007. *Pediatrics*, 124(5):1395–1403, 2009.

[43] Kopache, M. E. and Glinert, E. P. C2: a mixed textual/graphical environment for C. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, pages 231–238. October 1988.

[44] Landauer, J. and Hirakawa, M. From programming by demonstration to programming by WYSIWYG. *Journal of Visual Languages & Computing*, 8(5-6):621–640, 1997.

[45] Lane, D. M., Napier, H. A., Peres, S. C., and Sándor, A. Hidden costs of graphical user interfaces: failure to make the transition from menus and icon toolbars to keyboard shortcuts. *International Journal of Human–Computer Interaction*, 18(2):133–144, 2005.

[46] McCaffrey, C. *StarLogo TNG: the convergence of graphical programming and text processing.* Master's thesis, Massachusetts Institute of Technology, 2006. `http://dspace.mit.edu/bitstream/handle/1721.1/36904/80770344.pdf?sequence=1`.

[47] Meijer, E. and Drayton, P. Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages. In *Proccedings of the OOPSLA'04 Workshop on Revival of Dynamic Languages*. 2004.

[48] Moore, D., McGrath, P., and Thorpe, J. Computer-aided learning for people with autism – a framework for research and development. *Innovations in Education and Teaching International*, 37(3):218–228, 2000.

[49] Myers, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.

[50] Newschaffer, C. J., Croen, L. A., Daniels, J., Giarelli, E., Grether, J. K., Levy, S. E., Mandell, D. S., Miller, L. A., Pinto-Martin, J., Reaven, J., Reynolds,

A. M., Rice, C. E., Schendel, D., and Windham, G. C. The epidemiology of autism spectrum disorders. *Annual Review of Public Health*, 28:235–258, 2007.

[51] Parsons, S. and Mitchell, P. The potential of virtual reality in social skills training for people with autistic spectrum disorders. *Journal of Intellectual Disability Research*, 46:430–443, 2002.

[52] Parsons, S., Mitchell, P., and Leonard, A. The use and understanding of virtual environments by adolescents with autistic spectrum disorders. *Journal of Autism and Developmental Disorders*, 34(4):449–466, 2004.

[53] Rader, C., Brand, C., and Lewis, C. Degrees of comprehension: children's understanding of a visual programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 351–358. ACM, 1997.

[54] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[55] Roque, R. V. *OpenBlocks: an extendable framework for graphical block programming systems*. Master's thesis, Massachusetts Institute of Technology, 2007. `http://mit.dspace.org/bitstream/handle/1721.1/41550/220927290.pdf?sequence=1`.

[56] Rothermel, G., Li, L., and Burnett, M. Testing strategies for form-based visual programs. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 96–107. IEEE, November 1997.

[57] Schmidt, C. and Schmidt, M. Three-dimensional virtual learning environments for mediating social skills acquisition among individuals with autism spectrum disorders. In *Proceedings of the 7th International Conference on Interaction Design and Children*, pages 85–88. ACM, 2008.

[58] Seals, C., Rosson, M., Carroll, J., Lewis, T., and Colson, L. Fun learning Stagecast Creator: an exercise in minimalism and collaboration. In *Proceedings*

of the 2002 IEEE Symposia on Human Centric Computing Languages and Environments, pages 177–186. 2002.

[59] Sierra Romero, N. and Chapa Vergara, S. VITPE – visual tool for the program edition. In *Proceedings of the 1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*, volume 4, pages 3190–3194. October 1995.

[60] Streeting, S. Object oriented graphics engine, last accessed in March 2011. `http://www.ogre3d.org`.

[61] Tanimoto, S. L. VIVA: a visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[62] Van Zijl, L. and Chamberlain, M. A generic development platform for ASD therapy tools. In *Proceedings of the International Conference on Computer Supported Education*, volume 1, pages 82–89. April 2010.

[63] Wing, J. Viewpoint-computational thinking. *Communications of the ACM*, 49(2):33–35, 2006.

[64] Zloof, M. M. Query-by-example: a database language. *IBM Systems Journal*, 16(4):324–343, 1977.