SYSTEMS ASPECTS OF COMPUTER IMAGE SYNTHESIS AND COMPUTER ANIMATION

James F. Blinn

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, MS 201-209
Pasadena, CA 91109

ABSTRACT

The production of complex computer generated images generally requires the interaction of many different programs. This paper will describe some of the different techniques which can be employed to carry this out. Emphasis will be on the system developed by the author and used in the production of several planetary flyby animations and some sequences for the Cosmos series. Details of how this system was employed to produce some of the scenes in these films will also be discussed.

1. INTRODUCTION

Over the past few years I have been involved in the production of several computer animated sequences of various subjects. These have required the use of quite a wide range of techniques and have motivated the development of several different software packages. After completing these projects I have then had the opportunity to review the mechanisms developed in the "heat of production" for the various special purposes required and to attempt to draw conclusions about how the image making process can better be performed. Some patterns are beginning to emerge and this paper is a first attempt to document the conclusions I have come to. This is a somewhat empirical approach to systems design. That is, several special case systems are put together motivated just by the needs of some particular project. They are then are analyzed to see what things they seem to have in common. In doing this sort of examination it is important to realize that you cannot prove that your assertions are correct in the same sense you can prove a mathematical theorem. The best that can be said is that the mechanisms described below seem to work well for the problems to which they have been applied.

1.1 Multiple Program Structure

The first conclusion is that you cannot expect to develop one program that does everything. It is much better to have a collection of smaller programs, each of which is relevant to some small portion of the whole process. This allows the sections of the process to be thought about and debugged individually without needing to consider all aspects of the system at once. This conclusion is somewhat forced upon the current system since it is implemented on a small computer (a PDP 11) and a program cannot be very large and still fit inside the machine. This has a hidden advantages however, in that it forces the designer to think about the modularity of the system earlier in the game than if programs were allowed to grow to very large sizes before reaching the limits of the hardware. While we are now moving to a larger machine the principles of modularity will still apply.

The next conclusion is that it is useful to have several programs that do effectively the same things but in different ways. The different ways refer to the degree of generality/specificity of the program to the image being generated. On the one hand, one may employ a general purpose program that has very general but low level primitives. The generation of the desired image then requires a large amount of manual input of data and parameters. On the other hand, one may employ a special purpose program which "knows" how to make a class of images such as the one desired. In this case much of the data input and positioning will be done automatically by the program based on its knowledge of the "universe of discourse". The general but labor intensive approach is applicable to situations where only a few images are required of something not previously drawn. The latter case is applicable to situations where many images are required on a restricted class of subjects. For example, we have a general purpose articulated object animation system which will draw arbitrary articulated objects consisting of arbitrarily shaped element. The user must define the shape and placement of each object explicitly. On the other hand, there is a very special purpose simulation system for scenes in the solar system. One need merely say something like "draw a picture of Saturn as seen from the moon Mimas on Aug 28 1981". The program will automatically calculate the positions of the objects, pick a good view, and call upon the appropriate databases to produce the picture. An animation system benefits from having both these types of tools.

1.2 Loose Coupling

Once we have accepted the necessity of a diverse collection of individual programs in the system, the main problem becomes one of making them work together properly. This is mainly a problem of getting the necessary data to the appropriate program at the right time. Keeping track of which programs know about what data and how it moves from program to program is, I feel, the most important thing to know about a system in order to understand it. It is more important, in fact, than knowing the details of what the programs do with the data. The details of this process are, in fact, the main subject of this paper.

To maintain flexibility, the concept of "loose coupling" is important. This means that the system does not have to be "totally integrated".

Certain groups of programs may be able to interpret the same data base, others just some subset of it, others none at all. Various similar things may be done in different ways in different contexts. This may be contrary to what one might think of as the "ideal system", that is where everything is rigidly consistent and everything fits into one set of rules. The problem is that this is very constraining. It assumes the initial design was able to foresee all possible future needs and provide for them. Such a system stultifies experimentation with new ideas. An analogy can be made with various other natural and artificial systems, such as computer operating systems, the telephone system or the human brain. In each case, the most useful and flexible system is a conglomeration of (often incompatible) bits and pieces developed at different times for different purposes. We cannot allow the system to become too anarchistic, however. In parallel to the addition of new ideas to the system, effort must be continually expended to more completely integrate existing elements. The determination of when to follow the existing rules and when to be radically different is not really quantifiable and is more a craft than a science.

1.3 Unifying Techniques

There are two main unifying elements in the system discussed here. One is the output device, the frame buffer, and the other is the input mechanism, a common command language interpreter.

1.3.1 Frame Buffer Synergy

The fact that all image generation programs use the frame buffer as their output device produces what I call the "Frame Buffer Synergy". This occurs because the frame buffer enforces upon many programs a common database for representation of images, namely the pixel array. Different programs do different things to the frame buffer but the output of one program is automatically suitable as the input of another if they all operate on the frame buffer. A system, then, naturally accumulates a collection of different programs that "do things" to the frame buffer. They may be written at different times, by different people, without any knowledge or expectation of each other’s existence. But they all can be used to aid in the production of the same picture.

1.3.2 The Command Language Interpreter

The other major unifying mechanism is a common command language interpreter which most programs use to process their input. While the command syntax is very simple the interpreter system contains some subtle mechanisms to greatly enhance the programmer’s ability to utilize various previously defined code modules and subroutine libraries and integrate them into a new main program. To begin with, commands consist of a keyword followed by a list of parameters. The main program calls the interpreter to get a command, branches to code specific to that command, interprets parameters and goes back for the next command. Initially commands come from the terminal but a built—in command can redirect input to come from a file. Such file redirection can be nested. Parameter interpretation optionally allows parameters to take on symbolic values and provides for symbol/value assignment.

Finally and most importantly, the command language interpreter allows a group of commands to be interpreted by a separate set of library routines. By referencing the library, the programmer then automatically provides the user with any input commands needed to manipulate the entity that the library deals with. For example consider a library to manipulate transformation matrices. Programmer callable routines exist for maintaining a "current transformation", a matrix stack, and for applying rotations, translations, scales, etc. to the current transformation. The sub—command interpreter routine will then accept a command keyword received from the CLI parser and generate a call to the appropriate routine.

The command interpretation loop of the main program then includes a call to this sub—command interpreter as well as its own commands to draw some primitive object according to the current transformation. The user of the program will generally first enter the transformation commands to define the position of the object and then enter the drawing commands.

Such topical command groupings and their associated function libraries and internal databases have, in fact become a major programming element within the system. Such a structure corresponds in many respects with the concept of a "class" in Smalltalk systems. Ultimately a main program would simply consist of a series of initialization calls, the command interpretation loop, and nothing else.

The communication of data between programs can take place in many ways. Some methods are facilitated by certain operating systems. UNIX provides the useful mechanism of pipes, Multics allows dynamic linking of libraries, etc. Our system performs this communication with the logical equivalents temporary files, usually in the format of command lists. One program would then write out a command file in the appropriate format for another. This is slower than some other schemes but it has the advantage of leaving a record of the intermediate stages of picture generation for easy retry and debugging purposes.

1.4 Document Overview

The following sections will discuss the details of various of the image making tools and how they communicate. The programs are divided into three rough categories.

1) Modeling
2) Simulation
3) Rendering

We will describe them more in the order in which they were developed rather than the order of invocation or use.

2. RENDERING — LOW LEVEL

This section will deal with various low level rendering programs. The term low level refers to the fact that they each deal with one type of object and will typically be used in sequence by the high level rendering programs of section 5 to build up a composite image.

We have, in our tool kit, the standard collection of rendering programs which operate on databases consisting of polygons, quadric surfaces or patches. In addition there are several rendering programs for special cases of these shapes. For example there is a program that draws textured spheres, used for drawing planets. There is another program for drawing rings, etc. The special purpose programs take advantage of the special properties of their specific object to increase the rendering speed or improve the visual appearance.

2.1 General Operation of Rendering Programs

The particular technique used for high level image composition will determine the general strategy which must be used by the low level programs. In our case the high level image composition is performed by temporal priority. To be compatible with this technique, each low lever rendering program simply overlays its image in the frame buffer on top of what is there already. Currently, all programs operate in scan line order.

In each case, the general operation of the program is the same. The operator gives commands to set various viewing parameters, surface properties, and modeling transformations. Other commands cause the appropriate primitive element to be generated. For some special case programs, a single DRAW command will initiate the rendering of the appropriate object according the currently set viewing/shading parameters. For programs that deal with collections of more general primitives, each primitive drawing command passes the data down the pipeline to be accumulated in a buffer. The DRAW command then initiates the sorting and rendering of this buffer. For debugging/testing these commands may be entered manually on the keyboard. Alternatively, just the initial view selection comes from the keyboard and the input is redirected to a file to read a more complex model for rendering. When used with the high level image rendering programs, the view selection commands are automatically generated by another program.

2.2 Special Purpose Renderers

There are a wide variety of special purpose renderers in use for various purposes. For the most part they are fairly simple programs that overlay some fairly simple image in the frame buffer. Some examples are: a star field drawing program that references a large star database and draws anti—aliased dots on the screen, or a program that draws a two dimensional blurred spot to represent the sun. Some of the special purpose programs are more complex and deserve special mention.

2.2.1 PLANET (Draw Textured Ellipsoids)

An example special purpose rendering program is PLANET, which draws textured spheres. Since it draws only one sphere at a time it doesn't bother with Z sorting or testing. In addition to the the standard viewing and lighting specification parameters it has some other special purpose features necessary for space scenes. The most obvious is texture mapping. The texture map is referenced while the image is scanned out to set the surface color, normal vector perturbation, etc. Such maps are paged in and out of disk files while the image is rendered using a LRU paging strategy. Each "page" in this case is a 32x32 square pixel sub—array of the map. This method of subdividing the map minimizes the chance of thrashing in the texture file. Another special feature provides for shadowing effects of a ring structure surrounding the planet. For each pixel in the image the appropriate geometric calculations are performed to intersect the line between the planet surface and light source with the ring plane. The radius of this intersection is then used to index a one dimensional texture map containing the radial density distribution of ring particles. The brightness of the pixel is then reduced appropriate to the blocking effect of the rings. Another special feature is eclipse simulation. This is another shadowing effect dues in this case, to another spherical body in front of the sun. In this case, for each pixel, the programs determines the angular separation between the occulting body and the sun. This is used to calculate the proportion of the sun's disk which is blocked off by the eclipsing planet and again reduces the intensity of the pixel appropriately. This has the effect of simulating the umbra/penumbra properly for eclipses. Since these two shadowing calculations slow down the rendering process they may be enabled by a global switch within the program, and are thus only used when necessary.

2.2.2 RINGS (Draw Textured Translucent Disks)

Another special purpose program is RINGS which draws an arbitrarily oriented circular translucent disk. It will, in fact draw only half the disks where the break occurs at the plane parallel to the viewing direction which passes through the center of the disk. This split is used to properly draw a ringed planet by running RINGS to draw the back half, then PLANET to draw the disk of the planet, and RINGS again to draw the front half. The intensity and transparency of the rings are calculated using a reflection model appropriate to clouds of particles. The density and albedo of the ring particles is taken from a one dimensional texture pattern file which is indexed by the radius of the visible point from the ring center. In addition, there is included a procedural model for some radial structures in the rings which orbit the planet at different rates at different radii.

2.2.3 TEXFLY (Square Texture Mapper)

Another useful special purpose program is TEXFLY, which has a single textured square as its sole primitive. This is similar to a program called TEXAS by A.R. Smith at NYIT. The square can be arbitrarily scaled and oriented in three dimensions and rendered with various intensity and transparency patterns. The very simple geometry of this shape makes it run quite quickly. It has been used as a post processing step to overlay some surface features on the Voyager spacecraft.

2.3 General Purpose Renderers

2.3.1 The Three Pass Process

More general programs which operate on collections of primitives (polygons, patches etc.) in scan line order are usually separated into three passes. The first pass simply interprets viewing/modeling transformations, assigns shading parameters, and transforms the primitive objects and accumulates them in a buffer. The second pass sorts this buffer in Y order. The third pass renders the image from the Y sorted list. To maximize available buffer size and thus the complexity of objects we can draw these three passes are implemented as separate programs. The V sorted buffer is implemented as a temporary file. The I/0 operations on this temporary file do slow down the process a bit, but not a great deal. The advantage gained is that the rendering pass only needs maintain in main memory only those elements which are active on the current scan line.

Three sets of programs have currently been implemented using this strategy. They deal with polygons, bicubic patches, and "blobby molecules" used in the Cosmos DNA sequence. The databases to describe shapes to these programs are then simply command files which can be generated in a variety of ways. In addition, for speed, pass 1 of the polygon processor can also interpret a binary version of a command file. A simple pre—processor reads the text command file and generates the binary file. Thus objects which are going to be drawn repeatedly may be debugged via the text version of the model and then "compiled" into the binary version.

2.3.2 Special Purpose Pass 1 Processors

The use of temporary files for the Y list allows another degree of flexibility. For certain cases the geometry of a particular object allows a simpler internal representation than explicitly defining each individual primitive. In this case a special program can use this representation to generate the Y list file directly in sorted order. This would then replace the general purpose pass 1 program and eliminate the need to run the pass 2 sorter.

One example of this technique was used in the DNA simulation. In this case, the large macro—molecule is made of a relatively few monomers of 25 to 30 atoms each. A large storage and time reduction can be achieved by representing the molecule in terms of the locations and orientations of these monomers and storing the definition of a monomer only once. The V list of the individual atoms may be generated directly by sorting the monomers first. This list is then scanned in order of their Y appearance, expanding the monomers into the explicit atom list.

Another example concerns a simple terrain rendering scheme. The altitudes of a region of terrain are generated assuming a regular grid spacing. These can be stored in an appropriately scaled byte array. Pointers into this array are then sorted in Y. A global Y scan then directly generates a Y sorted polygon list by assuming the connectivity of the points due to the regular grid spacing. This technique was used to simulate a fly—over of a crater on the moon Mimas for the second Voyager Saturn encounter movie.

3. MODELING

The modeling process is basically that of database generation. Such databases generally consist of text files of commands to the various rendering programs, and binary files of texture maps.

3.1 Geometric Modeling

The most often used modeling program in the system is the text editor. Quite a few of the modeled objects were generated essentially manually by reading and measuring blueprints or other diagrams and simply typing the coordinates into a file, along with the appropriate commands for the renderer. This approach is not as unpleasant as it may seem however. In many cases, the actual coordinates or sizes of objects are already marked on the blueprints. In additions certain "medium level" primitive commands are provided in the pass 1 portion of the rendering program. These consist of single commands for such frequently encountered shapes as boxes, surfaces of revolution, or tubular struts. The medium level primitives are then automatically expanded into the proper set of low level primitives when the pass 1 executes. In addition, the inclusion of comments within this database file provides a valuable documentation for future modifications to the model. Finally, the use of symbolic parameters for various quantities is already built in to the command language interpreter. Parametrized models are therefore easy to create. It must be noted, however, that if this technique is used, any programs processing the model might need to keep these parameters in unexpanded form to properly process the now implicitly procedural model.

In addition to manual editing, there are a few simple menu/tablet driven modeling programs. These programs all allow more natural "drawing" modes of input to edit the location, sizes and types of object primitives. In each case the programs can read in a text file containing commands to the rendering program, build an appropriate internal data structure, allow the user to edit the structure interactively and finally write out a new command file. Such programs currently exist for patch design, polygon digitizing, and composite object design using a list of arbitrary primitive shapes. In the latter case the user is not so much designing shapes as a "tree of transformations" as discussed below.

Various special purpose design programs have also been written for various projects. For example several molecule generators were devised for the DNA project which placed Hydrogen atoms in appropriate places on molecules or which generated randomly positioned blobs to define an enzyme. Basically, given the definition of the format of the text commands for a particular rendering program it is fairly easy to quickly put together programs to generate data in that format via some desired algorithm.

3.2 Texture Pattern Generation

In addition to designing shapes it is also necessary to generate texture patterns for the various texture mapping programs.

Simple one dimensional textures (e.g. rings) are generated via a general purpose curve manipulation command. This provides a means of performing various simple arithmetic operations on tables of numbers derived from previously digitized images.

Two dimensional texture patterns are derived in a variety of ways. Generally, a texture pattern is taken from a region of the frame buffer so that any image synthesis or image processing program can be used to generate or alter a pattern. We have a collection of general purpose random number drawers, image rotation and stretching programs, image filtering programs, etc. In additions some special purpose programs have been written to process images of moons into maps. These effectively run the image synthesis process in reverse, distorting the image into a map projection.

Finally, frame buffer painting programs can be considered as a database generation tool for texture mapping. Certain special features have also been added here that are useful for spherical mapping. One interesting example concerns the generation of the terrain map for Mimas. A photograph of the moon gives a good general feel for the topography of the surface. Automatic methods for extracting this topography from the sighting parameter were not successful, however. A topographic map was finally obtained by using a painting—style program in "pantograph" mode. A map projected image of the surface was placed on the bottom half of the display and the operator traced out the visible craters. The craters were simultaneously applied to the actual map on the top of the screen by means of a special "carving" brush mode. This mode added or subtracted values from the image to raise or lower the generated terrain.

4. SIMULATION

A simulation program is basically the prime mover of the animation. Its main function is to alter various numerical parameters that define the appearance of the frame. A subsidiary, but very important, function is to provide a schematic preview of the animation on some fast output device. This is typically a line drawing display, the preview is effectively a "line test" of the animation.

4.1 Techniques

There are two general categories of parameters which change from frame to frame. They are 1) transformation parameters and 2) anything else. Transformation parameters are singled out especially here because they are so important. Typically a scene will be defined in terms of a "tree of transformations". This consists of the set of nested rotations, translations and scales that place each object in its correct position at its correct size. The nesting of these transformations allows objects to be arbitrarily articulated. The numerical values assigned to these transformations can then be varied to give a wide variety of motions. Many animations require no more than this. Other more general types require, in addition, alteration of brightnesses, texture patterns etc.

There are, in turns two general ways of specifying the way parameters vary, called here incremental and absolute. In the absolute mode, the value of each parameter is specified in a table for each key frame. When an intermediate frame is to be generated, some form of interpolation is performed on the surrounding table values. In the incremental mode, the program maintains the current state of the scene and is instructed to perform incremental modifications to it according to some rules. Absolute mode is usually more convenient to use when designing animation since it allows previewing of frames in arbitrary order or repeated playbacks of frame number ranges within the script. Incremental mode is necessary, however, for situations in which the connectivity of some data structure must be modified or in "particle pushing" types of physical simulations.

4.2 Tools

4.2.1 System Commands

One animation tool that everyone with a computer has immediately is the command language of the operating system. Most operating systems allow for commands to come from files and, in addition, allow for looping and symbolic variable manipulation within the command file. This mechanism has a tremendous advantage in that it is interpretive. Changes to the sequence of events necessary to make an image can therefore be made quickly with the text editor. No programs need to be changed. A disadvantage of using just the system command language is that it is quite difficult to perform line tests to verify that the animation will come out as desired. Also, since the command language was not designed for this purpose, its use sometimes becomes somewhat inelegant. The advantages in generality provided by the interpretive nature are substantial however and the method described below allows its use in combination with more specialized simulation programs.

4.2.2 General Purpose Articulation Program

Perhaps the most useful programming tool designed specifically for animation is the general purpose transformation tree articulator. This program internally stores a transformation tree and a set of line drawing of the primitives. The transformation tree structure can have "subroutines", i.e. higher order primitives defined in terms of transformation trees of lower order primitives. The program can then generate line drawings of the resulting environment by interpreting this structure. It allows the user to manipulate the parameter values for the various transformations in the tree and redraw the resultant image. Such parameters are assigned symbolic names. A named variable can be set either via keyboard commands, can be adjusted under knob control, or can be interpolated between keyframe values. Systems of this sort, dealing purely with line drawings, have been implemented by DeFanti, O'Donnell and Olson, and probably several others.

In order to interface to the rendering programs such a system needs just a few more 'hooks' to dump out the transformation tree with all parameter assignments made and with all tree 'subroutines' expanded explicitly. Garland Stern's BOOP system at NYIT has these capabilities as does the ARTIC program at SPL. In order to be applicable to the widest range of situations certain things must be done. The first of these refers to the definitions of the primitives. In order to be applicable to a wide range of primitive types (e.g. polygons, patches, blobby molecules or something new not invented yet) the ARTIC program makes no assumptions about the solid modeling scheme for the primitive. It sees only a definition containing a schematic line drawing version of the shape. The user is responsible for creating this file in any one of several ways. Most easily, there are several pre-processors for the various standard modeling primitives (polygons, surfaces of revolution, patches) which read the solid model file (which is going to be passes to the rendering program) and generates a line drawing version. Here, the solid model file is the "main" version of the database and the line drawing version is just a derived version of it. Alternatively, the user could generate a simplified form of the line drawing purely manually.

4.2.3 Special Purpose Space Simulator

For various applications the values of the transformation parameters may need to be generated by some functional calculation rather than by explicit numeric settings. An example of this is the space simulation program used for the Voyager fly-by movies. This is a special purpose program for simulating planetary astronomy. Its two main features are the mathematical modelling of the paths of the planets, moons and spacecraft according to Kepler's laws and a fairly sophisticated view selection algorithm for determining viewing position and direction in terms of some visually meaningful parameters.

4.2.3.1 Physical Simulation

The main structure of the modeled environment consists of a built-in transformation tree for the planetary system. This positions the moons and spacecraft relative to the planet, orients the planet and moons with their pole vectors pointing in the appropriate directions and spins them about their poles at the correct rates. In addition the spacecraft is oriented and articulated in an analog of the manner done by the onboard computer of the spacecraft. The values of positions, and rotation angles are made dependent upon the single parameter TIME. Whenever time is altered the appropriate calculations are performed to update these values.

4.2.3.2 View Selection

Since the positions and orientations of the objects are completely specified by the time of the simulation, the only freedom we have in specification of the image is the viewing location and direction. There are several viewing modes which may be employed to generate these two parameters.

Omniscient Mode

The main viewing mode is for an omniscient observer situated at some vector offset from any one of the simulated bodies, called the 'from' body. This vector may be specified in one of two ways, as a fixed x,y,z vector or as a fixed distance and an automatically calculated direction. The latter mode determines what direction relative to the 'from' body the observer must be in in order for the 'from' body to appear on the screen at a given x,y location. The viewing direction is also determined according to one of two modes. It may be explicitly specified in terms of its x,y,z coordinates. Alternatively it may be specified in terms of another simulated body, the 'at' body. In this mode the viewing direction is automatically calculated to be that which causes the 'at' body to appear at a given x,y screen location. For certain combinations of from/at modes there is no closed form equation to satisfy all the constraints. A solution is then determined numerically.

Camera Mode

An alternative viewing mode is to simulate a view through the onboard cameras of the spacecraft. In this case the spacecraft modeling parameters are examined and the appropriate view is generated according to the pointing direction of the camera and the spacecraft orientation.

Planet Surface Mode

This mode simulates a viewer sitting on the surface of one of the simulated bodies. The viewing position is specified in terms of the latitude, longitude and altitude above the surface. The viewing direction is specified in terms of the azimuth and elevation of the view direction relative to the local horizon and north direction.

4.2.3.3 Switching Viewing Modes

The various viewing modes provide a great deal of flexibility in finding interesting views in a given situation. One important feature is also necessary for smooth animation. Whenever the user switches modes the program will automatically calculate the appropriate numerical parameters for the new mode which will generate the >same< view. This allows easy transitions from one mode to another without introducing jumps in the picture.

4.2.3.4 Animation Commands

The animation of space scenes is carried out in absolute mode. There is a table of the values of the various viewing parameters as well as a few spacecraft articulation parameters, for each key frame. A new key frame is usually generated by adjusting various parameters under knob control. A command is then given to record the current parameters in the animation keyframe table at a certain keyframe number. When an animated sequence is played back, all the table values are interpolated appropriately and given back to the view generation routines.

When the space simulation program is instructed to make a color frame it writes out the values of all the internal parameters and invokes the frame buffer scene scheduler program described in section 5.

4.2.4 Special Purpose DNA simulation

The DNA simulation system is another special purpose articulation program which has many built—in parameter calculations and transformation tree structures. In this case, the physical simulation is quite complex while the view selection is quite simple.

4.2.4.1 View Selection

The interpolation of viewing parameters is done in a manner similar to the space simulation, but with a considerably simpler method of specifying viewing parameters. The view is specified simply by a field of view, center of interest point, distance from view point angular direction of observer and tilt of camera. These parameters may be adjusted via knob control. When a desired view was found they were stored in a keyframe table which was used for interpolation upon playback.

4.2.4.2 Physical Simulation

The DNA simulation program represents the molecular system as a collection of modules which are each rigid bodies and are connected together at rotatable joints. Each monomer is defined in terms of the location and orientation of its joints and as a list of the locations of its constituent atoms. Whenever a particular monomer is moved or rotated, a recursive connection tracer applies the same transformation to all other modules bonded to it. Each connected structure is assigned a velocity and tumbling speed which, upon each frame time simulation, are added to the current position and applied to the current orientation, respectively.

Since the animation of the molecular motion consisted of breaking and relinking bonds repeatedly, the absolute mode of simulation is not appropriate. Instead, the animation is driven by commands that 1) set the velocities 2) break and re—form bonds and 3) run the simulation forward by some number of time steps. The "script" for a sequence is, then, a list of such commands. First, commands to set some velocities, and break and reform bonds. Then a simulate command. Then more velocity/bond alterations. Then another simulate command, etc. The animation is then performed as the file is read. This makes moving directly to any given frame a bit difficult. If the desired frame is later than the current one, the file just continues to be read. If the desired frame is earlier than the current ones the program must be reinitialized from the beginning frame and run forward to the desired frame.

In fact, doing the entire film by this technique would be very difficult due to the complexity of the motion. Since much of the motion is actually initiated by the two enzymes in the film, two enzyme simulation routines were added to the program. Each of these routines consisted of a finite state machine which counts simulation time steps, performs some velocity alteration and/or bond relinking operation, resets the time step counter to a new value, and then changes to a new state. For example, the helicase simulation has two states, 1) prying apart and 2) moving down. At each time steps the first state rotates the two separate strands about their bonds by a small incremental angle and rotates the main double helix in the reverse direction by the same amount. The second state moves the helicase down one base pair. The polymerase simulation has four states involving 1) waiting for a nucleotide, 2) pushing the nucleotide in place, 3) retracting and moving to the next nucleotide position and 4) removing an erroneously matched nucleotide. The transition from state 3 to state 1 for example, generates a new incoming nucleotide by adding it to the database in such a position and at such a speed that it would fly into place just as state 1 expires and changes to state 2.

The randomly floating background nucleotides were driven by a simple program that generates their paths randomly, but in such a manner that they do not collide with the main DNA strand. The output of this program, when interspersed with enzyme directing commands then forms the main script of the movie.

When the simulator program is instructed to make a color frame, it writes the locations and orientations of the modules to a temporary file and invokes the module expansion program described in section 2.3.2.

5. RENDERING - HIGH LEVEL

High level rendering techniques concern themselves with building up images consisting of several disparate low level primitive types. The idea here is that the low level rendering programs do not need to know anything, or at least very little, about each other’s operation. The high level rendering technique causes them to be invoked in the proper manner.

5.1 Techniques

The main problem the high level rendering system must solve is the occlusion problem. There are several strategies which may be employed, of which the two most popular are Z buffers and Temporal priority (also called the Painter's algorithm).

5.1.1 Z Buffers

With Z buffers, the low level primitives must all reference a common Z buffer and will presumably have their Z values scaled into the same coordinate space. Otherwise they do not need to know about each other. Using this scheme, the high level scheduler can invoke the low level programs in virtually any order. This is convenient for complex texture mapping since all objects painted with a particular pattern can be rendered at once. Thus only one texture needs to be referenced at a time.

Z buffers also solve the problem of arbitrarily intersecting primitives easily.

Z buffers, however, have some severe problems for the space images considered here. For space scenes, the very large range of values in Z would require a Z buffer with very many bits of precision to have the resolution necessary to keep things properly hidden. In addition, anti-aliasing calculations are inconvenient with this scheme.

5.1.2 Temporal Priority

The basic idea behind temporal priority is to draw the objects in the scene in the order back to front. The later objects simply overlay the earlier objects. Temporal priority works well for space scenes because the different objects do not intersect (or at least are not supposed to).

5.2 Space Simulation Global Scheduler

The high level rendering scheduler for space scenes is a fairly specialized program which takes the frame state information file from the space simulator program and generates the command files to render the image on the frame buffer. It consists of four phases:

1) Global clipping — An enclosing sphere about each object (moon or spacecraft) is tested against the viewing volume. Those objects completely outside this volume are removed from further consideration.

2) Global Z sort — This then sorts the remaining probably visible objects in back to front order.

3) Generation of individual command files — A command file is generated for each visible object consisting of the transformation commands for placing it in viewing space, various surface lighting and texture map definition commands, etc. At this point some mutual shadowing calculations are performed. If a moon is in shadow behind the planet its brightness parameter, written out here, is decreased by 1/4.

4) Generation of global system command file — This is a system command file that runs the appropriate rendering programs in the z sorted order and tells them to read their commands from the appropriate file written during step 3. The appropriate rendering program to use for a particular body is taken from a table which is set up upon initialization of the high order renderer. Here also some shadowing calculations are performed. If a line from the sun to a moon intersects the planet on the sunlit side an extra command is added to the global file to run a program that places a black dot at the appropriate screen location.

Finally, control is given to the system to read commands, not from the file in step 4 but from another static file. This file in turn references the step 4 file and then does some clean—up operations, such as saving the image. The overhead for all this file manipulation may seem excessive but, in practice, it is negligible compared to the running time of the primitive renderers.

There are several useful features to this approach. First it allows for easy retry or debugging of an image since the commands and data files remain after the program terminates. Secondly it allows for some after—the—fact changes to the step 4 file by inserting suitable commands before or after its invocation by the global file. Two cases where this was used were during the title sequence of the Voyager Saturn movie. In these cases an explicit run of TEXFLY, for a record overlay, or the polygon renderer, for the title letters, was inserted just after the invocation of the step 4 file.

6. PRODUCTION

When the programs, databases and script definitions have all been prepared we are then ready to do production of the frames of the film. This essentially involves placing the simulator program in a loop generating frame information, passing it to the rendering program and saving the frames away somewhere.

6.1 Buffering Schemes

After a frame is generated it may be stored on various different media. The decision on where to save the frames depends on a set of trade-offs based on the resolution of the image (number of bytes to store), size of available mass memory (how much room you have to store it) and the need to recover from errors or make last minute changes to certain frames.

The available options are:

1) record directly on film as images are generated

2) record directly on video tape as images are generated

3) save images on disk/tape for later play-back to hard copy device

4) buffer intermediate data files for later image calculation

Note that any of these can be selected by editing the command file which controls the image saving. Option 4 is not so much a buffering scheme as a parallel processing scheme. It was used in the DNA sequence to accumulate Y sorted atom lists on tape for later farming out to several different computers.

To evaluate the other three options we first list some representative memory requirements and capacities

Video resolution 8 bit run length encoded frame	.10 to .25 Mbyte
Video resolution RGB run length encoded frame	.25 to .75 Mbytes
High resolution (2000x2000) RGB image	4.0 to 12. Mbytes

2400' Magnetic tape @ 1600 BPI	39.0 Mbytes
High density disk pack	256.1 Mbytes

From these figures we can form some conclusions:

Option 1 is dangerous since it provides the least opportunity to recover from errors. A single bad frame in a run that may last days can ruin the entire run. For the highest resolution images, however, it may be the only reasonable solution. It is hardly reasonable to require 8 tapes or one disk pack per second of running time. Later developments of higher density storage may alter this situation, however.

Option 2 is relatively safe since single frame video recorders have the capability to go back and insert replacement frames for bad ones. The disadvantage is the extra cost of the equipment and the fact that NTSC video is somewhat lower resolution than 16 mm film recorded in RGB.

Option 3 is quite safe since it allows digital editing of bad frames, i.e., a frame is recalculated and saved in a file which replaces the bad frame. It is somewhat slow, however due to the massive amounts of data transfer required. This happens to be the method currently used for our system at JPL. There are some convenient side effects to this mode.

Frames are stored one per file, in files named with some prefix string followed by the frame number. We can then generate the frames in non-sequential order and play them back in the proper order. One convenient usage of this technique is to generate widely spaced frames first to check the global appearance of the animation. By doing every 64 frames, then filling in the frames halfway between them, then filling in the frames halfway between them, etc. we will generate a sort of binary search for the movie. At any time we will have the entire movie done, just at different levels of jerkiness. On the next to last step, when all even numbered frames have been generated, it is possible to proceed to the odd numbered frames only for those scenes for which it is necessary. Thus saving time on slowly moving portions of the film.

Note that this frame/file naming convention allows the directory look up mechanism of the file system to do the frame sorting for us implicitly.

6.2 Checking and Filming

While frames are being generated a log file is maintained to keep track of which frames are finished and where they are stored. At various times during the production runs completed frames are played back on the frame buffer to verify that they are correct and erroneous frames are re—generated. Finally, the whole sequence is played back in sequence before a stop motion animation camera to create the finished film.