Project Number. CR1-1001

AIPaint: A Sketch-Based Behavior Tree Authoring Tool

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Jesse Bassett

David Becroft

Adrián Mejía

April 28, 2011

Approved:

____

Professors Charles Rich and Candace L. Sidner, Major Advisors

Abstract

Videogame designers who make use of a behavior authoring tool to define the behaviors of computer-controlled characters must typically split their attention between the tool and the game. We have built a behavior authoring tool with an improved design experience by merging these two contexts into one. This paper outlines the design and development of a game-independent behavior tree authoring tool called AIPaint. Its user interface allows the designer to build and edit behavior trees via a natural sketching interface overlaid on the game world. The use of AIPaint to create behaviors in two simple games is presented.

Acknowledgements

We thank Professor Jeffrey Rosenthal at the University of Toronto for his original soccer game code and the authors at <http://www.bennychow.com/pacman.html> for the Pac-Man code. We also thank Martin Field for his assistance with the Paleo sketch recognition software. Special thanks to Damián Isla for the presentation at WPI which inspired this project and for feedback throughout. Finally, we thank our advisors Chuck Rich and Candy Sidner for sharing their wisdom, guidance, and chocolate.

Table of Contents

List of Illustrations

1. Introduction

On a typical game development team, there is a designer and a programmer: the designer is responsible for deciding how the computer-controlled characters in the game should behave, while the programmer has the technical expertise necessary to implement these behaviors. Development involves a communication loop between the designer and the programmer – the programmer builds the AI, the designer revises the spec in response to playtests, the programmer revises the code, and so on (see Figure 1). Frequent iteration leads to more robust and polished AI, so nontechnical designers sometimes make use of behavior authoring tools, which empower them to edit behaviors on their own.

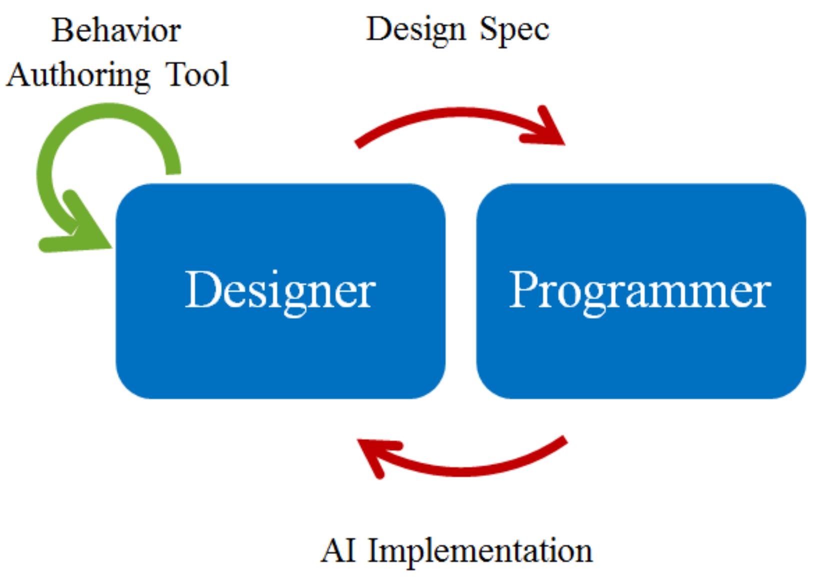
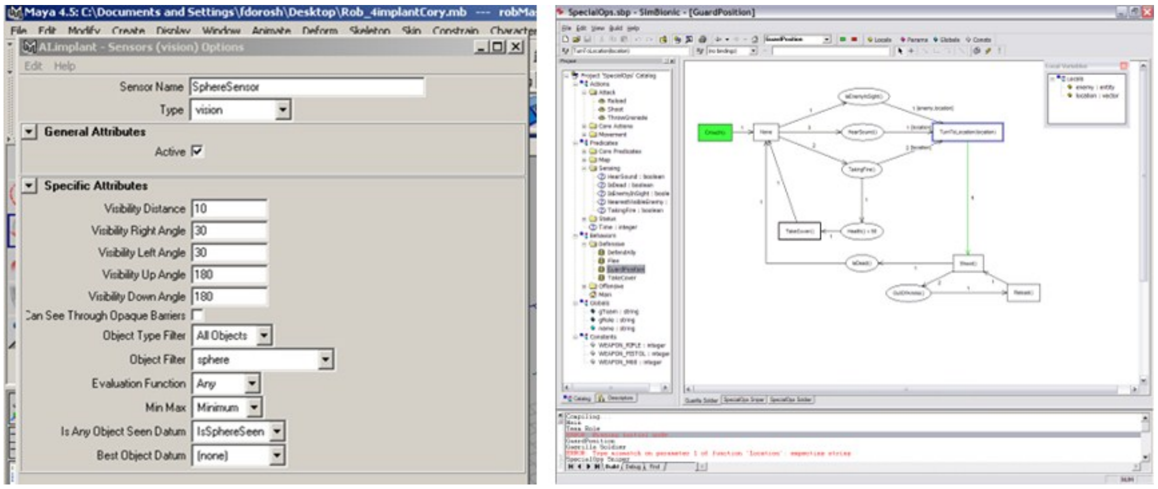


Figure 1: Designer-programmer communication in behavior authoring

For example, BrainFrame (Fu and Houlette 2002) provides a game-independent agent architecture and a graphical interface for editing finite state machines. Once BrainFrame is integrated with a particular game, a nontechnical designer can create behaviors by manipulating state machine 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 that could generate computationally efficient behavior representations while still offering designers a high degree of aesthetic expressiveness (Hecker 2008). Behavior authoring tools such as BehaviorShop (Heckel, Youngblood, and Hale 2009) have been written with this goal in mind. BehaviorShop provides an interface for creating agents based on subsumption architectures, which the tool's authors have shown to be an easier behavior representation than others for novices to understand.

As user-friendly as BehaviorShop and similar tools are, 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 (see Figure 2). We believe that this configuration limits the aesthetic expressiveness afforded by these authoring tools and that merging these two contexts into one improves the behavior design experience. This paper outlines the design and development of a game-independent behavior tree authoring tool called AIPaint. Its user interface allows the designer to build and edit behavior trees via a natural sketching interface overlaid on the game world, so that behavior authoring and testing may occur in the same environment.


Maya 4.5: C:\Documents and Settings\fdorosh\Desktop\Rob_4implantCory.mb --- robMa
File Edit Modify Create Display Window Animate Deform Skeleton Skin Constrain Characte
AI.implant - Sensors (vision) Options
Edit Help
SpecialOps.sbp - SimBionic - [GuardPosition]
File Edit View Build Help

Sensor Name SphereSensor
Type vision
General Attributes
Active ✓
Specific Attributes
Visibility Distance 10
Visibility Right Angle 30
Visibility Left Angle 30
Visibility Up Angle 180
Visibility Down Angle 180
Can See Through Opaque Barriers
Object Type Filter All Objects
Object Filter sphere
Evaluation Function Any
Min Max Minimum
Is Any Object Seen Datum IsSphereSeen
Best Object Datum (none)

Figure 2: The user interfaces of many of today's behavior authoring tools, such as AI.implant[1] (left) and SimBionic[2] (right), are independent of the game world.

The reader might note that EA Games' Madden NFL 2011 for the iPad includes a form of in-game sketch-based behavior authoring. It allows the player to direct football plays by drawing paths on the field using the iPad's touch screen; the simulated football players will follow these paths when the play begins. AIPaint is different in that it is a generic tool designed to be applied to many games – not just football plays. Also, the expressive power of AIPaint sketches is much higher than that of Madden's route sketches, because AIPaint allows the user to construct behaviors that involve making decisions and performing arbitrary actions, rather than simply following paths.

[1] Image from <http://www.gilgameshcontrite.com/Computer_AI/images/79.jpg>

[2] Image from <http://www.simbionic.com/images/sbscreen_large.gif>

2. Design Goals

In designing AIPaint, we used as our guiding metaphor the relationship between an actor and a director rehearsing a play (see Figure 3). Here, the game designer is like a director who needs to guide the behavior of group of actors, who represent agents. The director can watch the actors execute their behaviors, then tell them to stop when he sees something wrong. Next, he can issue some directions, and if the 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 rearrange the actors on the stage to set up the desired situation. It is important to note that a director specifies the behavior of his actors by communicating with them via an intelligent interface – not by reaching into their heads and manually adjusting their brains.

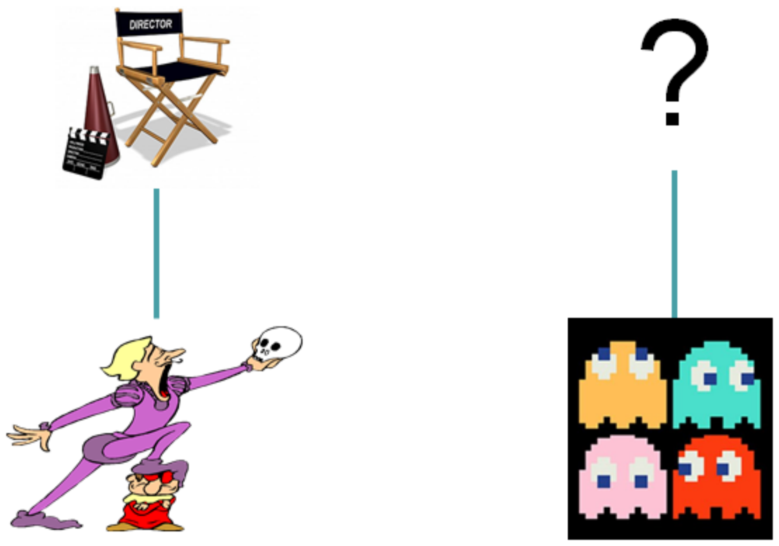


Figure 3: A theater director specifies the behavior of an actor in a play, just as a game designer uses some tool to specify the behavior of a videogame character. We are interested in the interface between the designer and the agent.

The metaphor of the game designer as a director led us to establish four design criteria for AIPaint. Our first criterion is that a game designer should 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 might imagine the designer often heading for a whiteboard or opening an image editor to draw diagrams, perhaps overlaid on game screenshots, in the same way a football coach or commentator might describe a play by marking up an image of the field with X's and arrows. For this reason, AIPaint presents a sketch-based interface, by which the designer may draw symbols – lines, circles, and arrows – on the game world to describe behaviors. This kind of interface lends itself very well to the use of a touch screen or tablet for input, but a mouse suffices.

Our second design criterion is that AIPaint should present a clear connection between the spatial and the symbolic. That is, the spatial relationships among game objects and the symbolic information that defines how an AI interprets the game world should be visible at the same time. In a behavior authoring tool in which the designer has to describe behaviors without being able to see the game world, this connection is less clear. One might compare the experience to that of painting a picture by typing in the coordinates of the desired brush strokes, without being able to see the canvas. The purpose of AIPaint's interface design is to put this more difficult aspect of behavior authoring into its proper context.

Our third design criterion is that AIPaint should produce behaviors in a simple, widely used representation. Therefore, we chose to use behavior trees (Isla 2005) in our implementation. The architecture leaves room for the possibility of using other AI representations, but this has not been explored by the authors.

Our final design criterion is that AIPaint should be a game-independent tool. A game implementation that uses AIPaint for behavior authoring must provide code that conforms to various interfaces, so that AIPaint can call upon the game to perform tasks such as collision detection or screen-space to world-space conversion. The game provides world state information to AIPaint, as well as agent objects that can execute primitive actions according to an AI representation. The game-independent portion of AIPaint contains the sketching interface and the code that builds an AI representation from the sketch input. Certain aspects of the game-independent portion of AIPaint are available to the game programmer for extension, and it is likely that each game that uses AIPaint will extend the sketching language with game-specific symbols or otherwise customize the tool.

3. Proof of Concept

Here we demonstrate, from the designer's point of view, the use of AIPaint for behavior authoring in two simple games. The first example is a Pac-Man game (see Figure 4), in which we recreate the original behavior of each of the ghosts. The ghost AI in Pac-Man is all about moving through space relative to other game objects, and is therefore a useful example for illustrating the way AIPaint's user interface creates a connection between spatial and symbolic information. The second example is a computer soccer game, which despite its simplicity is different enough from Pac-Man to illuminate the game-independence of AIPaint.

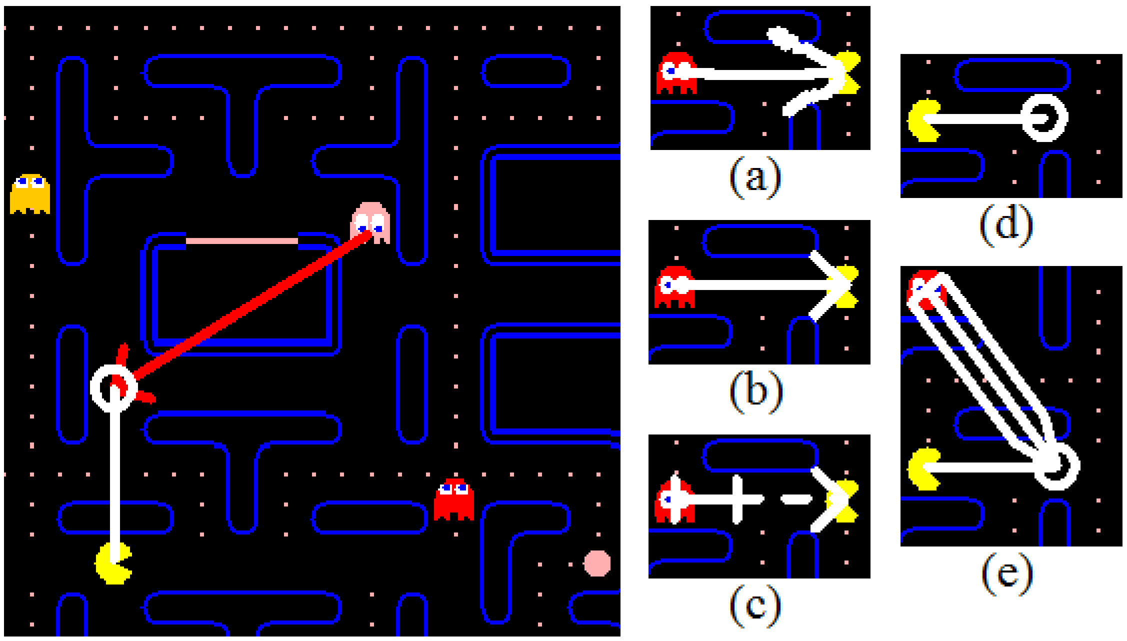


Figure 4: Pac-Man game running with the AIPaint tool. (a) A shape is drawn in natural sketch input. (b) The sketch input is recognized as an arrow shape and cleaned up. (c) A distance-conditional arrow with slider bar. (d) A temporary position variable relative to Pac-Man's position and orientation. (e) A "go twice as far" symbol connected to a temporary position variable.

Pac-Man

Each ghost in Pac-Man – Inky, Blinky, Pinky, and Clyde – has a unique behavior. We will examine the workings of AIPaint by demonstrating how a game designer would build each of these four behaviors.

Blinky

The behavior designer begins by running a build of his game in which the AIPaint data structures are initialized. Once the game is running, it will receive mouse and keyboard input until the designer activates the AIPaint UI, which pauses the game. The designer may then click on a ghost to edit its behavior.

Once the agent has been selected for editing, the designer can sketch on the game world to build behaviors. As the designer sketches, shapes are automatically recognized and cleaned up. Each of these shapes has a certain meaning when drawn on the game world. For example, Figure 4a shows a Pac-Man screen on which an arrow has been drawn from the red ghost, Blinky, to Pac-Man; Figure 4b 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.

In the original Pac-Man game, Blinky has a very simple behavior – he always moves towards Pac-Man. In fact, this arrow from Blinky to Pac-Man is all that is needed to represent his behavior. The designer can now copy Blinky's behavior into another ghost, Clyde, and edit the copy to build Clyde's original behavior.

Clyde

Clyde, the orange ghost, has a behavior that involves choosing between two actions. When he is farther 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 just outside one corner of the maze. We will need to compare the distance between Clyde and Pac-Man to some threshold value – eight tiles – and use the result to select one of the actions. This shape takes the form of an arrow from Clyde to Pac-Man with a hash mark in the middle that the designer can drag back and forth to specify the threshold value (see Figure 4c). To specify Clyde's complete behavior, the designer draws this decision arrow, along with an arrow from Clyde to Pac-Man and an arrow from Clyde to the corner of the maze.

Pinky

Pinky's behavior is to move to the space four units ahead of Pac-Man. The designer first draws a circle around the space four units ahead of Pac-Man and a line connecting the circle to Pac-Man (see Figure 4d). Now, the circle specifies a position relative to Pac-Man's position and orientation, and other shapes can be connected to this circle. To tell Pinky to move to this space, the designer draws a movement arrow from Pinky to the circle.

Inky

Inky's behavior is the most complex of the four ghosts. He considers a ray from Blinky to the space two units ahead of Pac-Man, doubles the length of this ray, and targets the position at the end of this new ray. To represent this behavior, the designer must make use of a special shape that means "go to the space twice the distance from point A to point B". As with Pinky, the designer first draws a circle and line to specify the space two units ahead of Pac-Man. Then, he draws this new shape, which looks like three parallel lines, from Blinky to the circle (see Figure 4e).

Soccer

We integrated AIPaint with a simple computer soccer game as well (see Figure 5) to demonstrate the game-independence of AIPaint. We implemented a blocking behavior, in which a player 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. AIPaint needs to know that when the designer draws an arrow to the player who last touched the ball, he might intend to make a statement about *whoever* last touched the ball, rather than that specific player. The game programmer must tell AIPaint how to make this generalization. Then the designer can specify the desired behavior by drawing a movement arrow to the member of the opposite team who last touched the ball, and AIPaint will make the appropriate generalization.

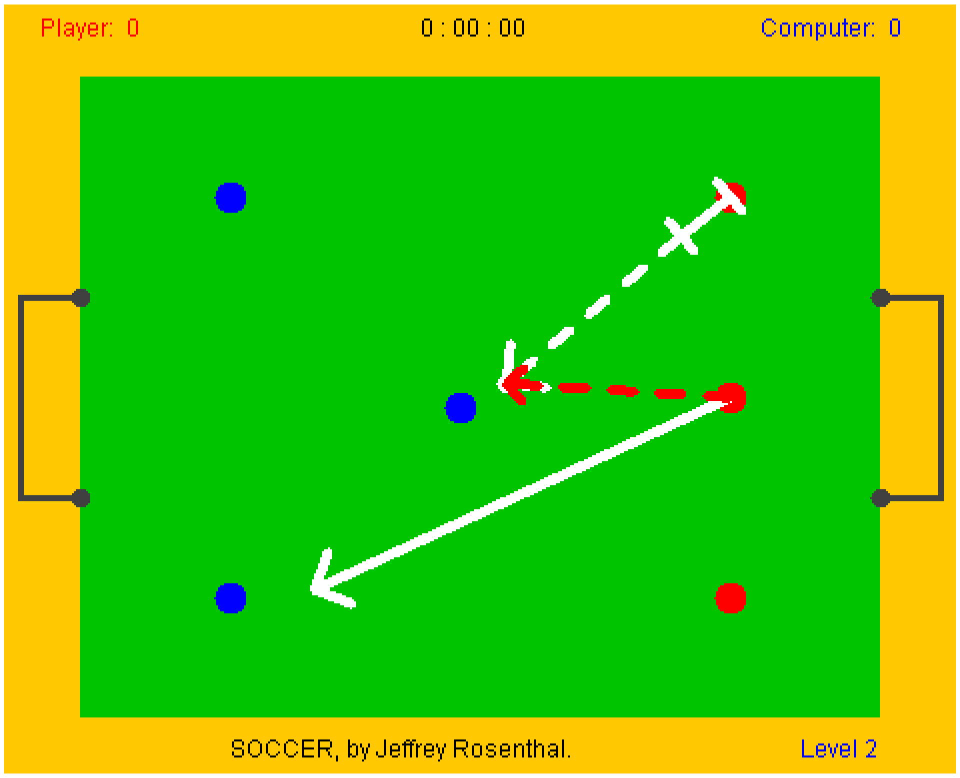


Figure 5: Soccer gameplay.

4. Architecture

The AIPaint software, which the authors implemented in Java, relies upon the game code to implement certain interfaces with game-specific behavior. Note that both the game designer and the game programmer are users of the tool, since the programmer needs to write some code to connect the game to AIPaint.

Interface to the Game World

AIPaint achieves game independence by communicating with the game world via Java interfaces. A game connects to AIPaint by implementing these interfaces and initializing certain data structures. See Appendix A for a full listing of these interfaces.

Primarily, the game is responsible for providing AIPaint with world state information, in the form of a collection of world features mapped to values. The game will typically obtain feature-value pairs by examining game objects, each of which must announce the feature that corresponds to its state and the current value of that feature. Game objects may also announce their orientation, if it might be meaningful to the game designer. Behavior tree decision nodes then make decisions during gameplay based on attribute objects, which represent computations on the values of world features. For instance, a particular attribute, when evaluated, might look up the values of the world features that correspond to two game objects' positions and compute the distance between them.

The Data Pipeline

AIPaint's data pipeline, illustrated in Figure 6, transforms sketch input into a form that can be used to build or modify a behavior tree. Using arrows drawn between game objects to shuffle decision nodes into their intended positions is a fairly large conceptual leap, so we break the process down into several more tightly scoped steps. Sketch input is first recognized and cleaned up into shapes, which are then mapped to statements, the set of which make up a language we call Sketcho. Statements are then translated into directions, which make up a language called Directo. Generalization and clarification questions to the designer support the process by bringing in contextual information. Finally, directions are used to effect changes to the behavior tree. Figure 7 illustrates the movement through the pipeline of examples of data involved in building Blinky's behavior.

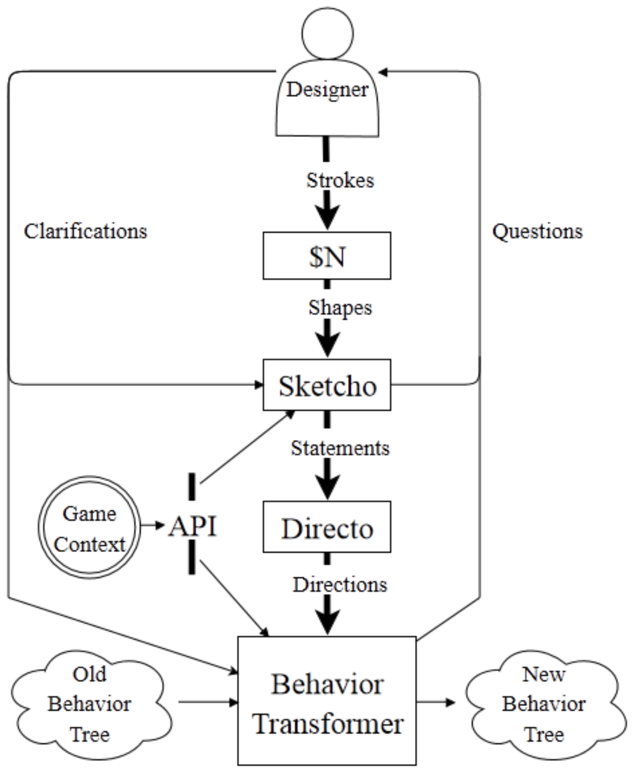


Figure 6: AIPaint data pipeline architecture.

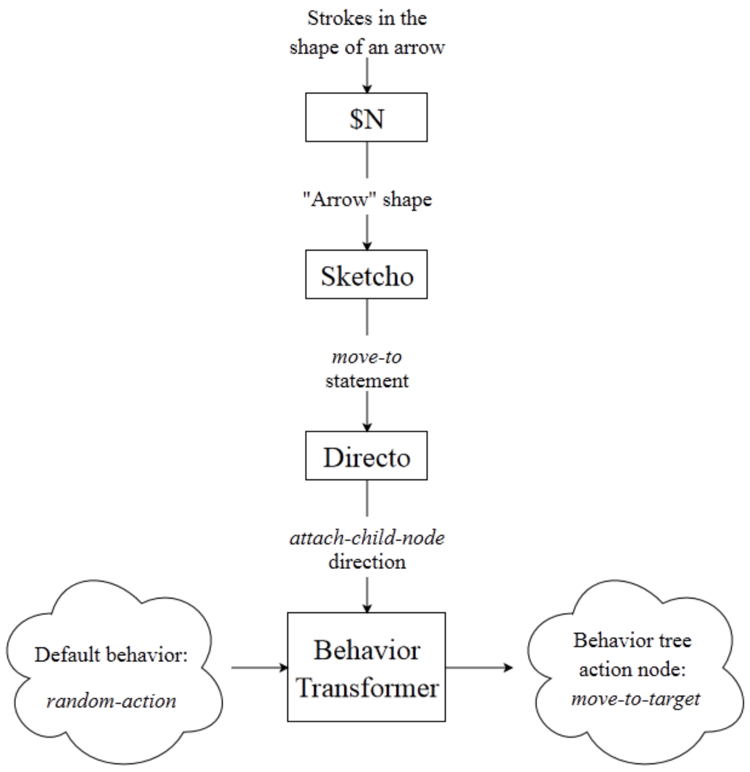

Strokes in the shape of an arrow
$N
"Arrow" shape
Sketcho
move-to statement
Directo
attach-child-node direction
Default behavior: *random-action*
Behavior Transformer
Behavior tree action node: *move-to-target*

Figure 7: The movement through the pipeline of data involved in building Blinky's behavior.

$N: Recognizing Shapes from Strokes

The first step is to recognize the shapes sketched by the designer. As the designer drags a mouse cursor or touch stylus across the screen, the input points are recorded and grouped into strokes. These are passed to a Java implementation of the $N multistroke recognition algorithm (Anthony and Wobbrock 2010), which returns a list of possible shapes paired with confidence values. To perform the recognition, the $N algorithm depends on a data set that defines each shape to be recognized; a game programmer can extend the Sketcho language by providing new shape definitions. The AIPaint 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.

Sketcho: From Shapes to Statements

When AIPaint decides that the input relatively unambiguously specifies a particular shape, it replaces the input strokes with the cleaned-up shape and requests that the game provide it with an instance of the statement object associated with that shape. This process is labeled as "Sketcho" in Figure 6; shapes are consumed and a syntax tree of statements is sent to the next stage of the pipeline. See Table 1 for a list of all statement types implemented by the authors for use in the Pac-Man game.

Statement objects encode the meanings in the designer's input. For example, the arrow in Figure 4b is a *move-to* statement, encoding the meaning that the ghost is meant to move to Pac-Man. Statement objects produce their meanings by translating themselves

into sequences of directions, which modify the behavior tree – this *move-to* statement produces directions that add a *move-to-pac-man* action node to the behavior tree.
Some statements are not translated into directions that modify the behavior tree, but still provide information to other statements in the sketch; these statements are called intermediate variable statements. For example, specifying Inky and Pinky's behaviors required the use of a statement that defines a space some distance ahead of Pac-Man. This statement says nothing about the behavior tree by itself, but the designer can draw a *move-to* statement that points to it. Together, these two statements produce an action node in the behavior tree telling the agent to move to the space *x* units ahead of Pac-Man.

Table 1: Types of statements implemented by the authors for use with the Pac-Man game. Game independence refers to whether the statement is defined in AIPaint or in the specific game implementation.

Statement Type	**Graphical Representation**	**Game Independent?**
PacManMoveToStatement	See Figure 4b	No
DistanceConditionalStatement	See Figure 4c	Yes
SketchoIntermediateVariable	See Figure 4d	Yes
PacManGoTwiceAsFarStatement	See Figure 4e	No

Clarification Questions

If any object finds that it requires additional input from the designer to complete some operation, it can call an AIPaint function to display a multiple-choice question and obtain the designer's response. For example, if the designer's statement input produces directions specifying that a decision node in the behavior tree should be removed, what should happen to the decision's child nodes might be ambiguous. Clarification questions are a way to disambiguate user input in such situations. The contribution of this mechanism to the Sketcho and Directo processes is noted in Figure 6 as “Questions” and “Clarification”.

Generalization

What happens if a designer wants to make a statement about a group of game objects, instead of just one? Perhaps the designer wants his agent to make a decision about an object that is selected dynamically each moment. As mentioned previously, the Soccer designer might want a player to move to the member of the opposite team who last touched the ball. The particular player specified by this statement changes often during gameplay.

To accomplish this effect, the game programmer provides AIPaint with generalizations that make sense for his game, in the form of game-specific attribute objects that select a “special” game object from a group. In this case, the programmer might specify a *last-player-who-had-the-ball* attribute that, when evaluated on a world state and a group of players, returns the member of the group who last touched the ball. Note that the game designer and game programmer need to work together to decide which generalizations will be useful for their agents to make.

When the designer draws a statement and attaches it to a game object, AIPaint automatically evaluates whether this object is special in some way – for instance, if this object is the player who last had the ball. If so, the statement is attached not to the particular game object, but to an attribute that produces a game object when evaluated at runtime. This is how game context factors into the Sketcho module in Figure 6.

Directo: From Statements to Directions

Once the designer has provided his sketch input, AIPaint collects the statements into a syntax tree and traverses it, having each statement translate itself into a list of directions that represent the statement's intent. This step is the "Directo" process in Figure 6. Directions describe modifications to a behavior tree, such as the addition or removal of decision or action nodes. See Table 2 for a list of all direction types implemented by the authors.

Directions are then passed to a behavior transformation module and applied to the behavior tree. Each direction operates on the behavior tree within the context of the existing behavior tree and the world state in which the direction was generated. Once the old behavior tree has been modified and the designer has unpaused the game, the agent will begin behaving according to the new behavior tree.

Figure 8 illustrates the transformation of Blinky's behavior tree into Clyde's. Blinky's tree contains only an action node, and the behavior transformer replaces it with a decision node that tests Clyde's distance to Pac-Man against a threshold of eight units.

Table 2: Direction types implemented by the authors.

Direction Type	**Effect**
AttachChildDirection	Adds a new node in the next available location
AddNodeDirection	Inserts a node at a particular location
RemoveNodeDirection	Removes a node from a particular location

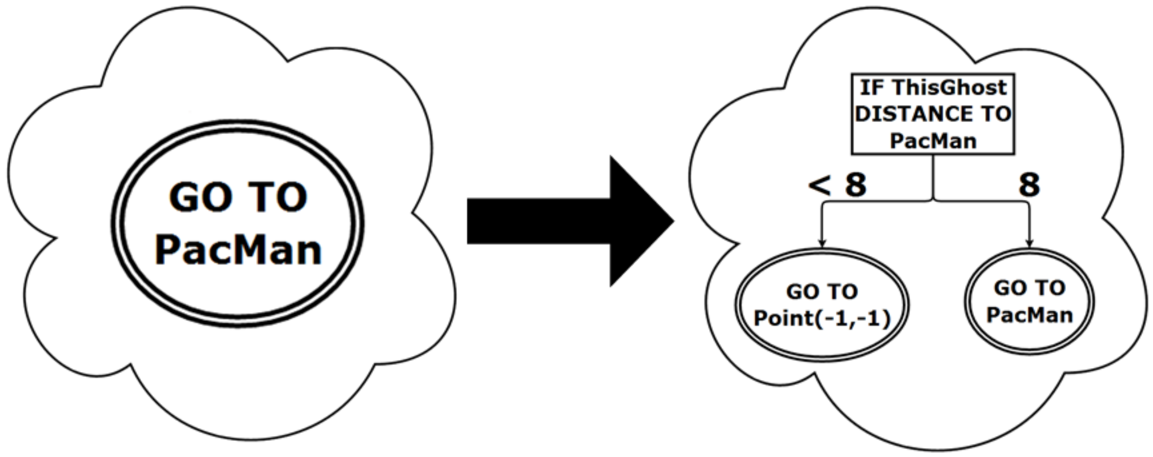


Figure 8: Transformation of Blinky's behavior tree into Clyde's.

Behavior Transformer: From Directions to Artificial Intelligence

The Behavior Transformer is responsible for applying directions to the existing behavior representation and returning the modified representation. The Behavior Transformer executes the directions with as little additional input from the designer as possible.

Building the Behavior Tree

The Behavior Transformer modifies the existing behavior tree by using the input list of directions to assemble a separate behavior tree. Then, the problem becomes one of merging two existing behavior trees, rather than modifying nodes or branches in a single behavior tree.

Merging Behavior Trees

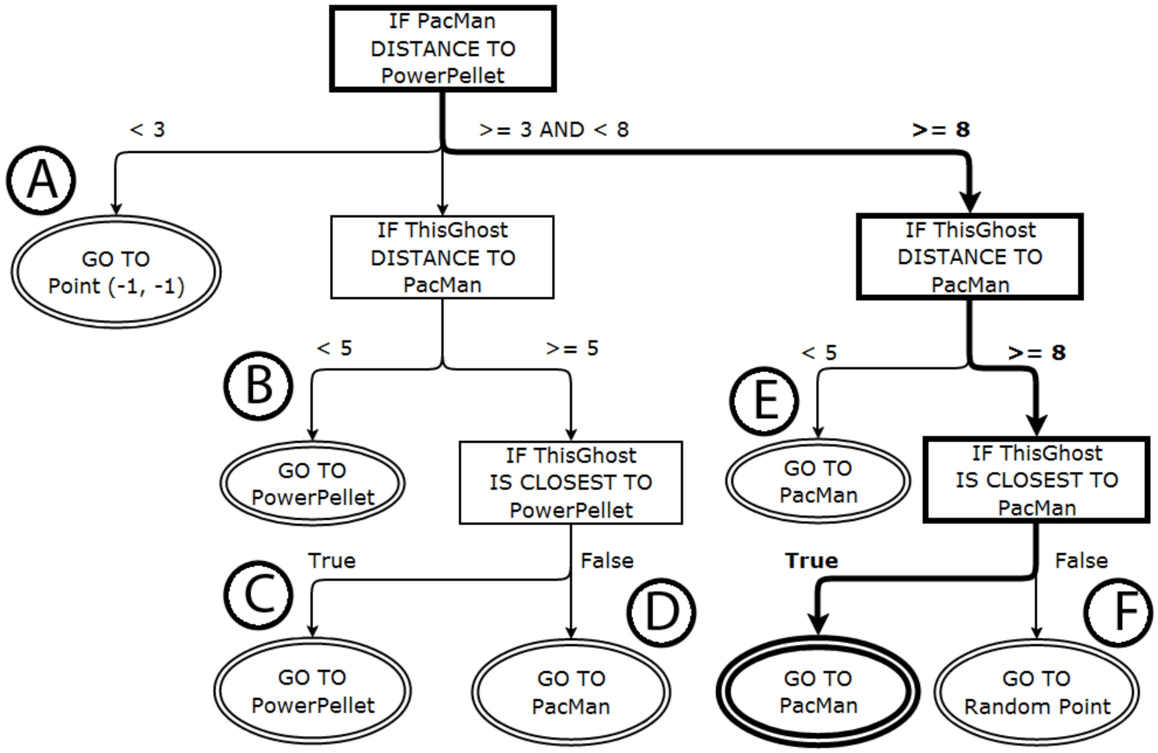

IF PacMan DISTANCE TO PowerPellet
< 3
>= 3 AND < 8
>= 8
A
GO TO Point (-1, -1)
IF ThisGhost DISTANCE TO PacMan
IF ThisGhost DISTANCE TO PacMan
< 5
>= 5
< 5
>= 8
B
E
GO TO PowerPellet
IF ThisGhost IS CLOSEST TO PowerPellet
GO TO PacMan
IF ThisGhost IS CLOSEST TO PacMan
True
False
True
False
C
D
F
GO TO PowerPellet
GO TO PacMan
GO TO PacMan
GO TO Random Point

Figure 9: Complex behavior tree, with the active path highlighted.

Merging two independent behavior trees according to the intentions of the designer can be complicated. If we let the incoming tree be denoted by *I*, and the existing tree be denoted by *E*, the merge between the two is only trivial if the following two conditions are met:

1. There is one and only one node N contained in *E* that considers the same attribute as the root node of *I*.
2. That none of the children of *N* are branched over the same interval as any of the children of the root node of *I*.

If these conditions are not met, *I* will have multiple possible places of insertion, and this means that the behavior tree can be inserted in a place not intended by the designer. Without any clear heuristic to determine the better location, the placement is arbitrary. If condition 2 is violated, the insertion of the new behavior tree could overwrite an existing subtree, losing data and possibly desired behavior.

The Active Path

To better the odds of meeting condition 1, we can reduce the tree by considering extra information that is passed along with the list of directions. The current world state is packaged with every set of directions, so we can use this to determine the currently active path of decision nodes through the tree (see Figure 9). This active path represents the context of the designer's input. For instance, if the incoming tree makes a decision on the attribute "IF ThisGhost DISTANCE TO PacMan", there are two places in the tree in Figure 9 that also make decisions on that attribute. The decision node that the designer most likely intends to modify is the one in the active path.

Thus, the first place to look in any merge is along this active path. If no decision node is found with a matching attribute, then the behavior transformer looks to the nodes adjacent to the active node – the action node at the end of the active path. Adjacent nodes are found by traversing the active path, and changing the world state to make neighboring nodes active. The heuristic importance of an adjacent node relative to the active node is calculated in two steps. First the distance from the adjacent node to the active node is calculated, and then the path from the adjacent node to the active node is determined. For that path, the total number of deviations from the active path is calculated. By adding the total number of deviations and the distance to the active node, it is possible to create a heuristic to determine the most relevant paths through the tree according to the current context. Using the adjacent nodes listed in Figure 9, the score of node D would be a total of 7. It takes five steps to reach that node, and it involves two deviations from the current world state. The final sorted order would be [F(2), E(3), A(4), B(6), C(6), D(7)], with the score of the node indicated inside the parentheses.

If there exists an adjacent node on which the incoming tree can merge, the Behavior Transformer will ask for clarification that the adjacent path is the intended context. If, in the tree in Figure 9, the designer meant to pose Pac-Man closer to the power pellets, the active path the designer may have intended would have been node C. If the designer accepts, the merge will take place there; otherwise, the search will continue to the rest of the tree.

If the merging algorithm goes through the whole tree without any success, it can try to reverse the merging process and merge the existing tree into the incoming tree. This is not a decision the algorithm is qualified to make on its own, and it must ask for clarification from the designer. The designer is presented with a dialog asking if he would like to try the reverse merge. If the designer accepts, the merging algorithm reverses the roles of the existing and incoming tree and tries to merge again.

If the designer declines or if the reverse merge fails, the Behavior Transformer can ask for more clarification on where to conduct the merging. It will ask about the priority level of the input decision, which indicates how far up the active path the Behavior Transformer should try to place the decision. If the designer decides to see more options, it will suggest places along the adjacent paths instead. This way the node can be inserted into the tree at the exact place where the designer desires it. This dialog takes place in natural language, so the attributes must have a method to describe themselves.

5. Design Debugging

Having built his agents' behaviors, the designer might make use of two important AIPaint features to debug his behavior designs: "Show Me What You're Thinking" and poseability. At this stage, the designer has finished drawing sketch input to build behaviors, and he simply wants to test that his agents behave as intended.

Show Me What You're Thinking

First, to pick up our example from earlier, assume that the designer wants to check that Clyde does indeed target Pac-Man when they are more than eight tiles away from each other. To verify this, the designer can ask Clyde to show him what he is thinking during gameplay (see Figure 10). Clyde will draw his current action on the screen as a statement – in this case, a move-to arrow from Clyde to Pac-Man. Note that this arrow is not drawn by the designer, but rather generated and rendered by AIPaint as a view on Clyde's behavior tree. The designer may then resume playing the game and observe how Clyde's current action changes with the situation: when Pac-Man moves, does the arrow target Pac-Man's new position? When Clyde comes within eight tiles of Pac-Man, does the arrow now point to the corner of the maze?

This feature involves pulling data through the pipeline in reverse. AIPaint displays the current action node by asking the game to produce a statement that expresses this particular action. For instance, a move-to statement attached to Pac-Man would express the action move-to-pac-man. The statement is then displayed to show the agent's current action.

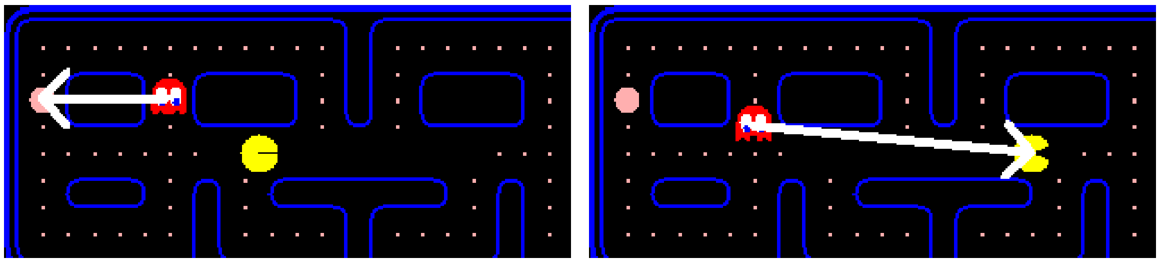


Figure 10: When "Show Me What You're Thinking" is active, AIPaint draws the agent's current action using the appropriate statement. Here, the agent is shown to target the corner of the maze when close to Pac-Man and to target Pac-Man when far away from Pac-Man.

Poseability

If the designer wants to test his agent's behavior in a very particular scenario that would take some time to bring about through normal gameplay, he might use poseability. Every game object that may be considered by agents' behavior trees must also allow the designer to manipulate its state at runtime. For example, the game object that represents Pac-Man makes it possible for the designer to set its position by clicking and dragging it to the desired location. Since every game object that an agent might consider must necessarily also be poseable in this manner, the designer is guaranteed the ability to configure any game scenario he wishes by simply dragging game objects about; he does not have to spend time trying to arrange unusual edge cases through gameplay. In our example, the designer can pose Clyde and Pac-Man closer together or farther apart and check that Clyde behaves as expected in each case.

6. Conclusions and Future Work

AIPaint could be made into a more powerful behavior editor with a few more features. First, agents might want to make decisions based on pieces of hidden world state that don't have graphical representations convenient for posing – for instance, whether Pac-Man is chowing down on a power pellet. AIPaint could enable the game to provide graphical representations of normally hidden state features that could be displayed to the designer for posing during AIPaint Pause.

Second, the choice of where to merge new decision nodes into an existing behavior tree has a significant impact on the resulting behavior, but AIPaint does not present the designer with a very powerful interface for making this choice. One way to make the interface more powerful without requiring the designer to edit the behavior tree directly might be to enable AIPaint to pose the scene while asking clarifying questions about what to do in particular scenarios, narrowing down the desired location of the new decision node.

Third, the statements displayed during "Show Me What You're Thinking" are presently immutable. Allowing the designer to edit them – for instance, by changing the target of a move-to statement – would be a natural and powerful extension of the user interface. This feature could also be extended to enable the designer to explore the decisions the agent is making as well as the actions it is taking.

Finally, designers are presently offered no interface for overriding AIPaint's eager generalizations. Designers should be notified when generalizations occur and should be enabled to opt out of them, as well as to specify that generalizations should be made when AIPaint would not do so automatically.

AIPaint might also be modified to support the construction of behavior representations other than behavior trees, such as planning systems, state machines, or subsumption architectures. The existing software architecture leaves room for other representations, but none have been implemented by the authors. A new behavior representation would require a new Directo language to modify it and might also lend itself to different sorts of sketches.

User testing would likely be of great benefit to AIPaint. It would be useful to gather data on whether AIPaint is any easier to use than other behavior authoring tools, whether Sketcho statements are easy to read, and what sorts of diagrams designers typically draw to express behaviors.

The AIPaint Java code and a video of the software are available at <http://www.cs.wpi.edu/~rich/aipaint>.

Appendix A: Interfaces to the Game World

```
/**
 *    This is an interface for the main handle that AI Paint has on
 *  the game. It should be implemented by one class that AIPaint
 *  will hold on to for the remainder of its execution.
 */
public interface Game {

      /**
       * Return poseable at the given position.
       * @param worldSpacePoint The position to test for a poseable
       * @return Poseable if it is in that position, null if none
       */
      public Poseable getPoseableAt(Vector2 worldSpacePoint);

      /**
       * Return all poseables that intersect with a circle
       * centered at the given point with the given radius
       * @param worldSpaceCenter The world space point representing
       * the center
       * @param worldSpaceRadius The world space radius for the
       * circle
       * @return List of poseables that collide with given circle
       */
      public List<Poseable> getPoseablesCollidingWith
            (Vector2 worldSpaceCenter, float worldSpaceRadius);

      /**
       * Update the game model.
       * @param gameTime The elapsed time since the last call of
       * tick
       */
      public void tick(int gameTime);

      /**
       * Draw the game
       * @param gameTime The elapsed time since the last call of
       * draw
       */
      public void draw(int gameTime);

      /**
       * Translate from screen space to world space
       * @param screenSpacePoint The point in screenspace to
       *  translate
       * @return A vector2 representing the world space point
       */
      public Vector2 getWorldSpacePoint(Vector2 screenSpacePoint);

      /**
       * Translate from world space to screen space
       * @param worldSpacePoint The point in worldspace to
       * translate
```

```
         * @return A vector2 representing the screenspace point
         */
        public Vector2 getScreenSpacePoint(Vector2 worldSpacePoint);

        /**
         * Function called at AIPaint initialization,
         * before any calls to tick or draw are made.
         */
        public void aiPaintInit();

        /**
         * Return AIAgent at the given position
         * @param worldSpacePoint The position to test for an AIAgent
         * @return AIAgent if it is in that position, null if none
         */
        public AIAgent getAIAgentAt(Vector2 worldSpacePoint);

        /**
         * Return MouseListener this game will use
         * @return MouseListener this game will use
         */
        public MouseListener getMouseListener();

        /**
         * Return MouseMotionListener this game will use
         * @return MouseMotionListener this game will use
         */
        public MouseMotionListener getMouseMotionListener();

        /**
         * Return KeyListener this game will use
         * @return KeyListener this game will use
         */
        public KeyListener getKeyboardListener();

        /**
         * Return the WorldState representing the current state of
         * the game model
         * @return WorldState object representing the current state
         * of the game model
         */
        public WorldState getCurrentWorldState();
}

/**
 * Represents an object on whose state an AIPaint agent may make
 * decisions. This object's state may be posed during design time
 * using the AIPaint UI.
 */
public interface Poseable {
        /**
         * Change the position of this poseable to the given position
         * @param worldSpacePoint The new position the poseable will
         * have
         */
        public void moveTo(Vector2 worldSpacePoint);
```

```
    /**
     * Gets the current worldspace position of the poseable
     * @return The current worldspace position of the poseable
     */
    public Vector2 getPosition();

    /**
     * Gets the WorldFeature that maps to this
     * Poseable's state from the WorldState
     * @return WorldFeature that maps to this Poseable's state
     */
    public WorldFeature getFeature();

    /**
     * Gets the ValueType that represents this
     * Posable's state, mapped from the getFeature() method
     * @param <ValueType>
     * @return ValueType that represents this Poseable's state
     */
    public <ValueType extends Value> ValueType getFeatureValue();

    /**
     * Get the groups this poseable belongs to, for purposes of
     * generalization
     * @return Iterable that gives all groups this poseable
     * belongs to
     */
    public Iterable<Group> getGroups();

    /**
     * Get a vector2 representing the orientation or heading in
     * which this poseable is traveling
     * @return Vector2 representing the orientation of the
     * poseable
     */
    public Vector2 getOrientation();

    /**
     * Gets the WorldFeature that maps to this
     * Poseable's orientation from the WorldState
     * @return WorldFeature that maps to this Poseable's
     * orientation
     */
    public WorldFeature getOrientationFeature();

    /**
     * Gets the ValueType that represents this Poseable's
     * orientation, mapped from the getOrientationFeature() method
     * @param <ValueType>
     * @return ValueType that represents this Poseable's
     * orientation
     */
    public <ValueType extends Value> ValueType
          getOrientationFeatureValue();
}
```

```
/**
 * Represents a computation on a WorldState.
 */
public interface Attribute {
	/**
	 * Evaluate the computation on the current WorldState
	 * @param worldState The current worldState
	 * @return A value representing the Attribute computation
	 */
	Value evaluate(WorldState worldState);
}

/**
 * Provides Attributes capable of generalizing over particular
 * Groups.
 */
public interface HeuristicFactory {
	/**
	 * Returns an Iterable containing all Attributes that are
	 * capable of
	 * generalizing the given Group.
	 * @param group
	 * @return
	 */
	public Iterable<Attribute>
		getGeneralizingAttributes(Group group);
}

/**
 * Defines a factory for constructing SketchoStatement objects.
 */
public interface StatementFactory {
	/**
	 * Returns an instance of the SketchoStatement with the given
	 * name.
	 * @param statement
	 * @return
	 */
	public SketchoStatement getStatement(String statementName);

	/**
	 * Returns an instance of the SketchoStatement that produces
	 * the given Action.
	 * @param actionProduced
	 * @return
	 */
	public SketchoStatement getStatement(Action actionProduced);

	/**
	 * Returns an instance of the SketchoStatement that produces
	 * a decision on the given Attribute.
	 * @param attributeProduced
	 * @return
	 */
	public SketchoStatement
		getStatement(Attribute attributeProduced);
}

/**
 * Represents an agent for whom AIPaint may be used to build
 * behavior.
 */
public interface AIAgent extends Poseable {
      /**
       * Returns the controller that stores this agent's behavior.
       * @return
       */
      public AIController getAIController();
}

/**
 * Represents a low-level action that an agent may execute.
 */
public interface Action
{
      /**
       * Executes this action on the given agent.
       * @param agent
       * @return
       */
      public Action execute(AIAgent agent);
}
```

References

Anthony, L. and Wobbrock, J. O. 2010. A Lightweight Multistroke Recognizer for User Interface Prototypes. In Proceedings of Graphics Interface 2010, 245-252. Toronto, Ontario: Canadian Information Processing Society.

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*, 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).