Game Painter: A Graphical Environment for Game Design and Development

James Morrow
Department of Computer Science
UNC Asheville
jmorrow1@unca.edu

ABSTRACT

Game Painter is a graphical environment for the development of two dimensional video games, which allows users to create games by creating rules, sprites, and levels. The name Game Painter comes from the way in which users interact with the software, which is by using a mouse to color in sections of a canvas. The project's intention is to provide a software development environment that is not based in writing code. Although some graphical environments for video game development do exist, many of them involve dragging icons which represent blocks of code. Game Painter is unique in that there is no code involved. The environment is accessible to users with no programming experience; and for those who have programming experience, Game Painter provides a direct and interactive approach to game development.

1. INTRODUCTION

Improvisation is a powerful creative tool in any discipline. A musician, for example, can make music improvisationally, by simply playing her instrument. In making music this way, there is a tight feedback loop between the musician and her instrument. Software development, on the other hand, often lacks this spirit of improvisation. In typical software development, a designer first conceives an idea, then a programmer implements the idea. Furthermore, the tools programmers use can impede creative flow. Some problems lend themselves to visual thinking and yet software is developed almost entirely through code. Game Painter is an attempt to address this lacking in software development.

2. BACKGROUND

2.1 Related Software

Some recent graphical programming environments involve arranging iconic graphics that represent blocks of code. One such example is RAPTOR. "RAPTOR allows students to create algorithms by

combining basic graphical symbols [8]." Another example is ScratchJr. "With Scratch, users program interactive art, stories, animations, simulations, and games by snapping together sequences of graphical blocks which represent instructions [7]."

The GUI of Game Painter, on the other hand, is completely code-free. Instead, users program game logic by drawing cells to two grids, one labeled "before" and the other labeled "after". If the game reaches a state where sprites neighbor a given cell in the fashion specified by the "before" grid, it will change that cell to the sprite specified by the "after" grid.

2.2 Cellular Automata

The way that game logic works in Game Painter is heavily inspired by cellular automata. Cellular automata are systems of cells in which a cell's state changes based on the states of its neighbors. Stephen Wolfram describes a family of cellular automata he calls elementary cellular automata [10]. For elementary cellular automata, cells can exist in one of two states, black or white. Only three neighbors can affect a given cell of an elementary cellular automaton. A rule in an elementary automaton is a statement about how a cell changes state based on the states of its neighbors. A neighborhood can naturally be encoded as a binary number. Black can coorespond to the digit 1 and white can coorespond to the digit 0. Since there are three neighbors, there are as many different arrangements of neighborhoods in the system of elementary cellular automata as there are 3-digit binary numbers, which is 2^3 [4]. This encoding of neighborhoods as binary numbers was a very useful starting point for developing the logic of Game Painter, which is described in section 3.2.

Users can use Game Painter to build automata as well as games. In Game Painter, an automaton is just a game that does the same thing regardless of input. Furthermore, a game in Game Painter is just a collection of automata mapped to different inputs. Only the automaton associated with the current input is active at any given time.

3. DESIGN OF GAME PAINTER

3.1 Overview

The painting metaphor is integrated into the design of the Graphical User Interface (GUI). Whether users are creating sprites, creating rules, or creating levels, the interaction of painting to a canvas is

fundamental. In the case of sprite creation, users paint colors to a canvas. In the case of rule creation and level creation, users paint sprites to a canvas. In all cases, the canvas is a grid of square cells. Users select paints from a palette, and they save their finished paintings to a reserved section of the screen called the gallery.

3.2 Implementation

The software is written in Java and uses Processing 2, a graphics library. All graphics are drawn to Processing's PApplet class, a subclass of the Applet class from Java's Abstract Window Toolkit (AWT). Most of Game Painter's architecture can be captured by two main categories: logic and graphics. The logic of game rules is discussed in the next section.

3.3 Incorporation of Cellular Automata

Game Painter is able to evaluate user-defined rules through a generalization of the encoding scheme described in section 2.2. In elementary cellular automata, neighborhood arrangements can be encoded as 3-digit binary numbers. This is due to the fact that elementary cellular automata involve 3 neighbors, which can exist in one of 2 states. Given these parameters there are only 256 elementary cellular automata [10]. If we remove the limit on the number of neighbors and the number of states, then there is an infinite number of possible automata. This is the case in Game Painter. Automata built in Game Painter can have an arbitrary number of states. Furthermore, neighborhoods can be of arbitrary shape and size. The number of states determines what encoding scheme is used. Generally speaking, if there are n states, then neighborhoods are encoded as base-n numbers.

When evaluating a level, Game Painter evaluates each cell in the level. For each cell, Game Painter converts the diagrammatic representation of the cell's neighborhood into a number. It then uses that number as an index to an array called ruleset. The value it gets back is a number cooresponding to a state. The cell currently being evaluated changes to that state. This straight-forward mapping between diagrams and numbers allows rule look-up times to be fast. However, there is a trade-off in space. As neighborhood sizes get bigger, the memory required to store rulesets gets exponentially larger.

3.3 Graphics Architecture

The graphics category is comprised mostly of three classes or class hierarchies: screens, grids, and cells. Screens are classes which implement the sprite creation, rule creation, and level creation screens. Screens draw graphics and handle input events received from Processing. The Grid class is the way in which canvases are implemented. Grids contain cells, which can be pixels, sprites, or any other subtype of the Cell class.

As discussed in section 3.1, the painting metaphor is used throughout the sprite creation, rule creation, and level creation screens. The architectural coorelate to the many-sidedness of the painting metaphor is the many-sidedness of the abstract types in Game Painter. Just as canvases are used as a UI element in the sprite creation, rule creation, and level creation screens, Grid is the type that implements each of these canvases. Just a paint can be either a pixel or a sprite in the UI, Cell is the type that implements both pixels and sprites.

3.4 User Feedback

The software was given to people with a variety of programming experience, on which qualitative testing was done. Some were experienced programmers and some had no programming experience. The goal is to make the software accessible to both programmers and non-programmers. User testers unanimously found the acts of defining sprites and levels intuitive. Some user testers, both programmers and non-programmers, found rule creation unintuitive, while others found it straight-forward. More work will be done in the future on refining the rule creation screen in order to align it with user expectation.

4. FUTURE WORK

4.1 Design

4.1.1 *Make Program Behavior Visible*

User testers found the connection between rule diagrams and the program behavior that results from them difficult to parse. The intended solution to this problem is to add visual feedback that highlights the connection between rules diagrams and program behavior. When the game evaluates a rule, it will highlight the cooresponding rule diagram.

4.1.2 *Support Creation by Abstraction*

Users of Game Painter are forced to make do with the tools given to them. In contrast, programmers can typically augment their programming languages in order to make it more suited for the problems on which they're working. Programmers can do this by creating abstractions, conceptual tools for solving different kinds of problems. The UI designer Bret Victor calls this "creation by abstraction" [2]. Game Painter does not yet give users ways in which to create by abstracting. One area of future work then is to create ways for users to create new abstractions. One idea is to add symbols that users can use as references to rulesets. Symbols would be displayed in the UI as icons. Another idea is to add an algebra of rulesets whereby rulesets can be operated on each other to produce to other rulesets.

It will be interesting to see how work in this area will affect the design of Game Painter. At the moment, the software emphasizes a graphical approach to game development. With the introduction of symbols and algebra, Game Painter will become a hybrid of a graphical approach and a symbolic approach. The user experience will likely change as well, as both visual thinking and symbolic thinking will be demanded of them. The basic approach to UI will remain the same. Users will continue to interact with the software by manipulating a GUI rather than by typing.

5. CONCLUSION

Video games are visual things, but software development is typically non-visual. This makes the developer relate to software more indirectly. Game Painter is an attempt to make the software development process more fluid and direct in order to address this problem.

In its current state, Game Painter is a completely graphical approach to game development. In the future, Game Painter will become more like a hybrid of a graphical approach and a symbolic approach to game development. The addition of symbolic features will allow users to create useful abstractions that allow them to better express their ideas. This way, Game Painter can combine what's good about a graphical approach to software development—it allows the developer to create software in a way that feels direct, fluid, and intuitive—with what's good about a symbolic approach

to software development—it allows the developer to create new abstractions that are geared toward the problem at hand.

REFERENCES

[1] Alfred L. McKinney. 2003. A recent radical graphical approach to programming.*J. Comput. Sci. Coll.*18, 6 (June 2003), 28-35.

[2] Bret Victor. (2012, September 1). Learnable Programming. Retrieved September 2, 2015.

[3] Bret Victor. (2011, October 1). Up and Down the Ladder of Abstraction. Retrieved September 2, 2015.

[4] Daniel Shiffman. "Cellular Automata."*The Nature of Code*. Ed. Shannon Fry. 1st ed. 2012. 323-354. Print.

[5] David G. Hannay. 1992. Hypercard automata simulation: finite-state, pushdown and Turing machines. *SIGCSE Bull.* 24, 2 (June 1992), 55-58. DOI=http://0-dx.doi.org.wncln.wncln.org/10.1145/130962.130971

[6] Joe Bergin. 2006. Karel universe drag & drop editor. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education* (ITICSE '06). ACM, New York, NY, USA, 307-307. DOI=http://0dx.doi.org.wncln.wncln.org/10.1145/1140124.1140212

[7] Louise P. Flannery, Brian Silverman, Elizabeth R. Kazakoff, Marina Umaschi Bers, Paula Bontá, and Mitchel Resnick. 2013. Designing ScratchJr: support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children* (IDC '13). ACM, New York, NY, USA, 1-10. DOI=10.1145/2485760.2485785 http://0-doi.acm.org.wncln.wncln.org/10.1145/2485760.2485785

[8] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. 2005. RAPTOR: a visual programming environment for teaching algorithmic problem solving. *SIGCSE Bull.* 37, 1 (February 2005), 176-180. DOI=10.1145/1047124.1047411 http://0-doi.acm.org.wncln.wncln.org/10.1145/1047124.1047411

[9] Mikael Kindborg and Kevin McGee. 2007. Visual programming with analogical representations: Inspirations from a semiotic analysis of comics. *J. Vis. Lang. Comput.* 18, 2 (April 2007), 99-125. DOI=http://dx.doi.org/10.1016/j.jvlc.2007.01.002

[10] Stephen Wolfram. 2002. *A New Kind of Science*. Champaign, IL: Wolfram Media.

[11] Wolfgang Slany. 2012. Catroid: a mobile visual programming system for children. In *Proceedings of the 11th International Conference on Interaction Design and Children* (IDC '12). ACM, New York, NY, USA, 300-303. DOI=10.1145/2307096.2307151 http://0-doi.acm.org.wncln.wncln.org/10.1145/2307096.230715