Delphi: A Source-code Analysis and Manipulation System for Bricklayer

Victor Winter
Department of Computer Science
University of Nebraska-Omaha
Omaha, Nebraska
vwinter@unomaha.edu

Betty Love
Department of Mathematics
University of Nebraska-Omaha
Omaha, Nebraska
blove@unomaha.edu

Chris Harris
Department of Computer Science
University of Nebraska-Omaha
Omaha, Nebraska
charris@unomaha.edu

Abstract

Delphi is a source-code analysis and manipulation system being developed to analyze and transform Bricklayer programs. The information obtained from Delphi analysis can be used to generate problem-specific text in the form of a mini-lecture. This opens the door to the automated integration of such texts with commercial animation software and text-to-speech (TTS) tools. The result is a scalable infrastructure capable of providing formative feedback to students in the form of an animated cartoon whose information is personalized (e.g. male/female actors, use of slang and dialects) and problem-specific. This feedback can be provided in a timely fashion and can ease technical burdens on educators that teach coding across the K-12 spectrum.

1 Introduction

This paper debuts *Delphi*, a source-code analysis system we are developing for analyzing Bricklayer programs. Within the Delphi infrastructure, it is possible to perform a wide range of customized analysis. The information obtained from such analysis can be used to generate problem-specific text in the form of a mini-lecture. This opens the door to the integration of such texts with commercial animation software and text-to-speech tools. The result is a system which provides *formative feedback* to students in the form of an animated cartoon whose information is personalized. Formative feedback is defined as "information communicated to the learner that is intended to modify his or her thinking or behavior to improve learning" [2].

The rest of the paper is organized as follows: Section 2 gives an overview of related work. Sections 3 and 4 give brief overviews of the Bricklayer ecosystem and Delphi respectively. Section 5 gives an example showing how Delphi can be used to provide animated formative feedback for pixel art artifacts constructed in Bricklayer. Section 6 looks towards the future and describes the kinds of analysis that is possible using Delphi, and Section 7 concludes.

DOI reference number: 10.18293/SEKE2017-010

2 Related Work

A literature review paper written by Keuning et. al focuses on tools that provide automated feedback for programming exercises [1]. They report on 69 feedback generation tools. A finding of their review is that tools (1) generally do not “give feedback on fixing problems and taking a next step”, and (2) cannot easily be adapted to specific needs of teachers. Delphi can address both of these concerns.

3 Overview of the Bricklayer Ecosystem

Bricklayer [5] is an online educational ecosystem designed in accordance with a “low-threshold infinite-ceiling” philosophy. Its purpose is to teach coding to people of all ages and coding backgrounds. A significant portion of the Bricklayer ecosystem has been developed specifically to help novices, especially primary school children, learn how to code. When executed, Bricklayer programs can produce LEGO® artifacts, Minecraft artifacts, as well as artifacts suitable for 3D printing. Bricklayer resides in a domain in which there is a strong connection between math, art, and computer science. The Bricklayer ecosystem is freely-available and can be found at:

`http://bricklayer.org`

Bricklayer programs are written in the functional programming language SML. Graphical capabilities are provided by the Bricklayer library. The output of Bricklayer programs are files which are input to third-party tools which include: LEGO® Digital Designer (LDD), LDraw, Minecraft, and STL viewers such as 3D Builder.

Bricklayer programs can also be developed using a block-based editor called *bricklayer-lite*. Bricklayer-lite is built using Google Blockly. A noteworthy capability of bricklayer-lite is that, in addition to producing a Bricklayer (graphical) artifact, the execution of a bricklayer-lite program will produce a well-formed and well-formatted Bricklayer program text which can be executed outside of the bricklayer-lite framework.

4 Delphi

Delphi is a source-code analysis tool that is being developed for Bricklayer. From an implementation standpoint, Delphi represents a non-trivial extension of the TL system (a general-purpose program transformation system) specialized to the domain of Bricklayer. The TL system [6][3] is a powerful and mature meta-programming system which has been used to develop tools for a variety of domains including: (1) a source-code analysis system for Java, (2) a compiler for an architecture independent microprogramming language, (3) compiler optimizations, and (4) automated test generators.

One of the primary design goals of Delphi is to provide a tool facilitating specification of custom analysis rules (e.g., domain specific or even problem specific analysis rules).

The basic capabilities of Delphi include dependency analysis as well as standard syntax-driven metrics such as source lines of code (SLOC) as well as metrics relating to the use of various programming language constructs (e.g., conditional expressions and curried function declarations).

5 An Example of Animated Formative Feedback

A very engaging and artistic activity in Bricklayer involves the creation of pixel art – an example of which is shown in Figure 1. This activity centers on the translation of pixel art images into Bricklayer code. Pixel art projects can be individual or group oriented and the choices of what pixel art image to code are enormous. Pixel art images found on the web range from very simple to extremely complex.

Given an assignment to “code up” a pixel art image, a student has a wide range of options. This enables the selection of images that have special interest or meaning to the coder. The selection process is also influenced by an assessment of the complexity of the image as well as a self assessment of the ability of the individual.

It is not uncommon for a student to employ a *greedy algorithm* when coding a pixel art artifact.

DOI reference number: 10.18293/SEKE2017-010

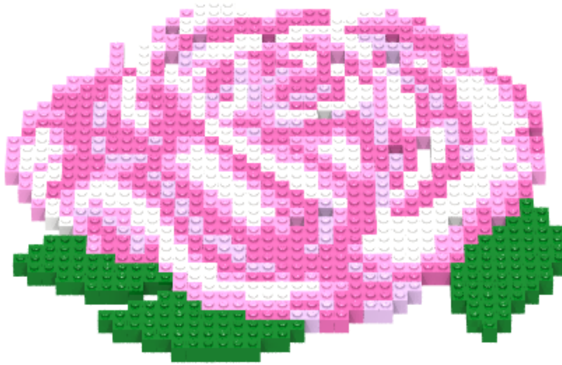

Figure 1: An example of a pixel art artifact created by an elementary school student.

When using such an algorithm one strives to place as many large bricks as possible and then places smaller bricks and so on. This approach can minimize the number of function calls in the source code. However, our experiences in working with student coding pixel art have led us to believe that the complexity of the boundary between the portion of the pixel art image that has been completed and the portion that remains to be completed becomes intellectually unmanageable when using a greedy algorithm. Though our results are anecdotal in nature, we have found that, due to its high cognitive load, a greedy algorithm for coding pixel art is fairly unreliable and, in the end, extremely labor intensive.

5.1 A Row-Major Algorithm

The most effective way to code pixel art is "one row/-column at a time". You code a row, run the code, validate that the row you coded is correct, and then move on and code the next row. We refer to such a construction algorithm as a *row-major algorithm.*

Bricklayer currently has five coding levels. Each level has greater expressive powers than the level that precedes it. Coding levels 1 and 2 only provide a standard set of brick shapes and colors for coding in the xz-plane. In particular, when subjected to the constraints imposed by a row-major algorithm, only 2×1 and 1×1 bricks are available for use. In levels 1 through 4, such bricks are placed using *put* function calls.

In order to be *minimal*, a program (i.e., code) implementing a row-major algorithm must satisfy a number of properties. First, the sequence of coordinates as which bricks are placed must satisfy a row-major order. Let (x_i, z_i) and (x_j, z_j) denote the coordinates of two put function calls, where a brick is placed at (x_i, z_i) before (x_j, z_j). To satisfy row-major order, $z_i < z_j \vee (z_i = z_j \wedge x_i < x_j)$.

A minimal program may not contain any brick *collisions.* A collision occurs when two or more put function calls place a brick in the same cell. In this case, Bricklayer's default behavior is to overwrite the contents of the cell according to the most recent put function call (i.e., the last brick you place is what you will see).

A minimal program should not be *locally compressible.* For example, two put function calls that place 1×1 blue bricks adjacent to one another should be replaced by a single put function call that places a 2×1 blue brick.

5.2 About Delphi Analysis

Through the use of 22 rules, Delphi is able to transform any input program p_{in} that creates a 2-dimensional artifact into a corresponding output program p_{out} such that (1) p_{out} constructs the 2-dimensional artifact according to a row-major algorithm, and (2) p_{out} is minimal according to the definition given above. During this program transformation, Delphi also records, in a database, which transformation rules are used as well as how they are used. The information stored in this database is then used to generate a natural language report describing the modifications that need to be performed in order to make the p_{in} compliant with a row-major algorithm. Figure 2 describes two of the rules used by Delphi.

Delphi is able to analyze and transform the contents of file hierarchies (e.g., hundreds of Bricklayer programs) at the touch of a single button. For each program that is processed a resulting text can be pro-

DOI reference number: 10.18293/SEKE2017-010

Condition	**Action**	**Rationale**
1. Two put function calls place bricks of the same color at adjacent coordinates (e.g., (0,0) and (1,0)). 2. The first put function call places a 2×1 brick and the second put function call places a 1×1 brick.	The second put function call can be removed.	The 1×1 brick placed by the second function call will overwrite the brick placed by the first put function call. However, since both put functions place the same color brick no visible change will occur as a result of the second put function call. Therefore, the second put function call can be removed without changing the appearance of the artifact.
1. In a sequence consisting of three put function calls, the first two put function calls place bricks of different colors at the same coordinate. Let (x, z) denote this coordinate. 2. The first put function call places a 2×1 brick, and the second put function call places a 1×1 brick. 3. The third put function call places a 2×1 brick at the coordinate $(x + 2, z)$.	Modify the first put function call so that it places a 1x1 brick at the coordinate $(x + 1, z)$, and then lexically commute the first and second put function calls (i.e., position the modified (previously) first function call after the second function call).	The first put function call places a 2×1 brick which will occupy cells (x, z) and $(x + 1, z)$. In contrast, second put function call places a 1×1 brick which will only occupy the cell (x, z). Since the bricks placed by the first two put function calls have different colors the bricks at (x, z) and $(x + 1, z)$ will have different colors. Furthermore, the third put function call places a 2×1 brick so its shape cannot be increased. Thus, the first put function call must be modified so that is places a 1×1 brick at $(x + 1, z)$. The first and second put function calls must then be commuted in order for the coordinates associated with put function calls to conform to a row-major order.

Figure 2: Descriptions of some rules used by Delphi.

duced explaining, at a level deemed appropriate for the author of the code, the changes that would need to be made in order to place the Bricklayer program under consideration into row-major form. This text can then be given as input to a text-to-speech (TTS) translator and the result can be given as input to an animation program.

Figure 3 shows a screenshot of a video created in CrazyTalk in the manner just described: The dialog was produced automatically using Delphi. This was then copy-and-pasted into the text-to-speech (TTS) translator for CrazyTalk. The animations were hand made, but used canned motions provided by CrazyTalk. The images of the Bricklayer program were constructed by hand as was the rendering of the artifact. Though the creation of this demonstration involved manual steps, in the future, animation tools, such as CrazyTalk, could be parameterized on such inputs. The result would be a system where the production of such custom videos would be completely automatic.

DOI reference number: 10.18293/SEKE2017-010

Watch the video at: `https://www.youtube.com/watch?v=o6aoW5rgOhU&feature=youtu.be`

A sample of a dialog generated by Delphi

The artifact created by the Bricklayer program in the file called panda.bl is not constructed according to a row-major algorithm. Not to worry, I will tell you how you can change your program so that it conforms to a row-major algorithm. The 1-by-1 brick created by the function call on line number 78 is overwritten by the function call that follows it. Therefore, it can be removed without changing the appearance of your artifact. Your program contains a pair of function calls that put a 2-by-1 brick and a 1-by-1 brick, having different colors, at coordinates whose x-value differ by 1. The function-call at line number 85 can be changed so that it puts a 1-by-1 brick instead of a 2-by-1 brick.

Figure 3: Animated problem-specific feedback provided by Delphi.

6 Future Work

One of the primary design goals of Bricklayer was to create a coding domain that is "example rich and problem dense" [4]. In such a domain, the creation of smooth learning curves become possible which can be supported through numerous examples and similar problems/exercises.

The learning curve in Bricklayer is, in part, realized through a sequence of concepts. In many cases the transition from one concept to the next can be understood in equational terms. This is by design and is facilitated by the underlying semantics of functional programming languages like SML (the language in which Bricklayer programs are written) in which referential transparency play a central role. Consequently, this also provides a domain in which a program transformation system, like Delphi, can play

DOI reference number: 10.18293/SEKE2017-010

an important role.

In traditional textbooks, new concepts are introduced using a *static* set of examples. Examples that were created when the textbook was published. With Delphi, it is possible to create examples *dynamically*. For example, a student program containing no user-defined functions can be transformed automatically by Delphi into an equivalent program containing user-defined function declarations and calls. This results in a "living textbook" in which learning examples are (transformationally) derived from individual programs created by students. Furthermore, it is also possible for such transformations to generate text explaining what was done. Such texts can then be fed into animation systems to produce personalized learning material.

7 Conclusion

We believe that educational technology has just scratched the surface regarding what is possible. Technologies lying within our near-term reach can be developed to provide significant technical support for educators interested in teaching coding. Technologies, such as Delphi, when added to the support services provided by online help desks can address teacher needs in effective and cost-efficient ways. Such a model also geographically decouples technical expertise from teaching expertise. One help desk can serve the educational needs of schools around the world. We believe that such approaches will be needed to solve the problem of teaching coding across the K-12 spectrum.

References

[1] H. Keuning, J. Jeuring, and B. Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 41–46, New York, NY, USA, 2016. ACM.

[2] V. J. Shute. Focus on Formative Feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[3] The TL System, 2010. `http://faculty.ist.unomaha.edu/winter/ShiftLab/TL_web/TL_index.html`.

[4] V. Winter. Bricklayer: An Authentic Introduction to the Functional Programming Language SML. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2014.

[5] V. Winter. The world needs more computer science! what to do? In D. Conway, S. A. Hillen, M. Landis, M. T. Schlegelmilch, and P. Wolcott, editors, *Digital Media, Tools, and Approaches in Teaching and Their Added Value*, pages 119–141. Waxmann Verlag GmbH, Germany, 2015.

[6] V. L. Winter. Stack-based Strategic Control. In *Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming*, June 2007.

DOI reference number: 10.18293/SEKE2017-010