3D Puppetry: A Kinect-based Interface for 3D Animation

Robert T. Held*, **Ankit Gupta**†, **Brian Curless**†, **and Maneesh Agrawala***
*University of California, Berkeley †University of Washington
{rheld,maneesh}@cs.berkeley.edu {ankit,curless}@cs.washington.edu

ABSTRACT
We present a system for producing 3D animations using physical objects (i.e., puppets) as input. Puppeteers can load 3D models of familiar rigid objects, including toys, into our system and use them as puppets for an animation. During a performance, the puppeteer physically manipulates these puppets in front of a Kinect depth sensor. Our system uses a combination of image-feature matching and 3D shape matching to identify and track the physical puppets. It then renders the corresponding 3D models into a virtual set. Our system operates in real time so that the puppeteer can immediately see the resulting animation and make adjustments on the fly. It also provides 6D virtual camera and lighting controls, which the puppeteer can adjust before, during, or after a performance. Finally our system supports layered animations to help puppeteers produce animations in which several characters move at the same time. We demonstrate the accessibility of our system with a variety of animations created by puppeteers with no prior animation experience.

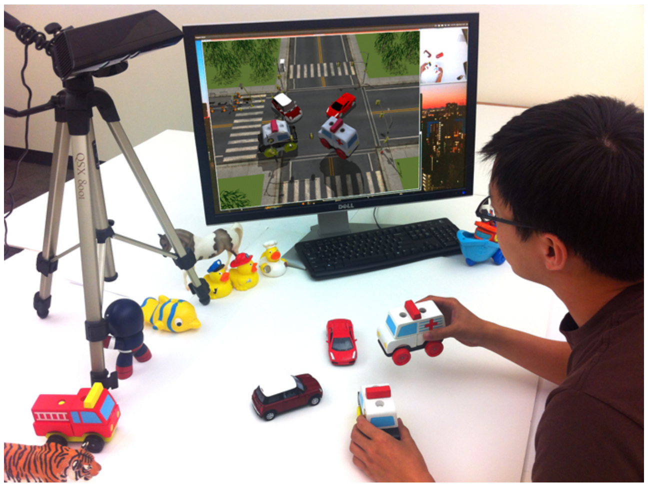
Figure 1. Our system allows puppeteers to use toys and other physical props to directly perform 3D animations.

Author Keywords
Tangible user interface, animation, object tracking

ACM Classification Keywords
I.3.6 [Computer Graphics]: Methodology and Techniques - Interaction Techniques; I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism - Animation.

INTRODUCTION
Today, the most common way to create 3D animated stories is to use high-end, key-frame-based animation software such as Maya [3], 3ds Max [2] or Blender [7]. While these systems provide fine control over the motion, position, and timing of each element in the scene, users must learn a complex suite of tools and commands to operate their high-threshold, high-ceiling interfaces. Building the expertise to use these tools effectively requires time and patience.

A simpler alternative is to film a performer moving physical puppets with their hands. Real-time, kinesthetic feedback from the puppets allows the puppeteer to concentrate on the performance, rather than how it is being captured. Such direct manipulation significantly reduces the learning curve required to create 3D stories, and even children can readily participate. However, the resulting video usually reveals the puppeteers and their controls. Moreover, video is difficult to modify once it is captured, and the physical nature of a video-recorded puppet show places practical limits on lighting, background complexity, and camera motions.

Motion capture offers another alternative for creating 3D animation that retains the real-time physical performance benefits of video recording a puppet show, but removes the constraints on lighting, background complexity, camera motions and puppeteer visibility. Such systems track the motions of physical props or puppets and map the motions onto virtual objects or characters rendered into virtual sets. However, current motion-capture implementations require complex and often expensive hardware setups (e.g. multiple high-frame-rate cameras, physical markers on the objects, specialized gloves on the hands, etc.) making them inaccessible to casual users [29, 31].

We introduce a 3D puppetry system that allows users to quickly create 3D animations by performing the motions with their own familiar, rigid toys, props, and puppets. As shown in Figure 1, the puppeteer directly manipulates these objects in front of an inexpensive Kinect depth camera [10]. We use the Kinect along with the ReconstructMe 3D scanning software [13] to build virtual 3D models of each puppet as a pre-process. Then, as the puppeteer performs the story,

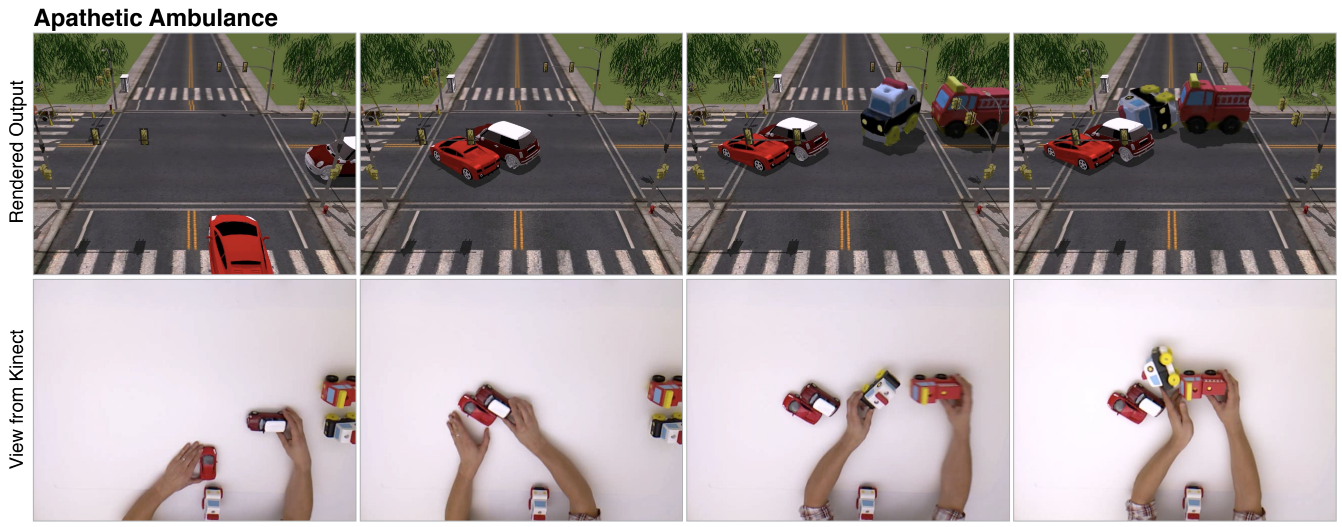

Figure 2. Example frames of our system in action. The bottom row shows the puppeteer manipulating physical puppets. The top row shows the result of our system tracking the puppets in real time and rendering them in a virtual set.

our system captures the motions of the physical puppets and renders the corresponding 3D models into a virtual set (Figure 2). Our system operates in real-time so that the puppeteer can immediately see the resulting animation and make adjustments to the motions on the fly.

We use inexpensive capture hardware and a relatively simple setup to make our system accessible to many users. Our system does not require any markers or specialized gloves for tracking. The performative mode of interaction, which closely resembles a physical puppet show, makes it familiar and approachable to novices. Kinesthetic and proprioceptive feedback gives the puppeteer constant and direct knowledge of the pose of each puppet as it is handled. For instance, the puppeteer can determine a puppet's pose simply by feeling its shape and knowing the spatial positions of her hands relative to the performance space. Moreover, with tangible 3D puppets, the physics of the real world can help to produce natural-looking motions. To animate a car crash, for example, a puppeteer can either manually move two toy cars simultaneously, or more easily, just send one car rolling into the other and rely on momentum to complete the performance.

Our puppet-tracking algorithm does impose a few restrictions on puppeteers. It assumes that puppets are rigid objects—it cannot track articulated joints or easily deformable materials such as cloth. In addition, if puppeteers move the puppets very quickly over long distances our system can lose track of them. However, in practice we have found that a wide variety of toys and props are made of suitably rigid materials and that puppeteers quickly learn the limitations on movement speed.

In addition to its basic tracking functionality, our system includes controls and effects to help users create animated stories. Because our system generates a 3D animated scene, users can adjust the camera, lighting, and background sets of the rendered animation at any time, before, during, or after the performance. In practice we have found that puppeteers sometimes move puppets outside of the tracking volume accidentally, at which point they no longer appear in the virtual set. To warn puppeteers before this happens, we reduce the opacity of the rendered puppets as they approach the edge of the tracking space. Finally, some stories may require a puppeteer to perform with more than two puppets at a time. We provide an interface for creating layered animations to facilitate such multi-puppet animations. Puppeteers can separately capture performances with different puppets, and then combine them into one animation.

We demonstrate the flexibility and accessibility of our approach with a set of animated stories created by first-time puppeteers with no prior animation experience. Our puppeteers combined several features of our system, including 6D camera and lighting controls, physical interactions between puppets, faded entrances and exits, and layered animations to tell engaging short stories. Our results indicate that novice users can quickly generate entertaining 3D animations with our system.

RELATED WORK

Our 3D puppetry system builds on several areas of related work.

Tangible Interfaces With tangible interfaces [16], the user handles physical objects to interact with a computer. These systems use a variety of methods to track objects, including computer vision, electronic tags, and bar codes [21]. For instance, one could use a small paddle to rearrange furniture in a miniaturized virtual room [20], or move two tracked cards together to make virtual characters interact [22].

Our system bears the most resemblance to interfaces that use the identity of everyday objects to guide interactions. Avrahami et al. [4], use a touch-sensitive tablet and stereo cameras to track physical props. Users can roll plastic soccer balls at the tablet to play a sports game, or directly place game tokens on the screen to play tic-tac-toe. Lego OASIS [34] identifies Lego figures using a depth sensor and then projects relevant animations into the physical space. For instance, they can detect a Lego dragon figure and project flames onto it. Finally, Johnson, et al.'s [18] sympathetic interface uses a plush toy with embedded sensors as input. The puppeteer manipulates the limbs of the puppet to control its virtual representation, which can interact with other virtual characters. These systems all use physical input to trigger preset events and interactions, whereas we design our interface to permit free-form creation of new animations.

Kinect-based Interfaces. Researchers have used the Kinect to produce new, low-cost, physical interfaces. The interfaces can track the human body [28] or hands [15] to provide interaction with virtual objects, including medical-imaging data [11]. Though our system also uses the Kinect, we track physical objects, rather than the puppeteer's body. The KinectFusion project [17] allows one to quickly scan a physical environment. We use a similar system, ReconstructMe [13], to convert physical puppets into 3D models. However, the rest of our interface bears less resemblance to these systems, as we focus on object tracking rather than scanning.

Object-based Motion Capture. Motion-capture generally refers to systems that record the movements of physical objects and converts them into animations. Our system most closely parallels object-based motion-capture systems, which track small physical objects and apply their movements to virtual characters [9, 25, 24]. While the prior systems focus on the detailed articulation of one character at a time, we provide a quick way for puppeteers to tell stories by performing with multiple rigid puppets at the same time. The other systems also require multiple cameras [9] or specialized sensing hardware [25, 24], while our system requires only a single Kinect and a tabletop.

Video Puppetry. The 2D video-puppetry system developed by Barnes, et al. [5] lets puppeteers perform animations with paper cut-out characters. It identifies, tracks, and re-renders the characters with the puppeteer removed in real time. Puppeteers operate our system in a similar way, but our use of 3D puppets enables more types of animations. For instance, 3D puppets can be rotated in three dimensions, whereas 2D cutouts are restricted to one axis of rotation and cannot rotate out of the plane. We also allow 6D control of the camera (3D translation and 3D rotation) during performance and playback, while camera changes in the video-puppetry system are limited to translations and zooms. Finally, the mass and shape of 3D objects let the user rely more on physics to tell a story. For instance, one could set a toy car rolling across a scene, which would not be possible with a piece of paper.

SYSTEM OVERVIEW

Our system consists of three main modules: Setup, Capture, and Render. During Setup, puppeteers scan physical puppets into our Puppet Database and build color models of their own skin. The Capture module consists of three components: a Puppet Identifier, a Point-cloud Segmenter, and a Pose Tracker (Figure 3). The Puppet Identifier uses 2D image features to detect the presence of each physical puppet and initialize the Pose Tracker. The Point-cloud Segmenter takes in a colored 3D point cloud from the Kinect and outputs individual point clouds corresponding to each physical puppet. The Pose Tracker then aligns each physical puppet's segmented point cloud with its stored 3D model to determine its pose. In the Render module, our system uses the captured poses to render the corresponding 3D model for each physical puppet in the virtual set. This module provides camera and lighting controls, and the option to swap 3D backgrounds.

SETUP

For tracking and rendering purposes, our system maintains a Puppet Database of defining characteristics for each trackable puppet. The database entry for each puppet consists of a 3D model and several sets of color images and depth maps associated with different example poses. If a puppeteer wishes to use a new puppet with our system, this information must be scanned in and stored in the database once as a pre-processing step. Finally, prior to running our system for the first time, each puppeteer must build a color model of their skin. This model is used by the Point-cloud Segmenter to isolate points that belong to puppets, and not the puppeteer's hands, and is necessary for accurate tracking.

Building the Puppet Database

The Puppet Database includes a 3D model of each physical puppet. To create a 3D model, the puppeteer first uses ReconstructMe [13] to scan the physical puppet with the Kinect sensor. As the puppeteer moves the Kinect around the puppet, casually capturing it from all sides, ReconstructMe converts the stream of incoming Kinect point clouds into an untextured 3D mesh of the puppet in real-time. The raw mesh also includes parts of the environment around the puppet, so we crop out non-puppet points and fill any remaining holes using the MeshMixer modeling software [27].

To add color to the model, the puppeteer loads the untextured mesh into our puppet-tracking system in *Painting mode* and places the physical puppet in front of the Kinect. Painting mode tracks only one puppet at a time, and uses a stripped-down version of the Capture module. It aligns the puppet's 3D model with the incoming Kinect point cloud, and then transfers the colors from the point cloud to the vertices of the 3D model. Figure 4 shows the result of scanning a toy using ReconstructMe and colorizing the model using our approach. Our Painting mode could also be used to colorize models obtained from other sources (e.g., laser scanners). The complete algorithmic details of our tracking approach are given in the Capture Section.

For accurate tracking it is essential that the 3D model match the shape of the physical puppet as closely as possible. However, our system does not require that puppeteers use ReconstructMe, and we also use models from other sources including Google's 3D Warehouse [12] and Autodesk's 123D Catch [1], which uses a multi-view stereo algorithm to produce 3D models from sets of digital photographs. In practice, we found the ReconstructMe pipeline to provide the simplest method for quickly generating suitably accurate 3D models of common objects such as toys, props and puppets.

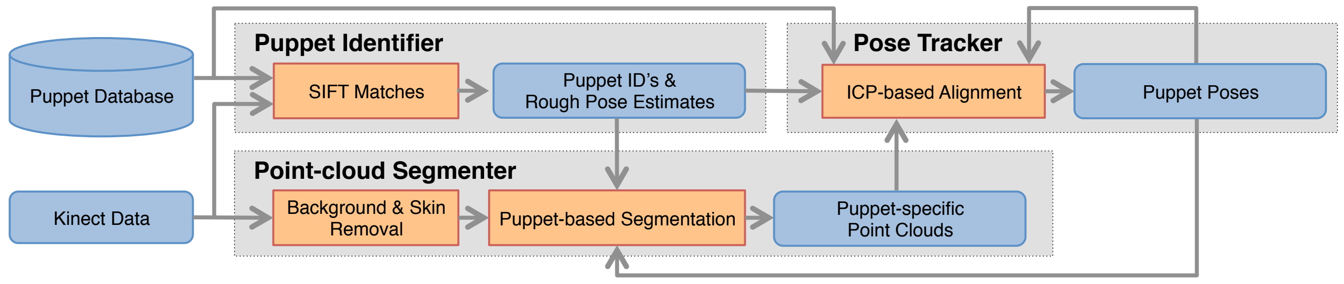

Figure 3. Overview of our system's Capture module. With each frame, the Kinect provides an RGB image and depth map. The puppet identifier compares SIFT features in those data to the features found in a database of image templates to identify puppets and roughly estimate their poses. The RGB and depth information are also combined into point clouds, which are processed to remove the background and the puppeteer's hands, and then matched to stored 3D models using ICP to estimate the puppets' 6D poses.

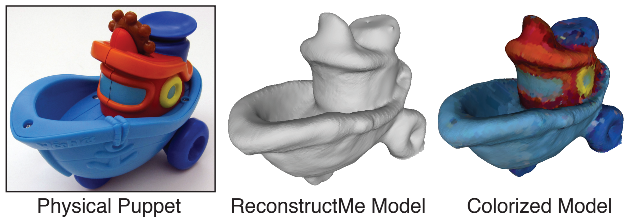

Figure 4. Overview of model-loading using ReconstructMe. Left: A photograph of the toy to be scanned. Middle: An untextured 3D model of the toy produced by ReconstructMe. Right: The same model, with vertex colors assigned using our system.

In addition to the 3D model, our Puppet Database includes roughly 30 color and depth template images for each puppet, along with the pose of the puppet in each image. Our system relies on these templates to detect physical puppets in the Kinect's field of view. Thus, they must sample the range of possible orientations and positions of the puppet throughout the Kinect's field of view. We capture these images with our Capture module in *Template-capture mode*. As with Painting mode, Template-capture mode only tracks one puppet. The puppeteer places the puppet in the Kinect's field of view and the system detects its pose. The puppeteer then presses a key to save the current image, depth map, and puppet pose and repeats this process for several unique poses throughout the Kinect's field of view. We have found that roughly 30 template images spaced in a 2x3 grid with 4-5 orientations in each position is sufficient for accurate puppet tracking. Cropped versions of example image templates can be seen in the bottom two rows of Fig. 5.

Skin Model

Our system uses the 3D point cloud from the Kinect to recover each puppet's 6D pose. However, to ensure accuracy, it needs point clouds that only contain points belonging to the puppets. Therefore, the Capture module must filter out any points that belong to the puppeteer's hands. It uses a histogram-based color model of the puppeteer's skin to perform this filtering. Our system creates this skin model once for each puppeteer as part of Setup.

The color model divides a 3D RGB color space into 96x96x96 bins. The puppeteer uses the Kinect to capture color images of his/her hands, and the system places the skin pixels into the color bins. If the number of pixels assigned to a bin exceeds a preset threshold, the system labels the color range associated with that bin as skin. The result is a tight volume of colors that belong to the puppeteer's skin. This choice of color model is powerful because it can handle arbitrary, non-linear boundaries. However, like any other color-based model, it prevents the use of puppets with colors that are very similar to the puppeteer's skin color.

CAPTURE

Figure 3 gives an overview of the Capture module, which we split into three components: a Puppet Identifier, a Point-cloud Segmenter, and a Pose Tracker. The inputs to the Capture module are the Puppet Database and Kinect data. The Kinect provides color images and depth maps, which are converted to 3D point clouds at 30 frames per second. For a given frame, the Puppet Identifier and Point-cloud Segmenter run first, and then send their data to the Pose Tracker, which determines the pose of each puppet.

Puppet Identifier

The Puppet Identifier serves two main functions: (1) it identifies puppets within the Kinect's field of view, and (2) it roughly estimates their poses as an initialization for the Pose Tracker. The Puppet Identifier runs continuously, but its output is only used in two situations: when a physical puppet first enters the field of view or when the accuracy of the system's pose estimate for that puppet falls below a threshold and tracking must be restarted. See the Pose Tracker module for more details.

Matching Image Features. To identify puppets within the Kinect's field of view, we first find scale-invariant feature transform (SIFT) [23] features within the template images in the Puppet Database. We use SIFT features because they reveal distinctive 2D image elements that are invariant to translation, rotation, and scale. Our implementation uses Sift-GPU [33] to achieve high-performance. We compute the SIFT features for the templates once as a pre-process when we first capture the template images in the Setup module.

During a performance, we compute SIFT features for each color image from the Kinect (or "RGB frame") and compare them to the pre-computed features of each template image. Given the set of feature matches for each template im-

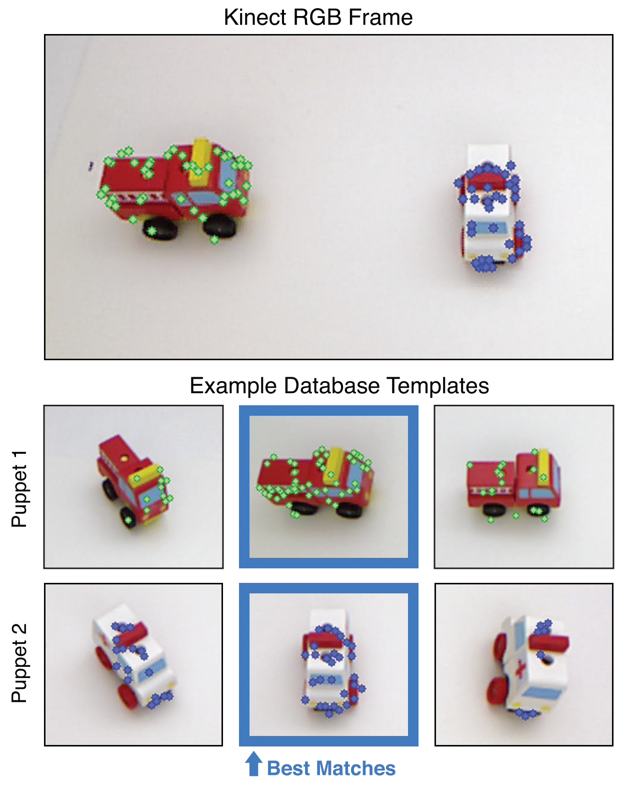

Figure 5. SIFT-based puppet identification. The RGB frame contains two physical puppets that our system has matched to the template images below it. We use color-coded dots to indicate the coordinates of the matched SIFT features. For each puppet, the best-matched template contains the most SIFT matches to the frame.

age, we use Lowe's [23] Hough-transform-based approach to find the subset of that are consistent with a single rigid transformation to the RGB frame. This procedure removes unreliable matches between the RGB frame and each template. Then, for each puppet in the database, we determine the best-matching template image based on the remaining feature matches. We eliminate templates with fewer than 20 feature matches and this usually leaves us with one template match (from the entire database) for each physical puppet in the Kinect's field of view (Figure 5). In some cases this procedure finds more than one best-matching template, each corresponding to a different puppet in the database, for a single physical puppet. In these instances we select the template with the highest ratio of matched features to total features in the template.

Rough Pose Estimation. We compute a rough pose estimate for each physical puppet that we match in the database. Both the template image and RGB frame have corresponding depth maps, which we use to recover the 3D coordinates of each pair of matched features. We then compute the optimal translation between the two sets of matching coordinates. We found that computing the full 6D rigid transform between the points using Horn's method [14] gave unreliable estimates of the rotation component. Instead, we compute just the 3D translation between the points and concatenate this with the transformation stored with the template to give a rough estimate of the physical puppet's pose. When needed, we use this estimate to label points in the Point-cloud Segmenter and to initialize the pose refinement in the Pose Tracker.

Point-cloud Segmenter

The Kinect 3D point cloud generated for each frame is undifferentiated and includes points that belong to each physical puppet as well as the background of the performance space and the puppeteer's hands. The Point-cloud Segmenter first removes the background points and the hands from the cloud and then splits the remaining points into separate clouds for each physical puppet (Figure 6).

Removing Background and Hands. To remove the background points our system captures a *background depth map* of the empty performance space each time the puppeteer first runs our system. Then, during the performance, if the depth of any point in the incoming Kinect point cloud is within 2% of the background, the Segmenter removes it from the point cloud. To eliminate points that belong to the puppeteer's hands, the Segmenter compares the color of each incoming point to the skin color model; if the color of the point matches any bin labeled as skin, the point is removed. Almost all remaining points in the cloud belong to physical puppets.

Segmenting by Physical Puppet. To further segment the point cloud, the Segmenter requires pose information for each physical puppet in the Kinect's field of view. In most cases the Segmenter uses the last known pose from the Pose Tracker. However, whenever a physical puppet is first introduced into the performance space the Pose Tracker cannot provide this pose information. In these cases we use the rough pose estimate produced by the Puppet Identifier. The Segmenter then iterates over each incoming point, computes its distance to the closest point on each visible puppet model, and assigns the point to the closest puppet. To reduce erroneous matches when two physical puppets are close to one another, if the ratio of distances between a point and two puppets is less than 1.2 (i.e., the point is about equally close to both puppets and could belong to either one) we do not assign it to any puppet.

Pose Tracker

The Pose Tracker is responsible for accurately estimating the pose of each puppet in the Kinect's field of view. It computes this estimate by aligning the segmented point cloud for a physical puppet with the corresponding 3D model in the database. The Pose Tracker refines the pose estimate for each puppet by aligning its 3D model from the database to the corresponding cloud produced by the Point-cloud Segmenter. We use the iterative closest point (ICP) algorithm [6] to compute the alignment.

The ICP algorithm aligns two sets of 3D points by 1) determining point-to-point correspondences between the sets, 2) calculating the least-squares optimal rigid transformation between the corresponding points [14], 3) transforming one of the sets to better align the points, and 4) iterating until the fractional reduction in error between iterations drops below a threshold. In our ICP implementation, one set of points is the segmented point cloud for a puppet and the other set is comprised of the vertices of its 3D model (Figure 7). Because

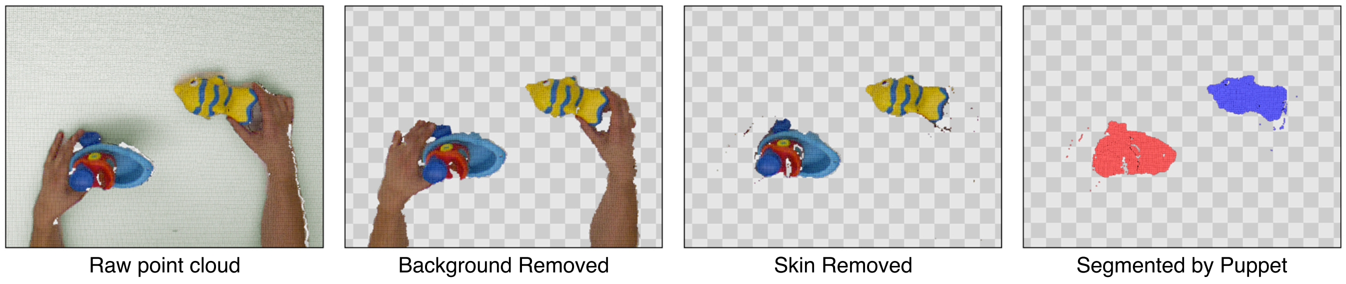

Figure 6. Steps of point-cloud segmentation. Beginning with the raw point cloud from the Kinect, our segmenter removes the background and the puppeteer's skin, and finally produces separate point clouds associated with each puppet.

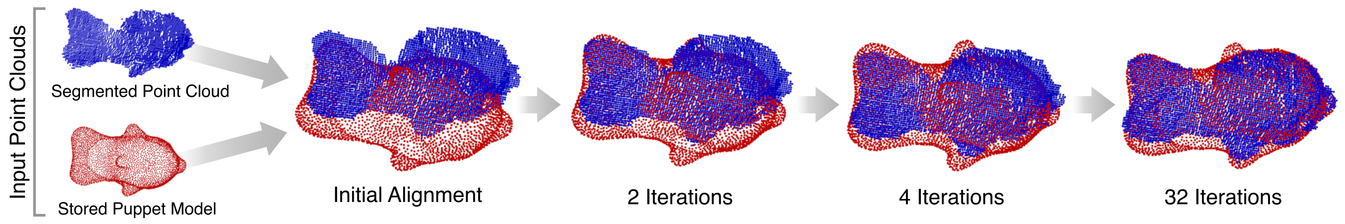

Figure 7. Progression of ICP alignment between a segmented point cloud and a Puppet Database model. Our ICP algorithm begins by transforming the puppet model using its last known pose (if available; otherwise it uses the rough pose estimate from the Puppet Identifier). Then each ICP iteration brings the clouds into closer alignment until the root mean square distance between their points meet an error threshold.

ICP is a greedy, local optimization algorithm, good pose initialization is critical both for finding the global optimum and for fast convergence. To initialize ICP in a given frame, we transform each puppet's 3D model using its pose from the last frame, if it exists. Otherwise we use the rough, SIFT-based pose estimate produced by the Puppet Identifier. Joung et al. [19] use a similar SIFT-based initialization for ICP in the context of 3D environment reconstruction.

Detecting Bad Poses After ICP finishes, we count the number of points in the segmented point cloud that do not have corresponding points in the puppet model. If this number accounts for more than 33% of the points in the segmented point cloud, we assume that we have an incorrect pose estimate. If a puppet fails this test for 20 frames in a row, then we consider the puppet "lost" and wait for a new rough pose estimate from the Puppet Identifier to restart tracking.

Real-time Performance. The point-to-point correspondence search is the main bottleneck of the ICP algorithm. Our implementation uses a kd-tree to accelerate this step of the algorithm. To ensure real-time performance, we also downsample the segmented point cloud as well as the 3D model points. For the segmented point clouds, we keep every n^{th} point, where n is the downsampling rate. For the 3D models, we use Poisson disk-sampling as implemented by Cline, et al. [8] to evenly reduce the number of points. We have empirically found that downsampling to roughly 1000 points for each segmented cloud and 5000 points for the 3D models achieves real-time tracking with minimal loss in accuracy.

Filtering. Sensor noise and imprecise database models can produce noticeable wobble in the tracked poses. To reduce the amount of wobble, we apply separate bilateral filters to the translation and rotation components of the puppet poses. The bilateral filter is an edge-preserving, smoothing filter that is commonly used to reduce noise while preserving strong changes in a signal [30]. It computes a weighted average of the input signal over a small neighborhood of samples to produce each output value. The weights are dependent on both the domain and range of the input signal. In our implementation, we use the last ten pose estimates for a puppet as the sample neighborhood. Since the poses change as a function of time, we treat time as the domain of the signal and either the translation or rotation component as the range. We represent the rotations as quaternions for this computation. We have found that the bilateral filters reduce wobble without introducing noticeable lag or damping intentional puppet motions.

Partially Occluded Puppets. We have found that the Pose Tracker typically requires that the segmented point cloud contain at least 125 points to produce robust pose estimates. However, during a performance, if the puppeteer partially occludes the Kinect's view of a puppet with her hands or another puppet, the size of its point cloud may fall under this threshold. In these cases we simply use the last known pose of the puppet, essentially leaving it in place. In practice we have noticed that this rule is rarely necessary, but that it significantly improves the tracker's robustness to occlusions in the few instances that puppets are mostly occluded.

Entrances and Exits. The segmented point clouds also fall below the 125-point threshold when physical puppets are entering or exiting the performance space. In this case, the thresholding approach we use to handle partially occluded puppets would create artifacts; a puppet exiting the performance space would have its 3D model permanently fixed to the last position in which its point cloud contained at least 125 points. To solve this problem, we add an additional check.

We compute the angle between the Kinect's optical axis and the ray from the Kinect's optical center to the centroid of the puppet's point cloud. If that angle lies within 2.5 degrees of an edge of the system's trackable volume, or the centroid lies within 5cm of the nearest depth detectable by the Kinect, and the puppet's point cloud contains fewer than 125 points, it is labeled as "lost" and not rendered. Our system labels a puppet as "found" only if the Puppet Identifier detects it and its point cloud contains more than 125 points.

Rendering

Our OpenGL-based renderer uses the poses from the Capture module to render each puppet model within a 3D virtual set. It operates in real time and provides visual feedback to the puppeteer during a performance. To allow control over the composition and appearance of a shot, we include 6D virtual-camera and lighting controls. We also provide entrance and exit effects to warn the puppeteer before he/she moves a physical puppet outside the performance space. Finally, we include layered animations to help puppeteers perform animations with more than two simultaneously moving puppets.

Camera Controls. Our 3D animation environment enables 6D virtual-camera control, which aids both the composition and performance of animations. For instance, the puppeteer may capture an animation with the virtual camera at one location, but decide during playback that another vantage point provides a more interesting view of the story. During performance, however, we have found that puppeteers usually place the virtual camera to mimic their point of view of the physical performance space. This setup aids performance by helping the puppeteer avoid difficult mental transformations between the physical space and the rendered scene.

Lighting Controls. Our system allows the puppeteer to adjust the illumination of the virtual set, including the position of a light source, the strength of the ambient and diffuse light levels, and whether or not to render shadows. Puppeteers can switch between day and night scenes by adjusting the strength of the ambient and diffuse components. They can also adjust the light position to cast dramatic shadows on the faces of the virtual puppets.

Entrance-and-exit Effects. The trackable volume of our system is typically smaller than the virtual set. As a result, puppets entering or leaving the trackable volume can spontaneously appear or disappear within the virtual set. Such events can distract from the story. Careful placement of the virtual camera can avoid capturing abrupt transitions, but then the puppeteer's scene-composition options become limited. We address the issue by adjusting a puppet's transparency based on its proximity to the edge of the performance volume. Similar to our process for determining whether we have lost tracking of a puppet, we compute the angle between the centroids of the puppets and the Kinect's optical axis. If that angle lies within five degrees of any side of the performance volume, the puppets are rendered with decreased opacity. We apply the same effect for puppets that move too close to the Kinect sensor for detection. Once a puppet comes within one degree of leaving the side of the volume or within 5cm of leaving the top of the volume, we render it completely transparent. This process renders the puppets invisible before the Pose Tracker labels them as lost. We find this rendering approach to be less distracting to a story than spontaneous entrances and exits. It can also help puppeteers set up animations. The puppeteer can slowly move a puppet to the edge of the tracking region until it is just invisible but still tracked, and then leave it in place. The puppeteer then knows exactly where the puppet will appear if it re-enters the set.

Layered Animations. A key benefit of our system is that it allows the puppeteer to concentrate on the performance, rather than the interface. However, some stories can be difficult to perform. For instance, a story may require three or more puppets to move at the same time. To aid these kinds of performances, we implemented layered animations, which allow the puppeteer to perform a single story using multiple capture sessions. As the puppeteer adds a new layer to a story, we simultaneously play back the previously captured sessions. This mode of performance loses kinesthetic feedback between puppets in different layers, but can be useful for performances with more than two puppets.

RESULTS

We created four animations to demonstrate our tracker, camera and lighting controls, 3D performance space, and layered-animation tools. "Apathetic Ambulance" (Figure 2), "Pool Intruder" (Figure 8), and "Traffic Fight," (Figure 8) were created by the first author, while "Ghost Duck" (Figure 8) was created by a novice user. We also conducted a user study wherein five novices (male and female, 10-30 years old) created animations using our system. All of the animations are on our project website[1]. We first describe the four initial animations and then present the user study.

Animation Details

In "Apathetic Ambulance," two cars crash into each other, and then a fire truck crashes into the police car that has arrived to help. An ambulance arrives, surveys the scene, and decides to go home. The puppeteer made use of physics in this animation to push the physical puppets into each other during the crashes. Our system tracked those motions in real time and reproduced the physical interactions between the puppets in the rendered result. The puppeteer also used the transparent rendering of puppets near the edge of the tracking volume. He placed the virtual fire truck, police car, and ambulance on the roads in the virtual set, and then moved them out until they were rendered invisible, but still tracked by the system. This setup made him confident about where in the virtual set the puppets would appear as they entered the animation. The puppeteer needed five takes to record the animation. He threw out one recording because the system did not accurately track the ambulance model. He discarded the other attempts because he did not like his physical performance (e.g., the motion of the ambulance puppet was too slow).

In "Pool Intruder," a duck goes to take a swim in his pool, but encounters a big fish who refuses to let him in. The duck recruits the help of a wheeled boat, which drags the fish away.

[1]http://vis.berkeley.edu/papers/3dpuppet

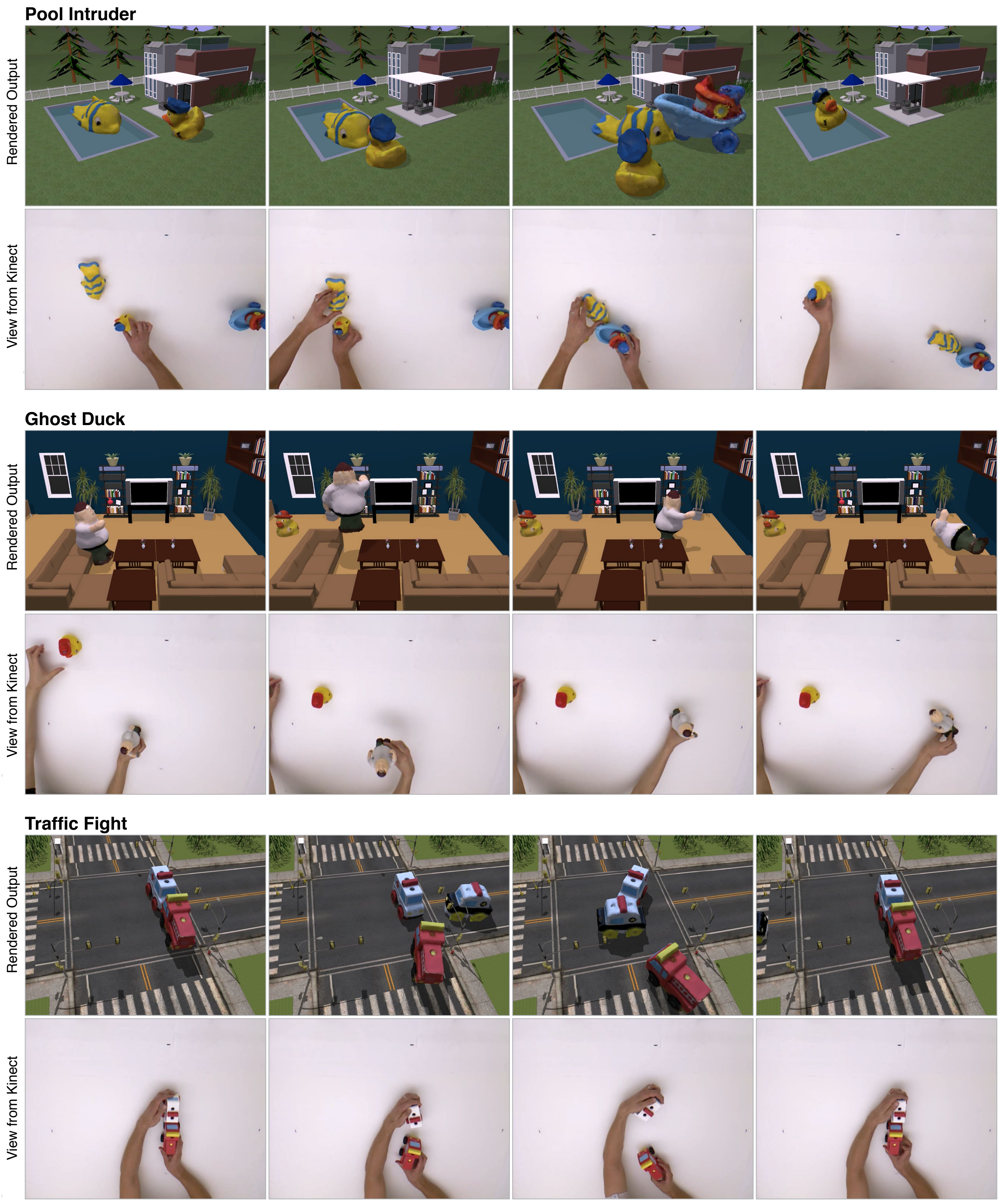

Figure 8. Summaries of three animations created with our system. For each animation, the bottom row shows the view from the Kinect sensor and the top row shows the rendered output of our system. "Traffic Fight" consists of two performances layered together; we include the raw input of only one of them.

Finally, the duck enjoys his pool. The puppeteer needed three takes to produce the final animation, after not being satisfied with the pacing of the first try or the relative positions of the puppets in second try.

In "Ghost Duck," a man enters his living room and goes to his couch. A ghost duck then enters the room and surprises the man, who tries to run away but hits a wall and falls down. The puppeteer creatively applied our system's variable-opacity rendering to give the duck a ghost-like appearance. To show the man's surprise, he made the puppet jump up and down, demonstrating how one can convey emotion even with rigid puppets. The jumps demonstrate how puppeteers can use the full 3D performance space to tell their stories. The puppeteer created this animation after using our system for 40 minutes, including the five minutes we used to explain its controls.

"Traffic Fight" demonstrates our system's layered-animation feature. An ambulance and fire truck fight in the middle of an intersection, move aside as a police car passes through, and then resume fighting. The animation uses three puppets that move simultaneously, and would have been impossible to perform in one take by one puppeteer. In this case, the puppeteer created two layers. In the first layer, he performed the motion of the police car. He then played it back and created a second layer with the fire truck and ambulance. Our system then combined the two layers into one story. This example demonstrates how layered animations rely heavily on the real-time performance of our system. The puppets in each layer interacted with one another, so the puppeteer had to continually monitor the earlier animation and adjust his second performance on the fly to make the interactions believable.

Implementation

We implemented our system with the Robotics Operating System [32], OpenNI Kinect driver [26], and SiftGPU software package [33]. Our system runs on a desktop PC with a 24-core, 3.33GHz Xeon CPU, 12GB RAM, and an NVIDIA GTX 580 video card.

For our physical setup, we place a Kinect sensor 75cm above a tabletop, which we use for a performance space. We orient the Kinect so its optical axis makes a 22.5-degree angle with the tabletop's surface normal. This setup provides a trackable performance volume of roughly 90cm by 60cm by 30cm. We found that this setup maximized the useful performance space while maintaining enough density in the point cloud to allow accurate tracking. While moving the Kinect further away from the table top increases the size of the performance volume it also leads to larger gaps between samples in the point cloud and degrades tracking. Our setup also allows the tabletop to serve as a physical proxy to the virtual ground plane. Thus, a puppeteer can rest a puppet on the tabletop and keep it within the animation while he/she manipulates other puppets. Finally, we chose an oblique view because it usually captures more of the distinguishing, frontal features of the physical puppets than a view pointed straight down.

USER EXPERIENCE

We observed five novice puppeteers (male and female, 10-30 years old) while they produced their first animations with our system. We began by helping each puppeteer create a skin color model. We then took about five minutes to explain the keyboard and mouse controls for changing camera, lighting, and background settings. After this introduction, we asked the puppeteers to play with our system and, when ready, to record a simple story.

The puppeteers began by exploring the capabilities of our system. They put physical puppets in the performance space and watched how quickly our system could detect them. Next, they moved the physical puppets throughout the performance space to test how well our tracker could follow them. They also cycled through the seven virtual sets we provided to explore how characters could interact within them. For instance, multiple puppeteers tried placing puppets under covered bridges and making them hide behind buildings. Three of the puppeteers expressed excitement when they first saw the virtual characters moving in response to their physical actions. In praising its simplicity, one claimed that he could have used it without any introduction.

While playing with our system, puppeteers learned not to occlude too much of each puppet with their hands. Otherwise, the point clouds could become too small for accurate pose detection, and our system would either render a puppet in the wrong pose or stop rendering it altogether. They also learned how quickly they could move each puppet without losing tracking. As we discuss in the Limitations Section, the maximum speed our system can track varies by puppet.

The lack of collision detection confused some puppeteers. They were surprised to see that puppets could pass through walls. We chose to omit collision detection between the virtual puppets and the objects in the virtual sets from our system because handling such collisions would require breaking the one-to-one, physical-to-virtual mapping of our interface. If a virtual wall stopped the motion of a puppet, then the virtual and physical positions of that puppet would no longer correctly match.

On average, our puppeteers took 16 minutes to play with the system before trying to record their first animations. Once they were ready to create an animation, we showed them how to record and export their own performances, and how to produce layered animations. While formulating their stories, they continued to play with our system to plan the motions of the puppets within the virtual sets. We found that first-time users typically needed three or four recording attempts to produce an animation that they liked, with more complex stories taking longer to produce. Puppeteers either discarded animations due to performance errors (e.g., the performance was too slow or a physical puppet's motion did not quite match the puppeteer's intention), or tracking errors. Practice with our system helped reduce the latter issue, as puppeteers learned to avoid motions that were too fast for our tracker.

The final animations produced by our novice puppeteers made full use of the capabilities of our system, including

6D puppet motions, camera and lighting controls, and layered animation. They typically adjusted the camera before performing the motions to set the view so that it resembled their view of the physical performance space but also ensured that all subsequent puppet motions would be visible. They also used camera controls to frame the output movie. One puppeteer chose to change the lighting parameters after she finished her performance. Once we explained our system, each puppeteer took less than one hour to record a story, with some only needing 30 minutes.

LIMITATIONS

Our system is primarily limited by the puppets and puppet motions it can track.

Our system only tracks rigid puppets; the Capture module does not account for articulated or deformable puppets. We also cannot track puppets with sizes much smaller than those in our example animations (the smallest being roughly 7cm x 8cm x 8cm). Smaller puppets do not produce enough unique SIFT features for identification. Puppets that pass the size limitation must also be unique in shape and texture so that they are identified and tracked properly. If two puppets share many of the same SIFT features, our Puppet Identifier can generate false matches. Finally, radially symmetric puppets produce errors during ICP alignment, as multiple orientations provide equally optimal alignment solutions. As a result, the output pose becomes more likely to rapidly jump between the optima and generate temporally inconsistent pose estimates.

Fast puppet motions can also produce tracking errors. We have found that the 3D shape of a puppet has a significant effect on the types of motions it can undergo without error. For instance, the duck puppets seen in our sample animations are the most prone to rotational errors due to their roughly radial symmetry. In general, geometrically simple puppets are more susceptible to tracking errors. For instance, the top of the Lamborghini puppet in the "Apathetic Ambulance" animation is almost completely flat. If that puppet moves quickly in a direction tangential to its roof, the optimal ICP-derived pose can slip and produce a mismatch between the physical and virtual poses. Thus, the puppeteer must take more care with the motions for that puppet.

CONCLUSION AND FUTURE WORK

We have presented a new interface for producing 3D animations by performing with physical puppets in front of a Kinect depth sensor. The system's direct, natural mode of input and simple hardware requirements make it approachable to users of any skill level. We allow puppeteers to tell stories by playing with toys and other physical props, which makes the method of interaction familiar and entertaining. Once puppeteers understand its limitations, the design of our system also allows puppeteers to focus on their physical performance, rather than the interface.

While puppeteers enjoy producing animations with our system, we found that they enjoy free-form play almost as much. In particular, they enjoy seeing how the physical puppets' motions translate into the context of the different virtual sets. This observation suggests other modes of play, for instance where the puppets could explore the virtual world and trigger preset animations, similar to the work in Lego OASIS [34].

Though we have demonstrated its utility for simple 3D animations, our system may be extended in a number of ways:

Integrated Puppet Loading. Our current implementation relies on third-party software like ReconstructMe [13] or 123D Catch [1] to provide 3D models for each puppets, followed by manual capture of the SIFT template images. A more streamlined implementation would combine 3D scanning and template-capture into one process, which could be incorporated directly into our interface.

Articulated Puppets. Our example stories demonstrated how puppeteers could impart emotion onto rigid puppets. However, articulated puppets would provide more expressiveness and widen the range of puppets available to puppeteers. This improvement would require substantial changes to our Puppet Identifier and Pose Tracker in order to handle every possible posture for each puppet.

Deformable Puppets. Our system currently cannot track soft, deformable puppets, including most stuffed animals. In order to track such puppets, we would need to modify our ICP algorithm, which assumes that the shape of a physical puppet does not change throughout a performance. The modification might include deformation models for the ways in which the shape of each puppet can deform.

Multiple Puppeteers. Currently we can easily handle multiple puppeteers using our system at the same time, as long as their skin tones closely match. We might also handle multiple skin tones by combining the skin model for each puppeteer. Beyond multiple puppeteers in one setting, we also envision remote, collaborative puppeteering sessions using our system.

ACKNOWLEDGMENTS

We would like to thank Dieter Fox for providing valuable insight into our tracking algorithm, Evan Herbst and Peter Henry for sharing crucial code, and Nicholas Kong for his help in making the video. We also thank the paper reviewers for their helpful suggestions. This work was supported by NSF grant CCF-0643552 and a gift from the Intel Science and Technology Center for Visual Computing.

REFERENCES

1. Autodesk. 123D Catch. http://www.123dapp.com/catch.
2. Autodesk. 3ds Max. http://usa.autodesk.com/3ds-max/.
3. Autodesk. Maya. http://usa.autodesk.com/maya/.
4. Avrahami, D., Wobbrock, J. O., and Izadi, S. Portico: tangible interaction on and around a tablet. In *Proc. UIST* (2011), 347–356.
5. Barnes, C., Jacobs, D. E., Sanders, J., Goldman, D. B., Rusinkiewicz, S., Finkelstein, A., and Agrawala, M. Video Puppetry: A performative interface for cutout animation. *ACM TOG (Proc. SIGGRAPH) 27*, 5 (2008), 124:1–124:9.
6. Besl, P., and McKay, N. A method for registration of 3-D shapes. *IEEE PAMI 14* (1992), 239–256.

7. Blender Foundation. Blender. http://www.blender.org.

8. Cline, D., Jeschke, S., White, K., Razdan, A., and Wonka, P. Dart throwing on surfaces. *Computer Graphics Forum 28*, 4 (2009), 1217–1226.

9. Dontcheva, M., Yngve, G., and Popović, Z. Layered acting for character animation. *ACM TOG (Proc. SIGGRAPH) 22* (2003), 409–416.

10. Freedman, B., Shpunt, A., Machline, M., and Arieli, Y. Depth mapping using projected patterns. Patent. US8150142 (2012).

11. Gallo, L., Placitelli, A., and Ciampi, M. Controller-free exploration of medical image data: Experiencing the kinect. In *Proc. CBMS* (2011), 1–6.

12. Google. Google 3D Warehouse. http://sketchup.google.com/3dwarehouse/.

13. Heindl, C., and Kopf, C. ReconstructMe. http://reconstructme.net.

14. Horn, B. K. P. Closed-form solution of absolute orientation using unit quaternions. *JOSA A 4*, 4 (1987), 629–642.

15. Iason Oikonomidis, N. K., and Argyros, A. Efficient model-based 3D tracking of hand articulations using kinect. In *Proc. BMVC* (2011), 101.1–101.11.

16. Ishii, H., and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. In *Proc. CHI* (1997), 234–241.

17. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., and Fitzgibbon, A. Kinectfusion: real-time 3D reconstruction and interaction using a moving depth camera. In *Proc. UIST* (2011), 559–568.

18. Johnson, M. P., Wilson, A., Blumberg, B., Kline, C., and Bobick, A. Sympathetic interfaces: Using a plush toy to direct synthetic characters. In *Proc. CHI* (1999), 152–158.

19. Joung, J. H., An, K. H., Kang, J. W., Chung, M. J., and Yu, W. 3D environment reconstruction using modified color ICP algorithm by fusion of a camera and a 3D laser range finder. In *Proc. IROS* (2009), 3082–3088.

20. Kato, H., Billinghurst, M., Poupyrev, I., Imamoto, K., and Tachibana, K. Virtual object manipulation on a table-top AR environment. In *Proc. ISAR* (2000), 111 –119.

21. Klemmer, S. R., Li, J., Lin, J., and Landay, J. A. Papier-mâché: Toolkit support for tangible input. In *Proc. CHI* (2004), 399–406.

22. Lee, G. A., Kim, G. J., and Billinghurst, M. Immersive authoring: What you experience is what you get (wyxiwyg). *Comm. ACM 48*, 7 (2005), 76–81.

23. Lowe, D. G. Object recognition from local scale-invariant features. In *Proc. ICCV* (1999), 1150–1157.

24. Numaguchi, N., Nakazawa, A., Shiratori, T., and Hodgins, J. K. A puppet interface for retrieval of motion capture data. In *Proc. SCA* (2011), 157–166.

25. Oore, S., Terzopoulos, D., and Hinton, G. E. A desktop input device and interface for interactive 3D character animation. In *Proc. Graphics Interface* (2002), 133–140.

26. OpenNI Organization. OpenNI. http://openni.org.

27. Schmidt, R., and Singh, K. meshmixer: an interface for rapid mesh composition. In *ACM TOG (Proc. SIGGRAPH)* (2010), 6:1.

28. Shotton, J., Fitzgibbon, A., Cook, M., Sharp, T., Finocchio, M., Moore, R., Kipman, A., and Blake, A. Real-time human pose recognition in parts from single depth images. In *Proc. CVPR* (2011), 1297 –1304.

29. Sturman, D. J. Computer puppetry. *IEEE Computer Graphics and Applications 18*, 1 (1998), 38–45.

30. Tomasi, C., and Manduchi, R. Bilateral filtering for gray and color images. In *Proc. ICCV* (1998), 839 –846.

31. Wang, R. Y., and Popović, J. Real-time hand-tracking with a color glove. In *ACM TOG (Proc. SIGGRAPH)* (2009), 63:1–63:8.

32. Willow Garage. Robotics Operating System. http://ros.org.

33. Wu, C. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). http://cs.unc.edu/ ccwu/siftgpu, 2007.

34. Ziola, R., Grampurohit, S., Landes, N., Fogarty, J., and Harrison, B. Examining interaction with general-purpose object recognition in LEGO OASIS. In *Proc. IEEE VL/HCC* (2011), 65–68.