OBJECT-ORIENTED COMPUTER ANIMATION

William E. Lorensen
Boris Yamrom

GE Corporate Research and Development

ABSTRACT

Recent advances in computer graphics hardware offer an opportunity to extend 3D visualization techniques into a fourth dimension, time. But, computer animation - the control and display of models, cameras, and lights in a simulated world - is a complex process and software that performs animation should hide this complexity from users. Furthermore, because computer animation is not mature, modern animation systems should be designed to add innovative techniques without discarding investment in existing software.

Praised by software researchers throughout the decade, object-oriented technology provides tools to deal with the complexity and change present in computer animation. Object-oriented design creates a natural partitioning of complex systems into manageable pieces called *objects*, allowing system architects to reuse existing software and to extend existing systems.

We used the object-oriented paradigm to design and create the 3D computer graphics animation system *OSCAR*, the *O*bject-oriented *S*Cene *A*nimato*R*. After a review of object-oriented terminology, we describe a design methodology, system architecture and implementation strategy for animation systems regardless of the object-oriented capabilities of the implementation environment.

1. INTRODUCTION

Two areas of computer science and computer graphics receive considerable attention in recent literature. Computer scientists tout the benefits of object-oriented systems, promising benefits in system design that will surpass those obtained using structured programming. Computer graphics researchers actively pursue realism, producing high quality, three-dimensional animation systems.

The following sections review object-oriented systems and 3D computer animation; present a design methodology that applies an object-oriented philosophy to a 3D animation system; and describe an implementation in the language C. A short example illustrates the system.

2. OBJECT-ORIENTED TECHNOLOGY

Object-oriented systems rely heavily on the software engineering concepts of

- Information hiding: Details of a system that do not affect other parts of the system are not visible from the outside.
- Abstraction: Entities of a system are grouped according to common properties and operations.
- Modularization: Parts of a system that have localized behavior are grouped together with well defined interfaces.

In object-oriented systems, these three principles complement rather than compete with each other.

Terminology

Object-oriented terminology combines the everyday terms objects and inheritance, with unfamiliar ones, such as instances and message passing. Here, we define common object-oriented concepts as they are used in the paper; Stefik and Bobrow [1] give more extensive definitions.

Object-oriented systems are characterized by abstract constructs called *objects* that contain data and procedures to manipulate that data. The data describe the local state of an object and are only accessible to the outside world through an object's procedures, called *methods*. A method is executed when the object receives a *message*. Objects communicate with other objects by sending messages. A *method dictionary* maintains the correspondence between messages and methods.

An *instance* is created by making a copy of a particular *class* of objects. Classes are templates for object creation, containing not only the data associated with the class, but also the methods for manipulating the data. These data are called *instance variables*.

Through a mechanism called *inheritance*, new classes obtain data and methods from other classes, changing or adding data and methods. Two types of inheritance exist in object-oriented systems: single and multiple. Using single inheritance, a class inherits data and methods from at

CH2759-9/89/0000-0588 $1.00

most one other class. On the other hand, multiple inheritance permits a class to inherit from more than one class.

Only the objects know their data structures and methods. Nothing outside the object can directly access these structures. If the implementation of an algorithm requires change in the object's data structure, only the object itself feels this change.

A variety of systems are available that apply object-oriented techniques. Smalltalk-80 [2], developed at Xerox's Palo Alto Research Center (PARC), is a system and language based solely on the object-oriented paradigm. Objective-C [3], a product of Stepstone, Inc., combines Smalltalk-style messaging, with the C language. C++ [4], from AT&T, is a superset of C that supports classes and inheritance.

3. OBJECT-ORIENTED DESIGN

Software engineers can apply the principles and properties of object-oriented systems to software design. This section describes an object-oriented methodology that is independent of the implementation.

Grady Booch [5] outlines an object-oriented design methodology for the Ada™ language that does not address inheritance (since Ada does not support it) and implies that object-oriented design can be done by extracting nouns (objects) and verbs (methods) from a requirements description.

We take an alternative approach that places more emphasis on the abstraction process and also deals with inheritance. This methodology is appropriate at the preliminary design stage, i.e. the architectural stage, of the software engineering life cycle. The primary effort is to define and characterize the abstractions. This approach to object-oriented design proceeds as follows:

1. *Identify the data abstractions for each subsystem.*
   Data abstractions are the classes of the system. Working from the requirements document, the abstraction process starts top-down where possible, although many times the requirements explicitly mention abstractions. Often the classes correspond to physical objects within the system being modeled. This is, by far, the most difficult step in the design process and the selection of these abstractions influences the entire system architecture.
2. *Identify the attributes for each abstraction.*
   The attributes become the instance variables for each instance of the class. Many times, for classes that correspond to physical objects, the instance variables are obvious. Other instance variables may be required to respond to requests from other objects in the system.
3. *Identify the operations for each abstraction.*
   The operations are the methods (or procedures) for each class. While some methods access and update instance variables, others execute operations singular to the class. If the new abstraction inherits from another class, inspect that class's methods to see if any need to be overridden by the new class.
4. *Identify the communication between objects.*
   Define the messages that objects can send to each other and suggest a correspondence between the methods and the messages that invoke them. Even if an object-oriented implementation is not planned, messages help the design team communicate and can be used in the next step, writing scenarios.
5. *Test the design with scenarios.*
   Scenarios, consisting of messages to objects, test the design's ability to match the system's requirements. Write a scenario to satisfy each user level function in the requirements specification.
6. *Apply inheritance where appropriate.*
   If the data abstraction process in step 1 proceeds top-down, introduce inheritance there. However, if abstractions are created bottom-up (often because the requirements directly name the abstractions), apply inheritance here, before going to another level of abstraction. The goal is to reuse as much of the data and methods that have already been designed. At this point, common data and operations often surface and these common instance variables and methods can be combined into a new class. This new class may or may not have meaning as an object by itself. If its sole purpose is to collect common instance variables and methods, it is called an *abstract class.*

The designer repeats these steps at each level of abstraction. Through successive refinements of the design, the designer's view of the system changes depending on the needs of the moment. Each level of abstraction is implemented at a lower level until a point is reached where the abstraction corresponds to a primitive element in the design. New levels of abstraction should provide some attribute or operation that cannot be expressed at the previous lowest level. For example, the abstraction of geometric primitives starts with a polygon, defines a rectangle at the next level of abstraction, and follows with a square.

4. COMPUTER ANIMATION SYSTEMS

Computer graphics representations have progressed from the early use of lines to produce wire frame images of three dimensional models, through simple shaded presentations, up to the current state-of-the-art realistic images. This is the result of success in defining more accurately the models' environment. Transparency, translucency,

™ Ada is a trademark of the Department of Defense

shadows, illumination models, and surface properties are a few areas where research continues to provide algorithms that produce more acceptable synthetic images. The current trend in computer graphics is to apply these advanced techniques to produce quality animations.

Related Work

Although dozens of films produced using 3D computer graphics have appeared over the years, the literature concentrates on the algorithms that produce the images, not on the animation systems themselves. This is probably because few general purpose animation systems exist. Many computer-generated films are produced by executing a sequence of unrelated programs through the control of command files. The major efforts seem to have been in the areas of image quality and realism, not on the animation interfaces themselves.

ASAS [6], the Actor/Scriptor Animation System, was used at Information International Inc., III, to produce sequences for the Disney movie *TRON* [7]. ASAS, implemented in Lisp, relies heavily on object-oriented concepts. ASAS actors are the participants in the animation, communicating by sending and receiving messages. Cues define actor behavior and their processing resides within the actors themselves.

MIRANIM [8] also extends a computer language, here Pascal. Abstract graphical types describe the participants in the animation. A sequence of scenes comprise an animation, where each scene is a sequence of statements manipulating actors, cameras, and lights.

Clockworks [9], developed at Rensselaer Polytechnic Institute, is a C-based object-oriented animation system that integrates modeling, animation control, and rendering. Both Clockworks and *OSCAR* use an animation script to control the animation process. The systems were developed in parallel and *OSCAR* benefits from the interactions with the R.P.I. staff. The major differences lie in *OSCAR*'s ability to interface with external programs that often exist in an industrial setting.

Several commercial animation systems are available. Wavefront Technologies (Santa Barbara, CA) offers modeling, animation and rendering products that run on high performance graphics workstations. Videoworks II, (MacroMind, San Francisco, CA) is a two dimensional animation system that runs on Apple Macintosh desktop systems.

5. OSCAR

Industrial computer graphics applications share some characteristics with those of the university and commercial film communities. Although software for modeling and rendering is common to both environments, university and commercial systems depend on artistic talent to communicate a message through the animation. In contrast, the industrial environment is driven by analyses of modeled phenomenon. For example, an artist interpreting robot motion in a work environment may show the robot execute apparently realistic motions, whereas an industrial robot motion must be predicted by sophisticated kinematic analysis. After all, the intent of such an animation is not to produce a pretty film but to gain insight into the interactions of the robot with its work environment. Also, if the animation is used as a marketing tool, prospective customers will not be impressed by an artist's interpretation of the robot behavior, but want to understand how the commercial robot operates in a real environment.

This is the key difference between this system and those described previously: this system relies on analysis rather than art. This raises an issue for industrial applications: the analysis software often already exists with its own user interface and data bases.

OSCAR, the *O*bject-oriented *SC*ene Animato*R*, provides an automated graphics animation capability to create, control, and manage 3-D computer-generated animation sequences. *OSCAR* automates the creation of high-quality film and video, showing the results of complex research, experiments, and other computer-generated analyses. Using an object-oriented script language as the user interface, the animation system provides automatic control of analysis, modeling, rendering, display, and filming processes. Interfaces have been developed for scientific analysis programs in the areas of molecular modeling, mechanisms analysis and robotics, with future interfaces planned for structural analysis and fluid dynamics. The object-oriented design produced a system that lends itself to interfacing with existing and future in-house and external software.

5.1. The Animation Process

A prerequisite to developing any software system, is understanding the problem to be solved. As in all disciplines, computer animation has its own terminology. Here we present the animation process, defining enough jargon for the reader to understand the subsequent system design.

The user of a computer animation system acts as the producer, writer and director of a film sequence. The producer manages the overall film production, keeping schedules, assigning tasks, and organizing resources. The writer creates a script based on the requirements of the customer. The director controls the animation, positioning

the props, actors, cameras, and lights. Several steps are required to produce an animation.

A review of the process illustrates the steps that can benefit from the computer animation system.

1. The Story. The writer, working with a customer, delivers a story that describes the role of each participant including their appearance, dialog, and actions. Every film is made with some purpose in mind. It could be to verify or understand some mathematical algorithm as it relates to a physical phenomenon, to explain an abstract concept to an audience, to market a product, or to provide entertainment. Writing is an artistic process that is difficult to assist with the computer.
2. The Script. The script contains the details of positioning and movement of the actors, cameras and lights in the animation. *OSCAR* provides a language for scripting to document changes and to produce the final animation. The script can be written with a text editor, or created with the *OSCAR Interactive Script Generator*. *OSCAR* scripts contain statements in an object-oriented language developed as part of the system.
3. Simulations. For scientific applications, experiments and simulations often provide the physics of the animation. Technical analysts setup and execute these simulations. For computer animations, these analysts are the scientists and engineers requesting the animation. *OSCAR* assumes that most animations depend on some computer model. Analysis runs must be made to provide the simulation results for the animation. *OSCAR* calls these simulation programs *analysts*.
4. Models. Each prop and actor in an animation must have a geometric model. During the animation, the director moves the models around the environment. Some analyses have geometric models associated with them, while others do not. For instance, a structural engineer, doing a stress analysis of a turbine, models the turbine with finite elements before the analysis is run. Here, the model and analysis are tightly coupled: both analysis and display require the same model. However, in a molecular mechanics calculation, simple cartesian points model the atoms, and connectivity relationships model the bonds. Here, the rendering process requires more sophisticated geometric models: spheres and cylinders. Models are created by programs called *modelers*.
5. Rendering. This step applies computer graphics algorithms to the computer geometric model, surface properties, lighting, etc. and produces shaded images for display. Rendering is done by programs called *renderers*.
6. Film Editing. Editors do post-production, creating titles, credits, and special effects such as dissolves and fades that add a professional touch to the completed sequence.
7. Recording. Recorders select frames for the final sequence and expose film or video tape.

5.2. Major Subsystems

The subsystem breakdown in *OSCAR* delegates authority for the steps in the animation process. Using an anthropomorphic flavor throughout the descriptions maintains the correspondence with conventional movie making metaphor. Figure 1 shows a diagram of the system.

Interactions between *objects* and programs are described below:

1. *Director* is a collection of *objects* that provide control over all the components of *OSCAR*. The Director reads and interprets a script, sending commands to the other modules to do the animation.
2. *Interactive Script Generator* provides a graphic user interface for writing scripts. Scripts contain the instructions describing what is to occur in the animation sequence. The *Interactive Script Generator* allows the user to position cameras, lights, and to describe the movement of objects while seeing a wire-frame image of the objects that will be presented in the final film as realistic images. User inputs are interpreted and stored in a script file. Although the *Interactive Script Generator* provides a user interface for both the novice and experienced user, the experienced user can achieve more control by writing or editing the scripts with a text editor.
3. The *Frame clerk* keeps track of the location of each finished frame of the film, notes whether it has been recorded, and archives frames when they are no longer needed. Frames can be kept in several places: online disk, magnetic tape, and optical video disk.
4. *Liaisons* are interfaces between *OSCAR* and external modules. The *Liaisons* translate *OSCAR*-specific information into a form their assigned modules (analysts, modelers, or renderers) can understand and vice-versa.
5. *Analysts* are external programs that do analyses in a variety of scientific fields. The interfaces to these software packages cannot be changed, so an analysis-specific *liaison* is the interface between each *analyst* and *OSCAR*.
6. *Modelers* are external programs that create the geometry of the objects for the animation. Typically, they use geometric primitives to build complex representations of structures. *Modelers* available for use include GEOMOD, Movie.BYU, Phigs+ [10], and Synthavison.

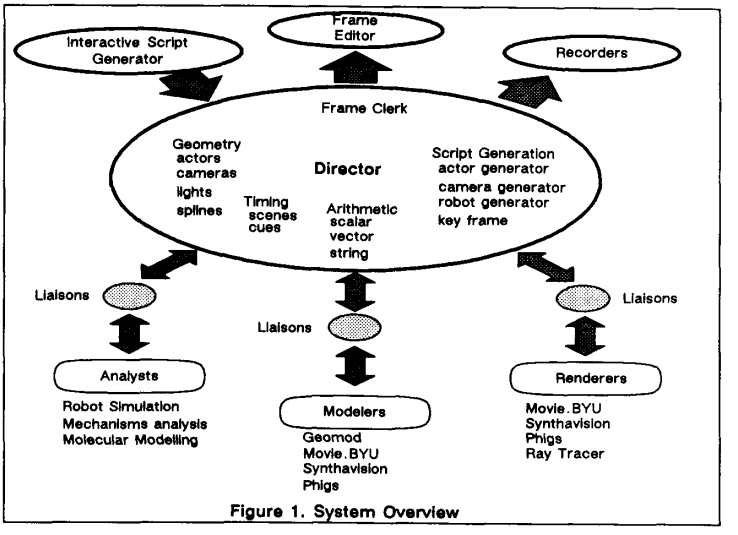

Figure 1. System Overview

Like the *analysts*, these systems also have defined interfaces, and each needs a *liaison* to translate between the *director* and the *modeler*.

7. *Renderers* are external programs and *objects* that take geometric information and environmental information (such as lighting and camera positions) from the script and create frames for later display and filming. These *renderers* include Movie.BYU, Phigs+, and Synthavison. There is a *liaison object* for each external *renderer*.
8. *Frame editor objects* do editing, and provide special effects and titles. These *objects* operate on frames.
9. *Recorder objects* do the filming of the sequences. Steps include obtaining finished movie frames from the *frame clerk*, displaying the images in a frame buffer, and recording the images on film or video disk.

5.3. Animation Language

Our animation language uses one statement structure that defines communication between *objects*. In an animation script, the user specifies an *object* and the messages for that *object*. In the excerpts from the syntax description of the language that follow, capitalized items and characters within double quotes are terminal symbols:

```
statement:= object messages ";"
object := NAME
messages:= message | messages message
message:= PREFIX "?" | PREFIX "!"
   | PREFIX ":" argument | PREFIX "=" argument
   | PREFIX "@" argument | PREFIX "+" argument
   | PREFIX "-" argument | PREFIX "/" argument
   | PREFIX "*" argument | PREFIX "^" argument
argument:= VALUE | NAME | STRING
   | "(" argument_list ")"
   | "[" object messages "]"
argument_list := argument | argument_list "," argument
```

where, *VALUE* is a floating point number, *NAME* is a string of characters, *STRING* is a quoted string, and *PREFIX* is an optional string. Special characters at the start of a line allow the user redirect input and output, invoke system routines, and print text at the terminal. The left and right square brackets allow the arguments to a message to be obtained from another *object*. The semantics of messages are implemented within the *objects* themselves. The following rules for message suffixes illustrate message semantics:

? requests for the value of an instance variable.

= sets an instance variable.

: requires arguments, but does not specifically set an instance variable.

@ defines an index.

+, –, /, *, and ^ terminate arithmetic operation messages.

! ends messages with no arguments.

Messages to the same *object* can be concatenated on a statement. A typical statement is:

```
ACTOR new: Abox
   position= (0,5,0) rotate_x: 30
   color=(1,0,1) on!;
```

This statement creates an *instance* of the *class* ACTOR with a position and color. The *object* is rotated about its local x axis. An alternate statement, that improves readability, provides the same results as above:

```
Abox := ACTOR {
    position= (0,5,0) rotate_x: 30
    color=(1,0,1) on!
};
```

To make another box with the same instance variable values as the first,

```
Abox new: AnotherBox;
```

A camera can be defined,

```
CAMERA new: Acamera position= (0, 20, 5) view_angle= 30;
```

and its view reference point can be set to the position of the box by sending a message to the box requesting its position,

```
Acamera focal_point= [Abox position?];
```

6. OSCAR CLASSES

Currently over one hundred classes exist in the system. These classes were selected using the design process described in Section 3. A few of the classes are summarized here:

Actors are the geometric objects of the animation. They have position, origin, orientation, color, and visibility. Their visibility is controlled with *on!* and *off!* messages. They have an associated model that is kept in a separate modeler object. By separating the actor's state and model, we can preview an animation with simple models, later substituting more sophisticated models for slower, but higher quality animations.

Scenes contain cues and renderers. In addition, scenes have durations (in seconds), resolution (in frames per second), and lists of actions to execute when they start, while they are active, and when they complete. An action is any valid script statement. A scene executes actions by sending *parse:* messages to the parser with the actions as arguments. Once a scene receives a *start!* message, it executes any start actions, and sends *tick!* messages to each of its cues. After the cues complete, the scene sends a *render!* message to each of its renderers. On a scene runs for its duration, it executes its end actions and sends a *complete!* message to each renderer.

Cues contain temporal information that controls the presence and behavior of a scene's participants. A start and end time define the interval that a cue is active. *Cues* have clocks that advance at a *cue*-specific resolution. When a *tick!* message is received from another *object* (typically a scene), the cue advances its clock and tests if it should become active. If so, it executes each of its start and tick actions and advances its clock. As long as a cue's clock remains within the interval, the cue executes tick actions when it receives *tick!* messages. Once the interval is exceeded, the cue executes its end actions.

Cameras are the means by which the animation is viewed. In our implementation, the Foley and Van Dam [11] viewing transformation pipeline is used. Cameras can be moved, rotated, and turned on and off. They have fields of view, up directions, and clipping planes. Cameras have no geometric representation so that if one is within the view of another, it is not visible in the animation. Although multiple cameras can be present in a scene, only one camera can be active at one time for a given renderer.

Lights illuminate the scene. They have position, color, and orientation. Lights can be moved and turned on and off. Multiple lights can be active at one time.

Renderers access cameras, lights, and the geometry and state of each actor to create raster or vector images. When a renderer receives a *render!* message, it requests position and orientation information from its assigned actors, lights, and cameras. Renderers that exist as objects in the system, produce images interactively. For external renderers, a renderer *liaison* creates a command file that can be run later to do the rendering. An abstract renderer exists that specifies the protocol for all renderers. To add a new renderer, the user must satisfy this protocol.

Editors are *objects* that contain cues and recorders. They are similar to scenes in that they send *tick!* messages to each of their cues and *record!* messages to each of their recorders. Editors manipulate the raster images created by renderers.

Recorders compose frames from multiple movie frames. Each recorder has a list of sequences of movie frames that it can display and record.

Other classes available within the system include matrix transformations, splines, scalars, vectors, key frames, and collections.

7. IMPLEMENTATION

The system, written in C, runs on Digital Equipment Corporation VAXs running VMS,® Sun Microsystems workstations running Unix,® and a Stellar GS1000 graphics computer (Newton, MA). The parser, produced using YACC [12] and LEX [13], is an *object* that other *objects* can send *parse:* messages to. Each class is a C module. C *struct*'s define the instance variables, but the structures themselves are static so that they are not visible outside the module. A standard header, required for each class, includes the *object* name, super class, debug information, and other general instance variables.

VMS is a trademark of Digital Equipment Corporation
Unix is a trademark of Bell Laboratories

Every method is a C procedure and each class has a method dictionary that contains the name of each message and the appropriate procedure to invoke. Single inheritance of both instance variables and methods has been implemented. The message handling is done through a *message object* that receives an instance name or pointer, message, and argument list. On receipt of a message, the *message object* searches the instance's method dictionary. If it finds a match, it invokes the appropriate procedure. If not, it searches the method dictionary of the *object*'s super class. This continues until the highest *object* in the *object* hierarchy is reached, when the *message object* reports an error to the user. An argument package, by keeping an argument stack, allows methods to have variable numbers of arguments. *Objects* return and receive arguments through this mechanism.

The system now consists of over 120,000 lines of C code. C macros generate the code for data structures and methods that query or update instance variables. A *class generator* object automatically generates C code from a script describing an object's instance variables and methods.

8. A SAMPLE ANIMATION

A short animation illustrates the capabilities of the system.

8.1. The Analysis System

The analysis package is an in-house system capable of predicting articulated robot motion given starting and ending positions of a robot hand. The user of this system interactively prescribes key positions of the robot hand, and using kinematic techniques, the robot program produces a graphic display of the motion of the robot. The program creates a file containing transformations for each joint and member as it progresses through the simulation. The analysis does not require elaborate geometric models, using prisms to model members and cylinders to model joints. However, for animation purposes, more realistic representations are used. Here, the robot has been modeled using GEOMOD [14], a commercial modeling package from SDRC. The transformations produced by the simulation are applied to the vertices of the polygons produced by GEOMOD. Liaisons for the robot simulation and GEOMOD provide the interfaces between each external program and the animation system. A PHIGS+ renderer produces the images on a Stellar computer.

8.2. Script Generators

A robot has many components, but, rather than burden the user with creating actors for each component, a *ROBOT* generator *object* scans user specified transformation and modeling files and generates a script that defines an actor for each joint. It also generates cues for key work points in the analysis. The ROBOT generator sends the script to the parser *object* and stores the script in a text file that can be edited later. The notion of generators reduces a user's effort in starting a script while still maintaining the flexibility offered by the language.

8.3. The Script

The scene has three cues. *Pan* moves the camera along a path defined by a spline. *Simulate* sends a *tick!* to the ROBOT liaison *object*. When the ROBOT liaison *object* receives the *tick!*, it interpolates the transformations for each joint of the robot and sends *position+* and *orientation+* messages to each joint's actor. The cue labeled *both*, combines the *pan* and *simulate* cues, setting the start time for the *simulate* cue to be 1 second, i.e. the robot movement will start 1 second after the camera pan begins.

```
-- First describe the scene
SCENE new: scene_1
   cues = (pan, simulate, both)
   renderers = aPhigs
-- Next define the participants.
-- For brevity, we omit repetitive statements
ROBOT new: ge_robot
   time=0
   transfile= trans2.dat
   joint_1=(robot_part_1_1)
   -- joints 2 - 9 skipped for brevity
   joint_10=(robot_part_10_1);
COLLECTION new: robotparts
   members=(robot_part_1_1,
   -- robot parts 2 - 9 skipped for brevity
   robot_part_10_1);
-- Each joint in the robot is an actor
ACTOR new: robot_part_1_1
   modeler=model_1_1;
-- robot part instances 2 - 10 skipped for brevity
-- Each actor has a modeler
GEOMOD new: model_1_1
   universal=trans3.uni
   object=trans1;
-- model instances 2 - 10 skipped for brevity
-- Define cameras and lights
CAMERA new: camera_1
   position=(228, 34.2, 90.0)
   view_up=(0,0,1)
   focal_point=(28.0, 34.2, 36.0)
   view_angle=45
   clipping_range=(5.,700.)
   on!;
LIGHT new: light_1
   position=[camera_1 position?] on!;
PHIGS new: aPhigs
   actors= robotparts
   cameras= camera_1
   lights= light_1;
-- path circle, below, is a spline left out for brevity
```

```
CUE new: pan
   duration=[ge_robot duration?]
   start_actions=
    ("path_circle time = 0;", "ge_robot time= 0.;)
   tick_actions =
    "camera_1 position= ([path_circle tick!]);";
CUE new: simulate
   duration= [ge_robot duration?]
   start_action= "ge_robot time= 0.;"
   tick_action="ge_robot tick!;";
CUE new: both
   duration= [ge_robot duration?] duration+ 1
   start_actions="
       ge_robot time= 0.;
       pan time=0 start=0; simulate time=0 start=1;
   ";
-- Give start times for each cue
pan start = 0; simulate start = [pan end?];
both start = [simulate end?];
-- Now run the animation
scene_1 start!;
```

The animation script creates instances of *classes* (defined in C modules) and specifies values for their instance variables. The *objects* interact with each other by sending messages. The animation process begins with the message *start!* to *scene_1*. This message causes *scene_1* to advance its own clock, send *tick!* messages to each of its cues, followed by *render!* messages to each of its renderers. The cues manipulate actors, cameras, and lights by sending messages to them. The renderers, on receipt of a *render!* message, ask their associated actors, cameras, and lights for current settings, and produce appropriate movie frames.

9. SUMMARY

OSCAR made its first film in December 1984 when it consisted of twenty five classes including *actors*, *cameras*, *lights*, *scenes*, *cues* and *renderers*. Since its initial implementation, *OSCAR* has grown to contain over ninety classes, fifty eight of which are subclasses of the original classes, i.e. over half of the classes share code with other classes.

During this project we have made several observations:

- Applying the data abstraction process to animation produces a natural user interface with familiar terminology.
- The abstraction step of the design is critical and requires the most effort.
- The object-oriented approach partitions a complex system into manageable pieces. No single *object* is complex, but the system as a whole can deal with the complexity of the modeled process.
- The system is less fragile than others we have written. We make changes and additions proceed without fear of breaking the system.

10. ACKNOWLEDGMENTS

Object-oriented design methodology and computer graphics application development is a team effort in our laboratory. Other team members include: Michelle Barry (now with Star Technologies) and Dan McLachlan (now with Ardent Computer). The Animation Project at the Rensselaer Polytechnic Institute Center for Interactive Computer Graphics contributed to the ideas described in this system. Jon Davis of GE provided interfaces to the robot simulation system.

11. REFERENCES

[1] Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations," *AI Magazine*, vol 6, no 4, pp. 40-62, 1986.

[2] Goldberg, A. and D. Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, 1983.

[3] Cox, B., *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1986.

[4] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

[5] Booch, G., *Software Engineering with Ada*, Benjamin Cummings Publishing, 1983.

[6] Reynolds, C., "Computer Animation with Scripts and Actors," *Computer Graphics*, vol. 16, no. 3, July 1982, pp. 289-296.

[7] "Disney Takes the Lead with TRON," *Computer Graphics World*, vol. 5, no. 4, April 1982, pp. 41-45.

[8] Magnenat-Thalmann, N. and D. Thalmann, *Computer Animation: Theory and Practice*, Springer Verlag, 1985.

[9] Breen, D., Apodaca, A. and P. Ghetto, "The Clockworks," TR-86016, Rensselaer Polytechnic Institute Center for Interactive Computer Graphics, 1986.

[10] "PHIGS+ Functional Description," Computer Graphics, vol. 22, no. 3, July 1988.

[11] J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[12] S. Johnson, *YACC: Yet Another Compiler Compiler*, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[13] M. Lesk, *LEX - A Lexical Analyzer Generator*, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[14] *GEOMOD Reference Manual*, Structural Dynamics Research Corporation, 1983.