

TakeToons: Script-driven Performance Animation

Hariharan Subramonyam^{1,2}, Wilmot Li², Eytan Adar^{1,2}, Mira Dontcheva²

¹University of Michigan,
Ann Arbor, MI
{harihars,eadar}@umich.edu

²Adobe Research
Seattle, WA
{wilmotli,mirad}@adobe.com

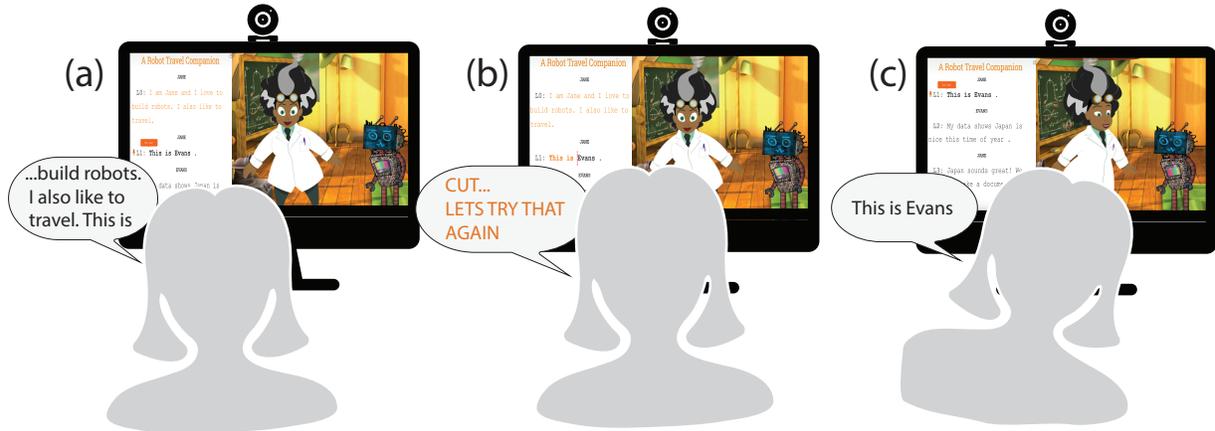


Figure 1. The performer reads a line from the teleprompter (a), which highlights the text and automatically scrolls to the next line. The performer forgot to move her head to the right, so she says, ‘cut! let’s try that again’ (b) to re-record the line (c).

ABSTRACT

Performance animation is an expressive method for animating characters through human performance. However, character motion is only one part of creating animated stories. The typical workflow also involves writing a script, coordinating actors, and editing recorded performances. In most cases, these steps are done in isolation with separate tools, which introduces friction and hinders iteration. We propose *TakeToons*, a script-driven approach that allows authors to annotate standard scripts with relevant animation events like character actions, camera positions, and scene backgrounds. We compile this script into a *story model* that persists throughout the production process and provides a consistent structure for organizing and assembling recorded performances and propagating script or timing edits to existing recordings. *TakeToons* enables writing, performing and editing to happen in an integrated and interleaved manner that streamlines production and facilitates iteration. Informal feedback from professional animators suggests that our approach can benefit many existing workflows supporting individual authors and production teams with many different contributors.

Author Keywords

performance animation; animation script; speech;

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 ACM 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.

UIST '18, October 14–17, 2018, Berlin, Germany

© 2018 ACM. ISBN 978-1-4503-5948-1/18/10...\$15.00

DOI: <https://doi.org/10.1145/3242587.3242618>

INTRODUCTION

Advances in performance animation systems are making it easier to create animated stories. Instead of specifying keyframes, as in traditional animation workflows, actors can directly create animations by acting out their characters. This approach is now part of professional workflows where software translates physical motion and audio to character animation [1]. However, performing motions is not the only task in creating a story. The complete workflow involves writing a script, coordinating actors, and editing the collection of recorded performances together into a complete animation. Since these tasks are typically done one at a time, earlier steps in the process cannot benefit from the context of later stages. For example, the script writer may not know or be able to influence, exactly what a voice actor sounds like or how the animation will appear in a given scene. Additionally, any changes to the script or performance require most of the animation and editing work to be redone. These limitations increase the time and cost of creating and iterating on animated stories.

We propose an approach for creating animated stories that integrates and interleaves writing, performing and editing. The key idea is to encode dialogue, scene changes, voice performance, and motion in an integrated and unified representation, what we call a *story model*, that allows us to support interleaving of the three main tasks. For example, as users perform, the recordings are automatically assembled into an animated story, which provides a quick preview of the results and context for subsequent performances. Changes to the script can be interleaved with performing without losing prior work. By integrating and interleaving parts of the workflow, animators can create and edit animated stories more efficiently.

We demonstrate this approach in the context of an end-to-end animation authoring system, TakeToons. With TakeToons, the user starts by writing a script, the natural first step in the process. TakeToons scripts allow users to add markup that indicates when animated events (e.g., changes to a character’s pose or appearance, cuts to different camera angles) occur within the dialogue. As the user reads and acts out the script, TakeToons automatically generates an animation from the performance and edits the footage together.

To support this animation workflow, TakeToons includes several key contributions. The TakeToons *script language* extends standard screenplay markup [12] to allow the writer to specify dialogue and on-screen performance behaviors for characters, sets, camera, and sound. TakeToons parses a script into a *story model*, a flexible internal data structure that captures the relationships between different events in the story. This model supports generating complex animations with multiple characters and layered animation where multiple events happen at the same time (e.g., one character reacts while another speaks). The *story controller* interprets user performances and maps them onto the relevant parts of the story model. To control the recording process, performers can interact with a physical interface (e.g., keyboard, touchscreen). When performance makes it difficult to be near a computer (e.g., running or jumping), the performer can also ‘be the director’ by controlling recording through speech commands. To support these commands, the story controller uses the story model to determine which speech *is* and *is not* part of the scripted dialogue.

TakeToons guides performers through a *teleprompter interface* that advances as the user speaks and displays the resulting animation in a *stage view* that provides a live preview of the animated content (see Figure 2). A single performer can go through the script one line at a time doing voice and physical performance simultaneously. TakeToons will auto-trigger predefined animations that are annotated in the script creating a rough cut of the whole animation in one pass. The system also supports redoing recordings and layering animation on top of the previously recorded content. Beyond the flexibility for an individual performer, TakeToons supports collaborative recordings. Different people can record different parts, and TakeToons will assemble the recordings into a final animation. Collaborative recordings can be synchronous or asynchronous. In asynchronous situations, previous recordings can give actors context, as TakeToons plays the previous recordings while the actor performs.

We used TakeToons to generate three animated stories that include interactions between multiple characters, scene changes, cuts between camera angles, automatically triggered animation cycles, and both voice and facial performances. These results demonstrate the range of story configurations and animation effects that TakeToons supports. We also obtained informal feedback from professional animators, who noted the practical benefits of our proposed workflow. Though our current implementation focuses on animation, we discuss how TakeToons’ approach can be applied to other speech-driven scenarios including lectures and presentations.

RELATED WORK

Scripts and animation

Previous work has explored techniques that leverage domain knowledge to automatically generate animation from natural language [6, 17, 21, 28]. In contrast, TakeToons allows animation authors to define how performances and automatically triggered animation should be combined using familiar script formats [16]. While our current approach does not perform any semantic analysis on the script, techniques that infer properties like mood, intent, scene changes, and character motion from written text could complement our proposed workflow by proposing where to insert animation events into the script.

Script-based editing of video and audio is an emerging research area [3, 27]. While TakeToons is the first to show how script-based editing can be useful in animation settings, previous approaches may be useful extensions to support performance feedback [26], editing after the performance is done [3], and misalignment between the script and the performance [29].

Performance animation systems

In performance animation live performance of actors is translated to actions of computer-generated characters. As the actor acts, the character moves, making it possible to animate without specifying individual keyframes [30]. TakeToons incorporates several features of existing commercial and research performance animation systems. Previous work has explored techniques for animating a character’s face through speech [5, 9, 10, 15] and expression changes [7]. TakeToons leverages a commercial performance animation system [1] that supports speech-driven mouth animation (i.e., lip sync) and expression-driven facial animation. To support a broader range of animated effects, Willet et al. [35] present a user interface for triggering predefined animations during live performances. TakeToons also uses predefined animations but triggers these events based on the script markup. In addition to character motion, we support triggering of scene changes and camera cuts, which allows actors to control such higher-level events during performance, as demonstrated by Barnes et al. [2]. Finally, TakeToons enables layering of multiple performances [11] to produce complex animations.

Speech commands for navigation and editing

In addition to generating lip sync animations from speech performances, TakeToons recognizes speech commands that allow actors to navigate the script for retaking performances and recording characters out of order. Previous work has shown that speech-based interfaces make creative editing tasks easier by abstracting interface complexity [19, 34]. Systems for video scrubbing and editing use a transcript to make it easy for users to issue semantic queries to find specific frames [24, 8]. In TakeToons, we support similar natural language queries for navigating the script. Wang and van de Panne [34] combine spoken input with mouse pointer input to ‘direct’ the actions of animated characters (e.g., “walk to here”). This approach is closest to ours in that it supports speech-based system commands for selecting characters, actions, start and end recording, replay and reset. However, we focus on using speech input to navigate the script, not to direct how the characters are animated.

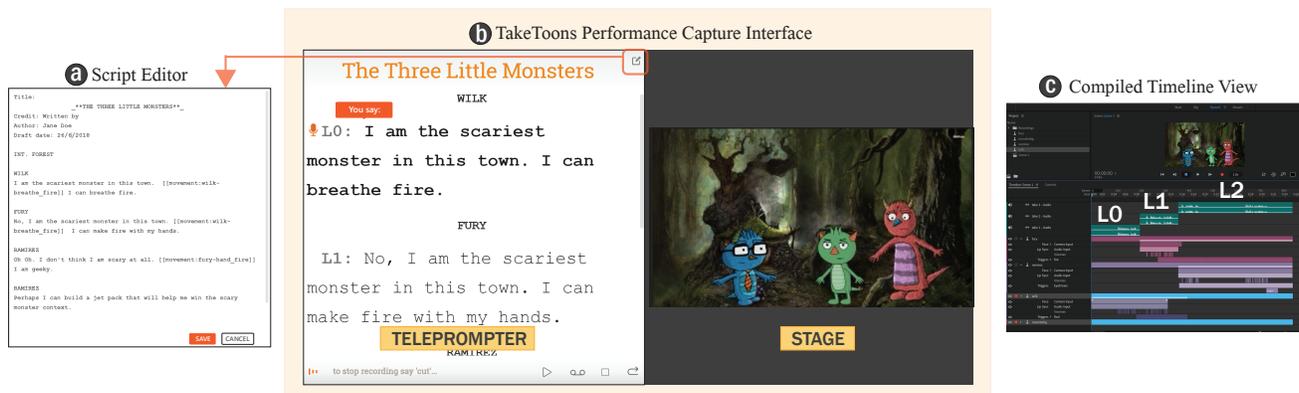


Figure 2. TakeToons (a) The script editor allows writing and editing the Fountain script. (b) The performance capture interface consists of a Teleprompter view (left) to read the script, and a stage view (right) for live preview of the animation. (c) The compiled animation is exported to Adobe Character Animator’s timeline for fine-grained editing. The action bar at the bottom of the teleprompter view provides hints for speech interactions (here: “say ‘preview animation’ or ‘start recording’”) and editing buttons for recording, stopping, doing retakes, and playing the animation as an alternative to speech commands.

USER EXPERIENCE

To demonstrate the key features and overall user experience of our system, we describe the process of creating an original animated story, “The Three Little Monsters,” using TakeToons.

Script

To create an animated story with TakeToons, our animator, Dave, starts in the same way most production workflows begin—by writing a script. Our system uses a script format that extends Fountain [12], a standard text markup language designed for screenwriting (Figure 2a). Dave can create standard lines of dialogue and scene changes. In addition, he can also add annotation for animation, camera control, and sound effects. For example, to indicate that a monster named Wilk should raise her arms while saying “so big!”, Dave can insert an annotation like “[[motion:Wilk-raise arms]]” at the appropriate spot in the script (see Figure 4).

Voice performance

Given a script, TakeToons allows one or more performers to produce the corresponding animated content through voice interactions. Figure 2b shows the user interface for these interactions, which consists of two side-by-side views: (1) the teleprompter view which shows the script (along with line numbers, and character names), and (2) the stage view for viewing the animation as it occurs in real-time. In our current implementation the stage is derived from Character Animator’s performance animation engine.

The interface shows both the teleprompter and stage views with the sets and characters loaded. Once a script has been loaded, a typical workflow can begin with Dave telling TakeToons to start recording either by pressing a button at the bottom of the teleprompter view or by issuing the speech command: “start recording.” Though Dave is not using full-body performance here, he prefers to verbally direct recording and playback. The teleprompter view highlights the first line in the script, prompting Dave to start performing that line (a microphone icon indicates recording is in progress).

As Dave reads the line, the teleprompter view highlights recorded words in real time, and the stage view shows a live preview of the corresponding animation. In this case, the animation includes lip sync for Wilk, one of our three monsters, who is speaking. In addition to the vocal performance, Dave can simultaneously control Wilk’s head motion and expression through facial performance. Any other actions (e.g., a pre-defined hand gesture) that are in the script will trigger automatically when Dave hits the associated line or word in his voice performance.

Interleaved performance and editing

If Dave stumbles over a line or is unhappy with his performance, he can stop recording by saying: “cut.” From here, Dave can tap on the line to go back in the teleprompter or can issue the redo command by saying, “let’s try that again.” In response, TakeToons resets the teleprompter to the beginning of the current line and rewinds the stage view to show the scene and characters in the reset state. After performing the line again, Dave can tell TakeToons to keep going by saying, “continue recording.” During a recording session, Dave always has the option of replaying ‘rendered’ segments (e.g., “play animation” for the last line) or the entire animation (“play from beginning”).

Beyond fixing performance errors with retakes, Dave can also interleave edits to the script and animation metadata with his performances. While recording, Dave sees that the fire animation occurs before he has completed the utterance “...I can breathe fire.” To rectify this, he opens the script editor view which is accessible from the teleprompter. He moves the position of the ‘fire-animation’ tag later in the dialogue and saves the script, which brings him back to the teleprompter view. TakeToons has automatically updated the timing of the fire animation to reflect the change. Similarly, Dave can delete a line, change line order, or even edit the dialogue using the script editor. When dialogue is edited, the teleprompter view indicates that the relevant line needs to be rerecorded. Other edits such as deleting and reordering lines automatically update the animation without requiring any additional recording.

Layering animation

In stories with multiple characters, the user may want to layer additional (unscripted) animations on top of a given line of dialogue. For example, when Wilk breathes fire on line three, Dave may want Ramirez (the other monster in the scene) to recoil a bit. To perform this type of layering, Dave says “*lets layer Ramirez on line three.*” The teleprompter and stage view reset to line three, and the system starts recording with Ramirez activated to respond to Dave’s performance. The other pre-recorded animation (including Wilk’s line of dialogue) is played back, while the new Ramirez animation is captured. For some types of performance (e.g., facial reaction shots) it’s easier for Dave to focus on only those events. TakeToons allows him to do this by recording the parts where Wilk is reacting separately from the parts when he is talking.

Character-by-character performance

While Dave can perform all the characters himself, TakeToons also supports multiple actors working collaboratively on different roles. For example, Dave may choose to play Wilk while getting a second actor, Jill, to perform Ramirez. To record in *character-by-character mode*, Dave says “*lets record Wilk.*” The teleprompter view adapts by reducing the font size for the lines that Dave does not need to perform, which preserves the broader context of the scene while keeping the focus on Wilk’s lines. If Jill hasn’t recorded her Ramirez lines yet, TakeToons can work as a virtual “scene partner.” The system will generate placeholder animations by synthesizing the speech and triggering the corresponding animations in the scene. This gives the Wilk character a more natural performance as Dave can respond to a performed Ramirez rather than just the text. If any of Ramirez’s lines have already been recorded by Jill, TakeToons uses these recordings rather than the placeholders. While in this asynchronous mode, Dave can edit animation triggers and ordering of Ramirez’s dialogue to blend with his performance. TakeToons handles such edits without requiring Jill to reperform her lines.

Wrapping up

Once he is happy with the performances, Dave can either directly publish the resulting video or export the various recorded files to a structured timeline representation for additional editing and refinement in a traditional animation editing environment (Figure 2c). When producing the timeline view, TakeToons automatically structures the recorded segments so they are properly time-aligned and in-sync with the script. Any timing edits that Dave makes to the timeline are propagated to the story model. All timing changes are preserved during animation playback (e.g., when Dave’s performance is played back as Jill records Wilk). In addition, TakeToons also preserves timing edits to triggered animation even when a performance is rerecorded. In this way, our system supports interleaving of fine-grained timing edits with performance.

SYSTEM ARCHITECTURE

At the highest level (see Figure 3), TakeToons is a tool for optimizing animation workflow by generating an animation from an annotated script, speech, and non-speech performance data (e.g., facial motion). The main system consists of two

main components: (1) a *script parser* that transforms a *TakeToons script* into a *story model* (our internal representation of an animated story); and (2), the *story controller* that manages the current state of the story model (recording and playback) as the user performs various parts of the script. For speech-to-text, TakeToons currently uses Google’s real-time API [13]. We use Adobe Character Animator’s performance capture and animation engines to map performance (voice, motion, etc.) to animation [1]. Control of Character Animator is achieved through a custom Python-to-Lua API. TakeToons’ is implemented in Python with a web-based (HTML and JavaScript) client for the teleprompter view. Here, we focus on TakeToons’ script parser, story model, and story controller, the novel contributions of our work.

TakeToons script language

While standard-format screenplay text [31] is flexible, it is primarily intended to convey the *story* and not other facets of the production. A screenplay, for example, does not generally provide instructions on camera positioning or detailed animation instructions. Each job in the animation process (e.g., directors, cinematographers, animators) has its own ‘formats’ for describing inputs (what the person should do) and communicating outputs (intermediate products to show others): storyboards, animatics (rough sketched animations), exposure sheets (details on animation cels at the frame level), pre-visualizations (camera movement simulations), ‘needle-drops’ (temporary musical scores), etc.

With TakeToons, many of these jobs become ‘compressed,’ requiring a more general format for communicating what should happen, and when, in the final product. Because the story is often the first production step and always the backbone of the animation, we opted to extend the screenplay language. Specifically, we utilize an extension of the Fountain screenplay markup language [12]. Fountain is a popular text-based format that is used by multiple screenwriter tools to generate screenplays in a standardized format.

Screenplay formats have a well defined structure for scene changes, character dialogue, and action [31]. The format has clear semantics to both capitalization and spacing (line breaks and indentation). For example, a scene change (all capitals) is unindented and of the form: “EXT. WRITERS STORE - DAY” indicating an external (outdoor) scene at the writer’s store during the day. Character speech is indicated by a centered and capitalized character name followed by lines of indented text representing the dialogue. Fountain simplifies the creation of this format by automatically applying the correct indentation, highlighting, etc. For certain actions (e.g., scene changes or knowing which character is speaking), TakeToons utilizes the standard script format.

For non-standard events, we use the double-bracketing syntax which standard Fountain parsers treat as a comment. Specifically, we use the format `[[type:character-action]]` where *type* indicates the kind of event we are triggering (e.g., ‘motion’ for animation, ‘camera’ for a camera change, and ‘sound’ for sound effects). The *character* and *action* are arguments to that trigger. For example, “`[[motion:Wilk-roar]]`” indicates that the Wilk character should perform the roar animation. Figure 4

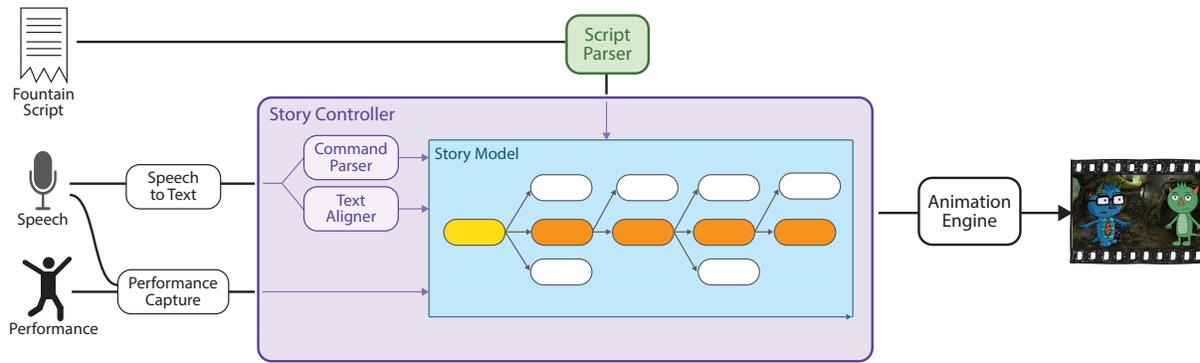


Figure 3. TakeToons Architecture. Our system takes a script, speech, and non-speech performance as input and generates an animated story. The key novel components are the script parser, story model, and story controller. We use Google’s real-time speech-to-text API and Adobe Character Animator to capture performance and map performances to animation.

(left) illustrates an example screenplay. By using this standard form, TakeToons’ script language has the benefit that it is flexible, familiar, and simultaneously backwards compatible with existing script editing software.

Story model

The TakeToons script parser (detailed below) translates, or ‘compiles’, the script into a *story model*. A TakeToons story model represents the sequence of events and actions that make up the final animation. Internally, a model is structured as a directed acyclic graph (DAG). Nodes represent dialogue in the script, script markup (camera, motion, etc.), and scene changes. The directed edges represent temporal relationships between these story elements (the semantics are roughly ‘performance’ for a node and ‘perform after’ for an edge). The DAG structure supports stories where actions overlap. For example, as a character talks, he might gesture with his hands or pace back and forth. At the same time, other characters in the scene may react by changing their facial expressions. The story model represents overlapping actions with nodes that share the same parent, indicating that corresponding (parallel) actions should occur after the parent node.

Figure 4 shows an example script and corresponding story model. The story model supports several types of nodes, each with an associated ‘play’ or ‘record’ action.

Speech nodes

Speech nodes correspond to lines in the script and are associated with a specific character or the narrator. To generate an animation, each speech node must have an audio recording that can either be performed by the user or automatically generated through text-to-speech. We use Google’s Text-To-Speech API [14] to synthesize speech for each script line.

Motion nodes

Motion nodes represent animation of a character. For example, Jane’s line introducing the robot Evans (Figure 4) includes a motion node where Evans walks in. This motion node is triggered by Jane’s line, “This is Evans” (a speech node). The animation itself may be performed directly by the user or generated from a pre-defined library of motions. In particular, our current implementation uses Adobe Character Animator

to generate motion in three ways: 1) automatic lip sync from a voice performance, 2) head motion and expression changes from a facial performance, and 3) triggerable, pre-authored animation cycles for a given character. Most speech nodes have an associated motion node (at the very least through lip sync). However there are exceptions where a motion node may overlap with multiple speech nodes or have no associated motion (e.g. narrator speech).

Sound-effect nodes

Sound effects are non speech audio events (e.g., sound of rain or thunder) that enhance the overall animation experience. In TakeToons, sound effect nodes trigger pre-recorded audio files.

Scene nodes

Scene nodes correspond to scene changes in the script. These nodes typically control the “set” (e.g., background artwork) and indicate which characters should be present in the scene.

Script parser

To generate the story model, the script parser first constructs a linear sequence of speech nodes from the lines in the script. To implement this component we extended the open source Fountain parser, Jouvence [18]. Within each parsed line, we create a speech node for each contiguous sequence of words between animation markup and connect adjacent nodes with directed edges. For each speech node, we automatically generate lip sync and facial performance motion nodes depending on the capabilities of the character. For example, most characters support lip sync, some are rigged to support head/facial animation, and the narrator does not support any motion. For each animation markup in the script, we create a corresponding motion or sound-effect node with a directed edge originating from the preceding speech node. Finally, we convert change-of-scene indicators in the script to scene nodes and connect these to the preceding speech node as well. The resulting graph has a (largely) fish-bone structure where the chain of speech nodes form the central spine with other nodes branching off (see Figure 4b). Though the model can support more general DAG configurations, in practice most story models follow this general pattern.

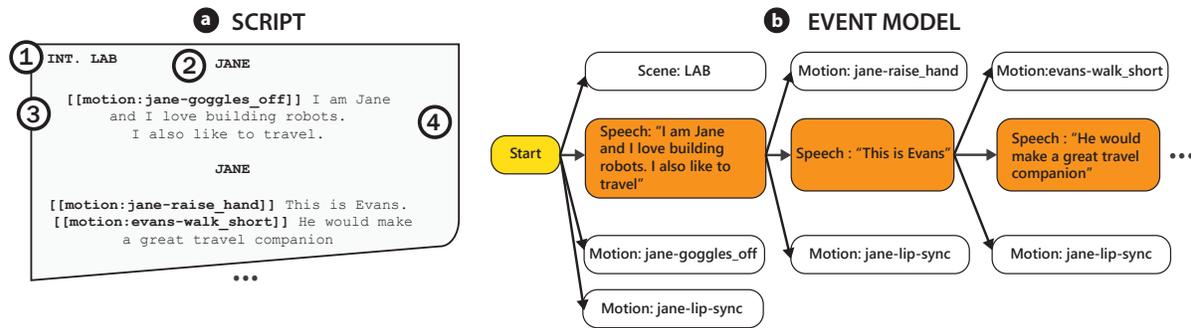


Figure 4. (a) A snippet of an augmented TakeToons script with scene information (1), character headings (2), dialogue (4), and animation markup (3). (b) The corresponding story model representation of the script.

The script itself can be changed even after recording has started. If changes result in new story nodes that require performance, the actor can record them. If a line or event is removed from the script, the corresponding node(s) can also be removed. By “replaying” the recorded story model (see below) the new animation will be re-rendered.

Story controller

The story controller enables recording and playback of the story based on real-time speech and performance input. It consists of a command parser, a text aligner, and a ‘playhead’ that points to the current active node(s) within the story model. The story controller ‘listens’ for both dialogue (i.e., record) and commands (i.e., directorial interactions). When in record mode, the controller actively attempts to align whatever the animator says against the script. If the controller believes there is a misalignment (i.e., the animator appears to have said something not in the script) the system will attempt to understand the mismatched text as a command (e.g., ‘stop recording’).

Script aligner

When the system enters the recording mode, the story controller expects subsequent speech to be a vocal performance. In this mode, the script aligner takes over and performs text alignment between the transcribed speech and the script segment associated with the active speech node that is being recorded. We use a simple alignment algorithm that tries to match each transcribed word in sequence. To account for transcription errors, the aligner performs fuzzy matching with a configurable amount of tolerance (default of two words). This both ensures that the system works even if the voice recognition system makes mistakes and to support performer ad libbing. To give the user real-time feedback while they are performing, the system updates the teleprompter view by highlighting each word that is successfully matched. When the aligner matches the last word in a speech node, the controller traverses the story model DAG and automatically moves the playhead to the next set of nodes. If there is a misalignment, the controller stops recording, and control is handed back to the command parser (described below).

While we currently use Google’s real-time speech-to-text transcription, this can be replaced with alternative engines. In

particular, a forced-alignment speech recognizer (e.g., [22]) may be effective, as it performs an alignment against a known script given an audio signal rather than attempting to decode the speech directly. However, because most algorithms of this type are off-line, and we require real-time alignment, we leave this extension to future work.

Playing and recording

Playing a story node plays any recorded content, auto-generated motion or synthesized audio. Recording a story node replaces any existing recording with a new performance. When nodes finish playing or recording, the story controller moves the playhead to the next node. In the record state, each node converts the user performance into recorded audio (for speech nodes) or animation content (for motion nodes). For scene nodes, sound-effect nodes, and motion nodes that trigger pre-authored animation cycles, recording simply specifies the exact time when the corresponding animation event should occur. In the play state, each node plays its recorded content at the appropriate time based on the story model graph structure. For unrecorded speech nodes, playback uses the synthesized speech. The system skips any unrecorded motion nodes that require an explicit user performance.

Figure 5 demonstrates different play/record configurations of a story model with two characters Jane and Evans. With the simplest, basic recording (Figure 5a), the playhead moves line by line (i.e., node by node) from the start and records the associated performance. This will record the dialogue and animation regardless of who the character is. With single character animation (b), the story model is configured to only record a single character’s performance (e.g., only Evans and not Jane). As the playhead moves across the nodes, those not associated with Evans are played (e.g., previously recorded lines or animations) but Evans’ are recorded. During a retake for Evans’ line (c), the playhead jumps back to the appropriate node and recording/playback proceeds from there. When layering a performance (d), playback and recording are done simultaneously. In this case, we are layering Evans performance for line 1 (he’s responding to something Jane is saying). Jane’s line is played while the actor’s performance for Evans is simultaneously recorded.

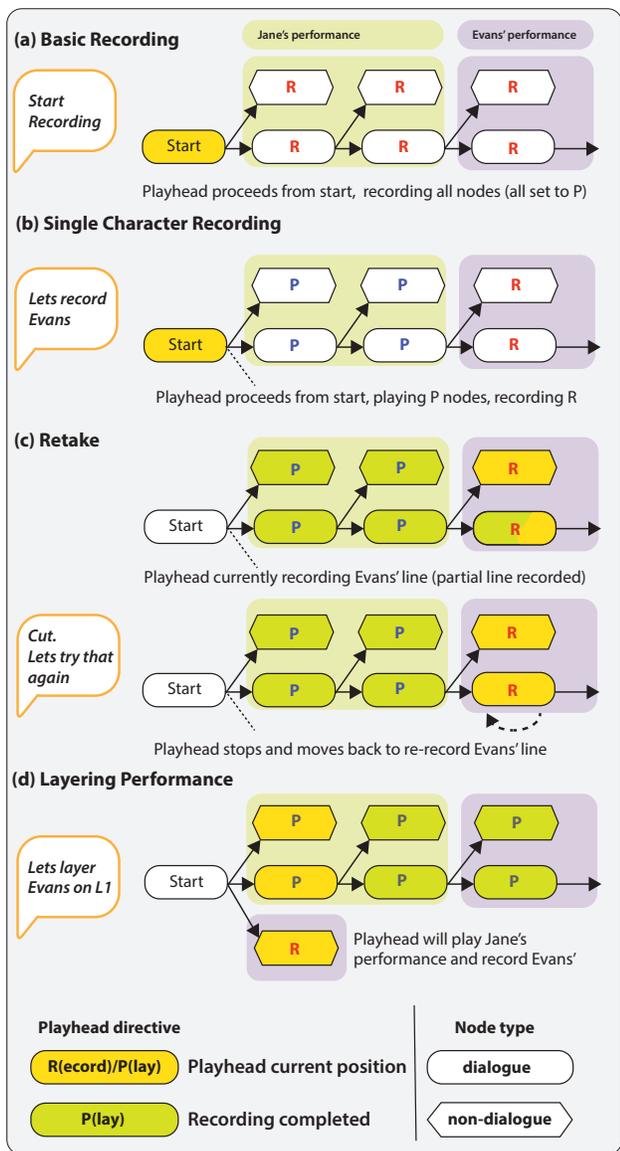


Figure 5. Story model states for different animation commands. Each node of the model can be set to record or play (or skip) based on the performance type. As the playhead moves along the DAG, those nodes that are to be recorded are captured from the actor’s performance. Nodes that are set to play (e.g., pre-recorded dialogue, timed animations, sounds, scene changes, etc.) are ‘performed’ by the system. In this diagram the nodes in the center of each DAG are speech nodes.

Command parser

While TakeToons supports most interactions through the interface, it can also handle non-dialogue voice commands for remote ‘directing.’ The command parser maps (non-dialogue) transcribed speech input to a set of pre-defined commands. Internally, we implement a template look-up that extends [23] (a similar model to Amazon’s Echo skills), to define a set of example phrases along with corresponding intents (play, record, retake, layer, stop). Command phrases may also contain placeholder tags for certain words that gets replaced during parsing. For example, the retake template for a specific line is: “*Lets*

redo line [‘line-number’, ‘four’].” Here [‘line-number’] is the tag and ‘four’ is the placeholder text. During performance, the speech command might be “*Lets redo line two*”, when parsed, ‘two’ is tagged as the line-number. This information is used for executing the intent– *retake*.

When the transcribed speech matches a command, the controller updates the playhead and the state of each story node to either **play** or **record**, depending on the command. When appropriate, TakeToons infers the context of the command. For example, in Figure 5c, the animator says “retake” or “let’s try again” when recording line two. Without needing to specify which line to retake, the playhead moves back to start of line two, and all events in that node are set for recording.

TakeToons supports both navigation commands and recording commands. Navigation commands include “*Go to where [character] says [dialogue]*,” “*Let’s redo last line*,” “*Let’s redo line [line number]*,” “*Let’s layer [character] on line [line number]*,” “*Play from beginning*”. Recording commands include “*Start recording*,” “*Stop recording*,” “*Let’s record [character name]*,” “*Cut*”. If the transcript does not match dialogue, or a known command, an error message is displayed.

Performance capture and animation

TakeToons uses Adobe Character Animator’s performance capture and animation engines. For performance capture, Character Animator uses a live camera feed to record facial performances and a microphone to record speech. The animation engine automatically maps the facial performances to head motion and expression changes and converts the speech input into lip sync (i.e., mouth motions) for rigged 2D characters. Character Animator also supports the creation of pre-authored, triggerable animation cycles for any character. TakeToons leverages all of this functionality to generate the live animation preview as the user records or plays the story model.

Furthermore, while our user interface does not expose a traditional animation timeline, our story model does map directly to a timeline with multiple tracks for different characters and segments that correspond to each recorded performance. In fact, our system maintains such a timeline, which the user can access and edit within Character Animator at any point. As noted in the User Experience section, working directly with the timeline may be useful for refining the animation. Timing edits in the Character Animator timeline are propagated back to the story model. When a segment is moved in time, a time change ‘delta’ is saved to the corresponding event node. During subsequent playback and performances, TakeToons factors in these edits when triggering animations.

RESULTS

We used TakeToons to generate three animated stories, as shown in Figure 6. The three are representative of different classes of animated stories and involve different kinds of acting, animation, and recording strategies. One story (Figure 6b) includes a single red monster character delivering a monologue in a similar style to animated interviews [4] and monologues [32]. The other two examples are more traditional multi-character stories. In total, the three stories involve six different characters performed by two actors. We obtained the



Figure 6. Example animations generated using TakeToons

characters (artwork and rigging) from the set of default assets distributed with Character Animator.

We designed the stories to include a range of common animation effects. All examples involve speaking characters whose mouths move based on the voice performances of the actors. In addition, we added head movements and expression changes to most characters (via facial performance) adding life to their motions. We also made extensive use of pre-defined animations, almost all of which were bundled with the character artwork. For example, in the red monster monologue, the script references several short cycling animations like fire and a heart that appear above the character’s head, as well as background props that help illustrate what she is saying. In the Jane and Evans story (Figure 6c), Jane uses various hand gestures to punctuate her lines. At the end of the three monsters story (Figure 6a), we play a longer jetpack animation over the blue Ramirez character, who dreams of building such a device to impress his friends. The stories also trigger pre-defined camera angles and sets to create shot boundaries and scene changes. By adding all of these animations directly into the scripts, TakeToons makes it possible to produce these

effects via the vocal performances alone, rather than through low-level manual editing and post-processing steps.

We also leveraged the layering functionality of TakeToons. In the three monsters story, we first recorded the vocal performances for all three characters. Then, we layered a facial performance of Ramirez recoiling as the green monster, Wilk, breathes fire in his direction. By allowing users to layer more freeform or improvised performances on top of pre-defined animations, TakeToons provides expressive control over the details of individual motions and character interactions. The combination of layering and automatically triggered animations gives content creators the option of allowing actors to provide vocal performances that control the timing of carefully crafted pre-defined animations. This flexibility in our approach makes it general enough to support a variety of animated stories across many different genres and visual styles.

INFORMAL EVALUATION

To gain insights on the potential impact of our approach for real animation workflows, we conducted informal feedback sessions with four professional animators. Three participants

work primarily on 2D animation, and the fourth focuses on 3D animation with some 2D segments. One of the 2D animators produces largely unscripted content that involves improvised conversations between himself and a collaborator. He typically does almost all of the production by himself. The other three animators focus on scripted content and have worked on production teams of various sizes. All animators had prior experience with performance-based animation systems.

At the beginning of each session, we described the overall workflow of our approach. As part of this description, we showed animators the Jane and Evans characters and their script (Figure 6c). We loaded the story script into our system and demonstrated how to issue commands and record individual lines. After this introduction, we asked participants to produce animated content for the story using the system. Figure 2 shows the arrangement of the script view and stage view that we used in the study. Finally, we conducted a freeform interview where the animators commented on their overall impressions and in what scenarios they would find the TakeToons' script-based workflow to be useful. Each session lasted roughly one hour.

Feedback

All participants said that our script-based workflow would be useful for animation production. The three who work on scripted content were especially enthusiastic. They felt that TakeToons could make it much faster to generate an initial cut of the animation. For some applications (e.g., kids' shows with simple animations, instructional content), they said that this output would likely be very close to the final content. For larger budget productions, TakeToons could streamline the creation of rough cuts and thus enable more efficient iterations. Among these animators, there was a consensus that adding animation annotations to the script was worth the small additional up-front effort given the downstream benefits in producing the animated content. They also noted that seeing a live preview of the animation while reading the script is valuable because it gives voice actors useful context for the characters and the scene, which they can incorporate into their performances. The animator who does not use scripts was uncertain if TakeToons would benefit his existing workflows, but believed it could help save time in scripted scenarios.

During the interviews, the animators proposed some interesting usage scenarios for TakeToons. One animator noted that he often works with pre-recorded audio from various voice actors. In this setting, he was excited to use TakeToons as a way to generate the animated content using the recorded speech as input. In addition to making it much faster to obtain a rough cut, he felt that it would be easier for him to identify and fix small animation errors (e.g., lip sync) while watching the real-time preview. Another animator noted the benefits of TakeToons for collaborative projects, where multiple performers and animators work on different parts of the story. By supporting out-of-order recording and layering, our approach makes it possible for each collaborator to focus on their specific task while seeing other auto-generated (i.e., using text-to-speech) or previously recorded performances for context.

Regarding additional features, all but one of the participants suggested that the teleprompter be overlaid on or near the stage where the animation occurs. This would make it easier for the actor to read their lines while also seeing the corresponding animations. One animator even proposed using word bubbles over the characters on stage as a way to display the lines. A related suggestion was to provide real-time guidance for the desired pacing for each line as the user reads. These guides could be generated based on durations of the corresponding animation events (e.g., a triggered animation cycle may take 2 seconds to complete), or explicitly annotated in the script. Finally, one animator suggested a visual interface for adding the various annotations to the script. For example, Character Animator has a panel that shows all of the pre-defined animations for each character. The participant wanted the option of dragging these animation triggers directly onto the relevant parts of the script.

Overall, we found the feedback to be very encouraging. Despite occasional speech recognition errors and some latency in our prototype implementation, the general sentiment was very positive. That said, it would certainly be valuable to conduct a larger, formal evaluation that provides more concrete validation for the approach.

DISCUSSION

Utility of TakeToons

Our implementation of TakeToons is centered around a persistent *story-model* that binds script-writing with performance capture and post-processing edits. This approach offers numerous advantages for both individual animators as well as conventional production teams. For individual animators, TakeToons provides a unified interface for performing and editing that can be controlled through voice-based commands. This minimizes the task switching cost between performing and controlling the animation. It also gives authors the freedom to refine the script and low-level animation details at any point in the production process. In other words, TakeToons provides greater efficiency and flexibility to 'one-man-band' productions.

For professional teams (with separate writers, performers, and editors), TakeToons provides a consistent (script-based) structure that makes it easy to share partial work, coordinate tasks, and iterate without having to redo all the downstream authoring work. For example, two or more actors can coordinate synchronous and asynchronous recording of multiple parts directly into a rough-cut. The script writer can make changes to the script based on an actor's voice performance without losing all downstream changes (e.g., substitute certain words based on the actor's pronunciation). Animation designers can modify detailed animation properties based on performances (e.g., distance and intensity of fire animation), and edit the sequence and timing of animations via the script or timeline without requiring actors to rerecord all their performance. As suggested in our feedback with animators, such collaborative editing and production capabilities can be useful in professional settings.

Finally, for novice animators, TakeToons offers a streamlined workflow for creating animations. For instance, given a script

and animation assets, parents can create their own renditions of popular children’s stories (e.g., The Three Little Pigs). This allows them to control the pace of the story, provide a more familiar voice to characters, and even engage their children in animation creation.

Limitations & Future Work

In our current instantiation of TakeToons we have mainly focused on those animations that involve scripted speech performances (i.e., dialogue and narration). While speech is a key component for many animations, not all animated stories contain spoken performance. Further, while we handle minor variations to scripted speech performance through fuzzy matching, we currently do not support extensive improvisations. However, these limitations can be handled in the future by extending our story model. For example, by mapping animation gestures and speech phrases to other non-performance events, the story model can be generated in real-time using an indexed look-up table. As an extension, to support interleaved editing and performance, the teleprompter view can be replaced with an alternate visualization such as storyboards, glyphs, or node-diagrams. This will allow for voice based control by directly referencing those elements, and can be triggered through a ‘wake-word’ (for ad libbing).

A second area for future work is to expand our focus beyond the end-user experience of professional animators. As noted earlier, while novice animators can work with pre-defined assets, understanding the script syntax for customization and editing can be a challenge. Future work can look at improving the script writing experience with features such as auto-complete or allowing the writer to highlight a piece of dialogue and choosing an animation that fits with the selection.

A limitation in performance systems, and one that impacts TakeToons, is that while writers and animators can directly edit the script after performance, these edits may introduce discontinuities in the cuts. For example, if a character jumps, walks, and falls and the walk is deleted, the final animation will include a sharp cut between a jump and fall at different locations. Future work can explore the best way to accommodate this type of editing. Further, the current interface does not include timing information, which can be useful in guiding performance and scripting animation. The TakeToons script can be extended with additional markup [33]. Other possible extensions may include supporting the review of multiple takes and authoring of stories with different moods [20].

Finally, we believe the core approach of TakeToons has applications beyond animation production. Writing a script is a key step in a wide variety of live and recorded performances, including educational lectures, business presentations, and conference videos. A natural extension of TakeToons would be to link scripts with the relevant metadata and triggered events for these other types of performances. For example, slide presentations could be enhanced by leveraging script annotations that automatically trigger slide transitions and animations based on the presenter’s performance [25]. In general, we believe our approach could benefit many script-driven authoring processes by enabling workflows that integrate and interleave the various steps of the production pipeline.

CONCLUSION

In this work, we present an approach for integrating and interleaving animation production workflows through an annotated script and performance-based animation. TakeToons compiles script metadata and annotations into a story model, and a story controller interprets and aligns performance with the story model. Finally, TakeToons assembles the story model recordings into an animated movie. Our approach supports a broad array of animation scenarios including individual performance, collaborative authoring, layered animation, and semi-automated story telling. We demonstrate how different types of common animation type can be constructed, and user feedback from professionals confirms the practical benefits of this approach in real productions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our study participants for their time and helpful feedback. We also thank Jovan Popović and Dave Simons for their advice and input and Dan Ramirez for his help with the demo video.

REFERENCES

1. Adobe. 2018. Character Animator CC. (2018).
2. Connelly Barnes, David E. Jacobs, Jason Sanders, Dan B Goldman, Szymon Rusinkiewicz, Adam Finkelstein, and Maneesh Agrawala. 2008. Video Puppetry: A Performative Interface for Cutout Animation. *ACM Trans. Graph.* 27, 5, Article 124 (Dec. 2008), Article 124, 9 pages. DOI: <http://dx.doi.org/10.1145/1409060.1409077>
3. Floraine Berthouzoz, Wilmot Li, and Maneesh Agrawala. 2012. Tools for Placing Cuts and Transitions in Interview Video. *ACM Trans. Graph.* 31, 4, Article 67 (July 2012), 8 pages. DOI: <http://dx.doi.org/10.1145/2185520.2185563>
4. Blank on Blank. 2017. (2017). <https://blankonblank.org/>
5. Matthew Brand. 1999. Voice Puppetry. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 21–28. DOI: <http://dx.doi.org/10.1145/311535.311537>
6. Werner Breidfuss, Helmut Prendinger, and Mitsuru Ishizuka. 2007. Automated Generation of Non-verbal Behavior for Virtual Embodied Characters. In *Proceedings of the 9th International Conference on Multimodal Interfaces (ICMI '07)*. ACM, New York, NY, USA, 319–322. DOI: <http://dx.doi.org/10.1145/1322192.1322247>
7. Ian Buck, Adam Finkelstein, Charles Jacobs, Allison Klein, David H. Salesin, Joshua Seims, Richard Szeliski, and Kentaro Toyama. 2000. Performance-Driven Hand-Drawn Animation. In *NPAA 2000 : First International Symposium on Non Photorealistic Animation and Rendering*. 101–108.
8. Juan Casares, A. Chris Long, Brad A. Myers, Rishi Bhatnagar, Scott M. Stevens, Laura Dabbish, Dan Yocum,

- and Albert Corbett. 2002. Simplifying Video Editing Using Metadata. In *Proceedings of the 4th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS '02)*. ACM, New York, NY, USA, 157–166. DOI: <http://dx.doi.org/10.1145/778712.778737>
9. Michael M Cohen and Dominic W Massaro. 1993. Modeling coarticulation in synthetic visual speech. In *Models and techniques in computer animation*. Springer, 139–156.
 10. Darren Cosker and James Edge. 2009. Laughing, crying, sneezing and yawning: Automatic voice driven animation of non-speech articulations. *Proceedings of Computer Animation and Social Agents, CASA (2009)*, 225–234.
 11. Mira Dontcheva, Gary Yngve, and Zoran Popović. 2003. Layered acting for character animation. In *ACM Transactions on Graphics (TOG)*. ACM, 409–416.
 12. Fountain. 2018. A markup language for screenwriting. <http://fountain.io>. (2018). Accessed: 2018-04-01.
 13. Google. 2018a. Cloud Speech API. <https://cloud.google.com/speech/>. (2018).
 14. Google. 2018b. Cloud Text-To-Speech API. <https://cloud.google.com/text-to-speech/>. (2018).
 15. Ricardo Gutierrez-Osuna, Praveen K Kakumanu, Anna Esposito, Oscar N Garcia, Adriana Bojórquez, José Luis Castillo, and Isaac Rudomín. 2005. Speech-driven facial animation with realistic dynamics. *IEEE transactions on multimedia* 7, 1 (2005), 33–42.
 16. Judith H Haag and Hillis R Cole. 1980. The Complete Guide To Standard Script Formats–Part 1: Screenplays. (1980).
 17. Kaveh Hassani and Won-Sook Lee. 2016. Visualizing natural language descriptions: A survey. *ACM Computing Surveys (CSUR)* 49, 1 (2016), 17.
 18. Jouvence. 2018. Fountain parser. <https://bolt80.com/jouvence/>. (2018).
 19. Gierad P Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. PixelTone: a multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2185–2194.
 20. Mackenzie Leake, Abe Davis, Anh Truong, and Maneesh Agrawala. 2017. Computational Video Editing for Dialogue-driven Scenes. *ACM Trans. Graph.* 36, 4, Article 130 (July 2017), 14 pages. DOI: <http://dx.doi.org/10.1145/3072959.3073653>
 21. Marcel Marti, Jodok Vieli, Wojciech Witoń, Rushit Sanghrajka, Daniel Inversini, Diana Wotruba, Isabel Simo, Sasha Schriber, Mubbasir Kapadia, and Markus Gross. 2018. CARDINAL: Computer Assisted Authoring of Movie Scripts. In *23rd International Conference on Intelligent User Interfaces (IUI '18)*. ACM, New York, NY, USA, 509–519. DOI: <http://dx.doi.org/10.1145/3172944.3172972>
 22. Michael McAuliffe, Michaela Socolof, Sarah Mihuc, Michael Wagner, and Morgan Sonderegger. 2017. Montreal Forced Aligner: Trainable Text-Speech Alignment Using Kaldi. In *Proc. Interspeech 2017*. 498–502. DOI: <http://dx.doi.org/10.21437/Interspeech.2017-1386>
 23. Nate Parrott. 2018. Commanding. <https://github.com/nate-parrott/commanding>. (2018).
 24. Amy Pavel, Dan B Goldman, Björn Hartmann, and Maneesh Agrawala. 2015. Sceneskim: Searching and browsing movies using synchronized captions, scripts and plot summaries. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 181–190.
 25. Hans Rosling. 2010. Hans Rosling's 200 Countries, 200 Years, 4 Minutes - The Joy of Stats. <https://www.youtube.com/watch?v=jbkSRLYSojo>. (2010).
 26. Steve Rubin, Floraine Berthouzoz, Gautham J. Mysore, and Maneesh Agrawala. 2015. Capture-Time Feedback for Recording Scripted Narration. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 191–199. DOI: <http://dx.doi.org/10.1145/2807442.2807464>
 27. Steve Rubin, Floraine Berthouzoz, Gautham J. Mysore, Wilmot Li, and Maneesh Agrawala. 2013. Content-based Tools for Editing Audio Stories. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 113–122. DOI: <http://dx.doi.org/10.1145/2501988.2501993>
 28. Jinhong Shen, Terumasa Aoki, and Hiroshi Yasuda. 2004. EMM Software System : Electronic Movie Making from Screenplay. *IPSJ SIG Notes* 2004, 86 (aug 2004), 51–56. <https://ci.nii.ac.jp/naid/110002780681/en/>
 29. Hijung Valentina Shin, Wilmot Li, and Frédo Durand. 2016. Dynamic Authoring of Audio with Linked Scripts. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 509–516. DOI: <http://dx.doi.org/10.1145/2984511.2984561>
 30. David J Sturman. 1998. Computer puppetry. *IEEE Computer Graphics and Applications* 18, 1 (1998), 38–45.
 31. D. Trotter. 2014. *The Screenwriter's Bible: A Complete Guide to Writing, Formatting, and Selling Your Script*. Silman-James Press.
 32. Trump Monologue from Our Cartoon President (Showtime). 2018. (2018). <https://www.youtube.com/watch?v=2uqjAbzrM2I>
 33. W3C. 2018. Speech Synthesis Markup Language. https://en.wikipedia.org/wiki/Speech_Synthesis_Markup_Language. (2018).

34. Zhijin Wang and Michiel van de Panne. 2006. Walk to here: a voice driven animation system. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 243–251.
35. Nora S Willett, Wilmot Li, Jovan Popovic, and Adam Finkelstein. 2017. Triggering Artwork Swaps for Live Animation. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 85–95.