AIPaint: A Sketch-Based Behavior Tree Authoring Tool

David Becroft*, Jesse Bassett, Adrián Mejía, Charles Rich and Candace Sidner

Worcester Polytechnic Institute, Worcester, MA USA
http://www.cs.wpi.edu/~rich/aipaint

Abstract

Current behavior authoring tools force game designers to split their attention between the game context and the tool context. We have addressed this problem by developing a behavior authoring tool that merges these two contexts. This paper outlines the design and implementation of a game-independent behavior tree authoring tool, called AIPaint, that allows a designer to create and edit behavior trees via a natural sketching interface overlaid on the game world. We demonstrate the use of AIPaint to author computer-controlled characters in two simple games and report on an observational evaluation.

Introduction

In a typical game development team, the designer is responsible for deciding how the computer-controlled characters (AI agents) in the game should behave, while the programmer has the technical expertise necessary to implement these behaviors. The development process involves frequent iteration, wherein the designer specifies a behavior, the programmer implements it, the designer evaluates it and changes it, the programmer changes the implementation, and so on. Behavior authoring tools in general attempt to reduce this expensive and inefficient iteration by allowing designers to create and change AI agents' behaviors on their own.

For example, BrainFrame (Fu and Houlette 2002) and many similar tools provide a graphical interface to a game-independent agent architecture, such as a finite state machine. Once BrainFrame is integrated with a particular game, a designer can create behaviors by manipulating state diagrams without having to write any code.

Chris Hecker was the first to introduce the idea of a "Photoshop of AI"—an ideal behavior authoring tool for designers that could generate computationally efficient behavior representations while still offering a high degree of aesthetic expressiveness (Hecker 2008). This goal has motivated the development of behavior authoring tools such as BehaviorShop (Heckel, Youngblood and Hale 2009). BehaviorShop provides a graphical interface for creating agents based on subsumption architectures, which are claimed to be easier for non-programmers to understand than state machines.

* Becroft is currently an employee of Microsoft Corporation.

As user-friendly as BehaviorShop and similar tools may be, they still require designers to split their attention between two contexts: (1) the tool, in which designers edit behaviors, and (2) the game, in which designers observe and test behaviors. We believe this split limits the aesthetic expressiveness afforded by these authoring tools and that merging these two contexts will improve the designer's experience. This paper outlines the design and implementation of a game-independent behavior tree authoring tool, called AIPaint, that allows a designer to create and edit behavior trees via a natural sketching interface overlaid on the game world. The name AIPaint is meant to suggest that we see this work as a small step towards Hecker's vision.

The existing system most similar to AIPaint is the sketch-based authoring facility in Madden NFL 2011 for the iPad. This game allows the player to define football plays by drawing paths on the field using the touch screen; the simulated football players will follow these paths when the play begins. AIPaint fundamentally differs from this system in that AIPaint is a generic tool designed to be applied to many games, not just football simulations. Also, as we will see below, the expressive power of AIPaint sketches is much greater than that of Madden's play sketches, because AIPaint allows the designer to construct behaviors that involve making decisions and performing arbitrary actions, rather than simply following paths.

More broadly, AIPaint falls within the tradition of visual programming-by-demonstration approaches, such as (Smith, Cypher and Tesler 2010). Compared with these approaches, however, our goals are more modest. We are not trying to achieve a completely general visual programming language—we would be happy if the designer simply bothers the programmer less often.

Design Goals

A guiding metaphor in our design of AIPaint is the relationship between a director and the actors rehearsing a stage play. We view the game designer as the director who needs to evaluate and modify the behavior of the actors (the AI agents). The director can watch the actors perform their behaviors and tell them to stop when he sees something he doesn't like. Then he can issue some directions, and if an actor is confused by the "input," he may ask questions to clarify the director's intent. When the director is finished "editing," he can resume the action.

Finally, if the director wants to rehearse a specific scene, he can prearrange the actors on the stage to set up the desired situation. Note that a director modifies the behavior of his actors by communicating with them via an intelligent interface, not by reaching into their heads and manually adjusting their brains. Also note that the game designer and director are both expert authors—AIPaint is intended for game development teams fluent in their craft. This metaphor of the game designer as a director leads us to four key design goals for AIPaint.

Our first goal is for the game designer to communicate behavior specifications to AIPaint similarly to how he might communicate them to an AI programmer in the absence of a behavior authoring tool. One can easily imagine the designer in such a situation heading to a whiteboard or opening an image editor to sketch diagrams, perhaps overlaid on game screenshots. We see a similar approach when a football coach or commentator describes a play by marking up an image of the field with X's and arrows. AIPaint therefore provides a sketch-based interface in which the designer draws simple symbols (lines, circles, arrows, etc.) directly on the game world to describe behaviors. This kind of interface works best with a touch screen or tablet, but a mouse also suffices.

Our second design goal is for AIPaint to naturally support the specification of relationships between the *spatial* and *symbolic* aspects of an agent's behavior. Our approach to this goal has been to make both the spatial relationships among game objects and the symbolic information that determines how an agent interprets the game world visible at the same time. AIPaint is therefore well suited for behavior authoring in which objects and conditions relevant to agents' decisions are manifested visibly in the game world. For instance, Pac-Man lends itself to AIPaint authoring, whereas a game in which conversation is the key element would not.

Our third design goal is for AIPaint to operate on behaviors in a simple, widely used representation. We therefore chose behavior trees (Isla 2005) for our current implementation. A behavior tree has action (behavior) nodes at its leaves and decision (choice) nodes internally. When a behavior tree is evaluated in a given world state, the current action is chosen by traversing the tree from the root node, choosing a child at each decision node according to the decision's outcome in the world state. Our architecture leaves room for substituting other behavior representations, but this would not be a minor change.

Our final design goal is for AIPaint to be a game-independent tool. A game implementation that uses AIPaint for behavior authoring must provide code that conforms to several well-specified interfaces (see details below), so that AIPaint can call upon the game to perform tasks, such as screen-to-world coordinate transformation, and obtain world state information. The game-independent portion of AIPaint contains the sketching interface and the code that builds a behavior representation from the sketch input. Certain aspects of the game-independent portion of AIPaint are also available to the game programmer for extension, and it is likely that each game that uses AIPaint will extend the sketching language with some game-specific symbols.

Proof of Concept

In this section, we demonstrate the use of our implemented AIPaint tool for behavior authoring in two simple games. The first game is Pac-Man[1] (see Figure 1), in which we use AIPaint to recreate the behavior of the classic Pac-Man ghosts. The ghost behaviors in Pac-Man are a good example of using the connection between spatial and symbolic information to define behaviors. The second example is a computer soccer game (see Figure 2), which despite its simplicity is different enough from Pac-Man to illustrate the game-independence of AIPaint.

Pac-Man

Each ghost in Pac-Man—Inky, Blinky, Pinky, and Clyde—has a unique behavior. We will demonstrate the use of AIPaint by showing how a game designer can build Blinky's and Clyde's behaviors (see video at website). We also built Inky and Pinky's behaviors using AIPaint, but they require the use of more complex features, such as intermediate variables, which are detailed in our technical report (Bassett, Becroft and Mejía 2011).

Blinky

The designer begins by running the current development build of the game, which includes the AIPaint tool. The game handles mouse and keyboard input as in normal game play until the designer presses a special key that pauses the game and activates the AIPaint sketching

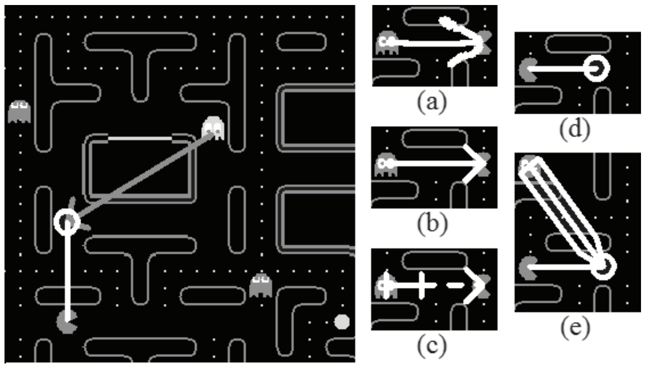

Figure 1: Pac-Man game running with the AIPaint tool. (a) A shape is drawn in natural sketch input. (b) The sketch input is recognized as an arrow shape and cleaned up. (c) A distance-conditional arrow with slider bar. (d) A temporary position variable relative to Pac-Man's position and orientation. (e) A "go twice as far" symbol connected to a temporary position variable.

[1] Pac-Man is a registered trademark of Namco Bandai Games, Inc.

interface. The designer then clicks on the red ghost (Blinky) to begin modifying its default random-walk behavior. In the original Pac-Man game, Blinky has a very simple behavior: he always moves towards Pac-Man.

Once an agent has been selected for behavior modification, the designer sketches on the game world to provide directions. As the designer sketches, shapes are automatically recognized and cleaned up. Each of these symbols/shapes has a specific meaning. For example, Figure 1(a) shows a Pac-Man screen on which an arrow has been drawn from Blinky to Pac-Man. Figure 1(b) shows this arrow after AIPaint's sketch recognizer has cleaned up the shape. The meaning of this arrow in the context of this game is that Blinky should move toward Pac-Man. This arrow is all that is needed to specify Blinky's desired behavior.

Clyde

The designer specifies Clyde's behavior by starting with a copy of Blinky. Clyde, the orange ghost, has a behavior that involves choosing between two actions. When Clyde is more than eight tiles away from Pac-Man, he targets Pac-Man's position. When he comes within eight tiles of Pac-Man, he instead targets a position near one corner of the maze. We thus need to compare the distance between Clyde and Pac-Man to a threshold value—eight tiles—and use the result to select one of two actions. This "distance-conditional arrow" shape takes the form of an arrow from Clyde to Pac-Man with a tick mark on the shaft of the arrow that the designer can drag back and forth to specify the threshold value (see Figure 1(c)). To specify Clyde's complete behavior, the designer draws this conditional arrow, followed by a simple arrow from Clyde to Pac-Man (the less-than-or-equal case) and a simple arrow from Clyde to the corner of the maze (the greater-than case).

Soccer

We also integrated AIPaint with a simple computer soccer game (see Figure 2) to demonstrate the game-independence of AIPaint. We then implemented a blocking behavior, in which an agent moves to the member of the opposite team who last touched the ball. The designer needs some help from the game programmer to set up this behavior. Specifically, AIPaint needs to know that when the designer draws an arrow to the soccer player who last touched the ball, he probably intends to make a statement about *whoever* last touched the ball, rather than that specific player. Once the programmer has entered this generalization rule, the designer can specify the desired blocking behavior by drawing an arrow to the member of the opposite team who last touched the ball, and AIPaint will make the appropriate generalization (see details below).

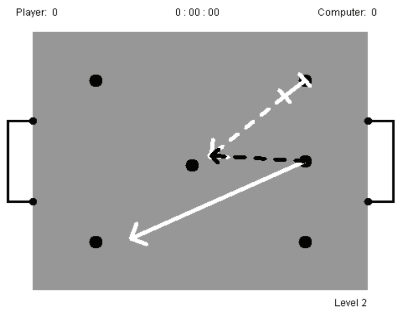

Figure 2: Sketch input for specifying behavior in simulated soccer game.

Architecture

AIPaint is a game-independent tool implemented in Java. Although the sketch interface described above is intended for unaided use by game designers, AIPaint does rely upon the game programmer to initially write some code to connect a specific game to AIPaint.

Interface to the Game World

AIPaint achieves game independence by communicating with the game world via Java interface classes. The game programmer needs to implement these classes and also write code to initialize certain AIPaint data structures.

Primarily, the game is responsible for providing AIPaint with world state information in the form of feature-value pairs, which is a very abstract and general data representation. The game-specific code will typically compute these feature-value pairs by examining properties of game objects, such as their type, position, orientation, etc. Behavior tree decision nodes make decisions during game play based on computations involving these features. For instance, a behavior tree node might look up the values of the world features that correspond to the positions of two game objects and compare the distance between them to a given threshold.

The Data Pipeline

AIPaint's data pipeline, shown in Figure 3, transforms the designer's sketch input into a form that can be used to create or modify a behavior tree. First the sketch input is recognized and cleaned up into *shapes*. These are then mapped to *statements* in a language we call Sketcho. Statements are then translated into *directions* in a language we call Directo. Finally, directions are used to effect modifications to a behavior tree. Clarification questions to the designer and game context information support the process by resolving ambiguities and suggesting generalizations.

$N: From Strokes to Shapes

The first step in the AIPaint data pipeline is to recognize the shapes sketched by the designer. As the designer drags his finger or mouse cursor across the screen, the input points are recorded and grouped into strokes. These strokes are passed to our Java implementation of the $N

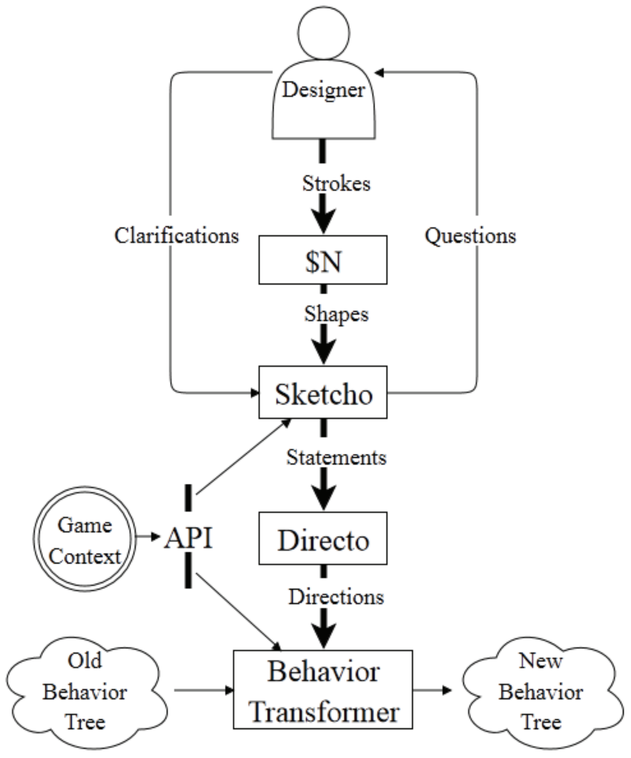

Figure 3: AIPaint data pipeline architecture.

multistroke recognition algorithm (Anthony and Wobbrock 2010), which returns a list of possible shapes paired with confidence values. To perform the shape recognition, the $N algorithm depends on a data set that defines the shapes to be recognized. A game programmer can easily extend AIPaint's sketching language by providing new shape definitions, enabling the designer to draw symbols for new actions and conditions.

Our implementation of the $N algorithm also returns the locations of control points in recognized shapes, which are specified as part of each shape's definition. Control points simply denote the points on a shape that might be meaningful, such as the head and tail of an arrow. Knowing the locations of the control points of a recognized shape makes it possible to perform shape clean-up.

Once a shape is unambiguously recognized, it is passed to the next stage of processing.

Sketcho: From Shapes to Statements

In the step labeled Sketcho in Figure 3, shapes are translated into statements that represent their meaning. For example, the arrow in Figure 1(b) is translated to a *move-to* statement and the shape in Figure 1(c) is translated to a *distance-conditional* statement.

Notice that Sketcho has two additional inputs, namely clarifications from the designer and game context information. These are needed to resolve possible ambiguities in interpreting the designer's intent, as explained below.

Clarification Questions

Although it does not occur in the authoring of the four Pac-Man ghosts, it is possible in principle for there to be ambiguity in the basic translation from shapes to statements. For example, if a distance conditional is to be added to a behavior that already contains a conditional, it may be unclear which test is done first. Sometimes, this ambiguity can be resolved from the game context, specifically the current decision states of all the nodes in the behavior tree (details beyond the scope of this paper). However, if this fails to resolve the ambiguity, our architecture also provides a mechanism for asking multiple-choice questions of the designer and providing the answers as input to Sketcho.

Generalization

Sometimes a designer wants to make a statement about a *class* of game objects, instead of a particular instance. Perhaps this is because the agent needs to make a decision about an object that is dynamically selected. An example of this phenomenon appeared in the preceding section, in which the soccer game designer wanted an agent to move to the member of the opposite team who last touched the ball. The particular opposite team member specified by this statement changes often during game play.

To support this kind of specification, the game programmer must, in effect, provide AIPaint with a set of generalization rules that make sense for this particular game. Concretely, the programmer defines a specially marked game feature that is to be used for generalization, and a procedure for keeping it up to date. For example, in Pac-Man, the *closest-pellet* is such a feature. In the soccer game, the programmer must define *last-player-who-had-the-ball.* The game designer and game programmer should work together to decide which generalizations will be natural for their game.

Thus whenever the designer draws a shape involving a game object, AIPaint automatically checks whether this object is currently the value of one of the special generalization features. If so, the argument to the resulting statement is not the particular game object, but a placeholder that produces the appropriate game object when evaluated at runtime.

Directo: From Shapes to Statements

The Directo step in Figure 3 collects the input statements into a syntax tree and then traverses the tree, replacing each statement with a list of directions for how to appropriately modify the behavior tree. Only this step and the next one (behavior transformer) would need to be changed if the AIPaint target behavior representation were changed from behavior trees to something else.

In the current version of AIPaint, directions include adding, removing and splicing behavior tree nodes. For example, the *move-to* statement in the creation of Blinky's behavior above gives rise to a direction that adds an appropriate action node at the root of the behavior tree.

Behavior Transformer

Directions are then passed to the Behavior Transformer module, which applies them to the behavior tree of the currently selected agent *in its current paused state*. The designer can then un-pause the game and the agent will begin behaving according to the new behavior tree.

Figure 4 illustrates the transformation of Blinky's behavior tree into Clyde's. Blinky's tree contains only one action node; the behavior transformer splices it with a decision node that tests Clyde's distance to Pac-Man against a threshold of eight units.

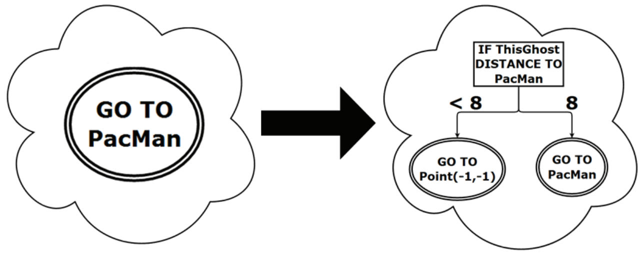

Figure 4: Example transformation of Blinky's behavior tree into Clyde's.

Behavior Debugging

In addition to creating and modifying agent behaviors by sketching directly on the paused game world, designers can also use two important AIPaint features for debugging: "Show Me What You're Thinking" and poseability.

Show Me What You're Thinking

Continuing the Pac-Man authoring example, suppose the designer wants to confirm that Clyde does indeed target Pac-Man when he is more than eight tiles away. The best way to do this is to press a special key that asks Clyde to "show me what you're thinking" while the game is running. In this mode, Clyde automatically draws his current primitive action on the screen as a shape, e.g., an arrow (see Figure 5). Note that these arrows are not drawn by the designer, but generated and rendered by AIPaint as a dynamic view on Clyde's behavior tree. As the designer plays the game, he can continuously observe how Clyde's current action changes with the situation: When Pac-Man moves, does the action arrow target Pac-Man's new position? When Clyde comes within eight tiles of Pac-Man, does the action arrow now point to the corner of the maze?

Implementing this feature involves essentially pulling data through the Figure 3 pipeline in reverse. AIPaint displays the current action by first producing a

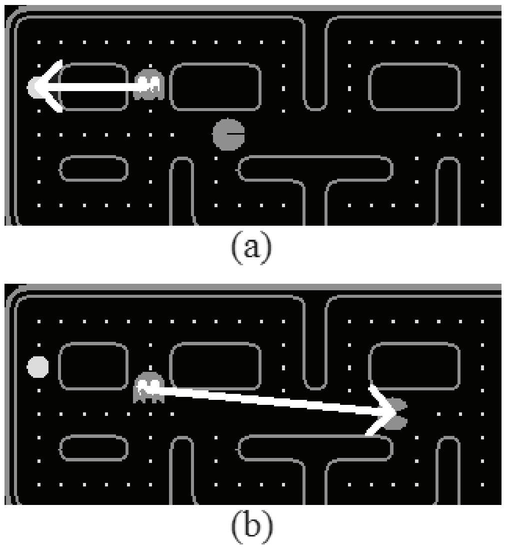

Figure 5: When "Show Me What You're Thinking" is active, AIPaint draws the agent's current action using the appropriate shape. Here, Clyde is shown (a) targeting the corner of the maze when close to Pac-Man and (b) targeting Pac-Man when far away.

corresponding statement. This statement is then back-translated to the corresponding shape, which is then displayed on the game screen.

Poseability

Often a designer needs to test an agent's behavior in a very particular scenario that occurs infrequently in normal game play. Instead of forcing the designer to get to the desired game state by playing, AIPaint allows the designer to "pose" a paused game by clicking and dragging visible game objects to desired locations. This feature is implemented via a set of required methods on the API for every visible game object that allow its position (and possibly other features) to be manipulated by AIPaint. For example, to confirm Clyde's correct behavior, instead of waiting until he happened to be close to Pac-Man and then farther than eight units, the designer could simply pose him and/or Pac-Man at various distances and see what happens.

Evaluation

We evaluated AIPaint using a small observational study, whose goal was primarily to give us some insights into the strengths and weaknesses of the current prototype. Four undergraduate students were given brief instructions on how to use AIPaint on a touch screen and then tasked with building the behaviors of the four Pac-Man ghosts. Each student was asked to rate themselves on a scale from 1 (unfamiliar) to 5 (familiar) on the topics of programming, videogames and videogame design. All students rated themselves 3 or less on programming, 4 or 5 on videogames and 3 or more on videogame design.

All the students completed all four ghosts. Other than difficulties getting the stroke recognition to work reliably, which we addressed by adding a special sketch training phase, the main complaint concerned confusion about whether to press the "execute" button after each shape was drawn or after all the shapes for a given agent were completed. Students also complained of difficulty expressing Clyde's behavior, which involves a *distance-conditional* statement paired with two *move-to* statements; students had little success deciding when to draw each statement and how the conditional statement would affect the behavior. On a 7-point Likert scale, students' responses to the statement "I found the AIPaint user interface easy to use" were one neutral (4), two agree somewhat (5), and one agree (6). Positive written comments included: "If only AIPaint could read the arrows I drew then I think it would work well" and "pretty nifty."

Evaluating software development tools is very difficult in general. For example, what should AIPaint be compared to? You cannot compare programmers using AIPaint versus programmers using Java, for example, because AIPaint is intended for non-programmers who may not write Java. You could compare AIPaint with another graphical tool, such as BrainFrame or BehaviorShop, but such a comparison would conflate the issue of state machines versus subsumption versus behavior trees with the key contribution of AIPaint, namely merging the authoring and game contexts. An evaluation of AIPaint would also benefit from the participation of expert game designers instead of students. A more robust evaluation remains as future work.

Conclusions and Future Work

We have demonstrated the feasibility and potential of a new type of general-purpose behavior authoring tool in which spatial/symbolic directions are sketched directly on the game world.

The biggest unanswered question is the *scalability* of this approach, i.e., for games with different kinds of decisions and behaviors or much bigger behavior trees with the same type of nodes we already use. Regarding the first kind of scalability, it is worth reiterating that we do not expect to totally eliminate the AI game programmer. Our architecture provides an API for defining new sketching primitives and their semantics, which we expect to be used for new games. For very large behavior trees, we expect that some additional visualization, such as a "scrapbook of situations," will be needed to manage complexity

In addition to fixing the problems highlighted in the user study, the following extensions would improve the system.

First, agents sometimes need to make decisions based on features of the game state that may not have graphical representations, such as the agent's current health. A solution to this problem is to define special graphical affordances for such features that are only visible when the game is paused for AIPaint interaction. For example, health could be indicated by a number next to the agent, and that number could then be included in a sketch.

Second, current clarification questions force the designer to think about the structure of the behavior tree representation to a greater degree than is desirable. A better approach might be to allow AIPaint to temporarily *pose* the game world in a way that highlights the implications of the designer's choice. For example, AIPaint might show the designer two different game states that would result from different agent decisions depending on the order in which conditional behavior nodes were merged.

Third, we could allow the designer to edit the shapes displayed in "show me what you're thinking" mode. For example, the designer could drag the head of an arrow from one place to another.

Finally, designers currently have no way to override the generalization rules in effect for their game. A simple mechanism for either asking before applying the rule or undoing the generalization afterward would be useful.

Acknowledgments

We thank Prof. J. Rosenthal at the U. of Toronto for his soccer game code and www.bennychow.com for the Pac-Man code. Special thanks to Damián Isla for inspiring this project and for his feedback throughout.

References

Anthony, L. and Wobbrock, J. O. 2010. A Lightweight Multistroke Recognizer for User Interface Prototypes. In *Proc. Graphics Interface*, Toronto, Canada, pp. 245-252.

Bassett, J.; Becroft, D.; and Mejía, A. 2011. *AIPaint: A Sketch-Based Behavior Tree Authoring Tool*, Project Report CR1-1001, Worcester Polytechnic Institute, Worcester, MA.

Fu, D. and Houlette, R. 2002. "Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games. *IEEE Intelligent Systems* vol. 17, no. 4, pp. 81-84.

Heckel, F. W. P.; Youngblood, G. M.; and Hale, D. H. 2009. BehaviorShop: An Intuitive Interface for Interactive Character Design. In *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*, pp. 46-51.

Hecker, C. 2008. Structure vs. Style. *Game Developers Conference*. http://chrishecker.com/Structure_vs_Style (accessed 2011).

Isla, D. 2005. Handling Complexity in the Halo 2 AI. *Game Developers Conference.* http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml (accessed 2011).

Smith, D.; Cypher, A.; and Tesler, L. 2000. Programming by Example: Novice Programming Comes of Age, *Comm. ACM*, 43(3).