Blair MacIntyre
blair@cc.gatech.edu

Marco Lohse
marco@cc.gatech.edu
GVU Center
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA

Jay David Bolter
jay.bolter@cc.gatech.edu

Emmanuel Moreno
motiv-8@mindspring.com
School of Literature,
Communication and Culture
Georgia Institute of Technology
Atlanta, GA, USA

Integrating 2-D Video Actors into 3-D Augmented-Reality Systems

Abstract

In this paper, we discuss the integration of 2-D video actors into 3-D augmented-reality (AR) systems. In the context of our research on narrative forms for AR, we have found ourselves needing highly expressive content that is most easily created by human actors. We discuss the feasibility and utility of using video actors in an AR situation and then present our Video Actor Framework (including the VideoActor editor and the Video3D Java package) for easily integrating 2-D videos of actors into Java 3D, an object-oriented 3-D graphics programming environment. The framework is based on the idea of supporting tight spatial and temporal synchronization between the content of the video and the rest of the 3-D world. We present a number of illustrative examples that demonstrate the utility of the toolkit and editor. We close with a discussion and example of our recent work implementing these ideas in Macromedia Director, a popular multimedia production tool.

1 Introduction

Augmented reality (AR) has been put forward as a powerful tool for education and training (for example, in architecture (Feiner, Webster, Krueger, MacIntyre, & Keller, 1995; Webster, Feiner, MacIntyre, Massie, & Krueger, 1996) and medicine (State et al., 1996)). A particularly compelling attribute of AR (when presented on personal displays, such as see-through, head-worn displays) is the ability to customize an experience to each individual viewer. In our own work, we are investigating how to incorporate nontrivial narrative content into AR environments, with a particular focus on education, training, and entertainment. We believe that video content, especially video-based actors and agents, can be an important element in such narratives, but only if the video actors can interact with the physical and virtual environment. For example, if we have a video of a human narrator, this figure will often need to point at, or otherwise interact with, other virtual objects and the physical environment being augmented.

Using 2-D images in place of 3-D objects is a common technique in 3-D graphics. Many VR systems replace complex objects such as trees with images. These images are typically placed on rectangles and given a "billboard" property: they are always oriented to face the viewer (like an ideal billboard). Similarly, many VR and graphics systems have placed video of varying quality into the world using similar texture-mapping techniques. For example, Criterion Software's RenderWare has allowed simple animated GIF images to be used as video textures for many years, and some high-end graphics hardware have supported the use of high-quality video as textures. Fortunately, it is now possible to use a reasonably

Presence, *Vol. 11, No. 2, April 2002,* 189–202

sized video as a texture on mainstream PCs; we can simultaneously render (from Java) multiple 256 × 256 pixel, fifteen-frames-per-second videos into texture memory on all of the Windows 2000-based PCs, and many of the laptop computers, in our lab.

Given that it is feasible to place video into an AR environment, we need easy-to-use tools that allow programmers and content creators to integrate 2-D video actors into a 3-D narrative in a straightforward manner. For flexibility, we are using AR testbeds built both in Java, on top of Java 3D, and Macromedia Director, using Shockwave 3D. Our work is motivated by programs like Director (that allow authors to assemble media—such as video, still images, and text—into an interactive screen-based environment) and by research systems such as Alice (Pausch et al., 1995) (that provide high-level tools for nonprogrammers to create VR environments). Building on the ideas put forth by such systems, we are designing authoring tools for assembling existing 2-D video and audio together with 3-D models into an AR environment.

Our aim is to create a collection of high-level design tools embedded in a general purpose programming environment that combines the power of systems like Alice and Director. We are prototyping such tools in both Java and Director because neither is ideal for our purposes. On one hand, Java (and Java 3D or OpenGL) is a rich and powerful programming environment, allowing us to test ideas that would not be possible in Director. On the other hand, the designers with whom we work are very familiar with Director (and its Lingo programming language), so we are enabling them to prototype AR experiences by extending Director to support 3-D AR. In either case, we do not need low-level editors for the various content elements; the designers we work with already have tools for editing video (that is, nonlinear editing systems) and 3-D objects (such as 3-D modeling and animation packages).

In this paper, we report on our first step toward the creation of our tool suite, the Video Actor Framework for Java, and the subsequent reimplementation of parts of this package in Director. The Java-based framework consists of the VideoActor editor and the accompanying Video3D runtime package. The editor provides support for nonprogrammers to specify the interactions between activities in the video and other elements of the 3-D AR environment. The tools are designed specifically to support video actors (although they can be used to integrate video of scenery or equipment). The central concept underlying the editor (and, therefore, the Video3D package) is the direct specification of a set of semantically meaningful synchronization points at keyframes throughout the video. When this specification is loaded into a Java program, a Video3D object is created that displays the video and automatically converts the 2-D synchronization points and paths to their equivalent in 3-D as the video plays. The Video3D package allows programmers to request notification of synchronization points when the keyframes are reached and to use these notifications as input to the rest of the program.

In section 2, we briefly discuss our ongoing work on building narrative AR experiences to set the context for the Video Actors work. In section 3, we present the narrative and software issues related to the integration of 2-D video into a 3-D programming environment. In section 4, we discuss the VideoActor editor and the Video3D objects and present some illustrative examples of these tools. In section 5, we briefly highlight the reimplementation of the Video Actor runtime in Director, before closing with a discussion of related and future work.

2 AR Narratives

The goal of our work on AR narrative experiences is to simultaneously develop the narrative theory to understand how to design these experiences and to build the authoring and runtime tools that are necessary to implement our designs (MacIntyre, Bolter, Moreno, & Hannigan, 2001). Our method is to approach AR as a new medium and examine it using media theory as one would any other medium (such as virtual reality, film, or stage). In particular, we focus on *remediating* (Bolter & Grusin, 1999) established media in an attempt to understand and develop media forms for AR. (Media forms can be thought of as sets of conventions and design elements that can be used by authors and developers to create meaningful information experiences for these users.) Media studies teaches us that remediation

is a critical tool because we never design in a vacuum, even when designing for a new medium. A user's expectations are (implicitly and explicitly) based on their experience with, and understanding of, all media forms; a lifetime of experiencing film, stage, television, and so on creates a starting point for the interpretation and understanding of any new experience. Understanding, and leveraging, these shared cultural expectations of the intended audience will allow us to create richer, more-engaging, and more-understandable AR experiences. Video and film are tremendously popular media, with a long and rich narrative history; by incorporating video into AR, we hope to import this rich tradition, while at the same time extending and refashioning it by allowing video actors to interact with the environment in various ways (Bolter & Grusin, 1999).

Developing and understanding media forms is also fundamental to the eventual success of a new medium such as AR at a more basic level; by understanding common elements and conventions, we can begin to develop tools that explicitly support them, significantly easing AR content creation. A recent example of this process is the rapid development of HTML tagging and other Web technologies after 1993, when graphic designers began to work on Web sites. The introduction of the inline image tag, for example, made the Web page a new media form that could be modeled on graphic design for print. Graphic designers then began to push on this new media and lead those responsible for standards to develop new versions of HTML and new markup languages (such as cascading style sheets) to provide them with the tools they needed. This acceleration of tool development is a normal part of the development of any new medium, and it happens as a matter of course once the low-level technology is practical and relatively widely available. The Video Actor Framework discussed in this paper is the first example of the tools we hope to develop through this exploration.

3 Video Actors

We could incorporate humanoid actors into a 3-D environment in one of two ways: create a 3-D model of characters and animate them, or create 2-D video of an actor playing the character and texture map the video onto a rectangle in the virtual world. Although animated 3-D models are potentially much more flexible, they are very difficult to create, and the resulting animated character rarely possesses the same qualities as a character acted out by a reasonably good actor (that is, in terms of body motion, facial expressiveness, and overall appearance). In the context of experiences designed to evoke emotional reactions, such as tours of historical sites, actors in video form can help bridge the gap between the computer-generated and the physical environment. An actor dressed as a wounded soldier at a war memorial is far more likely to emotionally involve the viewer than an animated version of that soldier.

However, 2-D video actors are not appropriate for all possible characters in an AR narrative. First, video actors must be filmed beforehand, so the range of interactivity is limited. Second, the video needs to be on a billboarded polygon (that is, oriented perpendicular to the line of site) to look correct. The first point is not a significant restriction: creating animations that react to, and interact with, a live human in a natural and believable way is extremely difficult, and still a topic of research. Thus, even if we were using animated 3-D characters, it is likely that many of their movements would be scripted ahead of time. The second point (that the video needs to face the viewer) is more serious. If the video is mapped onto a polygon that is not billboarded to face the viewer, the video will become distorted, with the edge of the polygon eventually becoming visible. A big attraction of AR is that it allows the user to move around freely in the environment and look at virtual objects from any angle. This freedom, however, makes it impractical to use video actors to represent real people with strict relationships to the physical world (such as people walking through a room, sitting in chairs, or talking to each other) because these characters cannot be billboarded.

Fortunately, in the context of education or entertainment, it is possible to create narratives that minimize these restrictions. Characters that directly address the viewer can be oriented to face them. Furthermore, authors can take advantage of the fact that, in these narra-

tive forms, the viewer may be willing to accept limitations on the appearance and behavior of the video actor. The stronger the narrative, the more the viewer may be willing to accept these limitations; the story (or the informational content) will help the user suspend disbelief.

For example, as viewers walk around the character that is addressing them, they may willingly "ignore" the fact that the actor is rotating in space to remain facing them. More fantastically, it may be perfectly acceptable for the guide at a historical site or a virtual personal assistant (akin to the butler in Apple's Knowledge Navigator video (Dubberly & Mitch, 1987)) to be presented as a disembodied character, floating in the air a few feet away from the viewer, who fades in and out of existence as needed. For true simulations and certain kinds of skill training, the actor might need to be more realistically constrained to the physical world. For example, if we wanted to use AR to learn how to juggle with someone else (how to pass clubs back and forth), the video actor would have to obey the laws of physics when you throw a ball at him or her.

3.1 Creating Video for Use as a Video Actor

As mentioned, when using a video actor, the video is texture mapped to a rectangle in the 3-D world. When the viewer looks at the polygon, we want the video image to look "reasonable." A few constraints must be satisfied so the video actor looks reasonable:

- The video actors should be sized predictably. Although we may want actors to appear bigger or smaller in the AR environment than they did in real life, we need to know how big they are in the video frame to easily specify their size in the 3-D world.
- The visible frame of the video should move in 3-D as it did when the video was shot. Therefore, if the actor walks across a stage, and the camera pans to follow him or her, the polygon on which the video is texture mapped must move similarly in the 3-D world. This implies we need to know the location of each video frame relative to the starting location.
- When the polygon rotates because of billboarding, the rotation should be around the center of the actor. This implies that we need to know the center of the actor in each video frame.
- The actor needs to be segmented out of the video background, so that the viewer sees a character and not a solid, rectangular video frame. This implies that the video must be created so that the background can be removed.
- Actors must realize that the audience consists of one person (the viewer wearing the AR gear). Some of the more experienced actors we have used were accustomed to stage acting. Because of their training, they acted in a way that was not appropriate for a single audience member who needs to believe the actor is in the same room, interacting directly with them.

The combination of hardware and software technologies we are using also imposed some restrictions. In particular, we cannot smoothly display video that exceeds approximately 256×256 pixels and fifteen frames per second (although we can display a few of those simultaneously). We expect some of these limitations to ease over time, especially as Java libraries are optimized and language facilities are improved to allow faster access to native data.

To satisfy the previous conditions, we made some assumptions about the video we would display and built the VideoActor tool to give the content creator control of the remaining variables. Because we cannot store video with transparency values, we assume that all videos have a solid background of a single color, which allows us to replace these pixels with transparent ones when we render individual frames onto the textures. We accomplish this by shooting video against a green screen, and keying the background to one color (black, in the examples in this paper).

We also assume that the actor is centered horizontally in the video frames, that the camera does not zoom in or out, and that, if the actor moves, the camera pans to keep him or her centered in frame. A sample frame from one of our videos (used in the "Thanksgiving" example from subsection 4.3.1) is shown in figure 1.

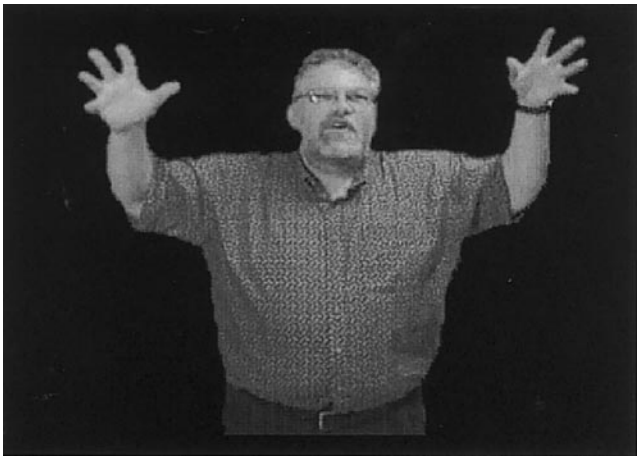

Figure 1. *To be used as a video actor, a video must satisfy a few criteria: the actor must be shot from the viewpoint of the eventual user and should remain centered in the frame, and the background should be one solid color (in this example, black) so that it can be easily removed.*

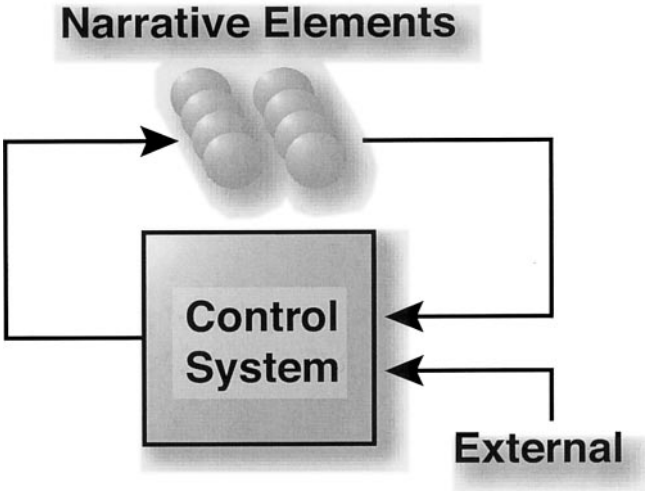

Figure 2. *An interactive story can be thought of as a control system. The controller uses both the state of the narrative elements and external inputs to control the story's progression.*

It is conceivable that some of these restrictions could be removed by using vision algorithms on the video (for example, use person-finding code and do automatic background subtraction (Wren, Azarbayejani, Darrell, & Pentland, 1997)). However, these restrictions are not overly burdensome and allow the content creators to create video that will be handled in a predictable way (an approach suggested by Pausch, Snoddy, Hazeltine, Taylor, and Watson (1996)). For example, they can include nonperson objects in the nonblack parts of the video, and they know they will be retained. Or, they can move the actor off-center to cause certain effects, if desired. Eventually, we may add vision algorithms to the VideoActor tool to automate certain tasks, such as tracking an actor's hand through the video.

3.2 Integrating Video Actors into a 3-D Testbed

Our AR testbed is built on top of Java and Java 3D, but the design of our Video Actor Framework is applicable to any object-oriented language and graphics library. Furthermore, the design of the library is independent of the particular AR technologies used; in the examples in this paper, we use an optical see-through display (Sony Glasstron LDI-100B) and a hybrid ultrasonic/inertial tracker (Intersense IS-600 Mark2) connected to an SGI Visual Workstation 320.

Regardless of the language used, at a fundamental level one can view an interactive 3-D narrative as a control system; it contains a collection of elements (actors, objects) that are controlled by some program as a function of both external inputs (such as scripts, behavior models, environmental context, user activity) and the state of the elements themselves, as shown in figure 2. Each element can be simple (such as a static 3-D model of a piece of furniture) or complex (such as a self-controlled, autonomous model of a animal that reacts to its environment). Independent of the complexity of an element, part of its state must be made available to the control program so it can be used to affect other elements in the story. For example, a script being executed by the control program might cause a secret door to open when the user moves a 3-D object in a room. Or, one autonomous creature (such as a cat) may run away from another creature depending on its type (such as a dog), or more abstract internal state such as its mood (angry).

The challenge of integrating video-based actors into an interactive 3-D narrative is that the video typically represents a complex animation (containing sound, gestures, and other movement), but these actions are not readily available to the programmer. Typically, a programmer decides when to start the video, but after that the timing is fixed. In the best case, the video is seen by the programmer as a sequence of images; in the worst case, it is seen as an abstract media object with a certain duration. To fully integrate the 2-D video actor into a 3-D narrative, therefore, we need to provide the control program with information about the activity contained in the video.

The control program needs to know two types of information about the video actor (as illustrated in figure 3):

- *How to position the video actor in the world:* the size of the video frame (in the coordinate system in which the video actor will be placed), the starting location of the "frame origin" of the actor, the motion of the frame origin through time, and the offset of the bottom of the video image from the frame origin.
- *How to synchronize the activity in the video with the rest of the narrative:* A set of actions to be synchronized and, for each action, a collection of parameters specified for each relevant keyframe. For simplicity, we define the frame origin of all video actors as the place that would be on the floor between their feet (as shown in figure 3). This allows them to be manipulated consistently.

We view each logically separate activity in a video as an *action*. Conceptually, an action may be synchronized with a narrative in two ways: *temporal synchronization* (between the actions of the video actor and other activity in the 3-D world) and *spatial synchronization* (between locations in the 2-D video frame and locations in the 3-D world). Our approach is to specify the keyframe for an action and to support temporal synchronization by notifying the programmer when the keyframe is shown. By specifying a 2-D point within the video frame as a parameter for the keyframe of an action, spatial synchronization is also supported.

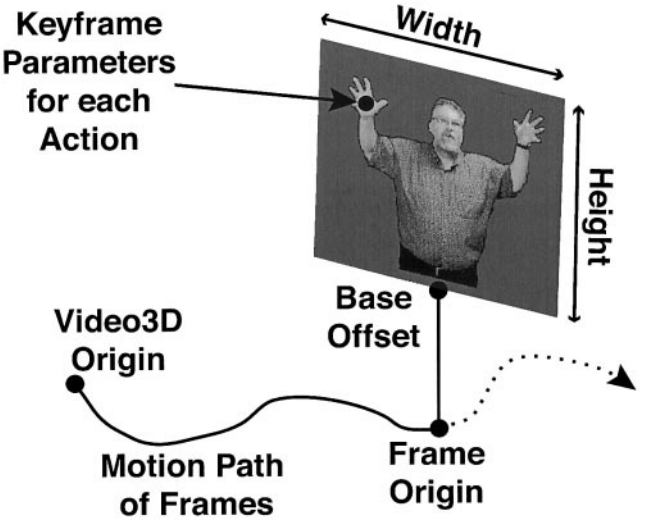

Figure 3. *The information that must be specified to integrate a 2-D video actor into a 3-D world.*

4 The Video Actor Framework for Java 3D

The Video Actor Framework for Java 3D is designed to allow content creators to take a QuickTime video they have created (using whatever tools they design), annotate the video to specify the semantic information about the actor and actions contained in the video, and use the video as an object in a Java 3D scene. The Video Actor Framework consists of two components. The first is the VideoActor Editor used by a content creator to specify the necessary information for integrating the video actor into a 3-D world (for example, the location and action information). The second is the Video3D Java package that supports reading in a script produced by the VideoActor Editor and integrating it into a Java 3D program. The Video3D package is designed to match the Java and Java 3D programming models. Although the terminology and structure would be slightly different if we used a different programming environment (such as C++ and OpenInventor), the basic functionality would remain the same. (In section 5, we discuss our recent work creating video actors in Macromedia Director.)

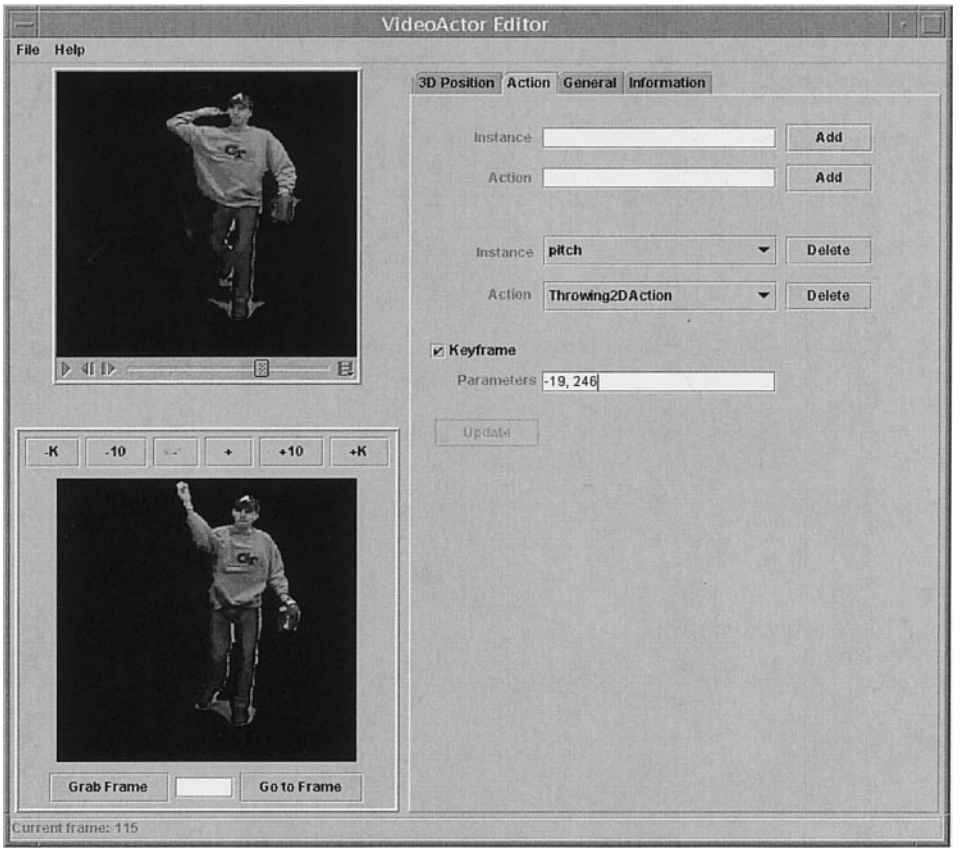

Figure 4. *The VideoActor Editor. The editor shows the video in the upper left with controls so the user can scan through the video. The image in the lower left is the current frame being worked on. To set the current frame, simply move the video in the upper left to the desired frame and click on the Grab Frame button in the lower left (or enter the frame number). This image shows the action editing pane of the editor. In this image, the designer is adding a Throwing2DAction called "pitch" to the frame in the lower left. The small white marker over the actor's hand is the 2-D location for this keyframe. (In the case of this action, this is the only keyframe.) The other panes allow the dimensions of the video and the 3-D path of the origin of the video frame to be specified.*

4.1 VideoActor Editor

A sample screen of the VideoActor Editor is shown in figure 4. The editor lets the user specify three categories of information:

- the width, height, and offset of the video actor from the frame origin in the world,
- the path of the frame origin, specified as 3-D locations for user-selected keyframes of the video, and
- the synchronization actions on the video.

We define the origin of the 2-D video to always be in the middle of the bottom edge of the texture. As previously described, the frame origin of the 3-D video actor (which is where the object is attached to the 3-D world) is defined as the center of the feet of the actor; an offset is specified to allow the bottom edge of the video to be raised or lowered depending on the relationship between the bottom edge of the video and the feet. For example, the video of the pitcher (shown in the editor window of figure 4) has a negative offset because the

feet are above the bottom of the video. Conversely, the Thanksgiving video ghost (shown in figure 1) has a positive offset because his feet are well below the bottom edge of the video. If the video image must move relative to the viewer (for example, if the video actor was moving, and the video camera panned to keep the actor in the center of the frame), the 3-D position of the frame origin is specified at keyframes in the video and interpolated between them; the frame origin is then automatically moved along this path when the movie plays.

The image in figure 4 shows how the synchronization actions are specified. As discussed in the next section, we implement the video actor's actions using the Java 1.1 event model, which gives us the flexibility we need. For any conceptual action, the user of the editor can define an action class (for example, the Throwing2DAction in figure 4, a generic "throwing" class). This class does not have to exist to be used in the editor, but it must be defined before the script file can be loaded. Each separate action in the video is defined as an instance of one of these defined action classes. For each of the action instances, any number of keyframes (from none to all video frames) can be selected to define the temporal synchronization between the video and the action instance. For example, the pitch instance in figure 4 has only one keyframe specified (the frame when the ball appears). Each keyframe can have an arbitrary list of parameters passed to it; it is up to the programmer who implements the action class to interpret the parameters. If the user clicks in the video image, the corresponding 2-D location will be added to the parameter list. For example, the pitch instance in figure 4 takes two parameters: the 2-D position coordinates where the ball is to appear in the video frame.

The script that the editor writes to disk contains all of this information, plus a pointer to the video file. This script can be read by the Video3D Java package, described in subsection 4.2. We envision adding many other features to the VideoActor editor, such as the vision-based algorithms discussed in subsection 3.1. Right now, the only automated feature of the editor is to preprocess the video to compute the bounding box of all nonblack content for each frame to accelerate transferring the individual frames to textures at runtime.

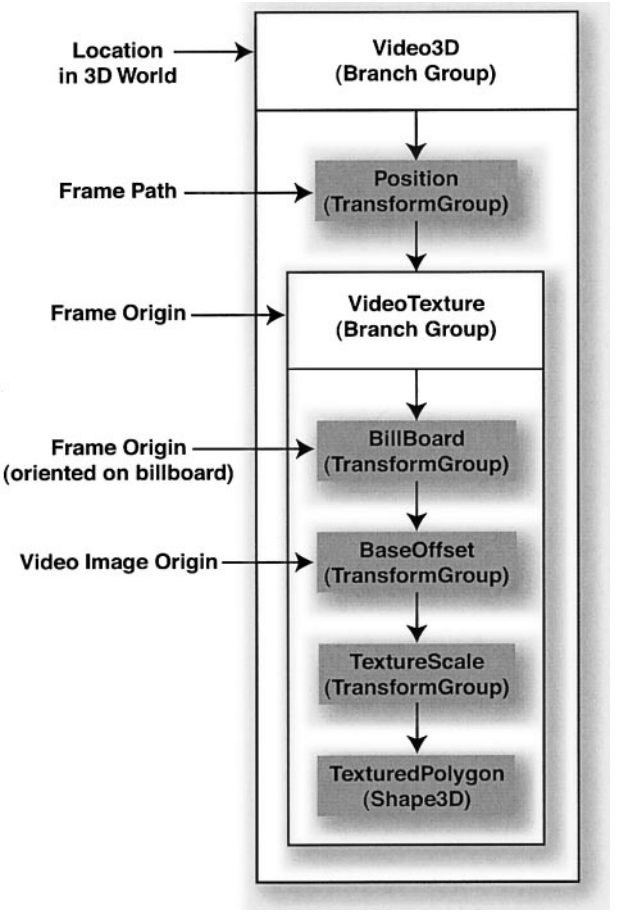

Figure 5. *The Java 3D object hierarchy inside a Video3D object. The Video3D and VideoTexture objects are themselves Java 3D BranchGroups. The transformation to move the video along its motion path is applied to the Position group. The frame origin is at the VideoTexture group. The BillBoard groups add the billboarding rotation. TextureScale is used to make the TexturedPolygon the correct size, as specified in the VideoActor editor.*

4.2 The Video3D Java Package

When a Video3D object is created by loading in a script from the VideoActor editor (see figure 9), a variety of objects are created, as illustrated in figures 5 and 6. In addition to creating the necessary Java 3D object hierarchy (as described in figure 5), the Video3D object creates a VideoTexture object that decodes the video onto a textured polygon and spatializes the audio track (if there is one). The VideoTexture object also drives

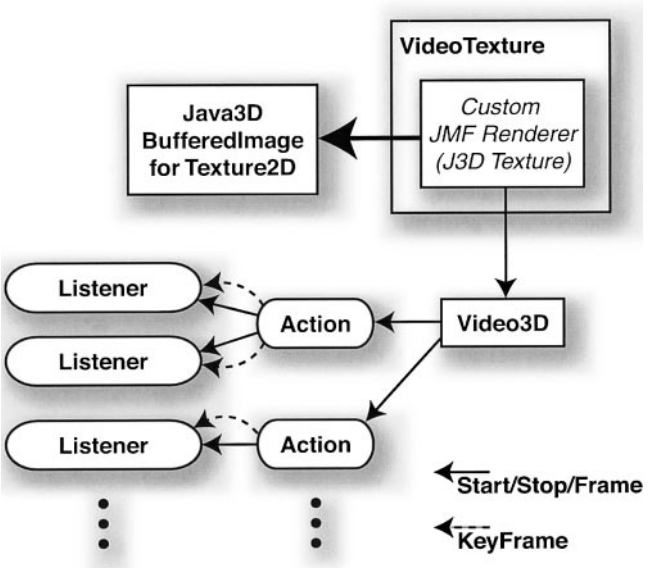

Figure 6. *The control/data flow through the Video 3D objects. All objects except the listeners are created automatically when a video is loaded. (The listeners are created by the control program to receive notification of actions in the video.) The heart of the Video3D package is a custom Java Media Framework (JMF) video renderer (that takes the video frame by frame and inserts it in a Java 3D Texture2D object) and a custom JMF audio effect (that spatializes the audio track of the video). Because the Video3D and action objects implement the listener interfaces, the custom renderer uses that interface to notify the Video3D object when the video starts and stops, and when a new frame is displayed. The Video3D in turn notifies all of its actions. Each action passes the notification on to any listeners, and also generates keyframe events for its KeyFrameListeners.*

the Listener notification scheme that implements the synchronization actions defined in the VideoActor editor (as described in figure 6).

The Listener model is based on the Java 1.1 event model and allows the Video3D object to drive external actions. Objects that are interested in events can register one of three kinds of listeners to receive notification of one of four events: *FrameUpdateListener* (notified each frame), *KeyFrameUpdateListener* (notified only each keyframe), *StartStopUpdateListener* (notified when the video starts or stops). The listeners are Java interfaces, and an object that implements any of the interfaces can register to have the notification methods called at the appropriate time (for example, the frame update method is *frameUpdate(frameNumber)*). A program can register any number of each of these kinds of listeners, and typically subclasses these listeners to implement the specific actions (such as those described in the examples in subsection 4.3).

When the Video3D object is loaded, it looks up the names of the action classes in the script file and dynamically loads them into the Java VM. It then creates the required number of instances of each action class and stores them internally. All of the action classes will have their basic functionality implemented and do not require any additional coding by the content creator. Only when the content creator wishes to receive notification of some action and use it to drive some other part of the narrative does a custom listener need to be set up. (An example of a custom listener is shown in figure 9.)

It is up to the programmer to ensure that all action classes have been implemented. When each action instance is created, it is passed a reference to all of its parameters for every keyframe. It is necessary to give the action all the parameters at the beginning because some actions require them. For example, to create a Java 3D interpolator to smoothly interpolate between points on the keyframes, all of the data is needed at the beginning. Setting up interpolators is vital for any action class that tracks a point in the video; at fifteen frames per second, the objects tracking the video appear to jump from frame to frame rather than move smoothly.

The final facility provided by the Video3D object is the *getCoorSystemTransform3D(src, dest)* function. Given any two Java 3D group nodes in the Video3D scene graph, this function returns the matrix that transforms a point from the source (*src*) to the destination (*dest*) coordinate system. If the source (or destination) is the TexturedPolygon node, it composes an additional scale on the matrix to convert from (or to) the 2-D coordinates in the video frame. Therefore, an action or listener function can convert the 2-D parameters generated by the editor into 3-D coordinates in any coordinate system in the AR environment in two lines of code

(one to get the transformation, one to transform the point).

We have implemented a variety of interesting actions so far, a few of which will be described in subsection 4.3. The action classes are all quite short and follow a similar pattern. The vast majority of the code deals with the work being performed by the action, not with talking to the Video3D framework.

4.3 Examples of the Java3D Video Actor Framework

In this subsection, we will briefly describe two simple AR experiences that we created using our framework.

4.3.1 Thanksgiving. Figure 7 shows two images from a simple animation we call Thanksgiving. The VideoActor uses an action built for this example called *Dropping2DAction;* each keyframe of this action takes a 2-D point and a text string as parameters (specified in the Video Actor Editor). It creates 3-D text for the string and drops it from the 3-D location (in world coordinates) of the 2-D point in the video frame. When the object hits the specified ground plane, it stops falling and stays there. The *Dropping2DAction* class file is only 130 lines of code.

In this example, the ghostly narrator is reciting a monologue about Thanksgiving. The ghost has been put on a path that moves slowly away from the starting point, and keyframes have been set on the *Dropping2DAction* so that certain words in the text appear on the ghost's hands or mouth and fall as he moves back. The end result is shown in figure 7(b).

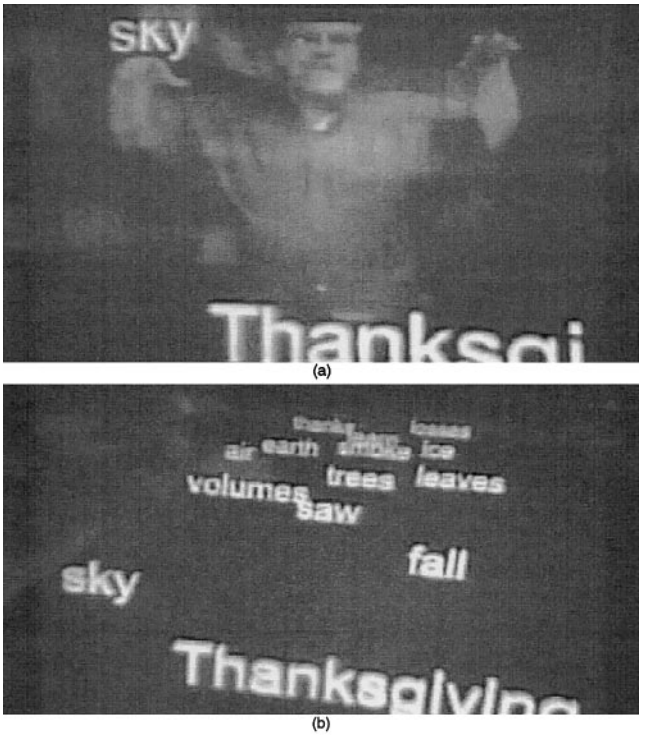

Figure 7. *Two frames from the Thanksgiving AR experience. In (a), the ghostly narrator is speaking to the viewer, while the occasional word from his speech drops to the floor (as a 3-D text object). In (b), the video has ended after the narration finishes, leaving only the dropped words behind.*

4.3.2. The Pitch! Figure 8 shows an image from "The Pitch!". This AR experience uses an action called *Throwing2DAction,* which is just more than 100 lines of code. Figure 9 shows the code to integrate "The Pitch!" in a Java program, including adding a simple update notification listener object. At each keyframe, the *Throwing2DAction* expects a 2-D point corresponding to a location in the image. It converts that point to 3-D world coordinates and then creates an interpolator from the point to the viewer's eye point, and uses that interpolator to "throw" an object at the viewer. This action requires the programmer to specify the 3-D object to be thrown, as shown in the figure.

5 Video Actors in Macromedia Director

Although the designers we work with are typically unfamiliar with Java and the Java 3D environment, they are extremely adept in Macromedia Director and Lingo (the programming language embedded in Director). Therefore, in addition to building AR experiences in Java and Java 3D, we have written interfaces to our tracking hardware in Lingo so that Director can be used to prototype AR experiences. This allows designers to

Figure 8. *A single frame from the "The Pitch!" AR experience. This frame occurs just after the keyframe specified in figure 4. At that keyframe, a ball was added to the environment on a path from the hand to the viewer. In this frame, the ball is in the air on its way to the viewer.*

```
/* An example notification class */
class PrintMsg implements
  KeyFrameUpdateListener {
    public void keyFrameUpdate(long currentFrameNumber) {
      System.out.println("This is the pitch at frame " +
          currentFrameNumber);
    }
};

/* Create a new Video3D object from a file.*/
video3D = new Video3D(viewPlatform, "h:\\editor\\pitch");

/* retrieve the throw action */
Throwing2DAction throw = (Throwing2DAction)
  video3D.getAction("Throwing2DAction", "pitch");

/* Create a sphere for the pitching action */
Sphere mySphere = new Sphere(0.05f);
throw.setContent(mySphere);

/* add a notifier to the pitching action. */
PrintMessage myMsg = new PrintMessage();
throw.addKeyFrameUpdateListener(myMsg);

...
/* add the video object to the 3D world */
scene.addChild(video3D);

...
/* start the video playing */
video3D.start();
```

Figure 9. *The code to integrate "The Pitch!" video into an AR narrative. The Throw2DAction is an action that automatically adds its content (a sphere, in this case) to the world on an interpolated path from the 2-D point specified for the keyframe (the pitcher's hand, specified in figure 4) to the viewer's head position at the time of the keyframe. As with any action, additional listeners can be added. In this example, a trivial KeyFrameUpdateListener is added that prints out a text message. Its KeyFrameUpdate method will be called when any keyframe is displayed (in this case, the single frame when the ball leaves the hand of the pitcher).*

explore AR as a new medium without needing to learn Java or rely on assistance from experienced Java programmers.

Although we have not implemented a graphical video actor editor for use with Director, we have implemented much of the runtime functionality, although in a very different manner. The key difference is that Director does not have a plug-in video rendering system that is analogous to the Java Media Framework that allows us to capture video frames and render them into Java 3D textures. (See figure 6.) Instead, we take advantage of Director's media handling system and expand each movie into a sequence of images stored in a Director external cast. (A cast is a collection of media and data elements.) Each cast also contains the soundtrack for the movie, the data for the actions, and a set of Lingo scripts that allow the movie to be played by copying the correct images to a 3-D texture map.

Although this approach was originally adopted out of necessity, it turns out to have a number of advantages. First, there is less overhead to movie playback because the video does not need to be decoded during playback; we can therefore have more simultaneous video actors in Director than in Java 3D. Second, we can store the video frames as 32-bit images with accurate alpha masks that are generated during the green-screen removal, so the background separation is more accurate and does not need to be computed on the fly (as was required in the JMF video renderer). Third, because all the video frames are available, we do not need to access them linearly, allowing other forms of playback. For example, in the case of idle video characters, such as those waiting for user input, we hope to use the video texture techniques of Schödl, Szeliski, Salesin, and Essa (2000) to create more-realistic idle video. (See section 7.)

5.1 Example: A Mad Tea Party

The most complex example of the use of video actors in Director is the *Mad Tea Party,* an AR experience based on a chapter from Lewis Carroll's *Alice's Adventures in Wonderland.* The user assumes the role of Alice and sits at the tea party with three interactive char-

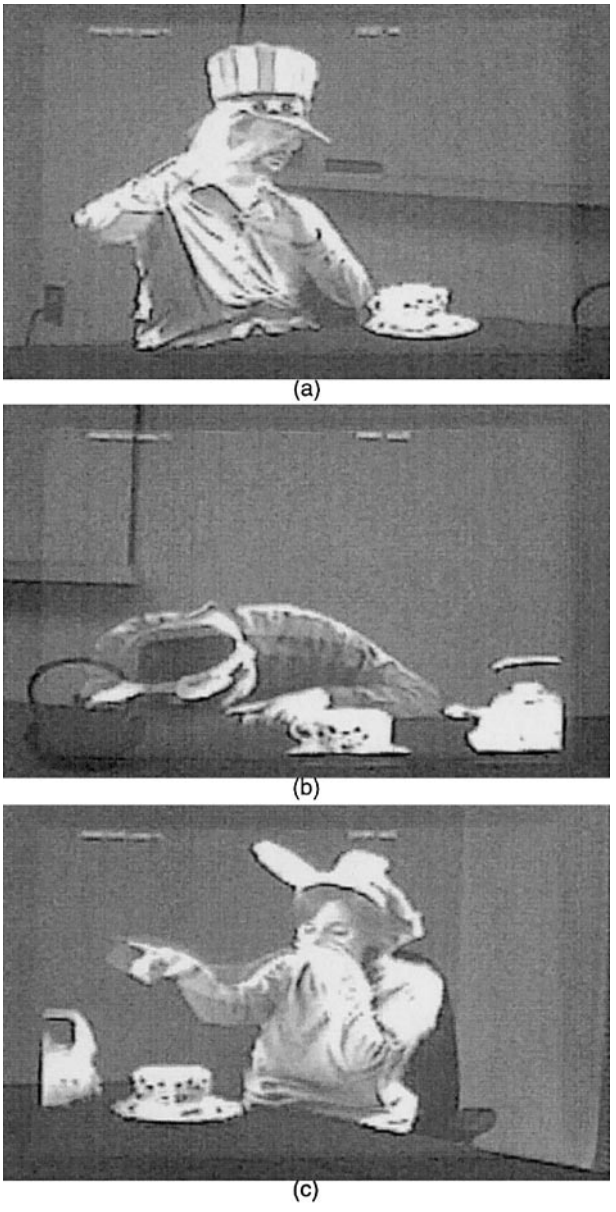

Figure 10. *The mad tea party. The user looks across the table at (a) the Mad Hatter to their left, (b) the Dormouse across from them, and (c) the March Hare to their right. The Mad Hatter has just been splashed with tea by the user, causing the March Hare to laugh at the Hatter. The Dormouse is asleep, but will soon wake up from the noise.*

acters: the Mad Hatter, the Dormouse, and the March Hare, as shown in figure 10. (For a more complete discussion of this experience, see Moreno (2001) and Moreno, MacIntyre, and Bolter (2001). The user's objective is to get directions to the garden located somewhere in Wonderland. The characters interact with the user and with each other. Each has a set of primitive actions that they can perform that generate events, including serving tea, receiving tea, sipping tea, asking riddles, and various reactions to events that may occur in the story environment. If properly provoked, a character may splash the user (or another character) with tea. The user can address the other characters, and has a range of gestures for virtually serving, receiving, sipping, and throwing tea. Initially, the gesture recognition and audio-level sensing were simulated using a "Wizard of Oz" interface (in which an operator pressed keys on the keyboard based on user actions). We have also implemented simple gesture recognition using a Polhemus magnetic tracker attached to the props, and audio-level sensing to capture when the user addresses a character.)

The characters have procedural behaviors that govern how each acts or reacts to the user or to other characters in the scene. Each action represents a primitive story element, and the progression of these elements builds the overall narrative experience. The tea party setting allows the user to be seated, simplifying position tracking. The teacups and teapot are part of the story and provide physical objects for interaction. The characters in the scene are simple, yet each provides opportunities for dramatic gestures.

6 Related Work

The idea of putting video into 3-D programs is not new: many VR systems have demonstrated the idea of texture mapping video to support "virtual videoconferencing" or to otherwise enhance the appearance of the virtual world (Carraro, Edmark, & Ensor, 1998), and 3-D packages like Criterion Software's RenderWare support simple video-based textures in 3-D. The question we are addressing in this paper is not "can you put video in 3-D?", but "how can you do it in a way that it is easy to use?" A substantial amount of work has also been done on creating lifelike animated characters (Hodgins, Wooten, Brogan, & O'Brien, 1995; Lasseter,

1987). These efforts are complementary to ours, as they are aimed at different problems; 3-D animated characters are usually used to "fill in" virtual environments with "plausible" characters, whereas video actors are used for primary interaction.

A large number of 2-D and 3-D authoring tools are available, from Discreet's 3D Studio to Macromedia's Director to Alice (Pausch et al., 1995), but none that we know of support the integration of 2-D and 3-D in as clean a manner. Apple's QuickTime lets authors add 3-D content to a 2-D video, which is the inverse of what we are trying to do. Some researchers are proposing toolkits that support the creation of 3-D animation objects, which encapsulate both the geometry and animated behavior of a 3-D element and include well-defined ways to link them together (Döllner & Hinrichs, 1998; Elcacho, Schäfer, Dörner, & Luckas, 1998). Although this work has a similar flavor, none of it supports 2-D content.

7 Discussion and Future Work

We would like to do a number of things in the future. As mentioned in the introduction, our eventual goal is to develop a full suite of tools for prototyping AR narratives in both Java and Director. Even without developing new tools, the VideoActor tool could be improved in a number of small ways, in addition to adding significant features such as feature tracking to make specifying the keyframes a bit easier.

Besides improving the tools themselves, we would like to explore two promising avenues in an effort to better integrate video actors into 3-D narrative. First, we would like to explore ways of warping the images to give the illusion of them being better integrated, as suggested by Carraro et al. (1998). Currently, 3-D objects cannot intersect the imagined volume of the actor, or the illusion will break when the object and the video do not interact in the expected way. For example, we cannot have a video actor bend over a table to write because their legs are behind the table and their arms must appear in front (on top) of the table.

Another exciting possibility would be to integrate the video texture techniques of Schödl et al. (2000). By creating video characters that can pause in a believable manner, we can expand the range of narrative possibilities available to us. Similarly, we plan on experimenting with capturing video simultaneously with multiple cameras positioned around the actor, which will allow us to create video actors that can be viewed believably from multiple sides. Unlike those working with multiple camera views of static objects, we do not plan on doing image-based rendering (Gortler, Grzeszczuk, Szeliski, & Cohen, 1996) or generating 3-D volumes using the data from multiple cameras (Matusik, Buehler, Raskar, Gortler, & McMillan, 2000), as such techniques would be prohibitively expensive in the context of video. Rather, we hope to develop techniques for smoothly switching between different cameras during playback.

Finally, the techniques developed here could also be used to tightly integrate any time-dependent 2-D content, such as the Flash animations created in Director, into a 3-D world. It might be possible to extract enough information from such an animation to support additional automation of the integration process.

Acknowledgments

The authors would like to acknowledge the members of the Augmented Environments Lab, IDT program, and members of our class at Georgia Tech for their influence on this work. We would especially like to thank Rob Kooper for his help with the software, Matjaz Divjak for his help with the JMF spatialized audio effect, and Blake Leland for authoring and acting in the Thanksgiving video. This work was supported by Siemens via a GVU Industrial Affiliate Grant, ONR under Grant N000140010361, and equipment and software donations from Sun Microsystems and Microsoft.

References

Bolter, J. D., & Grusin, R. (1999). *Remediation: Understanding new media*. Cambridge MA: The MIT Press.

Carraro, G. U., Edmark, J. T., & Ensor, J. R. (1998). Techniques for handling video in virtual environments. *Proceedings of SIGGRAPH 98*, 353–360.

Döllner, J., & Hinrichs, K. (1998). Interactive, animated 3D widgets. *Computer Graphics International 1998*, pp. 278–286.

Dubberly, H., & Mitch, D. (1987). The knowledge navigator. Video. Apple Computer, Inc.

Elcacho, C., Schäfer, A., Dörner, R., & Luckas, V. (1998). Performing 3D scene and animation authoring tasks efficiently: An innovative approach. Poster session presented at *Computer Graphics International 1998,* Hannover, Germany.

Feiner, S., Webster, A., Krueger, T., MacIntyre, B., & Keller, E. (1995). Architectural anatomy. *Presence: Teleoperators and Virtual Environments, 4*(3), 318–325.

Gortler, S. J., Grzeszczuk, R., Szeliski, R., & Cohen, M. F. (1996). The lumigraph. *Proceedings of SIGGRAPH 96,* 43–54.

Hodgins, J. K., Wooten, W. L., Brogan, D. C., & O'Brien, J. F. (1995). Animating human athletics. *Proceedings of SIGGRAPH 95,* 71–78.

Lasseter, J. (1987). Principles of traditional animation applied to 3D computer animation. *Computer Graphics (Proceedings of SIGGRAPH 87), 21*(4), 35–44.

MacIntyre, B., Bolter, J. D., Moreno, E., & Hannigan, B. (2001). Augmented reality as a new media experience. *Proceedings of the International Symposium on Augmented Reality,* 197–206.

Matusik, W., Buehler, C., Raskar, R., Gortler, S. J., & McMillan, L. (2000). Image-based visual hulls. *Proceedings of SIGGRAPH 2000,* 369–374.

Moreno, E. (2001). *Alice's adventures in new media: Towards a collaborative language for augmented reality.* Unpublished master's thesis, Georgia Institute of Technology.

Moreno, E., MacIntyre, B., & Bolter, J. D. (2001). Alice's adventures in new media: An exploration of interactive narratives in augmented reality. *CAST'01 Living in Mixed Realities,* 149–152.

Pausch, R., Burnette, T., Capehart, A., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., & White, J. (1995). Alice: A rapid prototyping system for 3D graphics. *IEEE Computer Graphics and Applications, 15*(3), 8–11.

Pausch, R., Snoddy, J., Hazeltine, E., Taylor, R., & Watson, S. (1996). Disney's Aladdin: First steps toward storytelling in virtual reality. *Proceedings of SIGGRAPH 96,* 193–204.

Schödl, A., Szeliski, R., Salesin, D. H., & Essa, I. (2000). Video textures. *Proceedings of SIGGRAPH 2000,* 489–498.

State, A., Livingston, M. A., Hirota, G., Garrett, W. F., Whitton, M. C., & Fuchs, H. (1996). Technologies for augmented-reality systems: Realizing ultrasound-guided needle biopsies. *Proceedings of SIGGRAPH 96,* 439–446.

Webster, A., Feiner, S., MacIntyre, B., Massie, B., & Krueger, T. (1996). Augmented reality in architectural construction, inspection and renovation. *Proc. ASCE Third Congress on Computing in Civil Engineering,* 913–919.

Wren, C. R., Azarbayejani, A., Darrell, T., & Pentland, A. (1997). Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 19*(7), 780–785.