From visual scripting to Lua

Mwawi F. Msiska
Dept. of Mathematical Sciences
Chancellor College
University of Malawi
Malawi
mfmsiska@gmail.com

Lynette van Zijl
Computer Science
Stellenbosch University
South Africa
lvzijl@sun.ac.za

ABSTRACT

We report on the ASD-Assist project at Stellenbosch University, and more specifically on our experience in developing a visual programming environment and language, which can easily be compiled to underlying Lua scripts for use on any compatible gaming engine. We show that such an environment and compiler can be successfully implemented, which results in a simple and efficient mechanism to generate front-ends for specialized 3D therapy tools.

Categories and Subject Descriptors

D.1.7 [**Visual Programming**]; I.2.2.3 [**Artificial Intelligence**]: Automatic programming—*Program transformation*

General Terms

Visual programming language

Keywords

Visual programming language, compiler, 3D therapy tool

1. INTRODUCTION

The incidence of autism spectrum disorders (ASD) in children is currently estimated to be 1 in every 500 children [5]. Children with ASDs are characterised by severe challenges in communication and social skills. Therapies are hence based on assisting with language, communication, and social skills. This implies that any therapies are language and culture specific (for example, teenage slang in London is different from teenage slang in Cape Town). It has been shown that 3D virtual reality tools are particularly beneficial for children with ASDs [13]. However, due to the cultural and language specifity, existing tools are not necessarily useful in the South African context. The ASD-Assist project at Stellenbosch University aims to assist professionals in the ASD field to set up and develop their own 3D therapy tools. To this end, we developed amongst others a streamlined 3D gaming engine (called Myoushu) that can work on older computers [14].

Therapy tools and educational games [1] can be developed on the Myoushu engine using the Lua scripting language [6]. Development of therapy tools in Lua is clearly suitable only for experienced programmers. The reader may note that there are two main components to such 3D therapy tools – on the one hand, the graphics, and on the other the logic associated with the therapy tool. The graphics require the creation or importation of backgrounds scenes, characters, and so on, and requires little programming experience. The logic, on the other hand, requires algorithmic thinking. We wished to present an easy to use interface which professionals in the ASD field (whom we assume to be non-programmers) can use to develop their own therapy tools, and specifically the logic underlying the therapy tool. Hence, we decided to implement a user-friendly visual programming interface, which can be automatically compiled into Lua scripts for execution on the Myoushu engine.

The idea of visual programming languages and visual programming environments is not new (see [3] for an overview). Such systems are typically quite general (for example, Alice [4]), and our aim was not to produce yet another visual language. Instead, we were interested in re-using existing well-defined and proven technology, and adapt that to produce Lua output for the Myoushu engine.

This article reports our findings on adaptable visual programming languages and environments in Section 2, and our design and implementation for the visual script to Lua compiler in Section 3. In Section 4 we discuss the implementation of an educational game in our system and the corresponding results. We conclude in Section 5.

2. BACKGROUND AND RELATED WORK

A visual programming language (VPL) is a programming language where the programmer constructs a program by graphically arranging graphical and textual objects – note that the spatial arrangement of these objects typically have a semantic meaning. A visual programming environment (VPE) is an interface which allows the programmer to manipulate the graphical and textual objects. VPEs have been shown to be an ideal platform for non-expert programmers to produce algorithmic solutions [1]. Hence, our choice of a visual programming approach is sensible. However, amongst VPEs and their associated VPLs, there is a wide variety of approaches (see [11] for a classification). This includes

[1]We use the terminology 'therapy tool' and 'game' interchangeably.

SAICSIT'12 October 01 - 03 2012, Pretoria, South Africa

dataflow languages, form-based languages, iconic languages, programming-by-example, and building block languages. For a full comparison of these different approaches, the reader is refered to [10].

Our aim was to develop a visual programming environment suitable to defining the logic of a 3D therapy tool, but which can easily be translated into Lua. As the building block approach can be considered as a visualisation of an underlying textual language, its translation to Lua would be considerably easier than any of the other approaches. In addition, other VPEs based on the building block approach have been shown to be easy to learn and effective to get novice programmers to develop their own programs [1]. Some of the best-known amongst these are Scratch [8], Alice [4] and StarLogo [2]. The reader may note that simply using one of these existing systems was impractical in our case. The main reasons included that these are general programming systems (not necessarily aimed at game or therapy tool development), the environment was prohibitively resource intensive and/or the translation into Lua was not practical.

VPEs based on building blocks such as Scratch, Alice and StarLogo can be implemented with the OpenBlocks library [12] to speed up the development of the VPE. A complete set of program building block types can be created by writing specifications in a single XML file. In the next section we describe our design and implementation of a visual script editor (VSE), based on the building blocks approach, aimed at novice programmers, and which compiles into executable Lua code.

3. DESIGN AND IMPLEMENTATION OF THE VSE

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

The development of a storyline, and creation of 3D objects, fall outside the scope of the VSE. However, knowledge of 3D objects, the specification of animations, and the state of the therapy tool all fall within the concept of the 'logic' of the therapy tool and hence within the domain of the VSE. We thus set out as requirements for the VSE:

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 point 1. above;
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 above. 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; which, coincidentally, is a by-product of using our VSE due to the nature of the visual programming representation technique that we have adopted.

Objectives 3 and 4 allow the author to discover the abstract contents of Myoushu 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.

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 [7]; 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.

We designed our VSE based on the OpenBlocks library [12], as explained in the previous section. The main characteristics here include the concept of a page, static and dynamic drawers where commands are stored, and a rapid minimization of the whole programming area for efficient navigation (see Figure 1).

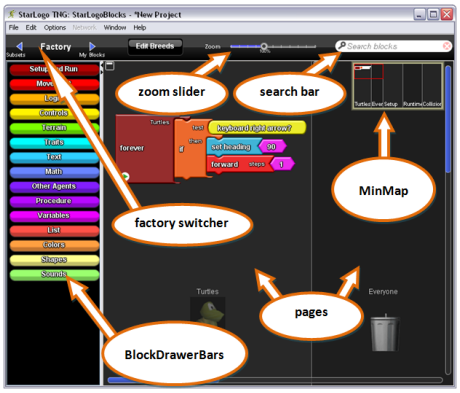

Figure 1: The OpenBlocks layout

Given the objectives as above, we designed separate modules for compiling the VSE blocks (compiler, verifier), graphical issues (scenes and collisions), GUI development, and interaction with the Myoushu engine (configuration). This high-level design in shown in Figure 2.

The scene object 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 GUI manager maintains information about GUIs that

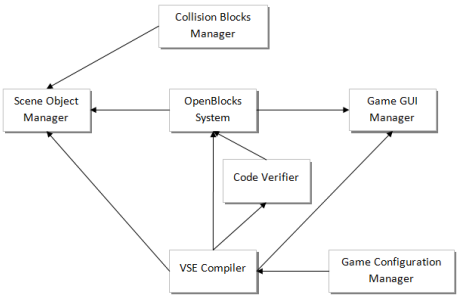

Figure 2: The VSE system overview

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 configuration manager sets up a complete configuration that can be exported to the Myoushu engine for execution.

In this article we are interested mainly in the issues surrounding the compilation of visual blocks into executable Lua code – for more technical details on the other parts of the VSE, the reader is referred to [10].

3.1 Error-free compilation process

The compilation 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 (see Figure 3).

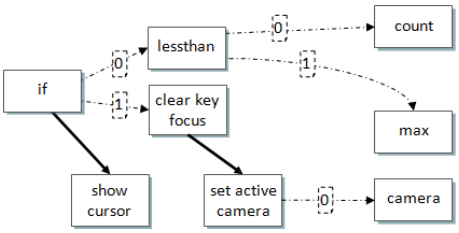

Figure 3: The abstract syntax tree

Although the building-block representation technique effectively prevents syntax errors in a visual program, there is still potential for compile-time errors. Before we discuss the handling of such errors, we first consider the case of compiling a visual program that has no errors.

Algorithm 4 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 Algorithm 4 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

```
while (b != NULL) {
   fixedCodeList = getFixedCodeList(b);
   socketList = getSockets(b);
   if (isEmpty(socketList))
      print next(fixedCodeList)
   else
     forall socket in socketList {
        print (next(fixedCodeList));
        generateCode (getBlockAt(socket));
        print (next(fixedCodeList));
     }
   }
   b = getBlockAfter(b);
}
```

Figure 4: `generateCode(Block b)`

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 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.

A possible problem emanating from non-insistence on connectedness is that some sockets can be left without plugs when the code generator module is evoked. 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 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 1, in the generated Lua code.

Table 1: Default values for missing blocks

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

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 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 amongst 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.

Next, we consider errors caused by deletion of blocks representing variable, procedure and function definitions.

3.2 Error-handling in the compilation process

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 [6], our VSE is statically typed in order to enhance the simplicity of ensuring correctness in visual programs [9]. 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.

A highly desirable property of the VSE is its extension by adding function libraries, and we now consider how the compiler handles such a situation.

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 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.

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 fiow 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 the 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 develops further. We thus decided to place the Lua code fragments 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 4 extracts the Lua code fragments from the language specific property elements of XML nodes representing blocks representing calls to the game engine's 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.

4. RESULTS

We used our VSE to re-develop an existing game (called *Journey to the Moon*) which currently runs on the Myoushu engine, and which was originally developed in Lua. This redevelopment would be advantageous, as the original can be directly compared to our translated version, and shortcomings in our VSE would become clearer. Indeed, our main aim was to confirm that the VSE is capable of supporting the specification of logic in virtual environments developed for the Myoushu engine.

Journey to the Moon is an educational game with a target audience of grade 1 learners. It contains two scenes – the first shows a laboratory, where a scientist has to get answers from the user in order to let a spaceship take off. The second scene features the spaceship flying through space, dodging and destroying asteroids. The user has to answer educational questions to prevent the spaceship being hit by the asteroids. Figure 4 shows the first scene.

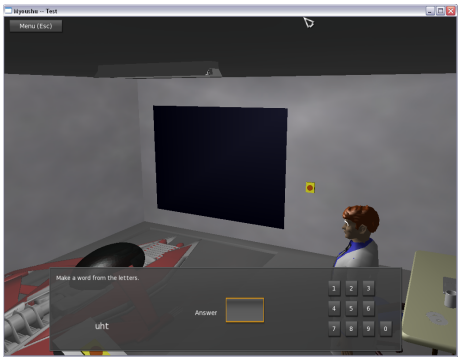

Figure 5: Screenshot from *Journey to the Moon*

The implementation of the game was successful, and our version corresponded correctly to the original implementation in Lua. However, some interesting problems were encountered, which led to improvements in the original design of the VSE.

Firstly, it was not possible to map the game engine's library functions to VSE blocks in a one-to-one fashion, lest one ends up with too many blocks representing library function calls, which would compromise learnability of the system. It also became clear that the addition of blocks representing library function calls can trigger the need for new data types, and consequently new variable types. The code generator was modified 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. Also noted was that GUI visual scripts were memory intensive. We thus implemented a modification 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.

A direct comparison between the original Lua game and the version compiled via the VPE is not necessarily meaningful (see [10] for more detail).

In conclusion, the reader may note that we did not conduct a user study. We deemed this not necessary at this stage, as the OpenBlocks framework was already adequately evaluated [1, 2]. However, a user case study will be conducted in the next stage of the project.

5. CONCLUSION

We reported on the development of a visual script editor which outputs Lua code, as part of the ASD-Assist project at Stellenbosch University. We showed that the code generator produces valid Lua code, and we conclude that this approach leads to a simple and efficient mechanism to generate the logical component of specialized 3D therapy tools.

As future work, we propose an extension to the VSE to allow for variable scoping (global, local and page local variables). Another more difficult issue that we intend to investigate, is a visual expression of hierarchy in order to design larger programs within the VSE.

6. REFERENCES

[1] T. C. Ahern. 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, Saratoga Springs, NY, 2008. IEEE.

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

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

[4] W. Dann and S. Cooper. Alice3: concrete to abstract. *Communications of the ACM*, 52(8):27–29, August 2009.

[5] E. Fombonne. Epidemiology of pervasive developmental disorders. *Pediatric Research*, 65:591–598, 2009.

[6] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2011.

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

[8] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *Transactions on Computing Education*, 10(4), November 2010.

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

[10] M. Msiska. A visual programming environment for authoring ASD therapy tools. Master's thesis, Stellenbosch University, 2011.

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

[12] R. V. Roque. OpenBlocks: an extendable framework for graphical block programming systems. Master's thesis, Massachusetts Institute of Technology, 2007.

[13] C. Schmidt and M. Schmidt. 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, Chicago, Illinois, 2008. ACM.

[14] L. van Zijl and M. Chamberlain. A generic development platform for ASD therapy tools. In *Proceedings of the Second International Conference on Computer Science Education*, pages 82–89, Valencia, Spain, April 2010.