PaintBoard – Prototyping Interactive Character Behaviors by Digitally Painting Storyboards

Daniel J. Rea
University of Manitoba
daniel.rea@cs.umanitoba.ca

Takeo Igarashi
The University of Tokyo
takeo@acm.org

James E. Young
University of Manitoba
young@cs.umanitoba.ca

ABSTRACT
The creation of interactive computer-controlled characters in interactive media is a challenging and multi-faceted task requiring the skills and effort of professionals from many fields. This work addresses authoring the interactive aspect of these characters' behaviors – how characters act automatically in response to a dynamic user-controlled character. We present PaintBoard, a system that enables users to prototype and test discrete, real-time, interactive movements in a 2D grid environment simply by digitally painting a storyboard. We designed and developed a novel authoring technique for creating behaviors (painting storyboards) and a novel algorithm based on machine-learning, that analyzes a storyboard to create a behavior that works beyond situations provided in the input storyboard. We conducted two exploratory studies that grounded the prototype design, and present the results of a proof-of-concept workshop with game developers. Finally, we performed a comparison of machine learning algorithms' performance on our storyboard data.

Author Keywords
End-user programming; interactive systems; sketch interface; prototyping; interface design; machine learning

ACM Classification Keywords
H.5.2. User interfaces: Prototyping, User-centered design

General Terms
Design; Human Factors

INTRODUCTION
Computer controlled characters are an integral component of modern video games and other interactive media. The creation of these characters is a difficult task that, at the professional level, can demand a broad range of specialized and highly-skilled collaborating individuals, including artists for creating 3D models and animations, writers and voice actors for dialogue, and a range of programmers to implement artificial intelligence and system logic. This is particularly challenging when the computer-controlled characters are highly interactive—when the characters must, in real-time, assess and interact appropriately to dynamic input from users and their environment. These highly-dynamic *interactive behaviors* can demand programming expertise and significant amounts of time. For example, in a role playing game, a designer may want a computer-controlled thief character to "sneak": avoid the user-controlled character when they are nearby while simultaneously approaching a treasure box, all without being seen. Such behaviors usually require the logical definition of multiple conditions based on user activity, and all details of the behavior that occur in each condition.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
HAI '14, October 29 - 31 2014, Tsukuba, Japan
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3035-0/14/10…$15.00.
http://dx.doi.org/10.1145/2658861.2658886

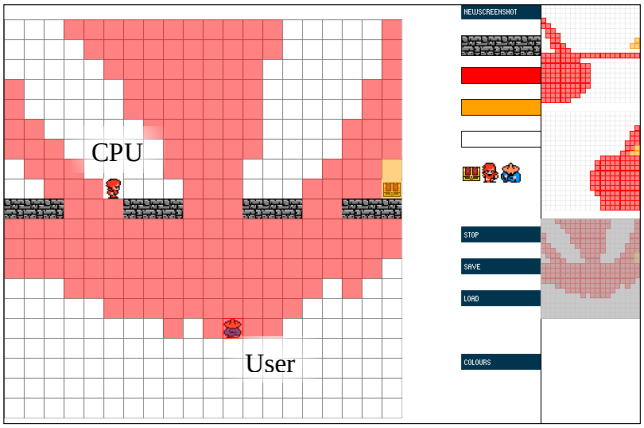

Figure 1: A participant sketches a storyboard to show that a computer-controlled interactive character (CPU) should approach yellow squares (the treasure) while staying out of the red squares (user controlled character's sight). PaintBoard extrapolates and generates the interactive behavior of sneaking to the treasure when the user's character is not looking.

Researchers have proposed various methods to reduce the amount of expertise and time required for creating content for interactive systems. This includes enabling people to create 3D models simply by sketching in 2D [12], to author advanced animations through simple mouse gestures [13], or to create complex interactive stories through point-and-click visual logic programming [15]. Simplifying the creative process further provides experts and non-experts alike with prototyping tools for quickly testing, visualizing, and sharing their ideas [18]. We extend this body of work with PaintBoard: a simple and visual prototyping method that aims to reduce the effort and skill required for the creation of interactive behaviors for computer controlled characters.

Paper sketching and other related low-fidelity techniques (such as storyboarding) are low-cost, fast, easy-to-use tools that support creativity and exploration [4,14] by assisting and

enabling rapid iteration of ideas, and by providing immediate visual feedback of those ideas [4,14,18]. They also enable and inherently support communication with others, as well as storytelling [11]. Because of this, low-fidelity techniques are part of standard toolkits across a broad range of fields including human-computer interaction [14], film [10], and animation [13]. PaintBoard leverages the benefits of these exploration techniques by enabling people to create behaviors by digitally painting rough ideas on virtual storyboards, similar to sketching, and by generating results that people can interact with, test, refine, and show to others.

In this paper we present an initial PaintBoard prototype that provides a novel storyboarding interface for painting interactive behaviors. We developed a feature set for representing the behaviors, and employed machine-learning using these features to generate real-time interactive behaviors from the user-drawn storyboards. To inform both the PaintBoard interface and algorithm design, we conducted interviews with industry professionals and analyzed results from a behavior-programming workshop with experienced programmers. We present the results of a proof-of-concept, hands-on PaintBoard workshop and an evaluation of our algorithm by comparing its performance in alternative configurations.

RELATED WORK

Existing research that simplifies the creation of interactive characters and systems has aimed to reduce the programming requirements typical to such tasks. One example uses accessible drag-and-drop textual representations of game objects combined with programming-like elements [15] to create story-based interactive worlds (their focus was not on interactive behavior movement). Another powerful work allows authors to explore complex interactive story narratives by specifying details about the characters and the world they act in [16]. Others enable users to build up an interactive behavior with gesture grammars and state machine logic [17]. PaintBoard focuses on prototyping, and because linear, detail-oriented thinking can hinder prototyping and exploration [20], our work aims to avoid requiring users to explain their behavior logically and enable them to describe it in a visual, story-like, less detail-oriented way.

Programming by demonstration, where a designer can author a behavior by simply providing a performance demonstration, is a common technique for authoring interactive behaviors. Although this is well established in animation [6,13] and robotics [19], most of this work has been for the creation of *static* behaviors without an interactive element. Interactive work has focused on, for example, learning reactive body language from motion-capture data with user-controlled parameters [7], or well-defined sequences of actions that fit into a state-machine model [9,17]. PaintBoard extends this work by targeting movement in response to unpredictable users in continuous real-time interaction throughout a 2D environment. Further, programming by demonstration can often require large numbers of repetitive demonstrations (e.g. [8]), real world data of many types of movement (e.g. [8]), or still uses programming in the process (e.g. [21]). To enable its use as a low cost rapid-prototyping tool, PaintBoard can work with as little as one example.

Perhaps closest to our work is the Puppet Master programing-by-demonstration project [22], which enables authors to rapidly prototype interactive animation or robotic motion behaviors similar to the ones targeted by PaintBoard. While Puppet Master emphasizes interactive movement "style" of two interacting characters, PaintBoard builds upon their results to cover multi-part behaviors (e.g., hide when seen, get some treasure when guards are not looking), and enable characters to interact with the environment (e.g., walls, important objects such as treasure chests). Further, Puppet Master's evaluations indicated that users had difficulty envisioning the result from their performance demonstration due to the mental load from Puppet Master requiring the user to successfully author the whole behavior in real-time in one attempt; To avoid such issues, we extend Puppet Master's approach by enabling more complex interactive behavior authoring in a visual way with frame-by-frame storyboards.

Low-fidelity prototyping techniques such as sketching have been used to successfully simplify other forms of digital content design, for example, by using 2D sketches to create 3D character models [12], or to design and implement user interfaces [14]. These tools are accessible as design and prototyping tools to both professional and amateur designers. We follow this approach by applying rough painting to simplify the creation of interactive motion behaviors

EXPLORATORY INVESTIGATION

To inform our interface and algorithm design, we conducted two exploratory studies. First, we performed semi-structured interviews with video game designers and developers to uncover common problems faced and workflows used during the creation process. To inform the design of our behavior generation algorithm, we conducted a second investigation (programmer study) with 26 undergraduate programmers, where we explored the range of interactive behaviors people may author, and analyzed implementations (computer code) to extract strategies and techniques used to implement them.

Interviews with Industry

We conducted four, one-hour semi-structured interviews with professional game designers and developers. Questions we asked include "How are certain interactive behaviors difficult to create, specifically because of interactions with the player?" and "How do you create these interactive behaviors?" We recorded the interviews and qualitatively analyzed transcripts through open coding to identify key themes.

Participants reported spending significant time planning behavior implementations due to the even higher time cost of actually implementing them with programming. In addition, participants heavily relied on experimentation and iterative prototyping (writing programs and observing results). This grounds our initial rapid prototyping motivation in the needs of real users. Further, participants reported having difficulty

communicating and understanding how an interactive behavior should look. We saw this reported by both technical developers as well as artistic designers. As visual tools improve communication [4,11], one aim of PaintBoard was leveraging painting's visual nature as the main interaction metaphor.

Programmer Study: Analysis of Implementations

To investigate programming strategies that may be useful for generating behaviors, we conducted a programming workshop to explore methods used by developers to implement interactive behaviors. We asked 26 fourth-year undergraduate Computer Science students in a Human-Computer Interaction class to program a set of behaviors; such participants have the skills to work in the video game industry. We used a medieval-theme (common within the role-playing video game genre) as a representative scenario. Participants were provided a simple graphical game board (that looked similar to PaintBoard, e.g., Figure 1) and Java API, and were tasked with creating three behaviors each. They were encouraged to develop their own behaviors, and were described "follow," "protect treasure," and "escape" as examples.

We received 78 unique behavior implementations that we categorized into 19 distinct types. The three most common of these were our suggested "follow the user" (24 participants), "protect treasure" (18), and "escape from the user" (13) behaviors. The remaining behaviors had less overlap (16 types over 23 implementations). Common behavior types, however had significant variation, for example, some "follow the user" implementations would stay close behind the user, others would walk side by side, and yet others would follow from a distance. Thus behaviors are envisioned differently by different authors; PaintBoard will need to accommodate not only a variety of behaviors, but allow for variations of those behaviors.

Our post-workshop analysis of the developers' behavior implementations illuminated strategies that participants used to define their behaviors. Across behaviors, we found that participants consistently leveraged a small set of commonly calculated quantities, such as the characters' visibility and relative positions, to decide on how the computer character should interact with the user; we distilled these into a set of features that our algorithm used to analyze painted storyboard input (detailed in Section 4.3). Additionally, programmers commonly specified points of interest, for example, a treasure chest to "guard," a "hideout" to run to, or even staying in "close proximity to the user." Thus PaintBoard enables users to define such goal areas with the interface.

PAINTBOARD PROTOTYPE

Our PaintBoard prototype provides a sandbox setting that enables people to author a behavior by digitally painting on a screen. Users paint a storyboard consisting of static *snapshot* panels that each convey the behavior in a specific situation. The storyboard represents a full behavior, and is used as input to the PaintBoard algorithm to generate the interactive result. The interface and logic was programmed in Java using Processing[1] and the ControlP5 library[2] was used for the GUI.

User Interface

The sandbox area of the interface is a 20x20 2D character movement grid where the user constructs snapshots from an overhead view (Figure 2a). Users can drag objects (from Figure 2d), including both the computer and player characters (maximum one of each), points of interest (treasure chest, maximum one), and environment (walls, from Figure 2c), onto the grid, and position them as desired.

Users can select a color from the palette (Figure 2c) and then paint on the grid by clicking and dragging the mouse; cells can only have one color at a time. Red paint denotes areas where the computer character should *not* go, for example, when painting a "sneak" behavior all grid cells that the user character can see should be red because a sneaking character does not want to be seen. Gold paint indicates *goal* areas: where the computer character wants to go (Figure 3). This differs from points of interest in that it is based on the user and computer characters' current configuration (e.g., "behind the user") and as such are more dynamic than stationary points of interest. Unpainted (white) squares are neutral and the character neither avoids nor tends toward them. In other words, the computer character should try to go toward gold areas, while passing through uncolored areas and avoiding red ones. Thus the PaintBoard interface addresses the features uncovered during our programmer study: visibility can be defined simply by painting where a character can see (Figure 3), relative position is defined by how the user and computer characters are placed, and the gold paint and points of interest enable the author to designate goals.

PaintBoard Interaction Flow

To facilitate rapid and iterative prototyping, PaintBoard enables users to quickly create behaviors, and easily test and modify them. While painting, users can test their behavior by pressing a single button ("play"), and the system instantly

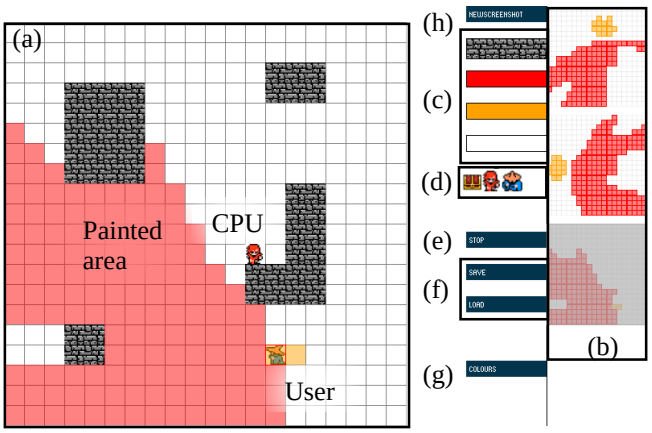

Figure 2: The PaintBoard interface: (a) sandbox area, (b) storyboard snapshots, (c) paint palette, (d) point of interest (chest) and characters, (e) play and pause, (f) save and load behavior, (g) debug mode, (h) new snapshot.

[1] http://www.processing.org/

[2] http://www.sojamo.de/libraries/controlP5/

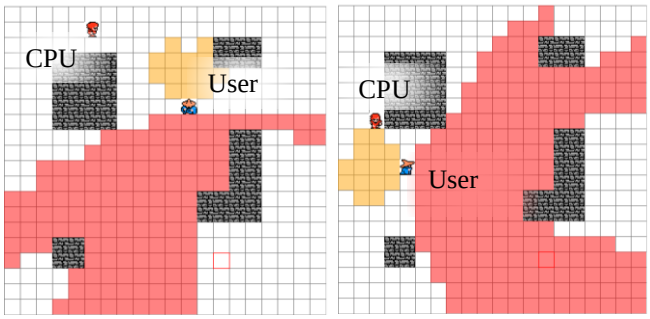

Figure 3: A sample two-part behavior storyboard of a character that sneaks up on the user. In this case, the computer character should not enter the sight of the user character, (red squares) and should stay close to and behind the user (gold).

compiles the behavior and generates a result that the users can interact with using the keyboard controls (arrow keys). At any time, the author can press "stop," edit any of the snapshots in the storyboard, or create a new snapshot to modify the behavior. While testing, if the computer character moves in an unintended way, the user can capture the real-time scenario as a new snapshot, and paint it.

PaintBoard has a debug mode that is identical to testing ("play" mode) except that it provides real-time visual feedback regarding what the character is trying to do. Based on the storyboard input, PaintBoard will display red and gold squares to indicate where – in the given situation – it believes the character should and should not go. For example, given the snapshots in Figure 3 as the storyboard input, Figure 2 is the debug-mode output, a visual representation of what PaintBoard has learned.

Enabling rapid and iterative prototyping of ideas was a major interface priority, as it has been shown to aid creativity and exploration [14,18,20]; PaintBoard enables users to quickly create, modify, and delete snapshots, as well as test the interactive results without requiring any programming or logic definition. With PaintBoard, authors create behaviors *in-situ* (e.g., [12,17]): they create in the same environment where the behavior will be used. This enables authors to bypass conceptual translations required when moving from an authoring to a testing medium (such as moving from visual programming to a game), helping visualize the final result [2].

Algorithm

The PaintBoard algorithm must analyze a painted storyboard and generate an interactive character that matches the qualities given in the storyboard.

Given an input storyboard and a new, live situation, PaintBoard first generates a new *synthetic snapshot* to have similar characteristics to the input examples (e.g. red paint in user's line-of-sight). For example, given the two snapshots as input from Figure 3, PaintBoard generated the painting in Figure 2a to have similar characteristics. Then, PaintBoard uses this synthetic snapshot to determine the interactive character's next move. After moving, the process is repeated, generating a new synthetic snapshot for each new situation.

Generating Synthetic Snapshots

We employ a Support Vector Machine (SVM) [5] classifier to iterate over all cells of the unpainted snapshot and label them as red, gold, or unpainted, creating the synthetic snapshot. We selected an SVM as a standard, fast classifier and used the LibSVM library [5] with its default settings.

To train the SVM, we calculate a feature vector for every cell in every snapshot, and label it with the color painted by the author (red, gold, unpainted). Thus, the SVM would ideally label cells of the synthetic snapshot to match characteristics (the features) of the training data (as show in Figure 2).

A challenge of using machine learning is to select representative features that capture the appropriate characteristics of the scene. This is non-trivial, and we developed our own domain-specific features given the lack of prior work; this problem and challenge is common in machine learning (e.g., [7,22]). PaintBoard uses the features identified in our programmer study, with the final set selected informally through experimentation. We call these *state features*, detailed below (Figure 4):

Cell Position Relative to the User in the Screen's Coordinate System. For example, in Figure 4 the bold cell is two to the left and three above the user. This captures absolute relation to the user (e.g., stay to the left side of the environment).

Cell Position Relative to the User's Orientation. For example, in Figure 4 the bold cell is two cells in front of and 3 cells to the right of the user character. This captures position from the user's point of view (e.g., stay behind them).

Cell Position in a Coordinate System Rooted at the User and Oriented to the Point of Interest. For example, in Figure 4 the bold cell is 2.6 cells behind and 2.5 to the left of the user and chest. This captures the context of the point of interest (e.g., do not go between the user and the chest). This is not used when there is no point of interest.

Visibility: we cast rays from the user to the cell and its neighbors to calculate visibility, with those blocked by walls not counted. For example, the bold square in Figure 4 has visibility 0.6 (6/9). This captures line of sight information (e.g.,

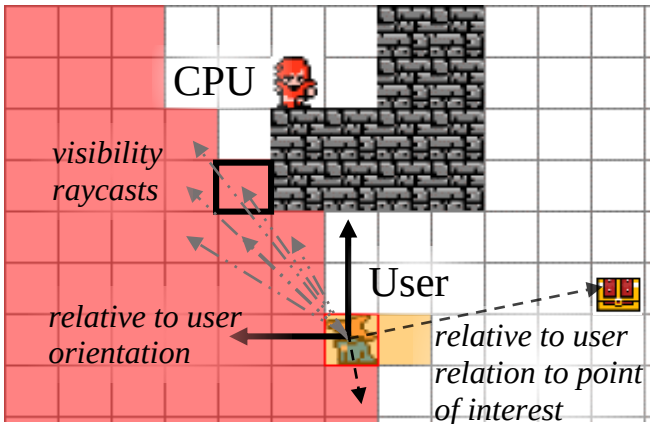

Figure 4: How state features are calculated for the bolded cell.

how visible cells are to the user), and the non-binary classification enables the computer character to capture the difference between being partially and fully seen.

Euclidean Distances from the Cell to the User and Point of Interest. For example, in Figure 4 the bold square is 3.6 and 7.3 from the user and point of interest respectively. This helps emphasize proximity (e.g., how close to be to the user).

These features form a multi-dimensional vector for each cell which is labeled with the color painted by the author. We use all vectors from all snapshots in a storyboard to train the SVM. During interaction, the features are calculated for each cell in real time and the SVM is used to classify (paint) them to generate the synthetic snapshot.

Using a Synthetic Snapshot to Generate the Behavior

After generating the synthetic snapshot, the computer character moves toward the closest goal space (gold paint), while not walking through walls, and avoiding red spaces if possible. The algorithm does a breadth-first search spiraling out from the computer character (we saw this in our programmer study), where walls are considered impassable. Red cells may not be completely avoidable, for example if all other paths are blocked. We address this by giving them a penalty distance of four. For the case when the character is stuck with large red areas between it and the goal, we added a path-length threshold of 20 so the character simply would wait in safety rather than traversing large red areas. Both of these quantities were selected through experimentation.

PAINTBOARD WORKSHOP

We conducted a proof-of-concept workshop to explore reactions to our PaintBoard approach and interaction design by potential end-users. Formal, targeted evaluations (e.g., investigating how PaintBoard can integrate into a designer's workflow) remain important future work.

We recruited five professional and hobbyist game developers for the 1.5 hour workshop, where they received a 15 minute tutorial on how to use PaintBoard to prototype interactive behaviors. Afterwards, they were given one hour to freely create any behaviors they wished (using the same medieval theme as above), and were given a 15 minute questionnaire at the end. Although each person worked independently, the atmosphere was friendly and collaborative, and people were having spontaneous discussions about their experiences. Researchers took notes throughout the workshop

Participants were asked to save their storyboards (through the PaintBoard functionality) for later inspection. In addition, we performed broad qualitative analysis on the notes and questionnaire answers in order to identify themes and insights of our participants' experiences on topics such as viability of painting as a behavior authoring technique, or ways that PaintBoard could be leveraged in real-world situations.

Results

Overall, participants were able to use PaintBoard to quickly and successfully prototype a range of interactive behaviors such as "follow the user," "hide," "obstruct the user," "guard an area," and "sneak to treasure." This was achieved in spite of minimal training, lending support to our painting and storyboarding approach and implementation.

Participants reported feeling that PaintBoard would be useful for planning and prototyping ideas:

> *In its current state, could be handy for prototyping and visualizing scenarios. - P3*
>
> *I would use this as a prototyping tool to make quick behaviors that I would then implement with code - P2*

and for communicating with others:

> *Easy to visually show others simple behavior that can be expanded to more complex situations. - P5*

Some noted that it may be useful for team members with less technical expertise:

> *I'm not sure if it'll be useful in my workflow (yet), but I think it'll be great for designers - P1*

Although this is far from rigorous proof, the previous two quotes highlight our motivation of making a tool that facilitates communication between designers and developers: the visual and interactive nature is important as the resulting behavior prototypes can be shared and discussed with coworkers. As PaintBoard requires no coding knowledge, it enables two-way communication as both designers and developers can modify behavior prototypes to enhance discussions.

Participants praised the benefits of PaintBoard's iterative nature, noting that it matches their existing workflows:

> *I like the iterative design process. Games tend to follow on iterative design, so this fits nicely. - P1*

Even though our participants were experienced programmers, they were very receptive to the use of painting in the behavior design process:

> *The abstraction of the concepts is very easy to understand … as well as the ability to alter states during play, and ability to watch the goal and avoid state change - P4*

All participants also felt the performance of the test mode was reasonable. However, some did show concern over PaintBoard's ability to scale up to more complex behaviors:

> *It's a bit hard to convey a behavior sometimes, but maybe that doesn't need to be a goal. It seems to work with simpler behaviors and I think it can be used as such usefully - P1*

There were several examples where the painted storyboard was very clear and descriptive from a person's perspective, but the resulting behaviors were not generated successfully. While this is a difficulty with the current learning algorithm, we believe that this is a success for the painting interface: it illustrates the ability to represent and communicate a desired result through our storyboards. See Figure 5, a storyboard produced by a participant in our workshop: it has easy-to-understand snapshots of specific behavior aspects and the overall storyboard clearly describes a complete behavior, but the generated behavior usually predicted only unpainted cells.

In addition to reflecting on the potential benefits of PaintBoard, participants described specific functionality that they believed could improve PaintBoard. For example, participants requested the addition of story branches, where a condition indicated in a snapshot may lead to a new set of snapshots. This could fit within the storyboard interaction, but would require new algorithmic solutions. Participants also suggested adding the ability to weigh painted cells, where some are more important than others (e.g. prefer not being seen over reaching the treasure), or the ability to make hard rules about the environment, for example, to mark specific squares in the environment which should always be avoided. While these would give more creative power and control to a PaintBoard user, such features should be added with careful consideration of the speed and simplicity of PaintBoard's interaction flow, else they may slow down PaintBoard's rapid and iterative nature.

COMPARISON OF CLASSIFIERS AND FEATURE SETS ON STORYBOARD DATA

To understand how our choice of learning algorithm and training features were linked to some of our users' behavior generation problems observed in our workshop, we analyzed the performance of different variations of our algorithm. We modified our algorithm on two dimensions: classifier, and feature set used to train the classifier. This analysis has three components: we developed a dataset and accuracy metric that could be used to test a given configuration, we tested the accuracy of each classification algorithm, and using the best performing classifier we greedily selected state features to form a possible variant feature set for training.

To build a dataset for training and testing classifiers, we recruited developers and had them use PaintBoard to paint behaviors. Participants were 4th year and graduate computer science students. They were given an explanation of how PaintBoard worked, including a demonstration of painting a "follow" behavior. Each participant was asked to paint three behaviors: escape from the user character, sneak up to the user character, and protect a treasure from the user character. For each behavior, the participant was instructed to create 10 different storyboard examples with each example fully defining a behavior and containing one or more snapshots. Thus, there were 30 example storyboards per user.

Comparison of Classifiers

We compared the performance of five different machine-learning classifiers for our behavior generation algorithm: SVM with a radial basis kernel (which we used in our initial PaintBoard implementation), SVM with a polynomial kernel,

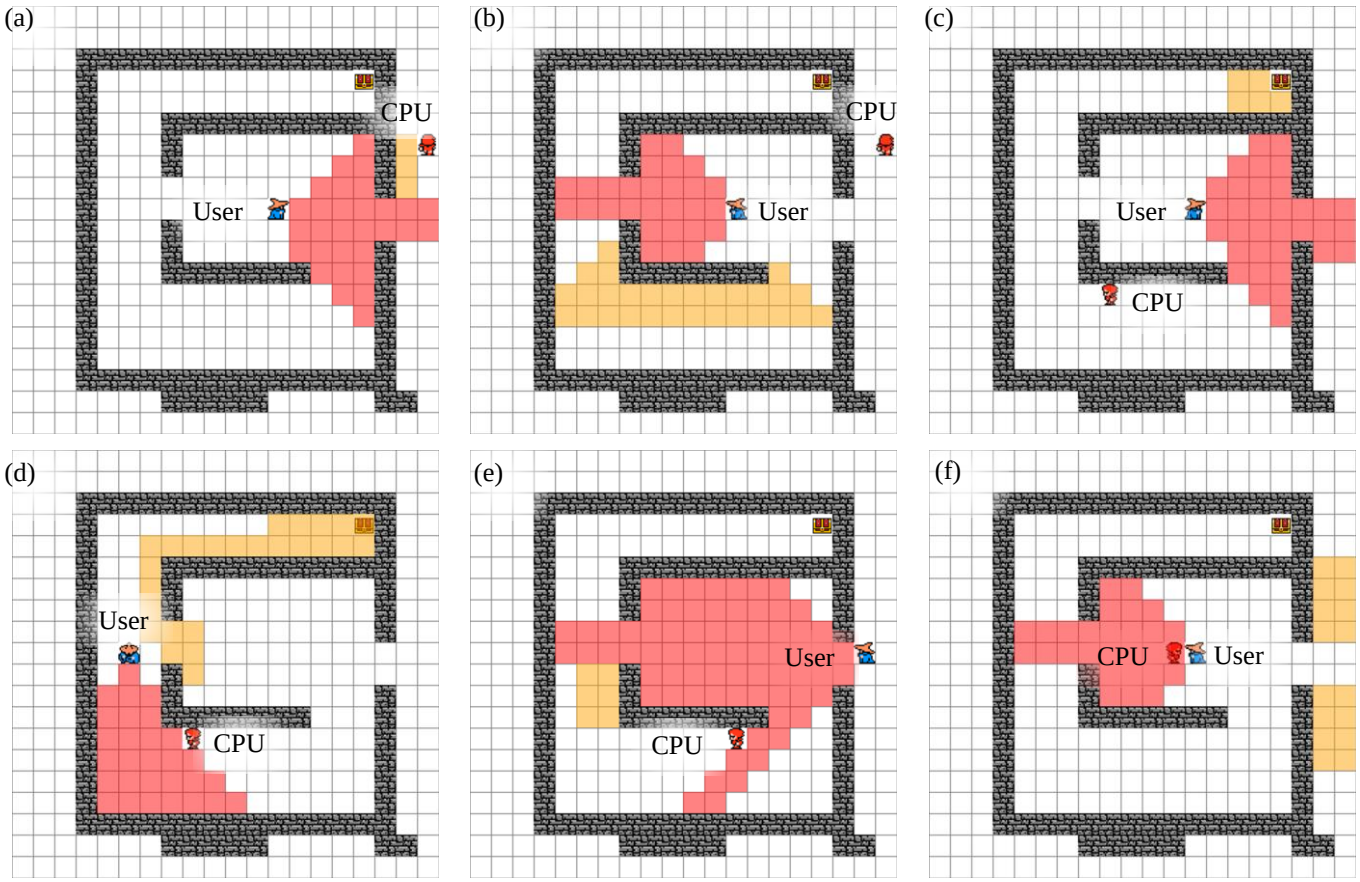

Figure 5: A storyboard authored by a participant during our workshop, showing how a computer character should sneak around a user to get treasure. (a) hide by the only entrance to the room (b) when the user is not looking, sneak into the room and stay out of sight (c) when the user is not looking at the inner hallway, run to the treasure (d) if the user is in the hallway, sneak around the other way (e) get as close to the treasure as possible without being seen, and (f) if spotted by the user, run out of the room.

K-Nearest Neighbors, Random Forest, and Naïve Bayes classifier. We used the implementations of these algorithms provided by the Java Machine Learning Library [1]. The two SVMs used the LibSVM default parameters; k was chosen as 5 for K-Nearest Neighbors as it had similar performance to higher values (e.g. k=10) with faster run-time; the tree count for the random forest was set to 100, based on the evidence suggesting random forests do not often overfit [3].

For each algorithm, we performed cross-validation to better understand its predictive accuracy with PaintBoard. For one participant's behavior, we trained PaintBoard with only one of the provided 10 storyboard examples: our target use case is rapid prototyping and, ideally, a user will paint minimal data and test in many situations. For each of the remaining examples, we generate a synthetic snapshot for the same situation and calculate accuracy as the percent of true positives (direct cell matches) between the two. This is done for each example, training a new classifier each storyboard and averaging the accuracies together. The accuracy of an algorithm is the mean accuracy over all participants and behaviors.

Results

We present accuracy of each algorithm in Figure 6.There was a main effect of the algorithm on the accuracy of the synthetic snapshot ($F(4, 120) = 11.9$, $p < .001$, $\eta^2 = .284$). Post-hoc tests (with Bonferroni correction) revealed that the radial basis function SVM performed better than Naïve Bayes ($p<.001$), and polynomial SVM ($p<.05$). All other comparisons were not significant.

The behavior type impacted the accuracy of the synthetic snapshot ($F(2, 120) = 8.0$, $p=.001$, $\eta^2 = .117$). Post-hoc tests (with Bonferroni correction) revealed that "sneak" was more accurate than "escape" or "protect" (10% more) $p<.005$. There was no interaction effect between algorithm and behavior type on accuracy.

One problem with interpreting the above accuracy results is that, in our data, a cell is much more likely to be unpainted (clear) than painted (red or gold); this biases classifiers to give us high accuracy for unpainted cells while possibly lowering the accuracy for the other colors. To provide insight we present a confusion matrix for the radial basis SVM (Figure 7), showing the average accuracy across all participants and behaviors as a percentage. Each entry can be read as "[entry value] percent of all author-painted [column] cells in the test data were predicted to be [row] by PaintBoard." For example, we can see that, across all participants and behaviors, 76% of gold painted cells were predicted to be unpainted in the synthetic snapshots. In our case, a possible solution is to carefully balance the data fed into the SVM by clustering the disproportionately large number of uncolored squares to a representative subset is similar in size to the other colors. Despite this limitation we highlight that labelling a square as unpainted is as equally important as painting it, and the results were robust enough for our workshop.

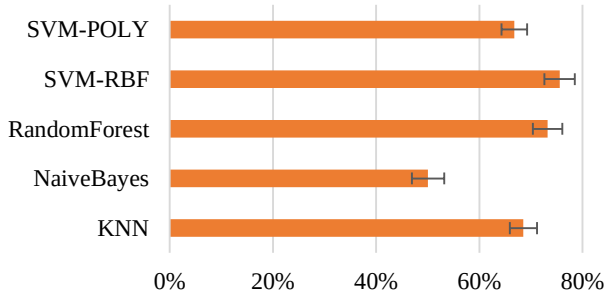

Figure 6: Accuracy of each algorithm for our dataset. Error bars are standard error. SVM polynomial kernel, SVM radial basis function kernel, Random Forest, Naïve Bayes, and K-Nearest Neighbors.

	SVM-RBF	*red*	*unpainted*	*gold*
	red	0.25	0.09	0.09
	unpainted	0.70	0.85	0.76
	gold	0.05	0.06	0.15

Figure 7: The confusion matrix for the radial basis function SVM. Columns are input labels, and rows are output labels.

Although our current implementation is sufficiently fast for interactive results, we analyzed execution time as a significantly faster algorithm could be important for future work. There was a main effect of the algorithm execution time per participant per behavior ($F(4, 42) = 19.4$, $p<.001$, $\eta^2 =.649$). Post-hoc tests (with Bonferroni correction) revealed that all algorithms ran at least 650% faster than polynomial SVM, $p<.001$. No other effects were found.

Evaluation of State Features

We explored variations of our feature set in order to explore if any of our features were poorly chosen. Using a naïve greedy selection approach using the radial basis kernel SVM, we first measured the accuracy (as above) with each state feature on its own, picked the single feature with the highest accuracy, and iteratively added features in the same fashion. We found no significant improvement of accuracy over our existing feature set as described earlier.

LIMITATIONS & FUTURE WORK

To expand the behavior vocabulary, PaintBoard should be extended to incorporate other state features such as the movement speed of the character. In addition, we should explore if PaintBoard should consider the order of the snapshots in the storyboard, as storyboards are inherently chronological. Along similar lines, it may be useful to explore how PaintBoard could use layers (e.g., as used in Adobe Photoshop). This may enable users to separate varying aspects of what they are authoring, and may be useful for representing speed, or other features such as character orientation, without cluttering the snapshots.

Our current studies served as a proof-of-concept of painting storyboards as a behavior generation tool, but targeted follow up studies with more rigorous evaluations need to be conducted. For example, we will conduct a follow-up study where developers and non-technical designers work together

to create a behavior, and use PaintBoard as the prototyping and communication medium. For future algorithmic work, investigating new state features may greatly improve accuracy. For evaluating behaviors, we relied on user self-reporting and naïve accuracy metrics; improving the behavior validation method is important future work.

CONCLUSION

This paper detailed PaintBoard: a novel interaction and algorithmic technique for prototyping interactive character behaviors by painting and storyboarding. We presented results from exploratory interviews and a programmer study, which informed our interface design and algorithm development. PaintBoard's algorithm was based on machine learning, and can generate real-time interactive behaviors based solely on a few painted examples. We presented a feature set (state features) that can represent important characteristics of paired interactive behaviors. Our proof-of-concept workshop highlighted the usability of PaintBoard's storyboard painting approach, showing how developers, with minimal training, can successfully and quickly prototype behaviors. Finally, we gave insight into the PaintBoard's algorithm with a comparative study. Overall, we believe that PaintBoard is a clear proof-of-concept for the approach of authoring interactive behaviors through visual painting and storyboarding.

ACKNOWLEDGMENTS

We would like to thank NSERC and JSPS for providing funding for this research.

REFERENCES

1. Abeel, T., de Peer, Y. V., and Saeys, Y. Java-ML: A Machine Learning Library. *Journal of Machine Learning Research 10*, (2009), 931–934.
2. Beyer, H. and Holtzblatt, K. *Contextual design: defining customer-centered systems*. Elsevier, 1997.
3. Breiman, L. Random Forests. *Machine learning 45*, 1 (2001), 5–32.
4. Buxton, B. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc. 2007.
5. Chang, C.-C. and Lin, C.-J. LIBSVM. *ACM Transactions on Intelligent Systems and Technology 2*, 3 (2011), 1–27.
6. Dontcheva, M., Yngve, G., and Popović, Z. Layered acting for character animation. *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*, (2003), 409.
7. Förger, K., Takala, T., and Pugliese, R. Authoring Rules for Bodily Interaction: From Example Clips to Continuous Motions. *Intelligent Virtual Agents*, (2012), 341–354.
8. Forte, D., Gams, A., Morimoto, J., and Ude, A. On-line motion synthesis and adaptation using a trajectory database. *Robotics and Autonomous Systems 60*, 10 (2012), 1327–1339.
9. Gebhard, P., Kipp, M., Klesen, M., and Rist, T. Authoring scenes for adaptive, interactive performances. *Autonomous agents and multiagent systems*, (2003), 725.
10. Goldman, D.B., Curless, B., Salesin, D., and Seitz, S.M. Schematic storyboarding for video visualization and editing. *ACM Transactions on Graphics 25*, 3 (2006), 862.
11. Greenberg, S., Carpendale, S., Marquardt, N., and Buxton, B. The narrative storyboard: telling a story about use and context over time. *interactions 19*, 1 (2012), 64–69.
12. Igarashi, T., Matsuoka, S., and Tanaka, H. Teddy: a sketching interface for 3D freeform design. *SIGGRAPH*, ACM Press (1999), 409–416.
13. Igarashi, T., Moscovich, T., and Hughes, J.F. Spatial keyframing for performance-driven animation. *ACM SIGGRAPH/SCA '05*, ACM Press (2005), 107.
14. Landay, J.A. and Myers, B.A. Interactive sketching for the early stages of user interface design. *SIGCHI*, ACM Press (1995), 43–50.
15. McNaughton, M; Cutumisu, M; Szafron, D; Schaeffer, J; Redford, J; Parker, D. ScriptEase : Generative Design Patterns for Computer Role-Playing Games. *Automated software engineering*, (2004), 386–387.
16. Pizzi, D. and Cavazza, M. From Debugging to Authoring : Adapting Productivity Tools to Narrative Content Description. *Lecture Notes in Computer Science 5334*, (2008), 285–296.
17. Shen, E.Y. and Chen, B. Toward gesture-based behavior authoring. *International 2005 Computer Graphics*, (2005), 59–65.
18. Shneiderman, B. Creativity support tools: accelerating discovery and innovation. *Communications of the ACM 50*, 12 (2007), 20–32.
19. Suay, H.B., Toris, R., and Chernova, S. A Practical Comparison of Three Robot Learning from Demonstration Algorithm. *International Journal of Social Robotics 4*, 4 (2012), 319–330.
20. Terry, M. and Mynatt, E.D. Recognizing creative needs in user interface design. *Creativity & cognition - C&C '02*, (2002), 38–44.
21. Wolber, D. Pavlov: Programming by stimulus-response demonstration. *SIGCHI*, (1996), 252–259.
22. Young, J.E., Sharlin, E., and Igarashi, T. Teaching Robots Style: Designing and Evaluating Style-by-Demonstration for Interactive Robotic Locomotion. *Human–Computer Interaction 28*, 5 (2013), 379–416.