

The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing

Li-wei He *

Michael F. Cohen *

David H. Salesin †

Microsoft Research

Microsoft Research

University of Washington

Abstract

This paper presents a paradigm for automatically generating complete camera specifications for capturing events in virtual 3D environments in real-time. We describe a fully implemented system, called the Virtual Cinematographer, and demonstrate its application in a virtual “party” setting. Cinematographic expertise, in the form of film *idioms*, is encoded as a set of small hierarchically organized finite state machines. Each idiom is responsible for capturing a particular type of scene, such as three virtual actors conversing or one actor moving across the environment. The idiom selects shot types and the timing of transitions between shots to best communicate events as they unfold. A set of *camera modules*, shared by the idioms, is responsible for the low-level geometric placement of specific cameras for each shot. The camera modules are also responsible for making subtle changes in the virtual actors’ positions to best frame each shot. In this paper, we discuss some basic heuristics of filmmaking and show how these ideas are encoded in the Virtual Cinematographer.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation—viewing algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques—interaction techniques.

Additional Keywords: cinematography, virtual worlds, virtual environments, screen acting, camera placement, hierarchical finite state machines

1 Introduction

With the explosive growth of the internet, computers are increasingly being used for communication and for play between multiple participants. In particular, applications in which participants control virtual actors that interact in a simulated 3D world are becoming popular. This new form of communication, while holding much promise, also presents a number of difficulties. For example, participants often have problems comprehending and navigating the virtual 3D environment, locating other virtual actors with whom they wish to communicate, and arranging their actors in such a way that they can all see each other.

*Microsoft Research, One Microsoft Way, Seattle, WA 98052. Email: {a-liwei | mcohen}@microsoft.com

†Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. Email: salesin@cs.washington.edu

In fact, these same types of problems have been faced by cinematographers for over a century. Over the years, filmmakers have developed a set of rules and conventions that allow actions to be communicated comprehensibly and effectively. These visual conventions are now so pervasive that they are essentially taken for granted by audiences.

This paper addresses some of the problems of communicating in 3D virtual environments by applying rules of cinematography. These rules are codified as a hierarchical finite state machine, which is executed in real-time as the action unfolds. The finite state machine controls camera placements and shot transitions automatically. It also exerts subtle influences on the positions and actions of the virtual actors, in much the same way that a director might stage real actors to compose a better shot.

Automatic cinematography faces two difficulties not found in real-world filmmaking. First, while informal descriptions of the rules of cinematography are mentioned in a variety of texts [1, 13, 15], we have not found a description that is explicit enough to be directly encoded as a formal language. Second, most filmmakers work from a script that is agreed upon in advance, and thus they have the opportunity to edit the raw footage as a post-process. In contrast, we must perform the automatic camera control in real time. Thus, live coverage of sporting events is perhaps a better analogy to the problem we address here, in that in neither situation is there any explicit knowledge of the future, nor is there much opportunity for later editing.

In this paper, we discuss an implementation of a real-time camera controller for automatic cinematography, called the Virtual Cinematographer (VC). We demonstrate its operation in the context of a “virtual party,” in which actors can walk, look around, converse, get a drink, and so on. The various actors are controlled either by human users over a network, or by a party “simulator,” which can control certain actors automatically. Each user runs his or her own VC, which conveys the events at the party from the point of view of that user’s actor, or “protagonist.”

The Virtual Cinematographer paradigm is applicable to a number of different domains. In particular, a VC could be used in any application in which it is possible to approximately predict the future actions of virtual “actors.” For example, in virtual reality games and interactive fiction, the VC could be used to improve upon the fixed point-of-view shots or ceiling-mounted cameras that such applications typically employ.

1.1 Related work

There are a number of areas in which related work has been explored. Karp and Feiner [11, 12] describe an animation-planning system that can customize computer-generated presentations for a particular viewer or situation. Sack and Davis [17] present the IDIC system, which assembles short “trailers” from a library of *Star Trek*, *The Next Generation* footage. Christianson *et al.* [3] have developed an interactive story-telling system that plans a camera sequence based on a simulated 3D animation script. All of these techniques

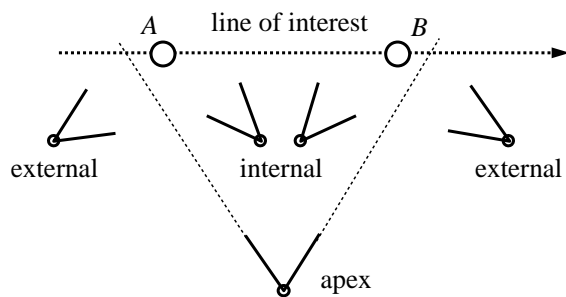


Figure 1 (Adapted from figure 4.11 of [1].) Camera placement is specified relative to “the line of interest.”

use an off-line planning approach to choose the sequence of camera positions. In this paper, by contrast, we are concerned with real-time camera placement as the interactively-controlled action proceeds.

A number of other systems concentrate on finding the best camera placement when interactive tasks are performed [8, 14, 16]. In particular, Drucker *et al.* [4, 5, 6] show how to set up the optimal camera positions for individual shots by solving small constrained optimization problems. For efficiency reasons, in the real-time setting we select shots from a small set of possible camera specifications so that camera positions can be computed using closed-form methods. The mathematics for defining low-level camera parameters, given the geometry of the scene and the desired actor placements, is described in a number of texts [7, 10]. We found Blinn’s treatment [2] to be the most helpful single source.

2 Principles of cinematography

It is useful to consider a film as a hierarchy. At the highest level, a film is a sequence of scenes, each of which captures a specific situation or action. Each scene, in turn, is composed of one or more shots. A single shot is the interval during which the movie camera is rolling continuously. Most shots generally last a few seconds, although in certain cases they can go on much longer.

2.1 Camera placement

Directors specify camera placements relative to *the line of interest*, an imaginary vector either connecting two actors, directed along the line of an actor’s motion, or oriented in the direction that an actor is facing. Common camera placements include *external*, *internal*, and *apex* views, as shown in Figure 1.

Cinematographers have identified that certain *cutting heights* make for pleasing compositions while others yield ugly results (e.g., an image of a man cut off at the ankles). There are approximately five useful camera distances [1]. An *extreme closeup* cuts at the neck; a *closeup* cuts under the chest or at the waist; a *medium view* cuts at the crotch or under the knees; a *full view* shows the entire person; and a *long view* provides a distant perspective.

Individual shots also require subtly different placement of actors to look natural on the screen. For example, the closeup of two actors in Figure 2(a) looks perfectly natural. However, from a distance it is clear that the actors are closer together than expected. Similarly, shots with multiple actors often require shifting the actor positions to properly frame them (Figure 2(b)).

2.2 Cinematographic heuristics and constraints

Filmmakers have articulated numerous heuristics for selecting good shots and have informally specified constraints on successive shots for creating good scenes. We have incorporated many of these

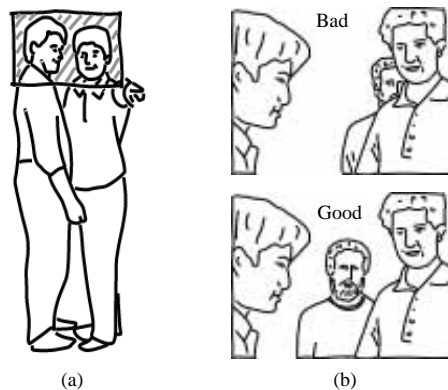


Figure 2 (Adapted from Tucker [18, pp. 33, 157].) Actor positions that look natural for a particular closeup look too close together when viewed from further back (a). Correctly positioning three actors for a shot may require small changes in their positions (b).

heuristics in the design of the Virtual Cinematographer. Some examples are:

- *Don’t cross the line*: Once an initial shot is taken from the left or right side of the line of interest, subsequent shots should remain on that side. This rule ensures that successive shots of a moving actor maintain the direction of apparent motion.
- *Avoid jump cuts*: Across the cut there should be a marked difference in the size, view, or number of actors between the two setups. A cut failing to meet these conditions creates a jerky, sloppy effect.
- *Use establishing shots*: Establish a scene before moving to close shots. If there is a new development in the scene, the situation must be re-established.
- *Let the actor lead*: The actor should initiate all movement, with the camera following; conversely, the camera should come to rest a little before the actor.
- *Break movement*: A scene illustrating motion should be broken into at least two shots. Typically, each shot is cut so that the actor appears to move across half the screen area.

A more complete survey of these heuristics can be found in Christianson *et al.* [3].

2.3 Sequences of shots

Perhaps the most significant invention of cinematographers is a collection of stereotypical formulas for capturing specific scenes as sequences of shots. Traditional film books, such as *The Grammar of the Film Language* by Arijon [1], provide an informal compilation of formulas, along with a discussion of the various situations in which the different formulas can be applied.

As an example, Figure 3 presents a four-shot formula that will serve as an extended example throughout the remainder of this paper. The formula provides a method for depicting conversations among three actors. The first shot is an external shot over the shoulder of actor *C* toward actors *A* and *B*. The second and third shots are external shots of actors *A* and *B* alone. The fourth shot is an internal reaction shot of actor *C*. Arijon [1] stipulates that an editing order for a typical sequence using this setup would be to alternate between shots 1 and 4 while actors *A* and *B* talk to actor *C*. When *A* and *B* begin to talk to each other, the sequence shifts to an alternation between shots 2 and 3, with an occasional reaction shot 4. Shot 1 should be introduced every now and then to re-establish the whole group.

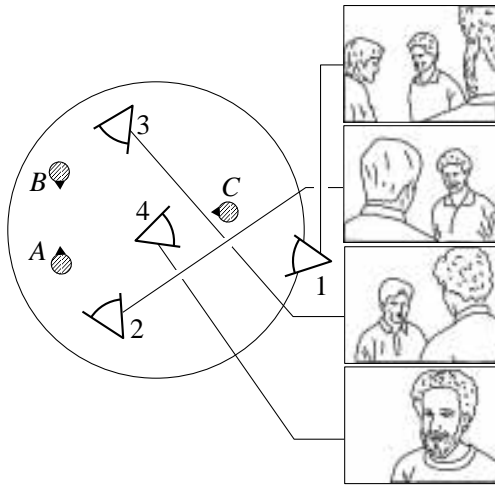


Figure 3 (Adapted from Figure 6.29 of Arijon [1].) A common formula for depicting a conversation among three actors.

The particular formulas preferred by any individual director lend a certain flavor or *style* to that director's films. In the Virtual Cinematographer, the style is dictated by the particular formulas encoded. (In fact, as each of the authors of this paper worked on the VC, a slightly different style emerged for each one.)

3 The Virtual Cinematographer

The Virtual Cinematographer is one part of the overall architecture shown in Figure 4. The other two parts consist of the *real-time application* and the *renderer*. The real-time application supplies the renderer with any static geometry, material properties, and lights. At each time tick (i.e., at each frame of the resulting animation), the following events occur:

1. The real-time application sends the VC a description of *events* that occur in that tick and are significant to the protagonist. Events are of the form (*subject, verb, object*). The subject is always an actor, while the object may be an actor, a current *conversation* (comprising a group of actors), a fixed object (e.g., the bar), or *null*.
2. The VC uses the current events plus the existing state of the animation (e.g., how long the current shot has lasted) to produce an appropriate camera specification that is output to the renderer. The VC may query the application for additional information, such as the specific locations and bounding boxes of various actors. The VC may also make subtle changes in the actors' positions and motion, called *acting hints*. These are also output to the renderer.
3. The scene is rendered using the animation parameters and description of the current environment sent by the application, and the camera specification and acting hints sent by the VC.

3.1 The VC architecture

The cinematography expertise encoded in the Virtual Cinematographer is captured in two main components: *camera modules* and *idioms* (see Figure 5). Camera modules implement the different camera placements described in Section 2.1. Idioms describe the formulas used for combining shots into sequences, as described in Section 2.3. The idioms are organized hierarchically, from more general idioms near the top, to idioms designed to capture increasingly specific situations. This structure allows each idiom to simply re-

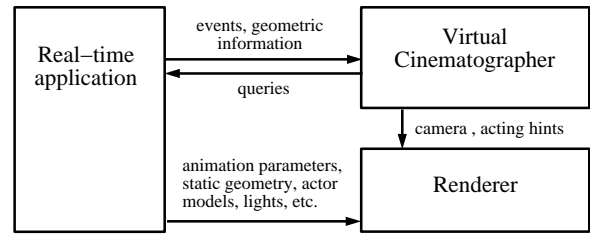


Figure 4 System including the Virtual Cinematographer.

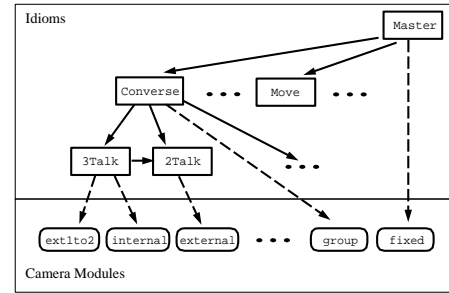


Figure 5 The Virtual Cinematographer structure.

turn control back to a more general idiom when unforeseen events are encountered.

3.2 Camera modules

Each camera module takes as input a number of actors, called *primary actors*; the exact number depends on the particular camera module. Each camera module automatically positions the camera so as to place the actors at particular locations on the screen and allow for pleasing cutting heights. In addition, the camera module may decide to reposition the actors slightly to improve the shot. Finally, the camera placement is automatically chosen so as to not cross the line of interest.

3.2.1 Example camera modules

Sixteen different camera modules have been implemented, several of which are shown in Figure 6. The most heavily used camera modules include:

- `apex(actor1, actor2)`: The `apex` camera module takes two actors as input and places the camera so that the first actor is centered on one side of the screen and the second actor is centered on the other. The camera distance is thus a function of the distance between the two actors.
- `closeapex(actor1, actor2)`: This camera module also implements an `apex` camera placement. However, it differs from the previous camera module in that it always uses a close-up camera distance. To compose a more pleasing shot, this camera module may move the actors closer together, as discussed in Section 2.1 and illustrated by A' and B' in Figure 6.
- `external(actor1, actor2)`: The `external` camera module takes as input two actors and places the camera so that the first actor is seen over the shoulder of the second actor, with the first actor occupying two-thirds of the screen and the second actor the other third.
- `internal(actor1, [actor2])`: The `internal` camera module places the camera along the same line of sight as the `external` camera module, but closer in and with a narrower field of view, so that only the first actor is seen. If only one actor is specified,

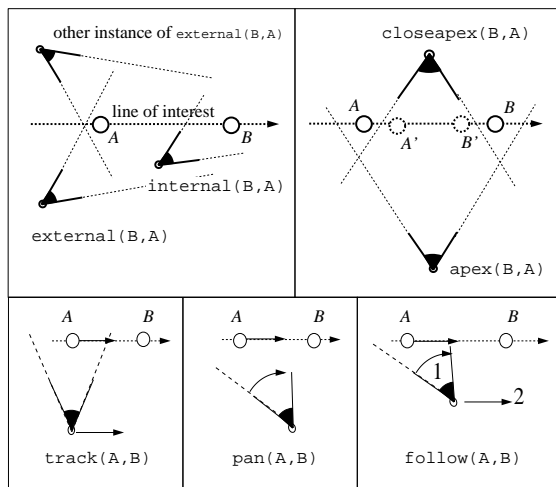


Figure 6 Some camera modules.

then the line of interest is taken to be along the direction the actor is facing.

- `ext1to2(actor1, actor2, actor3)`: This camera module implements an external camera placement between one actor and two others. It places the camera so that the first two actors are seen over the shoulder of the third actor, with the first two actors occupying two-thirds of the screen, and the third actor the rest of the screen (see camera 1 in Figure 3). This camera module may also sometimes perturb the actors' positions to compose a better shot.
- `{track | pan | follow}(actor1, actor2, mindist, maxdist)`: These three related camera modules are used when *actor1* is moving (Figure 6). They differ from the preceding modules in that they define a moving camera that dynamically changes position and/or orientation to hold the actor's placement near the center of the screen. The `track` camera sets the camera along a perpendicular from the line of interest and then moves with the actor, maintaining the same orientation. The `pan` module sets itself off the line of interest ahead of the actor and then pivots in place to follow the motion of the actor. The `follow` module combines these two operations. It first behaves like a panning camera, but as the actor passes by it begins to "follow" the actor from behind rather than allowing the actor to move off into the distance.
- `fixed(cameraspec)`: This camera module is used to specify a particular fixed location, orientation, and field of view. We use it in our application to provide an overview shot of the scene.
- `null()`: This camera module leaves the camera in its previous position.

3.2.2 Respecting the line of interest

Recall that the line of interest is defined relative to the two actors in a shot. Most of the camera modules can choose one of two *instances*, corresponding to symmetric positions on opposite sides of the line of interest (Figure 6). The rules of cinematography dictate that when the line of interest remains constant, the camera should remain on the same side of the line. When the line of interest changes, for example, when one of the two actors in the shot changes position, the choice is not as well defined. We have found that a good rule is to choose the instance in which the camera orientation with respect to the new line of interest is closest to the orientation of the previous shot.

3.2.3 Influencing the acting

The camera modules are also able to subtly improve a shot by influencing the positions of the actors in the scene. Since the real-time application is primarily in charge of manipulating the actors, the changes made by the VC must be subtle enough to not disturb the continuity between shots (Figure 10).

For example, the `closeapex` camera module moves the two primary actors closer together if their distance is greater than some minimum, as in Figure 6. The `ext1to2` camera module adjusts the positions of the three primary actors so that no actor is obscured by any other in the shot. Some camera modules remove actors altogether from the scene, to avoid situations in which an actor appears only part-way on screen or occludes another primary actor in the scene. For example, the `internal` camera module removes the second actor from the scene.

3.2.4 Detecting occlusion

Camera modules are also responsible for detecting when one or more of the primary actors becomes occluded in the scene. In the case of occlusion, at each time tick, the camera module increments an occlusion counter, or resets the counter to zero if the occluded actors become unoccluded. This counter can be used by the idioms to decide whether to change to a different shot.

3.3 Idioms

At the core of the Virtual Cinematographer is the film *idiom*. A single idiom encodes the expertise to capture a particular type of situation, such as a conversation between two actors, or the motion of a single actor from one point to another. The idiom is responsible for deciding which shot types are appropriate and under what conditions one shot should transition to another. The idiom also encodes when the situation has moved outside the idiom's domain of expertise — for example, when a third actor joins a two-person conversation.

In the VC, an idiom is implemented as a hierarchical finite state machine (FSM) [9]. Each state invokes a particular camera module. Thus, each state corresponds to a separate shot in the animation being generated. Each state also includes a list of conditions, which, when satisfied, cause it to exit along a particular arc to another state. Thus, a cut is implicitly generated whenever an arc in the FSM is traversed to a state that uses a different camera module. The FSM's are hierarchically arranged through call/return mechanisms.

We will introduce the concepts involved in constructing idioms by way of examples. In the first example, we construct an idiom for depicting a conversation between two actors, called `2Talk`. In the second example, we use this idiom as a primitive in building a more complex idiom, called `3Talk`, for depicting a conversation among three actors.

3.3.1 The 2Talk idiom

The `2Talk` idiom (Figure 7) encodes a simple method for filming two actors as they talk and react to each other. It uses only external shots of the two actors. The `2Talk` procedure takes as parameters the two actors *A* and *B* who are conversing. It has four states. The first state uses an `external` camera module, which shows *A* talking to *B*. The second state is used for the opposite situation, when *B* talks to *A*. The third and fourth states use external camera placements to capture reaction shots of each of the actors.

When the idiom is activated, it follows one of two initial arcs that originate at the small circle in the diagram of Figure 7, called the

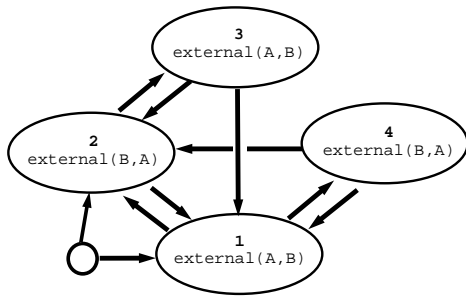


Figure 7 The 2Talk idiom.

entry point. The arc to be used is determined by the following code:

```

DEFINE_IDIOM_IN_ACTION(2Talk)
  WHEN ( talking(A, B) )
    DO ( GOTO (1); )
  WHEN ( talking(B, A) )
    DO ( GOTO (2); )
END_IDIOM_IN_ACTION
  
```

This code, like the rest of the code in this paper, is actual C++ code, rather than pseudocode. The keywords written in all-caps are macros that are expanded by the C preprocessor. This code tests whether *A* is talking to *B*, or *B* is talking to *A*, and transitions immediately to the appropriate state, in this case, either 1 or 2, respectively.

As a state is entered, it first executes a set of *in-actions*. The *in-actions* are often null, as is the case for all of the states in the 2Talk idiom. Once the state is entered, the state's camera module is called to position the camera. The state then executes a sequence of *actions* at every clock tick. The actions can be used to affect conditional transitions to other states. Finally, when exiting, the state executes a set of *out-actions*, again, often null.

In the 2Talk idiom, the camera modules are defined as follows:

```

DEFINE_SETUP_CAMERA_MODULES(2Talk)
  MAKE_MODULE(1, external, (A, B))
  MAKE_MODULE(2, external, (B, A))
  LINK_MODULE(1, 1, "A talks")
  LINK_MODULE(2, 2, "B talks")
  LINK_MODULE(3, 1, "A reacts")
  LINK_MODULE(4, 2, "B reacts")
END_SETUP_CAMERA_MODULES
  
```

`MAKE_MODULE(module_id, type, parameter list)` creates a new camera module of the designated type with the specified parameters and gives it an identifying number. `LINK_MODULE(state_id, module_id, name)` associates the specified camera module with the specified state. Thus for example, whenever state 1 is entered, an external shot of actor *A* over the shoulder of actor *B* will be used.

The first action code to be executed in each state is specified in a block common to all states, called the *common actions*. This is primarily a shorthand mechanism to avoid having to respecify the same (*condition, arc*) pairs in each state of the idiom. The common actions in the 2Talk idiom are:

```

DEFINE_STATE_ACTIONS(COMMON)
  WHEN ( T < 10 )
    DO ( STAY; )
  WHEN ( !talking(A, B) && !talking(B, A) )
    DO ( RETURN; )
END_STATE_ACTIONS
  
```

The first statement checks to see whether the total time *T* spent so far in this state is less than 10 ticks; if so, the current state remains un-

changed. (However, an exception mechanism takes precedence and may in fact pre-empt the shot, as discussed in Section 3.3.2.) If the shot has lasted at least 10 ticks, but *A* and *B* are no longer conversing, then the idiom should return to the idiom that called it. The action statements are evaluated sequentially. Thus, earlier statements take precedence over statements listed later in the code.

The variable *T* is a global variable, which is accessible to any state. There are several other global variables that can be used in state actions:

- *Occluded*, the number of consecutive ticks that one or more of the primary actors has been occluded;
- *IdiomT*, the total number of ticks spent so far in this idiom;
- *D[A, B]*, the distance between the actors (measured in units of "head diameters");
- *forwardedge[x]*, *rearedge[x]*, *centerline[x]*, the edges of the bounding box of actor *x*, relative to the screen coordinates.

There are also a number of predefined control structures:

- *STAY*, remain in the same state for another tick;
- *GOTO(x)*, transition to state *x*;
- *RETURN*, return to the parent state;
- *CALL(idiom, parameter list)*, execute the specified idiom by passing it the specified list of parameters.

Finally, the actions code above makes use of a domain-specific subroutine called `talking(A, B)`, which returns true if and only if the current list of events includes `(A, talk, B)`.

State 1 of the 2Talk idiom is used to depict actor *A* talking to *B*. In addition to the common actions, the list of actions executed at each clock tick when in state 1 are:

```

DEFINE_STATE_ACTIONS(1)
  WHEN ( talking(B, A) )
    DO ( GOTO (2); )
  WHEN ( T > 30 )
    DO ( GOTO (4); )
END_STATE_ACTIONS
  
```

If *B* is now talking to *A*, then a transition to state 2 is required to capture this situation. Otherwise, if an actor has been in the same shot for more than 30 ticks, there should be a transition to state 4 to get a reaction shot from the other actor.

State 2, which addresses the case of actor *B* talking to actor *A*, is completely symmetric: the code is exactly the same except that *A* and *B* are swapped and states 1 and 3 are used in place of states 2 and 4.

For completeness, the action code for state 3 is shown below:

```

DEFINE_STATE_ACTIONS(3)
  WHEN ( talking(A, B) )
    DO ( GOTO (1); )
  WHEN ( talking(B, A) T > 15 )
    DO ( GOTO (2); )
END_STATE_ACTIONS
  
```

Note that this state can make a transition back to state 1, which uses the same camera module as is used here in state 3. In this case, the two shots are really merged into a single shot without any cut. Finally, state 4 is symmetric to state 3 in the same way that state 2 is symmetric to state 1.

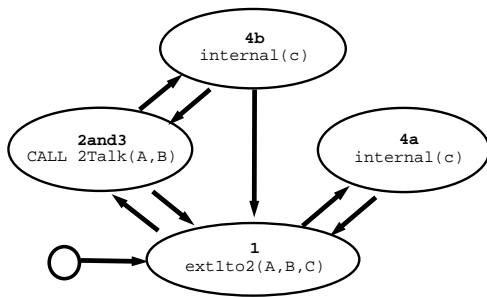


Figure 8 The 3Talk idiom.

Since the out-actions for 2Talk are null, we have now completely described the 2Talk idiom. The next section shows how 2Talk can be used as a subroutine for a higher-level idiom that handles conversations among three actors.

3.3.2 The 3Talk idiom

The finite state machine for 3Talk, which handles conversations among three actors, is shown in Figure 8. This idiom implements the cinematic treatment of three actors sketched in Figure 3 and illustrated in Figure 10. The 3Talk FSM has the same types of components as 2Talk: it has states and arcs representing transitions between states. In addition, this FSM also uses the *exception* mechanism, as discussed below.

The idiom has four states. The first state, labeled 1, is an establishing shot of all three actors, corresponding to the first camera position in Figure 8 and the second shot in Figure 10. The second state, labeled 2and3, is a parent state that calls the 2Talk idiom, corresponding to cameras 2 and 3 in Figure 3. Finally, the last two states, labeled 4a and 4b, capture the reaction shot of the first actor; these two states correspond to camera 4 of Figure 3.

All four states have actions that are similar to the ones described in 2Talk. The two states 4a and 4b have been implemented as separate states because they function differently in the idiom, even though they both shoot the scene from the same camera. State 4a is used in the opening sequence or after a new establishing shot, allowing shots of all three actors to be alternated with reaction shots of actor C. By contrast, state 4b is used only after a two-way conversation between actors A and B has been established, in this case to get an occasional reaction shot of actor C and then quickly return to the two-way conversation between A and B.

The one state that differs significantly from the states considered earlier is the state labeled 2and3. First, unlike the previous states, state 2and3 does have in-actions:

```

DEFINE_STATE_IN_ACTION(2and3)
  REG_EXCEPTION(left_conv, C, LEFT_CONV);
  REG_EXCEPTION(too_long, 100, TOO_LONG);
  REG_EXCEPTION(reacts, C, GET_REACTION);
  CALL( 2Talk, (A, B) );
END_STATE_IN_ACTION
  
```

These in-actions set up a number of exceptions, which, when raised, will cause a child idiom to exit, returning control to the parent state. Each REG_EXCEPTION command takes three parameters: the name of a function to call to test whether or not the exception should be raised; an arbitrary set of parameters that are passed to that function; and the exception name, which is an enumerated type. The final in-action of state 2and3 calls the 2Talk idiom, passing it actors A and B as parameters. The 2Talk idiom is then executed at once. All of the exceptions are implicitly tested before the actions in every state of the child idiom are executed.

The 2Talk idiom will return either when it executes a RETURN in one of its actions or when one of the exceptions is raised. In either case, control is returned to the parent state at that point, and its actions are executed. The actions for state 2and3 are:

```

DEFINE_STATE_ACTIONS(2and3)
  WHEN( EXCEPTION_RAISED( LEFT_CONV ) )
    DO( GOTO(1); )
  WHEN( EXCEPTION_RAISED( TOO_LONG ) )
    DO( GOTO(1); )
  OTHERWISE
    DO( GOTO(4b); )
END_STATE_ACTION
  
```

If either the LEFT_CONV or TOO_LONG exception has been raised, then a transition is made back to state 1 to get another establishing shot. Otherwise, a transition is made to get a reaction shot.

The out-actions of state 2and3, evaluated just before the transition to the new state is made, are used to remove the exceptions that were set up by the in-actions:

```

DEFINE_STATE_OUT_ACTION(2and3)
  DELETE_EXCEPTION(LEFT_CONV);
  DELETE_EXCEPTION(TOO_LONG);
  DELETE_EXCEPTION(GET_REACTION);
END_STATE_OUT_ACTION
  
```

3.3.3 Idioms for movement

Capturing screen motion (Figure 9) presents special problems. In particular, it may be desirable to end a shot not only when an event is triggered by the real-time system, but also when an actor reaches a certain position on the screen (such as the edge of the screen). The global variables `forwardedge[x]`, `rearedge[x]`, `centerline[x]` are used to facilitate these kinds of tests. These variables are measured in a *screen-coordinate system of the actor*, which is set up relative to the orientation of each actor x. The edge of the screen that the actor is facing is defined to be at +1, and the edge to the actor's rear, -1. The center line of the screen is at 0. Thus, for example, a state can see if actor x has just reached the edge of the screen by testing whether `forwardedge[x]` is greater than 1. It can test whether the actor has walked completely off the screen by testing whether `rearedge[x]` is greater than 1.

4 The "Party" application

For illustration, the Virtual Cinematographer has been applied to a simulated "party" environment. The party application runs over a network, so that multiple participants can interact in a single virtual environment. Each participant controls a different actor (their *protagonist*) at the party. The actors can walk, look around, converse with each other, or go to the bar where they can drink and talk to the bartender.

The user interface allows the user to invoke (*verb, object*) pairs, which are translated into (*subject, verb, object*) triples in which the protagonist is the subject. Current *verbs* include *talk*, *react*, *goto*, *drink*, *lookat*, and *idle*. Each invocation of a verb causes a change in the action of the protagonist shortly after the corresponding button is pushed.

An additional interface button allows the actors who stand alone or in a conversation to "vote" on whether to accept or reject a new actor *signaling* to join in the conversation. The *signal* verb is implicitly generated when an actor approaches within a short distance of the object of the *goto* verb.

At each time tick, the party application running on each client workstation sends an update of the current actions of that client's protagonist to a server. The server then broadcasts a list of (*subject, verb, object*) triples of interest to each protagonist's private VC. Triples of interest are those involving the protagonist (or others in the same conversation as the protagonist) as subject or object. The party application is responsible for all low-level motion of the actors, including walking, mouth movements, head turning, etc.

5 Results

The party application, renderer, and Virtual Cinematographer run on a Pentium PC. They are implemented in Visual C++, except for the user interface code, which is written in Visual Basic. The renderer uses *Rendermorphics*[®] to generate each frame. The full system runs at a rate of approximately 5 ticks per second, of which the majority of time is spent in the renderer.

Figures 9 and 10 depict the *Moving* idiom in action and the hierarchy of *Converse*, *3Talk*, and *2Talk*. Individual frames are shown on the left, with a corresponding shot from above on the right (which includes the camera itself circled in black.) In Figure 10, the arcs are labeled with the condition causing a transition to occur between states. (Heavy lines indicate the path taken through the idiom. Dotted lines indicate jumps between idioms caused by the calling and exception mechanisms.)

Figure 10 depicts the use of the hierarchy of film idioms. The sequence begins in the generic *Converse* idiom, which specifies the group shot. The *converse* idiom then calls *3Talk*, which follows an *ext1to2* shot by calling *2Talk*. Eventually *2Talk* is interrupted by an exception and by the *3Talk* idiom. Note the subtle rearrangement of the characters and the removal of extraneous characters in the *3Talk* idiom.

6 Conclusion and future work

This paper has described a Virtual Cinematographer whose architecture is well suited to a number of real-time applications involving virtual actors. The VC has been demonstrated in the context of a networked "virtual party" application. By encoding expertise developed by real filmmakers into a hierarchical finite state machine, the VC automatically generates camera control for individual shots and sequences these shots as the action unfolds.

There are a number of areas for future work. Although the camera modules have proved quite robust, they can fail for a few frames due to unexpected occlusions, or they may miss a critical action due to minimum-length shot constraints. Some of these issues can be resolved by redesigning the idioms in the current structure. We are also looking into incorporating simple constraint solvers, such as the ones proposed by Drucker [4, 5, 6] and Gleicher and Witkin [8]. In addition, we would like to expand the input to the VC to include such ancillary information as the emotional state of the scene or of individual actors. For example, if the situation is tense, faster cuts might be made, or if one actor is scared, the camera might be lowered to give the other actors a looming appearance. We would also like to apply similar rules for automatically lighting the scenes and actors in a cinematographic style.

Acknowledgements

We would like to thank Daniel Weld and Sean Anderson for their significant contributions during an earlier phase of this work. We would also like to thank Jutta M. Joesch for her help in editing the paper.

References

- [1] Daniel Arijon. *Grammar of the Film Language*. Communication Arts Books, Hastings House, Publishers, New York, 1976.
- [2] James Blinn. Where am I? What am I looking at? *IEEE Computer Graphics and Applications*, pages 76–81, 1988.
- [3] David B. Christianson, Sean E. Anderson, Li-wei He, David H. Salesin, Daniel S. Weld, and Michael F. Cohen. Declarative camera control for automatic cinematography. In *Proceedings of the AAAI-96*, August 1996.
- [4] Steven M. Drucker, Tinsley A. Galyean, and David Zeltzer. CINEMA: A system for procedural camera movements. In David Zeltzer, editor, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 67–70, March 1992.
- [5] Steven M. Drucker and David Zeltzer. CamDroid: A system for implementing intelligent camera control. In Michael Zyda, editor, *Computer Graphics (1995 Symposium on Interactive 3D Graphics)*, volume 28, pages 139–144, April 1995.
- [6] Steven M. Drucker and David Zeltzer. Intelligent camera control in a virtual environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.
- [7] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1990.
- [8] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 331–340, July 1992.
- [9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.
- [10] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1994.
- [11] Peter Karp and Steven Feiner. Issues in the automated generation of animated presentations. In *Proceedings of Graphics Interface '90*, pages 39–48, May 1990.
- [12] Peter Karp and Steven Feiner. Automated presentation planning of animation using task decomposition with heuristic reasoning. In *Proceedings of Graphics Interface '93*, pages 118–127, Toronto, Ontario, Canada, May 1993. Canadian Information Processing Society.
- [13] Christopher Lukas. *Directing for Film and Television*. Anchor Press/Doubleday, Garden City, N.Y., 1985.
- [14] Jock D. Mackinlay, Stuart K. Card, and George G. Robertson. Rapid controlled movement through a virtual 3D workspace. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 171–176, August 1990.
- [15] Joseph V. Mascelli. *The Five C's of Cinematography*. Cine/Grafic Publications, Hollywood, 1965.
- [16] Cary B. Phillips, Norman I. Badler, and John Granieri. Automatic viewing control for 3D direct manipulation. In David Zeltzer, editor, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 71–74, March 1992.
- [17] Warren Sack and Marc Davis. IDIC: Assembling video sequences from story plans and content annotations. In *IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994.
- [18] Patrick Tucker. *Secrets of Screen Acting*. Routledge, New York, 1994.

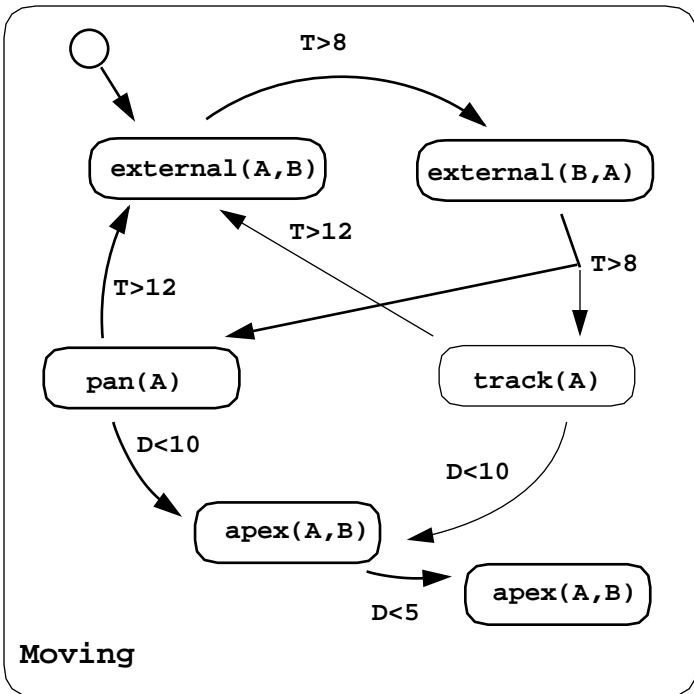
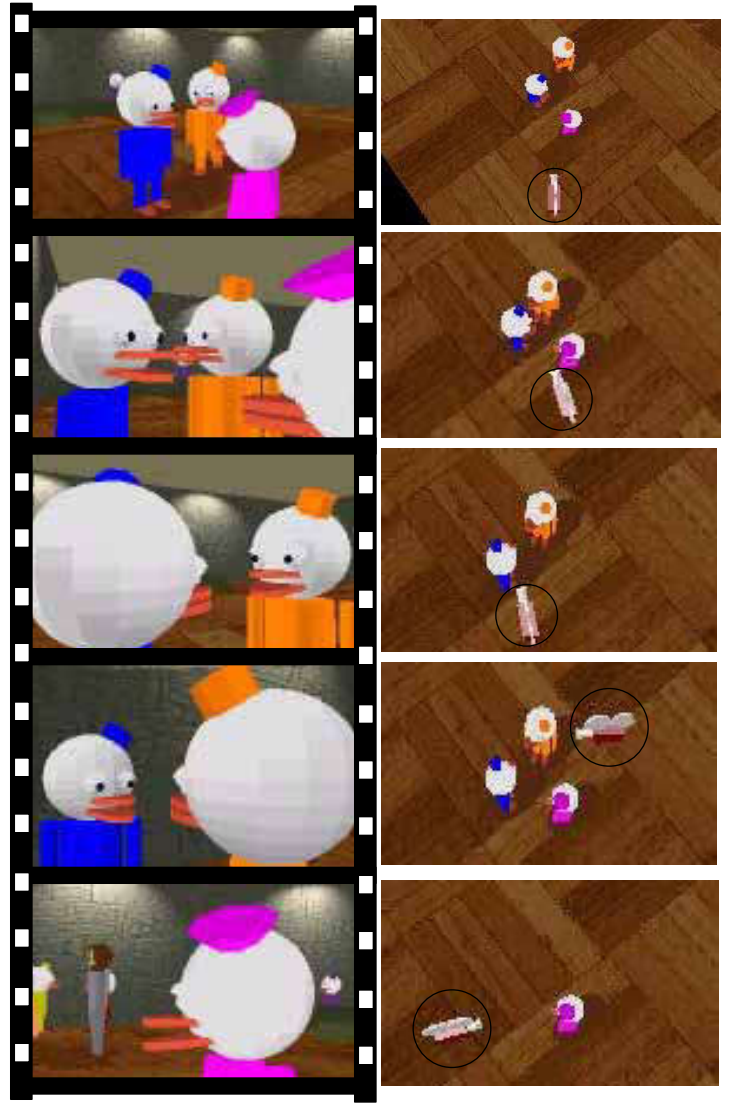
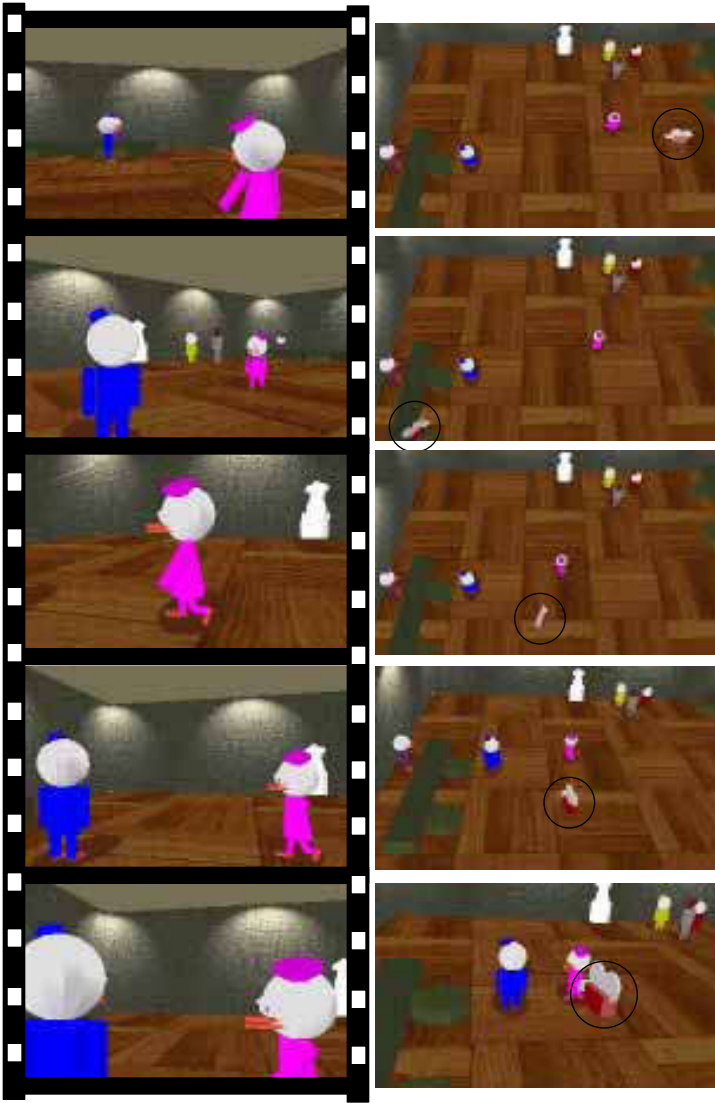


Figure 9: Idiom State Transitions

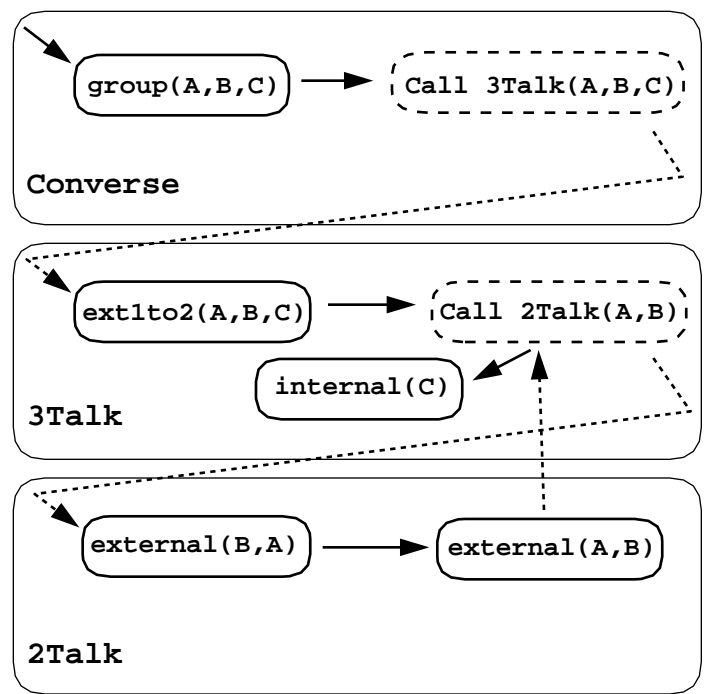


Figure 10: Idiom Hierarchy