CRD

Corporate Research and Development

Schenectady, New York

AN OBJECT-ORIENTED GRAPHICS ANIMATION SYSTEM

by

W. Lorensen, M. Barry, D. McLachlan and B. Yamrom
Information System Operation

86CRD067

June 1986

Technical Information Series

GENERAL ELECTRIC

AN OBJECT–ORIENTED GRAPHICS ANIMATION SYSTEM W

W. Lorensen, M. Barry, D. McLachlan, B. Yamrom

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 concepts. Computer graphics researchers actively pursue the goal of realism to produce high–quality, three–dimensional animation systems. In following sections, this report will first review past work in object–oriented systems and 3D computer animation. Then, a design methodology is presented that applies the object–oriented philosophy to a 3D animation system developed at General Electric Corporate Research and Development. Finally, an implementation using C, a portable systems/application language, is described.

2. OBJECT–ORIENTED SYSTEMS

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 the context of object–oriented systems, these three principles complement rather than compete with each other. No compromises are required to apply all three to their limit.

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 the *object* and are only accessible to the outside world through the *object's* procedures, called *methods,* that are executed when the *object* receives a *message.* Messages are the only means by which *objects* can communicate with each other, and they provide a uniform mechanism for inter–object communication. An *object* is created by making a copy or *instance* of a particular *class* of *objects.* Classes are the only abstractions of these systems. Not only do

Manuscript received *3/13/86*

they define the data structures associated with the class but also the methods for manipulating the data. These data are called instance variables. When an object is created by instancing a class, the object receives not just the data structures but also the methods to manipulate the data structures. Through the mechanism called inheritance, new classes can be created by sharing the description of other classes and changing or adding to their data and methods.

A final important point: only the objects know their data structures and methods. Nothing outside the object can directly access these structures. If a new algorithm is required to do a task, the data structure may require change, but only the object itself is affected by this change. This property applies the software engineering notions of information hiding and modularity to data as well as procedures.

2.1 Prior Work

One early object–oriented system was Ivan Sutherland's Sketchpad [I]. Sketchpad is a general purpose graphics system for interactive creation and editing of pictures on a graphics display. Geometric transformations were applied to master (class) definitions of objects resulting in an instance of the geometric object. Although the concept of an object–oriented system was not defined in 1963, Sutherland's user–interface had many properties in common with such systems.

The Smalltalk effort [2] at Xerox's Palo Alto Research Center (PARC) is most often associated with object–oriented systems. This system uses the concepts of objects, messages, and classes to produce a programming environment and a user interface. It differs from other object–oriented systems in that it does not have any conventional typing and procedural constructs that might violate the rigorous application of objects and message passing. The only construct in the system is an object that even controls program flow. Each class has methods it uses to process messages, all of which takes place inside the objects. New classes can be defined by adding data and methods to other classes called super classes. When a message is received by a Smalltalk object and an associated method is found, it is executed; otherwise, the message is passed to the object's super class. This process proceeds until the message is either recognized or rejected. This hierarchical inheritance property allows the Smalltalk system to rely heavily on previous software and to build incremental systems without substantial software development.

The Flavor System [3] of Symbolic's Lisp Machine is an implementation of objects in a dialect of Lisp. This system extends the hierarchical inheritance concepts of Smalltalk classes to allow non–hierarchical combinations of classes called flavors. When a new flavor is created, it can inherit the attributes of multiple flavors. Methods for handling messages are defined as combinations of

methods from the other *flavors.* Conflicts can arise when two combined *flavors* process the same message in different ways, but the *flavor* concept resolves these conflicts in a uniform, prescribed manner. The Lisp Machine Window System [4] is a practical implementation of *flavors.* The Window System manages communication between processes and the user. The user communicates to processes through windows via a keyboard and a pointer device (mouse). Methods for windows have been classified into several *flavors.* For instance, all windows use the basic *flavor minimum–window.* It contains the minimum functionality that a window must have to behave as a window. The *window flavor* adds more sophistication. This *flavor* is the *minimum–window* plus methods (called *mixins)* to handle stream input, draw borders, draw labels, and do graphic operations. When a window is instanced, the process specifies the *flavor* and any initial values for internal states. The Lisp system returns an *object* descriptor of the particular window created. To communicate with the window, the process sends messages to the window *object,* the messages then prescribe a uniform syntax and semantics for *object* communication. For example, to draw a vector in a particular window, the process sends the message *:draw–line* to the window and a vector will appear. In summary, the user has complete control over the characteristics of the window and can easily communicate with multiple windows by sending messages to the appropriate *objects.* The cursors, menu system and keyboard are also handled via the *flavor* concept.

3. OBJECT–ORIENTED DESIGN

The principles and properties of object–oriented systems can be applied to the software design process. Examples in the previous section illustrated implementations of object–oriented systems. In this section, the emphasis is on the design process without regard to implementation. The methodology outlined is appropriate at both the preliminary and detailed design stages of the software engineering life cycle.

3.1 Booch's Methodology

Grady Booch *[5]* outlines one approach for designing systems for *objects.* His procedure consists of three steps:

1. Define the problem.
   This is the conventional process to describe the system to be built. Iterations with the other steps are required as new aspects of the system are discovered.
2. Develop an informal strategy.
   A written narration is used to describe the operations and *objects* of the system.

3. Formalize the strategy.
   Using simple rules and the narrative description of the system, identify the *objects* (nouns), attributes (adjectives), and operations (verbs) required.

Booch uses Adam to describe the visible interfaces to each *object* and the explicit operations that can be applied to the *objects.* This process is successively applied to refinements of the system. Booch uses this methodology to solve five design problems ranging from leaf counting on trees to a heads–up display system for fighter pilots.

At first glance, Booch's approach seems attractive. One just writes down a description of the system to be modeled, underlines the nouns (these become *objects),* then the verbs (these become operations) and translates the design into Ada. But he has not described a design methodology, only given guidelines for translating a design into an object–oriented implementation. Although his guidelines are useful for that step, moving from the problem definition step to the informal strategy is the most difficult step of the design process.

3.2 A New Methodology

This section presents an object–oriented methodology that is intended to define the overall design of the system to be developed. The new method does not extract *objects* from the design but builds the design from abstractions of the objects themselves. Therefore, the primary effort in this approach is the definition and characterization of the abstractions. The following steps are involved:

1. Identify the data abstractions for each subsystem.
   These data abstractions will be the *objects* of the system. Often the *objects* correspond to physical *objects* within the system being modeled. If this is not the case, the use of analogies, drawn from the designer's experience on past system designs, is helpful. This is by far the most difficult step in the design process and the selection of these abstractions influences the entire system design.
2. Identify the attributes for each abstraction.
   The attributes become the *instance variables* for each *object.* Many times, if the *objects* correspond to physical objects, the required instance variables will be 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 for each *object.* Some methods are required to access and update instance variables. Others execute operations singular to the *object.* The details of the methods' implementations need not be specified now, only the functionalities.

Ada is a trademark of the Deparment of Defense

4. Identify the communication between *objects.*
   This step defines the messages that *objects* can send to each other. This protocol must be decided on by the design team. Consistency in message naming should be a primary goal.

5. Test the design with scenarios.
   Scenarios consisting of messages to *objects* are needed to test the design's ability to match the system's requirements. In fact, the designers should write a scenario of messages for each requirement in the system specification.

6. Apply inheritance where appropriate.
   Once some *objects* have been designed, common data and operations often surface. These common instance variables and methods can be combined into a *class.* This 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.

This process must be repeated at each level of abstraction. Through successive refinements of the design, the designer's view of the system changes depending on the needs at 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. A new level of abstraction should provide some attribute or operation that cannot be expressed at the next lowest level. For example, the abstraction of geometric primitives might start with a polygon. **A** rectangle would be defined at the next level of abstraction, followed by a square.

4. COMPUTER GRAPHICS 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 thoroughly the models' environment. Transparency, translucency, shadows, illumination models, and surface properties are a few of the areas where research has produced algorithms resulting in more acceptable synthetic images. The current trend in computer graphics is to apply these advanced techniques toward the production of quality animation.

4.1 Prior Work

Although dozens of films produced using 3D computer graphics have appeared over the years, the literature has concentrated on the algorithms used to produce the images, not on the animation systems themselves. This is probably because few general purpose animation systems exist. Most computer–generated film production is done 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.

Reynolds [6] has worked for several years on the ActorIScriptor Animation System, ASAS. Originally started as a master's thesis at MIT, ASAS was used at Information International Inc., III, to produce sequences for the Disney movie TRON [7j. ASAS is implemented in Lisp and relies heavily on object–oriented concepts. ASAS actors are the participants in the animation, communicating by sending and receiving messages. Once an actor is started it remains a part of the animation until it stops itself or is stopped by another actor. Actors can also be given cues to appear or disappear. The processing of the cues is contained within the actors themselves. ASAS, as is characteristic of other Lisp–based systems, can use all the power of the Lisp interpreter by extending the capabilities of Lisp through new functions and forms.

The **MIRA** System **[8]** also extends a computer language, here Pascal. Abstract graphical types are defined to describe the participants in the animation. An animation is described by a sequence of scenes, where each scene has a name and is a sequence of statements manipulating actors, cameras and decor. Decor includes graphical objects that do not change within a scene.

The DIAL System [9], developed at Brown University, uses a clever two dimensional notation to describe temporal relationships. Events contain instructions that should be executed when the event is active. Time is represented on a horizontal line with special characters denoting execution, continuation of an event, and event suppression. The timing of an event is defined by giving the event name and a time line to control the execution of the event.

The efforts of a commercial computer graphics animation company, Pacific Data Images, are described in Reference 10. The company produces commercial computer–generated animation for broadcast television. Few specifics of the system are given but the overall production process is outlined. This systems differs from the previous systems in that an animation language interpreter was written in a high level language, C, rather than extending the language itself.

Finally, reference 11 describes a procedural–based animation system. Procedural systems are like object–oriented systems since a procedurally–modeled object is entirely described by its procedure and parameters. Rather than using message passing for event control, this system uses data flow. Each function has pre–defined input data paths through which it receives information and output data paths through which it transmits information.

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 artistic interpretation of a robot in a work environment may show the robot execute apparent 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 how the robot behaves, but need to understand how the commercial robot operates in a real environment. This is the key difference between this system and those described previously: the reliance on analysis rather than art. This also illustrates a problem that this animation system must address: the analysis software often already exists with its own user interface and data bases.

OSCAR, the Object–oriented Scene AnimatoR, provides an automated graphics animation capability for the efficient creation, control, and management of 3D 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, and robotics, with future interfaces planned to structural analysis and factory simulation software. The object–oriented design has produced a system that lends itself to interfacing with existing and future in–house and external software.

5.1 The Animation Process

Several steps are required to produce an animation. It is useful to review the process to see what steps can benefit from the proposed animation system.

1. Determine the intent of the animation.
   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. This creative step is left to the user.
2. Create a story and write a script.
   Creation of the story is a mental process that cannot be assisted, but the

description of the story as a script is useful for documenting, changing, and creating a final product. The script can be written with a text editor, or created with the *OSCAR Interactive Script Generator*. The scripts in *OSCAR* consist of statements in an object–oriented language developed as part of the system.

3. Run any simulations.
   OSCAR assumes that most animations depend on some computer model. Analysis runs must be made to provide the simulation results for the animation. Sometimes, the simulation may be complete before the script is written. Simulations are run by external programs called *analysts.*

4. Create geometric models of participants.
   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, more sophisticated geometric models are required for the rendering process: spheres and cylinders. Models are created by programs called *modelers.*

5. Render the geometric models.
   This step involves applying computer graphics algorithms to the computer geometric model, surface properties, lighting, etc. and producing shaded images for display. Rendering is done by programs called *renderers.*

6. Preview the animation.
   Fast, interactive preview facilities catch conceptual and mechanical errors in the animation. Within *OSCAR,* a special *renderer object* allows the user to preview animation on the Evans and Sutherland PS300.

7. Edit the completed frames.
   Frame editor objects create titles, credits, and special effects such as dissolves and fades that add a professional touch to the completed sequence.

8. Shoot the film or video.
   Recorder objects do this final step in the process that involves selection of frames and exposure of the film or video.

5.2 Major Subsystems

The subsystem breakdown in *OSCAR* delegates authority for the steps in the animation process and corresponds to software which is already available. We use an anthropomorphic flavor throughout the descriptions so that the correspondence with conventional movie making is maintained. This appears to be a

natural abstraction to use in the animation system. Figure **1** contains a diagram of the system.

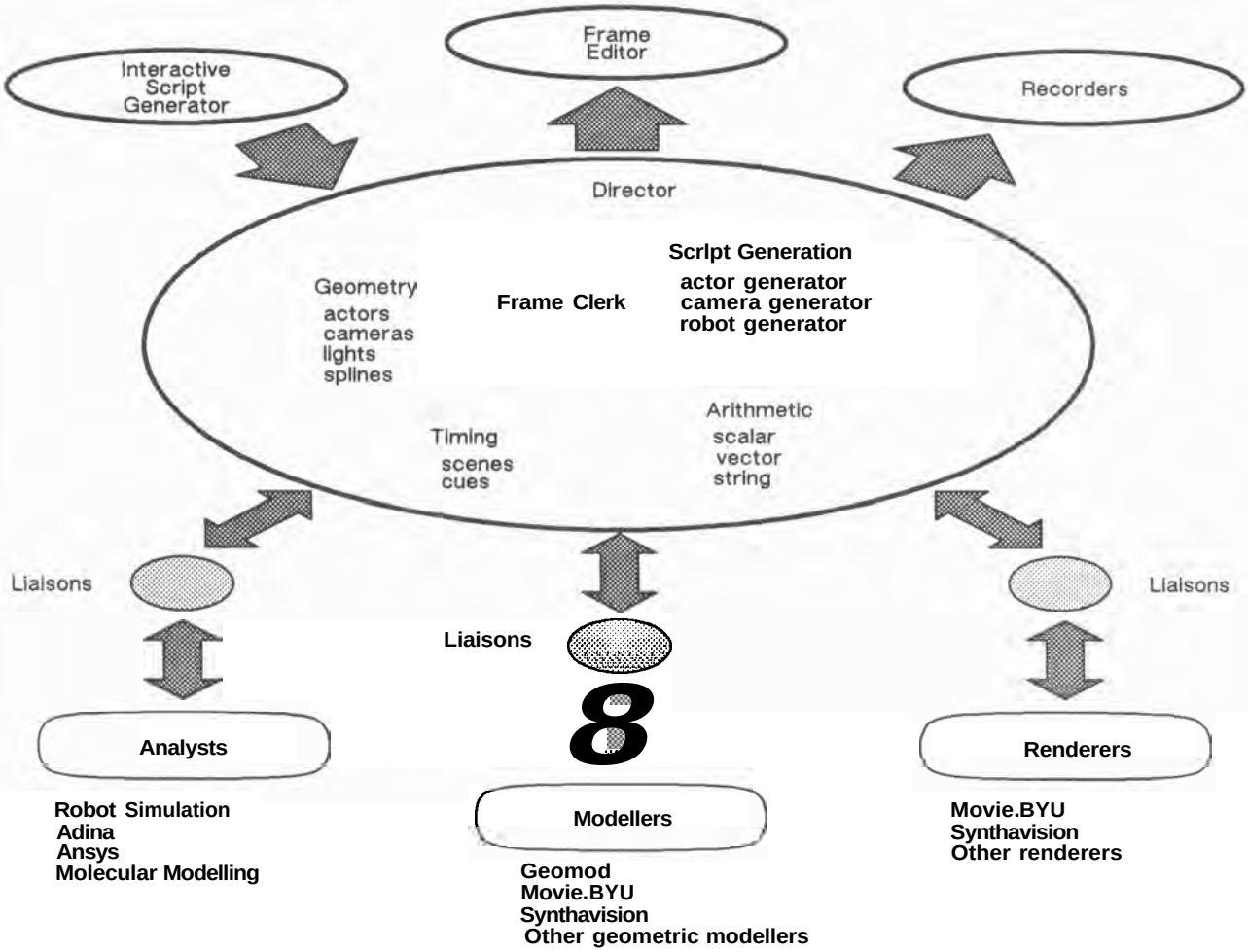

Figure 1.

Interactions between *objects* and programs are described below:

1. *Director* is a collection of *objects* that provides control over all the components of *OSCAR*. The Director reads and interprets a script and sends commands to the other modules to do part of the animation.
2. *Interactive Script Generator* is an *object* that 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. The data that the user inputs is interpreted and stored in a script file. Although the *Interactive Script Generator* provides a user interface for both the novice and experienced user, the ex-

perienced user can achieve more control by writing or editing the scripts with a text editor.

3. Frame clerk is an object that 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 objects that provide an interface 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. Some examples of the analysis packages that are or will be supported include: **ADINA,** a non–linear finite element analysis program; **ANSYS**, a general purpose finite element analysis system; MNDO and Gaussian 80, molecular mechanics programs; and an in–house robot simulation program. The interfaces to these software packages cannot be changed, so an analysis–specific liaison is required to 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, and Synthavison. 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 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 Design

Our animation language uses one statement structure that defines communication between objects. In each statement, 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
```

In the above description, *VALUE* is a floating point number, *NAME* is a string of characters, *STRING* is a quoted string, and *PREFIX* is an optional string. The language also allows for C [12] language style comments. Some special characters at the start of a line allow the user to do redirection of 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:

? indicates a request for the value of an instance variable.

= indicates the setting of an instance variable.

: is used for messages that require arguments, but do not specifically set an instance variable.

@ is used for indexing messages.

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

! is used for actions not requiring arguments.

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

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

Here an *instance* of the *object* ACTOR is created with a position and color. The *object* is rotated about its local x axis. An alternate statement, that improves readability, is provided to produce 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,

```
AnotherBox := Abox {};
```

A camera can be defined,

```
Acamera := CAMERA {
        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?];
```

5.4 OSCAR Classes

Currently over sixty classes exist in the system. These classes were selected using the design process described in Section **3.2.** A few of the classes are summarized here:

Scenes contain cues and renderers. In addition, scenes have durations (in seconds); resolution (in frames per second); and lists of actions to be done at the start, during their existence and after the scene is completed. Once a scene is started with a *start!* message, it executes any start actions, and then proceeds to send *tick!* messages to each of its cues. After the cues have processed their ticks, the scene sends a *render!* message to each of its renderers. On expiration, the scene 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 *cue's* time interval, during which it is active, is defined by a start time and end time. *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 sees if it should become active. If so, executes each of its start and tick actions and advances its clock. As long as its clock's time remains within the interval, the cue's tick actions are executed each time it receives a *tick!* message. Once the time interval is exceeded, the cue executes its end actions.

Actors are the geometric objects of the animation. They are described by models that are accessed through liaisons to specific modelers. They have position, origin, orientation, color, and visibility. Their visibility is controlled with *on!* and *off!* messages.

Cameras are the means by which the animation is viewed. In our implementation, the Foley and Van Dam [13] viewing transformation pipeline is used. Cameras can be moved, rotated, and turned on and off. Cameras have no geometric representation so that if one is in the field of view of another, it is not seen in the animation. Although multiple cameras can be present in a scene, only one camera can be on at one time for a given renderer.

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

Renderer Liaisons are *objects* that convert the geometric data structures of actors into a format that can be rendered by a specific renderer into raster or vector images. Typically, when a renderer liaison receives a *render!* message, it requests position and orientation information from its assigned actors, lights, and cameras. Normally, the renderer liaison creates a command

file that can be run at a later time to do the renderering. Specific renderer liaisons inherit some instance variables and methods from a generic renderer class. This class contains data and procedures that are applicable to all renderers. A specific renderer liaison need only provide code for details required by its assigned renderer.

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 are used to manipulate the raster images created by renderers.

Recorders are objects that 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, and collections.

6. IMPLEMENTATION

The system has been implemented in C on a Digital Equipment Corporation **VAX** 111780 running VMS, a Sun Microsystems workstation running Unix, and an IBM PC/AT running Xenix. The parser was produced using YACC [14] and LEX [15]. The parser is an object so that other objects can pass parse: messages to it and do parsing at run time. Classes are implemented as C modules. C struct's are used to define the instance variables, but the structures themselves are declared static so that they are not visible outside the module. There is a standard header required for each class and it includes the object name, super class, debug information, and other general instance variables.

Every message is implemented as a C procedure and each class has a method dictionary that contains the name of each message and the appropriate procedure to invoke. Hierarchical inheritance of both instance variables and methods has been implemented. The'message handling is done through a message object that is passed an instance name or pointer, message, and argument list. The message object searches the instance's method dictionary. If a match is found, the appropriate procedure is invoked. If not, the method dictionary of the object's super class is searched. This continues until the highest object in the object hierarchy is reached, at which point an error is printed at the terminal. Variable argument handling is implemented by an argument package that keeps a stack of sets of arguments. Objects return and receive arguments through this mechanism.

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

The system now consists of over 100,000 lines of C code. Extensive use is made of C macros to reduce the code for data structures and methods that query or update instance variables. Reference 16 contains a detailed description of the implementation.

7. A SAMPLE ANIMATION

This is a description of a scene showing the results of a robot simulation system.

7.1 The Analysis System

The analysis package is an in–house system capable of predicting articulated robot motion given starting and ending positions of the 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 geometry of the robot required by the analysis system is not elaborate. Prisms are used to model members and cylinders are used to model joints. However, for display purposes, more realistic representations are used. Here, the robot has been modeled using GEOMOD [17], a commercial modeling package from Structural Dynamics Research Corporation, SDRC. The transformations produced by the simulation are applied to the vertices of the polygons produced by GEOMOD. The renderer used is MOVIE.BYU [18]. Liaisons for the robot simulation, GEOMOD, and MOVIE.BYU provide the interfaces between each external program and the animation system.

7.2 Script Generators

A robot usually has many components. Rather than burden the user with instancing actors for each component, a *ROBOT* generator object scans user specified transformation and modeling files and generates a script. This script defines each joint as an actor with its required modeler liaisons. Also, cues for key work points in the analysis are generated. This generated script is sent to the parser object by the *ROBOT* generator and is also stored in a text file that can be edited at a later time. We have found this concept of a generator to be useful in reducing the user's effort in starting a script while still maintaining the flexibility offered by the language.

7.3 The Script

Three cues are used in this scene. The pan cue moves the camera along a path defined by a spline. The simulate cue sends a tick! to the ROBOT liaison object. When the ROBOT liaison object receives a tick!, it interpolates the transformations for each joint of the robot and sends position and orientation increments to each joint's actor. The cue labeled both, combines the pan and simulate

cues. However, it sets 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_1  := SCENE {
            cues = (pan, simulate, both)
            renderers = byu
        /* duration = 3 * robot duration + 5 seconds */
            duration = [ge–robot duration?]
            duration + [ge–robot duration?]
            duration + [ge–robot duration?]
            duration + 5
};

/*
 *  Next define the participants.
 *  For brevity, repetitive statements have been eliminated.
 */

ge–robot := ROBOT {
            time=0
            transfile= trans2.dat
            joint_l=(robot_part_1_1)
            /* joints 2 - 9 skipped for brevity */.
            joint_10=(robot_part_10_1)
};

robotparts := COLLECTION {
            members=(robot_part_1_1 ,
            /* robot parts 2 - 9 skipped for brevity */
            robot_part_10_1)
};

/*
 * Each joint in the robot is an actor
 */

robot_part_1_1    := ACTOR {
            modeler=model_1_1
};
```

```
/* robot part instances 2 - 10 skipped for brevity */

/*
 * Each actor is assigned a modeler
 */

model_1_1 := GEOMOD {
            universal=trans3.~ni
            object=trans1
};

/* model instances 2 - 10 skipped for brevity */

/*
 * Define cameras and lights
 */

camera_1 := CAMERA {
            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_1 := LIGHT {
            position=[camera_1 position?]
            on!
};

byu := MOVIE_RENDERER {
            actors= robotparts
            cameras= camera_1
            lights= light_1;
};

pan := CUE {    /* path circle, below, is a spline left out for brevity */
            duration=[ge_robot duration?]
            start-actions= ("path-circle time = O;", "ge_robot time= 0.;)
```

```
            tick_actions = "camera_1 position= ([path_circle tick!]);"
};

simulate := CUE {
            duration= [ge–robot duration?]
            start_action= "ge_robot time= 0.;"
            tick_action="ge _robot  tick!;"
};

both := CUE {
            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!;
```

Notice that the animation script consists of creating instances of *objects* (which are defined as C modules) and specifying values for their instance variables. The *objects* interact with other *objects* by sending messages. The whole animation process is started 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 typically 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.

8. SUMMARY

An object–oriented approach to the design and implementation of a computer graphics animation system has been described. During this project we have made several observations:

- Applying the data abstraction process to the animation production cycle has resulted in a natural user interface using familiar terminology.
- The abstraction step of the design is critical and requires the most amount of time and effort. The path taken at this point in the design will drive the design of the system.
- The object–oriented approach allows the natural partition of complex system into manageable pieces. No single object is complex, but the system as a whole can deal with the complexity of the process being modeled.
- The system seems less fragile than others written previously. Objects can be modified and added without fear of breaking the system.

The system as described in this report serves as a starting point for a system to be enhanced over the next few years. Liaison objects will be introduced to interface to more analysis, modeling, and rendering systems. From this system, we intend to learn more about the application of the object–oriented approach to other areas of computer graphics and computer science.

9. ACKNOWLEDGMENTS

The Animation Project team at the Rensselaer Polytechnic Institute Center for Interactive Computer Graphics has made many technical contributions to the ideas proposed in this report. This group is engaged in a parallel effort to build a graphics animation system. Jon Davis of General Electric has provided interfaces to the robot simulation system.

10. REFERENCES

[1] I. Sutherland, Sketchpad: A Man–Machine Graphical Communication System, PhD Thesis, MIT, 1963.

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

[3] Lisp Machine Manual. Symbolics Inc., 1983.

[4] D. Weinreb and D. Moon, Introduction to Using the Window System, Symbolics Inc., 1983.

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

[6] C. Reynolds, "Computer Animation with Scripts and Actors," Computer Graphics (Proc. of SIGGRAPH '82), 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] N. Magnenat–Thalmann and D. Thalmann, "The Use of High–Level Graphical Types in the Mira Animation System," IEEE Computer Graphics and Applications, December 1983, pp. 9–16.

[9] S. Feiner, D. Salesin and T. Banchoff, "Dial: A Diagrammatic Animation Language," IEEE Computer Graphics and Applications, September 1982, pp. 43–54.

[10] R. Chuang and G. Entis, "3–D Shaded Computer Animation – Step by Step," IEEE Computer Graphics and Applications, December 1983, pp. 18–25.

[11] H. Hedelman, "A Data Flow Approach to Procedural Modeling," IEEE Computer Graphics and Applications, January 1984, pp. 16–26.

[12] B. Kernighan and D. Ritchie, The C Programming Language, Prentice Hall, 1978.

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

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

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

[16] W. Lorensen, M. Barry, D. Mclachlan, and B. Yamrom, Object–Oriented Software Development in a Non–Object–Oriented Environment, General Electric TIE Report, Schenectady, New York, 1986.

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

[18] H. Christiansen and M. Stephenson, "MOVIE.BYU – A General Purpose Computer Graphics Display System," Symposium on Applications of Computer Methods in Engineering, vol. 2, p. 759, UCLA, 1978.