MARS: A TOOL-BASED MODELING, ANIMATION AND PARALLEL RENDERING SYSTEM

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Murat Aktıhanoğlu

December, 1992

MARS : A TOOL-BASED MODELING, ANIMATION AND PARALLEL RENDERING SYSTEM

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND
INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Murat Aktıhanoğlu
December, 1992

B02158

T
385
.A28
1992

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Bülent Özgüç (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Dr. Cevdet Aykanat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Dr. Fatih Ulupınar

Approved for the Institute of Engineering and Science:

[illegible]

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

MARS : A TOOL-BASED MODELING, ANIMATION AND PARALLEL RENDERING SYSTEM

Murat Aktıhanoğlu
M. S. in Computer Engineering and Information Science
Supervisor: Prof. Dr. Bülent Özgüç
December, 1992

Abstract: This thesis describes a system for modeling, animating, previewing and rendering articulated objects. The system has a modeler which models objects, consisting of joints and segments. The animator interactively positions the articulated object in its stick, control vertex or rectangular prism representation into the keyframes. interpolates inbetweens and previews the motion in real time. Then the data representing the motion and the models is sent to a multicomputer (*iPSC/2 Hypercube*[1]). The frames are rendered in parallel by distributed processing techniques, exploiting the coherence between successive frames, thus cutting down the rendering time significantly. The main aim of this research has been to make a detailed study on rendering of a sequence of 3D scenes. The results show that due to an inherent correlation between the 3D scenes, a much more efficient rendering than the conventional sequential one can be done.

Keywords: 3D Modeling, Computer Animation, Rendering, Parallel Processing, Distributed Rendering, Temporal Coherence.

[1]iPSC/2 is a trade mark of iNTEL Corporation

ÖZET

MARS : BİR MODELLEME, CANLANDIRMA VE PARALEL BOYAMA SİSTEMİ

Murat Aktıhanoğlu
Bilgisayar Mühendisliği ve Enformatik Bilimleri Bölümü Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Bülent Özgüç
Aralık, 1992

Bu araştırma, bir modelleme, canlandırma ve boyama sistemini tanımlamaktadır. Sistemin modelleme bölümünde, eklem ve parçalardan oluşan modeller yaratılmaktadır. Canlandırma sürecini oluşturan kişi daha sonra bu nesneleri anahtar çerçevelere yerleştirip, canlandırma sürecini oluşturmak için ara çerçevelerin ara değerlerini bulma işlemini başlatır. Boyama işlemi içinse model ve canlandırma bilgileri bir hiperküpe gönderilir. Tüm çerçeveler burada paralel bir şekilde 'dağıtımlı-işleme yöntemiyle ve çerçevelerin arasındaki benzerlikten faydalanılarak boyanır. Boyama işlemi bu şekilde önemli ölçüde kısaltılır. Bu araştırmanın ana amacı bir dizi çerçevenin boyanması üstüne ayrıntılı bir inceleme yapmaktır. Sonuçlar, bir canlandırma filminde varolan -çerçeveler arasındaki benzerlikten- yararlanarak geleneksel boyamadan daha etkili bir boyama yapılabileceğini göstermektedir.

Anahtar kelimeler : Modelleme, Canlandırma, Dağıtımlı İşleme, Zamansal Benzeşim, Anahtar Çerçeve, Hiperküp Topolojisi.

ACKNOWLEDGEMENTS

I wish to extend my thanks to my supervisors Prof. Dr. Bülent Özgüç and Assoc. Prof. Dr. Cevdet Aykanat, who have guided and encouraged me during the development of this thesis.

I am grateful to Prof. Dr. Yılmaz Tokat for his valuable guidance and encouragement through this thesis.

I express my gratitude to Asst. Prof. Dr. Fatih Ulupınar, who provided me with his valuable suggestions about my research.

My sincere thanks are due to my parents for their moral support.

Finally, I would also like to thank to all of my friends who helped and cooperated during thesis.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

During the last fifteen years, three-dimensional computer animation has become widely accepted as a powerful tool in a variety of applications, from entertainment industry (television, cinema, video) to education and business. Computer animation, although greatly assisted by the computer, is a very labour intensive job. Even with very sophisticated equipment and tools, animators work months for a high-quality computer generated animation. Mostly, the equipment (special architecture computers, video cards, single frame recorders) and the development tools (modeling, animation and rendering softwares) cost so much that only large, professional communities can afford them.

MARS (Modeling, Animation and Rendering System) is an ongoing study to provide a framework or environment for developing *high-quality* and *cost-effective* computer-generated animations. The animator is presented with an interactive, flexible, powerful and fast system.

There have been several goals while designing Mars. First, the system was not intended to be designed for a specific application. Usually, animation tools are designed and implemented such that they only perform some specific tasks. For example, there are animation tools that animate only human body models, or only some scientific phenomena. We have designed and implemented Mars such that an animator can create any imaginable character and animate it without limits. Mars is a multi-purpose animation generator.

Second, Mars tools were designed such that any of the tools can easily be replaced

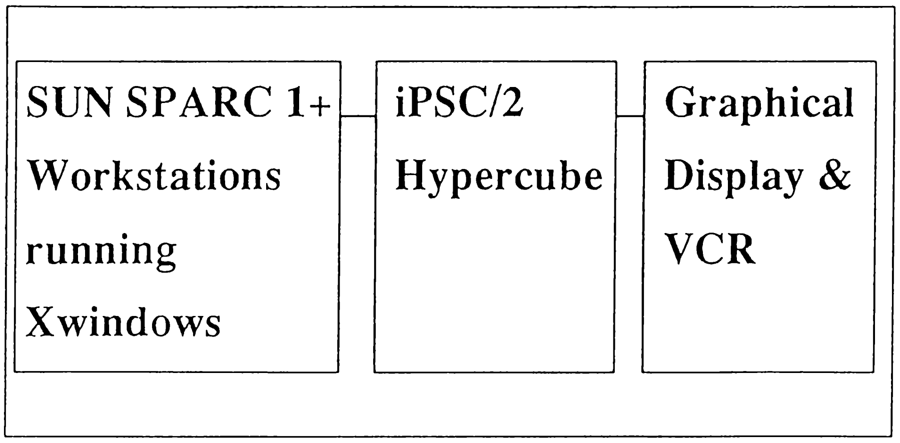


Figure 1.1. General view of the MARS system

or new tools can be added to the system, without disturbing the integrity of the environment. This property is very important since all techniques and algorithms in computer graphics are due to a rapid change, in an effort to create realistic looking pictures. Because of this rapid change, tools often get out of date. For example, an animation tool that has been written 10 years ago is no longer in use, because many things in rendering have changed, like ray tracing and radiosity. We have designed Mars such that any part of it can be easily replaced, in order to update and improve the system.

Third, Mars was designed such that it makes the most of the current resources in the development environment (Figure 1.1). Mars is poor man's high-quality graphics supercomputer. It employs each architecture in the development environment such that it exploits the most efficient parts of each architecture and as those efficient parts are put together, a virtual super graphics computer comes into the picture.

To generate a computer animation, characters of which structured models are defined in three dimensions, are needed. This structure should be well defined and flexible to allow the animator to create any imaginable character. Then, the characters should be placed and oriented appropriately for each frame, and then the characters and the background objects should be painted (rendered) with respect to the lighting conditions and the camera position.

The process of generating computer animation by Mars can be broken down into three phases (Figure 1.2) : modeling, animation and rendering. In the modeling phase, the *modeler* creates a three-dimensional model of the scene and its components.

In the animation phase, the *animator* describes how the model will change its place and orientation over time, thus generates the keyframes and subsequently, using a simpler model of the object, views the described motion in real time. Those two phases (modeling and animation) are done on Sun Sparc computers, running Xwindows, because the most important notion in these two phases is the user-interface. The modeling and animation tools should be highly user-interactive to involve the user further in the animation. If the user cannot easily control what he has created, the tool would become useless.

The most important tool of the system is the *renderer*. In the rendering phase, the renderer running on the iPSC/2 multicomputer currently with 32 processors, takes the model and animation data and for each frame in the sequence, generates a two-dimensional image using the specified shading algorithm and the camera position.

In Mars, an animator can use graphical primitives or B-spline surfaces. The models can have any number of joints, thus any general object can be modeled using the modeler. By scaling different segments of the model, many different versions of a model can be created.

To create a frame with models in desired positions and orientations, Mars modeler has a user friendly interface. The animator can switch between models by clicking on them, and orient the current model by selecting an axis, an angle, and a joint through a menu [12, 18].

The rendering process which is the most expensive phase of the three phases, is done by a multicomputer. The patches are rendered on 32 processors concurrently. One of the most important concepts in rendering is distribution of load among processors, i.e. load-balance.

Moreover, the rendering process is done even more efficiently by exploiting the temporal coherence that exists between the frames of the animation. Thus, rendering time is quite short, compared to the traditional straight ahead renderers that work on uniprocessor machines and do not use the principle of coherence.

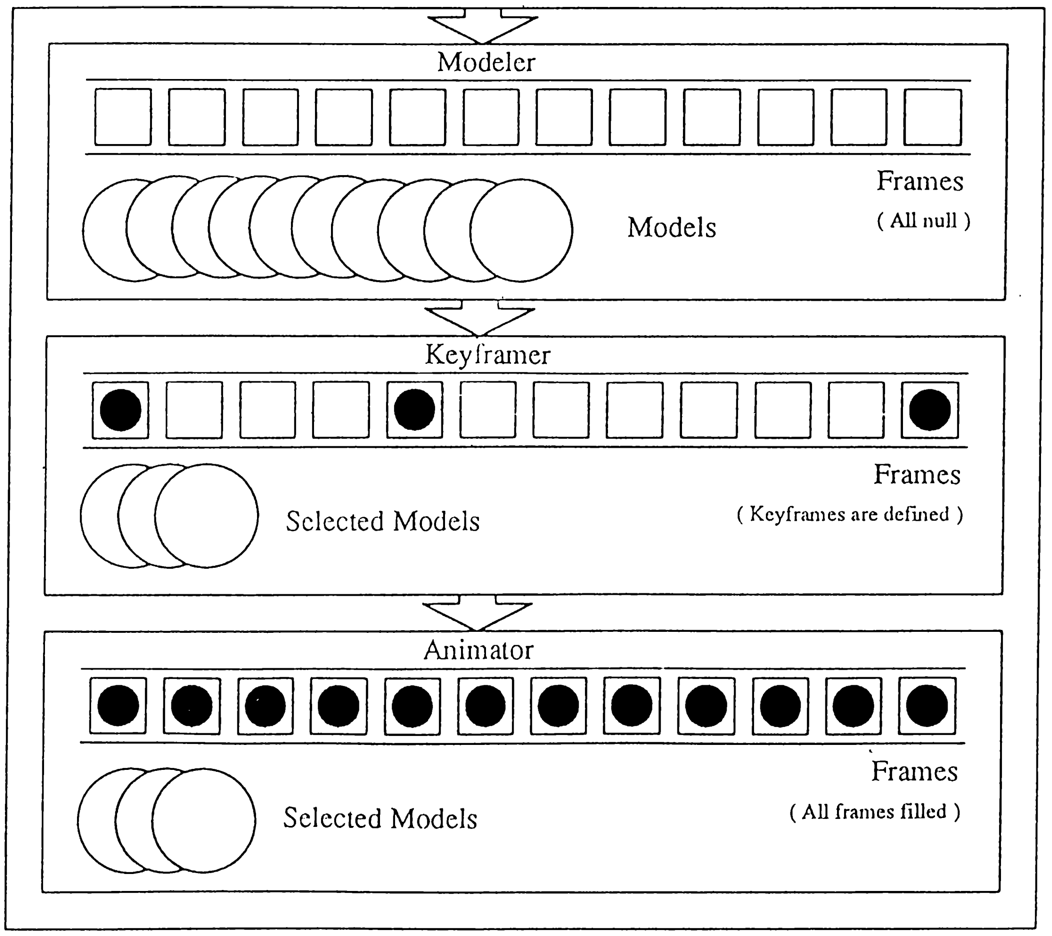


Figure 1.2. Process of Generating Computer Animation

Chapter 2

MARS MODELER

For a computer program to generate correct inbetweens, it has to be given the three-dimensional definition of the articulated objects that will be animated. Then the computer manipulates those given points with respect to a methodology, which is the principle of cascaded transformations. Each segment is defined with respect to its parent segment, and it is affected from all transformations that is applied to its parent segment. That is, if the parent segment is rotated 90 degrees, all its sibling segments are also rotated 90 degrees with respect to the parent segments joint position. The structure of the representation of a model is an important issue, because it is directly related to how easily a model can be manipulated.

The problem of presenting a three-dimensional definition to the computer has been well researched in the past. There have been many approaches and studies on modeling in three dimensions: [6, 9, 15, 16, 17, 27].

Most of the time, methods like imitating the real objects are used and according to the needs of the application, sometimes very simple and irrelevant models are employed. For example, to model a human body, simple spheres have been used, since the aim was to investigate the human behavior, as how they walk, sit, etc. Because of the complexity of the models that further resemble reality, such models have been limited to special applications where reality is of more importance than computation time.

The Mars Modeler treats a model as a composition of a library of predefined or ready graphical primitives. The model consists of joints and their base segments and

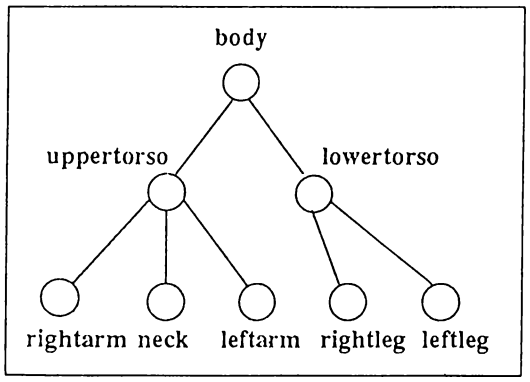


Figure 2.1. Structure of a Model

the modeler connects those base segments with each other as specified in the joint definitions. The connection of segments to each other is of great importance. That is, for a realistic animation the segments of a model should act meaningfully in any kind of orientation. If a segment looks unrealistic in some orientation, it means the joint structure of the model is not planned carefully enough. In fact, each model has a different nature of segment connection and the problem of modeling the joints in a realistic way is a topic of importance in computer animation.

The models of the Mars system consist of joints connected to each other and their base segments defining the shape of the model.

2.1 Joints

The joints form an n-ary tree structure (Figure 2.1), i.e. there is no limit on the number of child joints of a joint (so, we can define caterpillars!). Each joint has a parent joint and n children. Two vectors are used to define the X and Y - *axis* of the joint. The Z - *axis* is the vector product of X and Y - *axis* of that joint.

A joint can be rotated about each of its local X, Y and Z axes, thus has up to three degrees of freedom. This is actually the most ideal situation, but most joints cannot move in each of the three axes. For the sake of ease-of-realism from the animator's point of view, an upper and a lower limit are specified for the rotation of the joints about each axis. Consequently, a joint may be restricted to one or two degrees of freedom by permitting the joint to rotate about only one or two of the

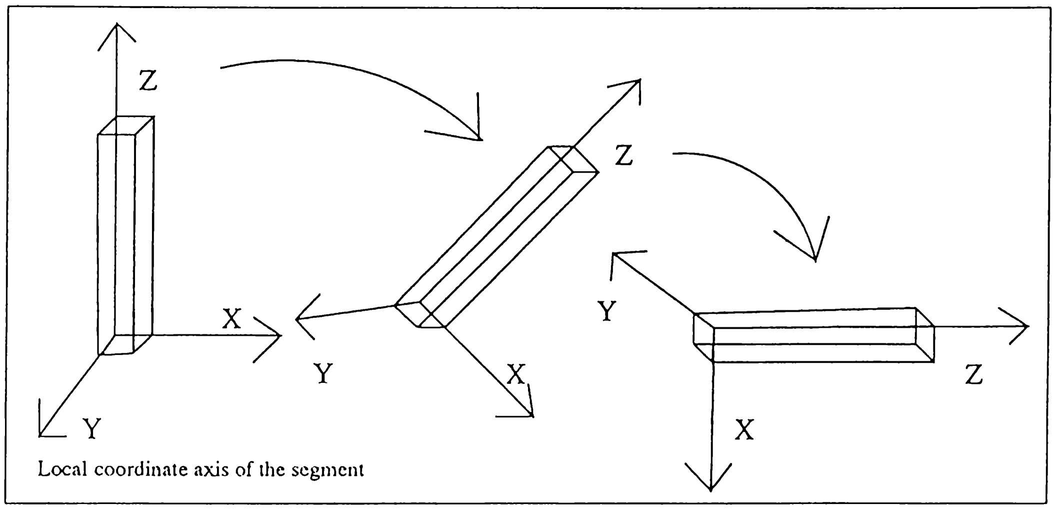


Figure 2.2. A joint rotates with its coordinate axes

axes. Using this method, simple joints, such as fingers (hinge joints), and complex joints such as shoulders (ball-and-socket joints) can be simulated. Also, each joint has its own coordinate axis, which is given in the definition of the model. Most of the time, the *Z* axis is along the direction of the segment as a convention, but this can be modified to suit the needs of the animator.

As a joint is rotated along its coordinate axes, the axes are also rotated, so it does not matter what the orientation of the joint with respect to the world coordinate axes is. The local coordinate axes are always aligned in the same way with respect to the segment (Figure 2.2).

2.2 Segments

Each joint has a base segment that is defined with respect to its local coordinate axis. Detailed, realistic looking models are always preferred and desired in computer animation but the animator does not want to mess with those detailed models while planning the motion. Detailed models are difficult to manipulate. It takes more time to orient a complex model than a simpler one. Mars has a multi representation of the models: one real and others simpler representations. To achieve this, segments of a Mars model are defined as *Bezier surfaces* [9, 21, 22, 26], but instead of directly giving a set of *Bezier control points* for each segment, the user first defines the

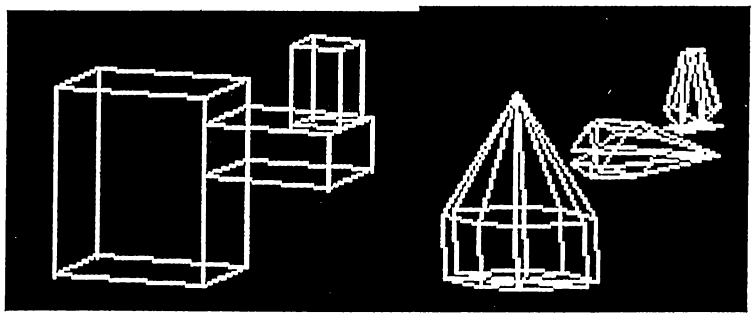


Figure 2.3. The *sbb*'s of a model and the actual model

segment bounding box (sbb) of the segment, which is simply the rectangular prism that bounds the segment itself, and then a set of *Bezier control points* which are in normalized form. The Bezier control points are defined in a unit cube so that all the points have coordinates with values between 0 and 1.

Eventually this normalized segment definition is scaled with respect to the defined *sbb*. If no *sbb* is defined, a unit cube is assumed (Figure 2.3). As each segment's *sbb* is defined, we use this simpler representation in the positioning and previewing phases. This speeds up the respective processes.

For each frame of a film, a transformation matrix is kept for each segment of each model (Figure 2.4).

This matrix is generated from the local and the world coordinate axes and the joint positions. It is updated at every frame should the segment change its place or orientation [25].

2.3 Modifying a Model

To create a model, we scale the normalized base segment definitions so that the segment fits into the segment's *sbb*. If one changes the dimensions of an *sbb* without changing its Bezier control points, a new model would be created. It then becomes straight forward to create many models from the same segment definitions. Changing the dimensions of the *sbb*s and the directions of the local coordinate axes of the model, the base segment is sheared, resulting in a number of different models created from

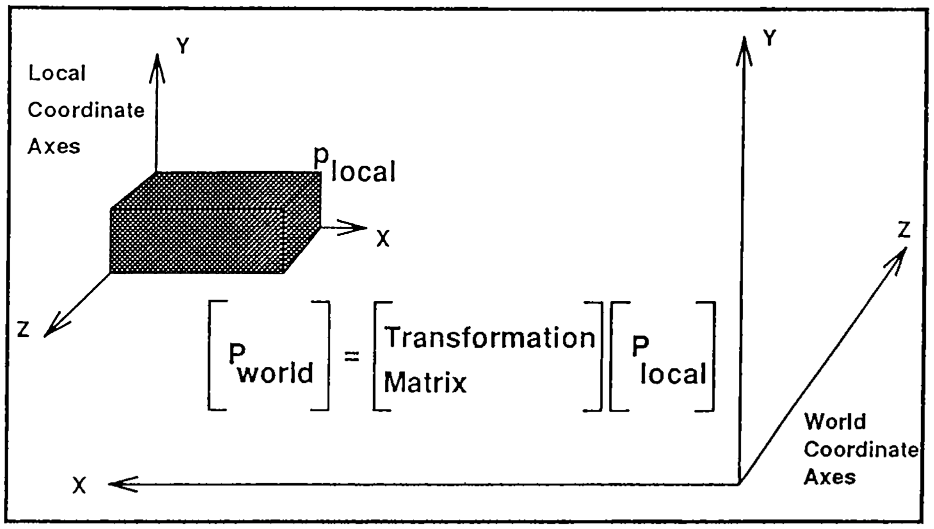


Figure 2.4. Transformation matrix kept for each segment

the same normalized segment definitions. This gives us flexibility in creating our models. Creating just one set of *Bezier control points*, representing a specific segment (e.g. human torso), would be enough to create many types of human bodies (Figure 2.5).

2.4 Animation-Preview Model Type

Mars uses a simpler representation of the model during the motion planning phase, so that extensive calculations need not be done to position and orient the models. The animator can interactively change the position and orientation of the model and see it instantly. Then, when the keyframes are prepared, he can view the whole motion in real time. He can frequently switch back and forth between the positioning and previewing phases without having to wait for long minutes for the computer to calculate the inbetween positions of the complex model.

The animator has three choices for the simpler model :

- stick model
- control vertex model
- rectangular prism model

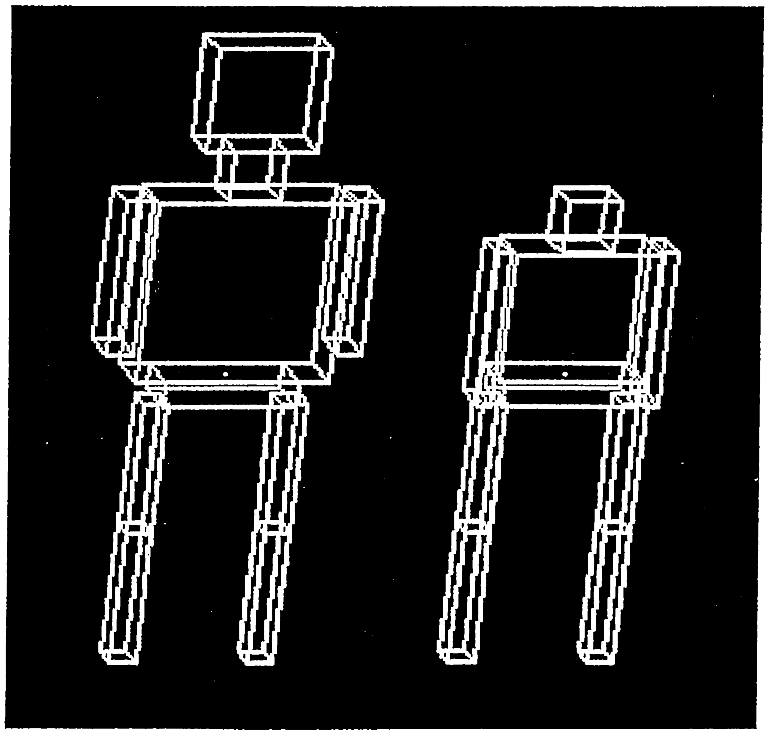


Figure 2.5. Two different models created from the same segments with different parameters

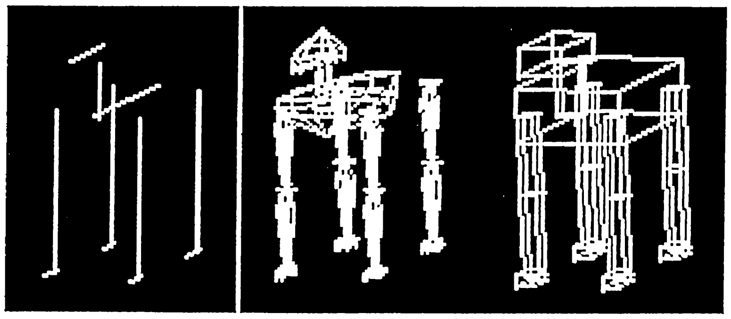
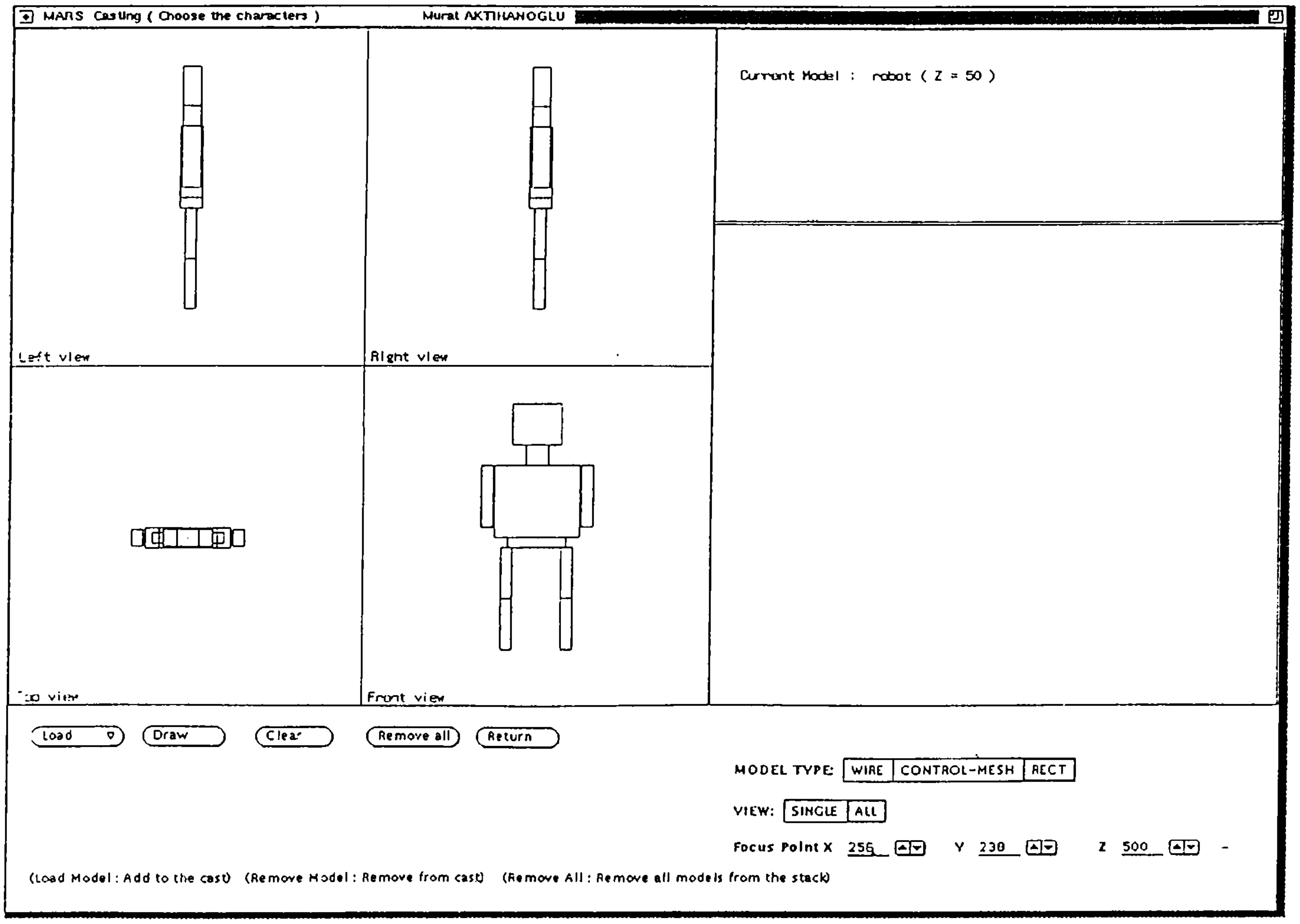


Figure 2.6. Stick, Control vertex, and Rectangular Prism

The stick model is simply a wire man, the control vertex model is composed of the interconnection of the control vertices of the model data and the rectangular prism model is a composition of rectangular prisms of the *sbbs* of each joint (Figure 2.6). While choosing the models through interactive menus, the animator can view the model from many directions. The model selection screen of Mars can be seen in Figure 2.7. The animator can view the model in any of the three representations. The representation of the model for the previewing and keyframing should be chosen according to the needs of the models and the animation scene. If real-time playback is achieved by using the control-vertex representation, the animator can clearly see the animation characters and the scene. If flickers occur during playback with the control vertex model, then a simpler representation should be chosen in order to view the animation clearly. Timing is very important in an animation because of the issues like anticipation, staging, slow in and out and etc. The success of an animation depends wholly on the timing arrangements of the animator, so the animator should have a strict control on timing.



Current Model : robot (Z = 50)

MODEL TYPE: WIRE CONTROL-MESH RECT

VIEW: SINGLE ALL

Focus Point X 256 Y 230 Z 500

(Load Model : Add to the cast) (Remove Model : Remove from cast) (Remove All : Remove all models from the stack)

Figure 2.7. Model screen of Mars

Chapter 3

ANIMATION

Animation is to give a series of pictures the feeling of motion by using the persistence of vision phenomenon of the human eye. The lower limit for the eye to perceive a series of pictures as continuous is 15 pictures/second. Below this rate, the eye can detect each picture separately. Above this limit, human eye perceive a sequence of still pictures as continuous and moving. Those series of pictures can also be thought of as samples of a real motion taken at regular intervals.

Traditional animators started to use computers first in the painting of drawings. To do this, all the pencil drawings were being scanned into the memory of the computer and then the lines were being enhanced and painted by very simple seed-fill algorithms. Usage of computers has speeded up the preparation of animations so much that, animators started thinking about employing those perfect partners more in the process of animation preparation as inbetweeners. Inbetweeners were those people who drew the inbetweens from the drawings of the chief-animators and painted them. There were so many faults most of the time, either some paint leaped over the line or the inbetween did not look realistic.

This led to the development of tools that produced the inbetweens from the drawings given by the chief-animator. This process is called 2D inbetweening. 2D inbetweening always caused deformations because the information given to the computer lacked one dimension (Figure 3.1).

The thought of giving the computer the 3D definition of the characters was the next step. This has been done by means of the algorithms that could produce the

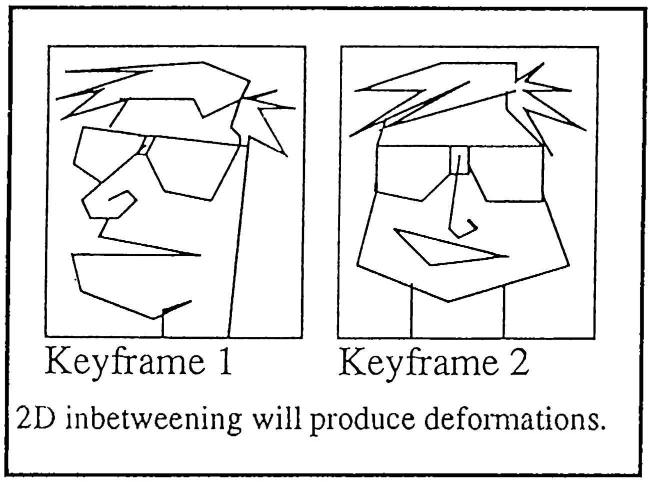


Figure 3.1. 2D Inbetweening

2D view of scene by applying some rendering techniques (hidden surface elimination and shading) to the 3D data. Also, by the use of homogeneous coordinate systems, fast matrix operations were introduced, to apply rotations and translations to the 3D data of the animation characters. Those developments led to the result that computers started producing meaningful inbetweens from the given 3D keyframes (keyframes are those frames prepared by the chief-animator).

However, there were further problems. There were many possibilities and techniques to make a character move on the computer screen, or to display a series of frames.

3.1 Display Techniques

The frames of an animation can be displayed on the screen of a computer in one of the three ways:

Read from disk and display

Frames of the animation can be rendered and stored on a storage medium, to be displayed later on. Another program reads those frames from the storage one by one and display them on the screen. Very fast-access disk-drives can reach the 25 frames per second speed, which is the default value for the PAL system (30 frames per second for NTSC system). Storing only the differences between frames may speed up this process.

Frame Buffer Animation

Frames, which are rendered and stored on a storage medium can be read into the memory and can be displayed from the memory, which can write to a screen faster than directly from storage medium. But it is obvious that only short sequences of frames can be shown this way, as RAM sizes are small compared to the size of a frame. Using very large random access memories, this technique can be employed. There are special hardware designed to do frame buffer animations like Abekas6000[1]

Realtime

The other way is to render the frames and show them instantaneously on the screen. To use this technique, one should use either very simple shading models and algorithms or very fast graphics-devoted machines. Because of the necessity to render each frame in less than a $1/25^{th}$ of a second, this is most of the time reserved for applications with very simple shading requirements. Flight simulators are a good example for real-time display animations. They use flat shading and very sophisticated graphics devoted computers.

3.2 Animation Techniques

As computer science and computer graphics techniques improved, many animation techniques other than keyframe animation has been put into use [1, 2, 3, 4, 7, 11, 14, 19, 23, 27]. These techniques are:

- Algorithmic Animation
  - Kinematic Algorithmic Animation
  - Dynamic Algorithmic Animation

[1]Abekas6000 is a trade mark of Abekas industries.

- Goal-directed Animation
  - Goal-directed animation by kinematic laws
  - Goal-directed animation by dynamic laws
- Procedural Animation
- Keyframe Animation
  - Image based interpolative animation
  - Semi goal-directed keyframe animation
  - Joint parameters interpolative animation

3.2.1 Algorithmic Animation

Most phenomena can be successfully animated using abstract motion specification methods, like keyframing, etc. When an animator wants to animate an elastic ball hitting a wall and bouncing back, he has to work very hard to make the whole sequence look realistic. Usually the timing, which is required for a realistic animation, is very hard to achieve manually.

Algorithmic animation is a method that is developed to achieve the realistic animation of physical phenomena of which laws are well defined. Motion specification is done algorithmically, in which physical laws are applied to the parameters of the animated characters. We can classify algorithmic animation system into two as applying kinematic physical laws and as applying dynamic physical laws to the characters. But sometimes the system can admit laws which is apparently specified by the animator, that is the physical laws of the animator.

Kinematic Algorithmic Animation

The algorithmic animation systems which use kinematic laws are called the kinematic algorithmic animation systems. In this type of motion specification strategy, the animator assigns an initial velocity and an initial acceleration to the animation character or any segment of it. The laws used are:

$$x = v * t = a * t^2$$

where x is the distance taken by the object, t is the time, v is the velocity and a is the acceleration of the object.

By using those laws, functions which specify the trajectories of the animation characters are found and employed in the animation.

The kinematic laws usually produce acceptable but not very realistic motions. They are used when computation time has to be short and dynamic laws cannot be employed.

Dynamic Algorithmic Animation

The algorithmic animation systems which use dynamic laws, in addition to the kinematic laws are called the dynamic algorithmic animation systems. In this type of motion specification strategy, the animator assigns an initial force (torque) to the animation characters or any segment of it. Each segment of each character has a specified mass and the rules used for animating the characters are as follows:

$$x = a * t^2$$
$$F = m * a$$

where x is the distance taken by the object, t is the time, F is the force applied on the object, m is the mass and a is the acceleration of the object.

This kind of motion specification is very expensive in terms of processing time and it is only used when strict realism is needed. The results are highly remarkable. The bouncing of an elastic ball can only be successfully animated through the use of such dynamic rules.

3.2.2 Goal-directed Animation

Sometimes the motion that the animator wants to create is a very specific motion that can be specified to the machine by simple English-like commands like walk, sit, nod, jump, run.

If the animation program is equipped with a strong knowledge-base, it can perform those actions by simple English-like commands. The motion planning and control is done entirely by the machine and the animator only specifies a command.

Usually, a simple command like *run* is composed of several different actions. What is done is that motion units are defined like *lift left leg* and those are composed into more complex motion units.

This motion specification method eases the job of the animator, but it also puts strict limits to what the animator can do with the models. If a specific motion is not defined in the knowledge-base, the animator cannot move the models in that way. The quality of a goal-directed animation system depends on the amount of information embedded in the system and how the motion units are implemented. There are two approaches in the implementation of the motion units:

Goal-directed Animation by kinematic laws

Goal-directed animation systems that use kinematic motion specification rules are used commonly. Although this usually does not give realistic and satisfactory results, its performance superiority over other methods lets the animator produce acceptable results. The basic tools of kinematic motion control are position, displacement, velocity and acceleration of the models.

Goal-directed Animation by dynamic laws

Goal-directed animation systems that use dynamic motion specification rules are employed in systems that require high realism. The output of dynamic motion control is highly realistic but it is as much expensive as it is realistic. There is a trade off between realism and computation times.

In dynamic control, energy, force and torque are employed in addition to position, displacement and velocity. The analysis of real physical variables produces very realistic results but the knowledge-base for such a system is huge and the processing time is very long. Real-time animation systems cannot employ such techniques.

3.2.3 Procedural Animation

An animation scripting language is used in the specification of motion of the characters. This approach is used in applications where the motion of the models can be procedurally defined.

Examples of such an scripting language is CINEMIRA [10] and ASAS [26] (Actor Script Animation System).

In this approach the animator writes an program to produce a sequence of animation. The program is either written in a high level language, and the animation produced through a graphical interface, or it is written in a specially designed animation scripting system.

The animator cannot see the result until the script is complete. and this is a major disadvantage. Another disadvantage of this approach is that the effects that will be created by the language cannot be intuitively guessed by the animator, that is the language is very abstract and cannot serve as a user-interactive system. For example the program in Figure 3.2 does not give a good feeling of what will happen in the film.

What the script does is that it spins two cubes on the screen but it is not intuitive. The use of procedural animation is restricted to specific areas, where the animators are experienced programmers and the motions required for the animation are procedurally definable.

3.2.4 Keyframe Animation

Keyframe animation is the oldest technique used to generate computer animations. It is adopted from traditional animation. The idea, as referred above, is the same with traditional animation: the animator gives the motion parameters for some specific frames, which are main breakpoints of the desired motion. Then the computer generates the motion parameters for other *inbetween* frames as it has the three dimensional definitions of the models. There are different techniques in keyframe animation with respect to motion specification or interpolation methodologies to obtain the inbetweens.

```
(script myprogram
  (local: (runtime 96)
          (midpoint(half(runtime)))
  (animate (cue (at 0)
           (start(spin-cube-actor green)))
           (cue (at midpoint)
           (start(spin-cube-actor blue)))
           (cue (at runtime)
                (cut))))
```

Figure 3.2. A sample script

Image based interpolative animation

Input to the computer is given as two dimensional pictures and the computer performs two dimensional inbetweening (Figure 3.1). This technique except for some special conditions produces results that are incorrect. Shear effect is observed in the inbetween frames and models seem distorted. User intervention is needed to achieve a sequence of frames without distortions.

Joint parameters interpolative animation

In this approach, the animator specifies the joint parameters of models for some specific frames, which are called keyframes. Then the computer interpolates those joint parameter values and generates the inbetween frames. This approach is very efficient and widely used in computer animation.

Semi-goal directed keyframe animation

This approach is the same with joint parameters interpolative animation except that the animator specifies the keyframes in a more user-interactive fashion. For each keyframe, the animator positions the models by using some simple, English-like commands. This technique is suggested by Özgüç and Mahmud [16] and the system developed by this method efficiently melts many approaches in one pot and gives flexibility to the animator in choosing a way to specify the keyframes.

The Mars Animator animates the 3D articulated rigid models using the parametric keyframe interpolation method among the many methods in literature.

3.3 Motion Specification

Motion control has always been a problem in computer animation, as the data to be manipulated is 3D and our tools (mice, digitizers, lightpens, cursors) are 2D. It is very hard for an animator to visualize a 3D object on a 2D screen, and it is even harder for him to manipulate that object by using 2D manipulators.

From all these points, we conclude that an *interactive* manipulation of our 3D animation characters is very hard to achieve, but we want to position and orient our characters any way. We have to give keyframes to the computer, so that it produces an animation.

First, it would be better to examine the criteria of control, that an animator would like to have over the 3D data of the animation characters. The animator should be able to apply

- translations to the characters
- rotations (about any of the 3 fundamental axes) to the characters
- rotations (about any of the 3 fundamental axes) to the segments of the characters

There are many possibilities for achieving those criteria. The computer may be loaded with a knowledge base (as with goal-directed approach) and the animation may be abstracted from the low-level details of the transformation, or the animator may be required to go into the depths of the motion specification and do the transformation at a very low level.

Indeed, there is a trade-off between high and low-level motion specification schemes. As the motion specification level gets higher, computer does more job, the animator's job gets easier, but the animator's control over the characters is reduced. As the motion specification level gets lower, the computer does less job, the animator works more, but his control over the characters increases.

In Mars, we chose a way in between: Joint parameters manipulation. The translations and rotations are performed as follows:

Translation :

To translate an object, a new coordinate:

$$P_{new} = (P_{x_{new}}, P_{y_{new}}, P_{z_{new}})$$

is needed. $P_{x_{new}}$ and $P_{y_{new}}$ can be obtained by the mouse, and we obtain $P_{z_{new}}$ by using a slider. As those parameters are chosen by mouse and slider, the object is moved to its new position.

Rotation :

To rotate a segment of an model or the model itself (which is the main segment of the model), a quadruple of the form:

$$R = (modelname, segmentname, axis, \Theta)$$

is needed. Modelname is the name of the model, segmentname is the name of the segment of the model, axis is either of the three fundamental axes x, y, z and Θ is the rotation angle about this specified axis.

Modelname is selected interactively from the screen. Θ is selected by using a slider. Axis is selected by an exclusive toggle menu. The rotation is applied to the model with the given modelname, axis and Θ as the segment is selected from a pop-up menu. The instance of choosing a segment from the segments pop-up menu can be seen in Figure 3.3. This way, many segments can be rotated with the same parameters consecutively.

Full control of the motion of the models is given to the animator. Each segment can rotate along any of the three axes and by this method, all joints can access any point in 3D space. The animator can create any alignment position of the models by selecting each joint one by one and rotating them.

3.4 Interpolation

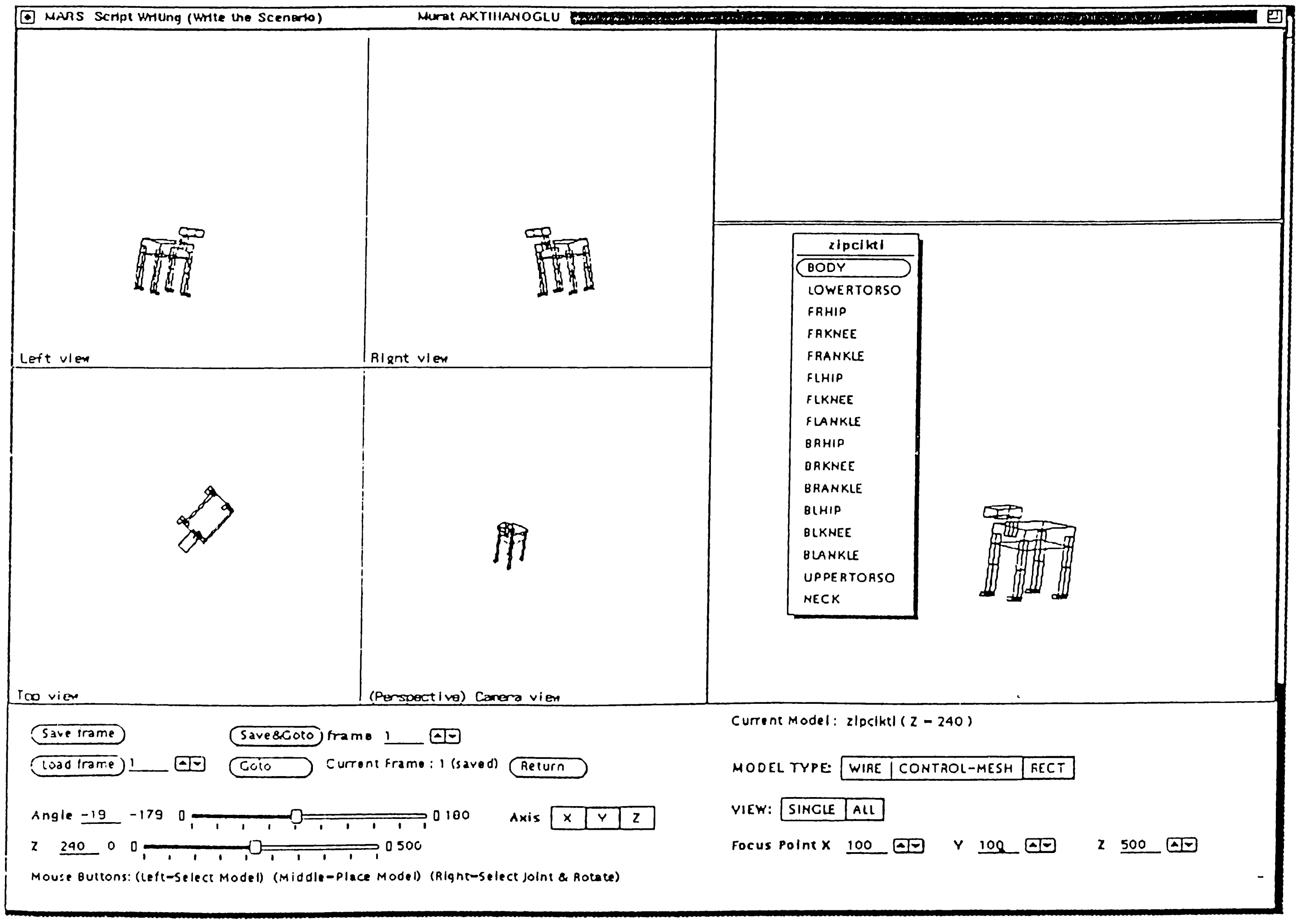
When all keyframes are ready, the computer interpolates the parameters of the joints of each model and creates the frames between the keyframes. This is done by an interpolation scheme that depends mainly on the matrix operations. This interpolation is performed as shown in Figure 3.4.

In fact, what is done is that, we find the matrix that transforms the first position to the final position. Then from this transformation matrix, a rotation axis and a rotation angle are found. After this angle is divided into the number of inbetween frames, a new transformation matrix is formed from these axes and the rotation angle step. The details of this process is as follows:

An animation tool should find the inbetween positions and orientations of a moving segment fast and correct. In our implementation of Mars, we have used a mathematically-based approach to find the transformation matrices for the inbetween positions.

In Mars, every segment has a transformation matrix that transforms the local defined segment points to the world coordinates for each keyframe. Each segment is defined in a local coordinate axis to ease the manipulation of the shape of the models. Also, a defined segment can be used in many orientations and sizes in many models.

Let the local defined points be P_{local} and their projections onto the screen with respect to the world coordinates be P_{world}. The transformation matrix that transforms P_{local} to P_{world} is $M_{local_{to}world}$. So



Current Model: zipciktl (Z = 240)

frame 1

Load frame 1

Current Frame : 1 (saved)

Angle -19 -179 0 180

Z 240 0 0 500

Focus Point X 100 Y 100 Z 500

Mouse Buttons: (Left-Select Model) (Middle-Place Model) (Right-Select Joint & Rotate)

Figure 3.3. Animation screen of Mars

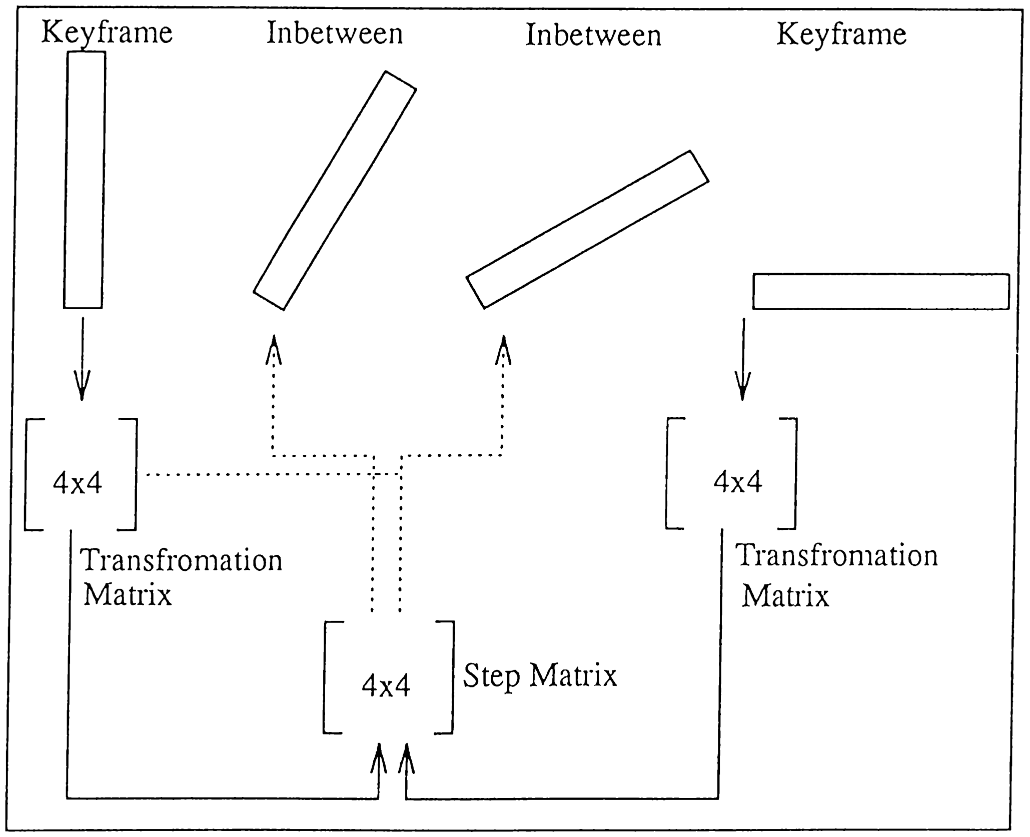


Figure 3.4. Matrix Interpolation Scheme

$$P_{world} = M_{local_{to}world} * P_{local}$$

for any frame. Note that P_{local} does not change throughout the film (if some morph is not employed in the film!).

We want to find the inbetween positions for two points with a known step size N. We have all the transformation matrices of all segments of all models for each keyframe. Let us consider a specific interval that is to be interpolated. Let the starting keyframe have the label s and the final keyframe f.

The starting position is given by $M_{s_{local}s_{world}}$ and the final position is given by $M_{f_{local}f_{world}}$. From those two matrices, we can find the transformation matrix that transforms starting position to the final one. It is given by

$$M_{s_{local}f_{local}} = M_{f_{world}f_{local}} * M_{s_{local}s_{world}}$$

We know $M_{s_{local}s_{world}}$, and as those transformation matrices are orthogonal,

$$M_{f_{world}f_{local}} = M^{-1}_{f_{local}f_{world}} = M^{T}_{f_{local}f_{world}}$$

Using this 4x4 homogeneous coordinates transformation matrix $M_{s_{local}f_{local}}$, we can find the axis and the angle of this transformation which can be seen in Figure 3.5.

The angle Θ of the transformation is given by:

$$\Theta = cos^{-1}((trace(M_{transformation}) - 1)/2)$$

This angle obviously can take infinitely many values but we chose the smallest positive value as a convention.

Then the arbitrary axis around which the rotation is done is found first by finding the 4x4 vector matrix K:

$$K = (M_{transformation} - M^{T}_{transformation})/2$$

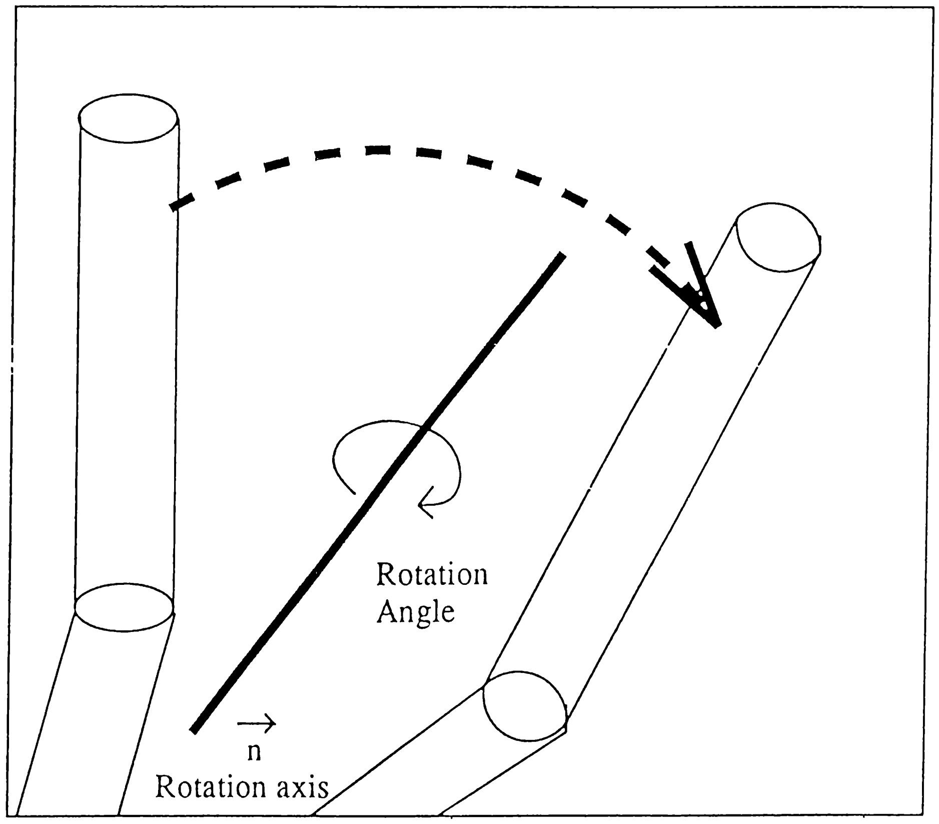


Figure 3.5. Axis and angle of transformation

Then the column matrix k is found from K and finally the column vector n (the axis of rotation) is found by:

$$n = k/sin(\Theta)$$

After finding the arbitrary axis and the rotation angle, we divide the rotation angle by frame number of first keyframe minus the frame number of the next keyframe, to find the step rotation angle Θ_{step}. The step transformation matrix $M_{transformation_{step}}$ is found by:

$$M_{transformation_{step}} = cos(\Theta_{step}) * I + (1 - cos(\Theta_{step})) * n * n^T + sin(\Theta_{step}) * N$$

All the transformation matrices in that interval are updated using this step transformation matrix.

3.5 Previewing

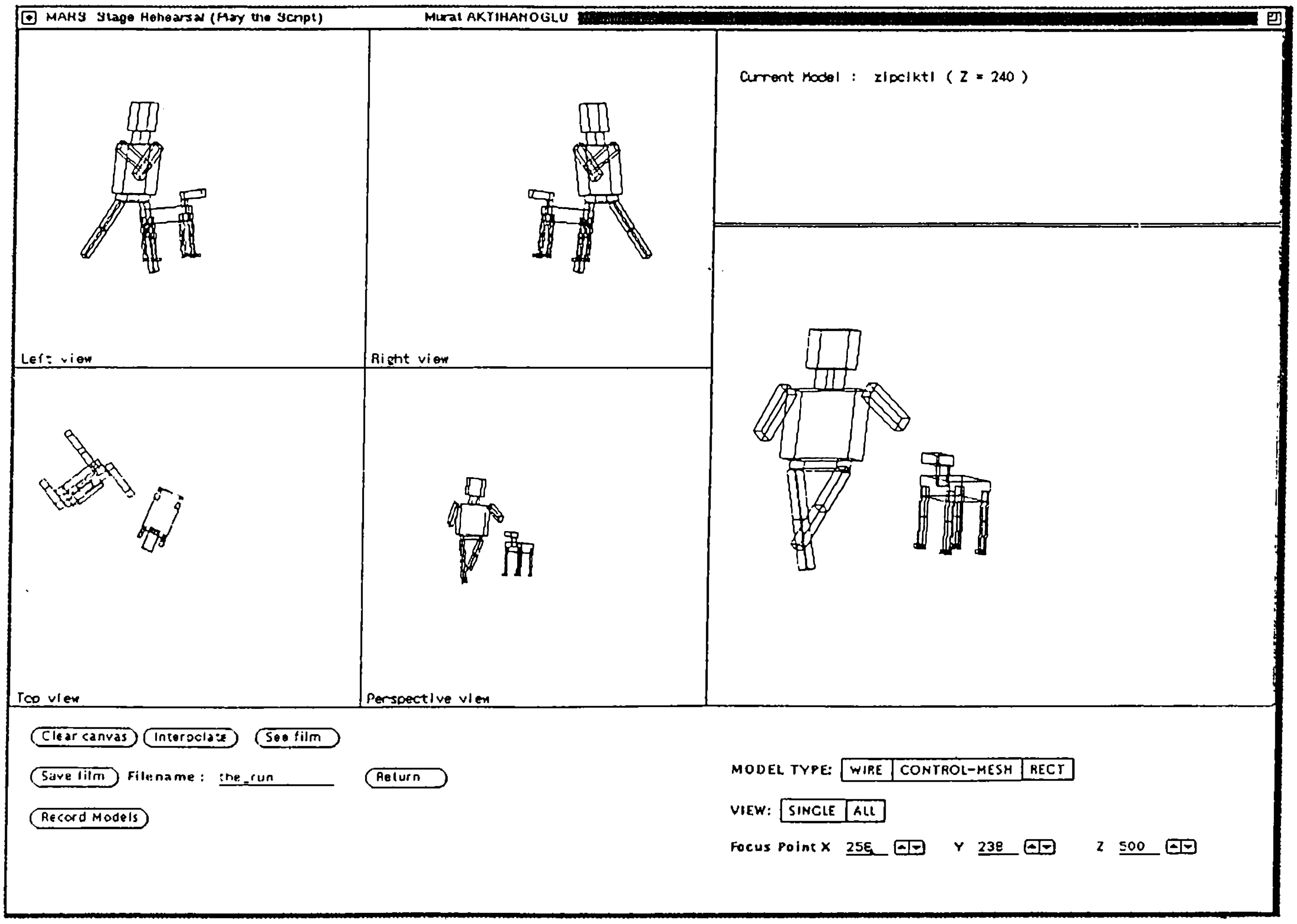
When the transformation matrices for all the segments of all models for all frames are found, the sequence of animated frames becomes ready to be viewed by the animator. The interpolation and previewing screen of Mars can be seen in Figure 3.6.

The player draws each frame on the screen consecutively one by one. For this process, it makes many matrix multiplications of the form:

$$P_{world} = M_{transformation} * P_{local}$$

for each point of the model. If these multiplications are done for a complex model with many points, the animation on the screen would be very slow and does not give the feeling of moving models.

Instead Mars uses one of the three skeleton views according to the needs of the animator. This way, real-time playback can be achieved. The modes of the preview model type has been explained in the Modeling chapter.



Current Model : zipcikti (Z = 240)

Filename : the_run

MODEL TYPE: WIRE | CONTROL-MESH | RECT

VIEW: SINGLE | ALL

Focus Point X 256 Y 238 Z 500

Figure 3.6. Interpolation and previewing screen of Mars

The benefit of choosing a *multi representation* becomes clear at this stage. The animator can view the created motion in real time, so editing is easier and faster. By means of this preview phase, the animator can see the created motion and make changes, before the long process of rendering begins.

Up to the previewing phase, the model is treated with its stick, control points or *sbb* representations. At this stage, the Mars animator creates the *Bezier surfaces* from the control points or directly reads the patch definitions of segments from file.

After editing, when the desired motion sequence is achieved, Mars sends the model's data and the motion data to the multicomputer for rendering of the scene.

3.6 Communication between the Animator and the Renderer

After all the previewing is done and the desired motion sequence is achieved, the scenes get ready to be rendered. This expensive process is done on a multicomputer, to cut down the total film-making process. This means that the data representing the models and the animation should be transmitted to the multicomputer in an appropriate form.

After previewing, we have the model data and the motion sequence data. The important point in this stage is the way this model and motion data is communicated between the Animator and the Renderer. There are a number of ways to.do this. The criterion of optimized communication is that this data should be well compressed and it should have no redundancy as well as containing all the necessary information about the models used and the specifications of the motion.

The format of this data is very significant. There is a trade off between the data size and the data interpretation time. If the animator sends the data to the renderer in a very compact form, it takes more time for the renderer to achieve the data. For example, the data for the inbetween frames which the animator has might be omitted from the communication packet because the renderer can find those values by itself if it is given the necessary database. This obviously increases the processing time of the renderer and since the animator already has this data, it would take less time for the renderer to read it from file than compute itself. So, we have to be careful

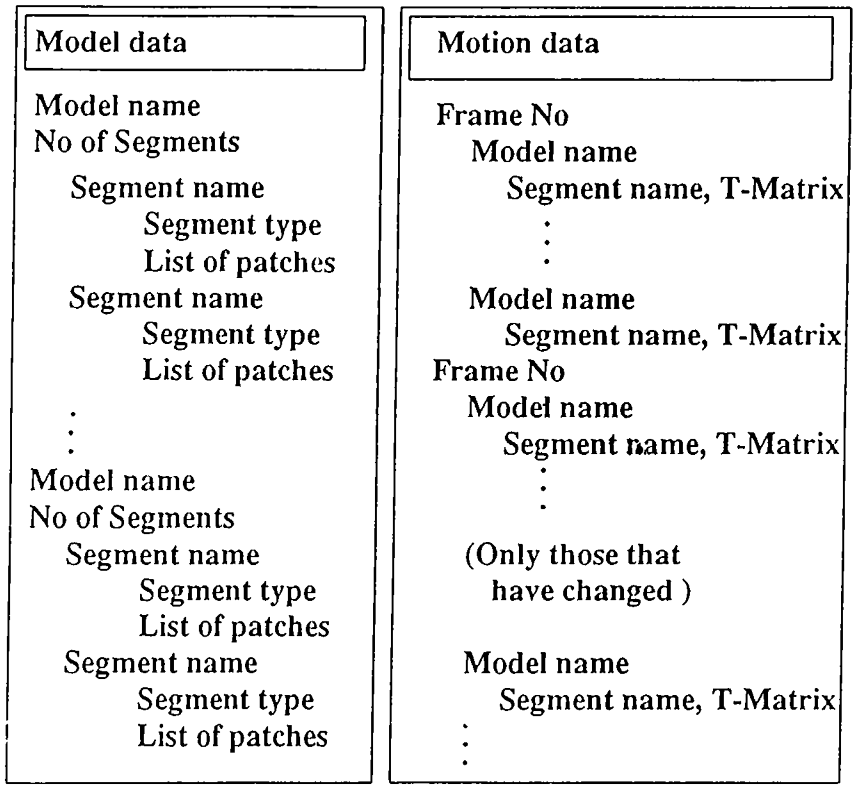


Figure 3.7. Data formats for communication

about what to insert into and what to omit from this data. The communication format of Mars is shown in Figure 3.7. The model data communication is straight forward. Each model has some segments, and each segment has its own definition. But for motion data, the transformation matrices for each segment of each model is communicated only for the first frame of the film. Then, a transformation matrix of a segment is transmitted to the multicomputer if the segment has changed its place or orientation since the previous frame. This provides a significant compression of data since only the necessary matrices are transmitted. This is also exploited in the processing of data, as will be seen in the next section.

Chapter 4

RENDERING

Rendering is the process of producing realistic images or pictures. Visual perception involves mainly physics and positioning of the surfaces and objects observed. In the rendering process of a three-dimensional scene that is composed of three-dimensional objects and surfaces, two issues are considered: how the surfaces reflect the incident light, that is the illumination model of the objects and which surfaces and objects are seen and which are hidden.

4.1 Illumination Model

When light energy falls on a surface, it can be absorbed, reflected or transmitted. It is the reflected or the transmitted light that makes a surface visible. What makes us see an object colored is that some wavelengths of the incident light may be absorbed more than other wavelengths. When a white light falls on a surface and red and green components of the light is absorbed by the surface, the surface is visually percepted as blue. To give this effect computationally in computer generated pictures, there are mainly two illumination models used by the computer scientists: Phong [5] and Cook-Torrance [20] illumination models.

Phong model is an illumination model that deals with only a few illumination parameters, but yet still gives acceptable results. Another model, the Cook and Torrance illumination model is a more realistic model. It deals with a lot of parameters like the *Fresnel term, attenuation factor, surface distribution function* and the *non-uniform reflectance hemisphere of the surface.* As the Cook and Torrance model is a more physically-based model than the Phong model, which is roughly an approximation of the former, it requires more computations than the Phong model. The more complex the illumination model is, the more expensive the computations but the more realistic our pictures are. More realistic pictures are always justified in computer graphics but after all, Phong shading model is used almost all the time instead of Cook and Torrance model which is very expensive. Phong model provides realism enough to avoid all those parameters. In our implementation of rendering a sequence of animated film frames, we have used Phong's algorithm.

4.2 Hidden Surface Removal

For rendering a scene, first the hidden surfaces should be removed, and the projection of the scene onto the two dimensional screen must be performed. This process also includes the rasterization on the projected surfaces.

A comparison of hidden surface removal algorithms may be found in [24]. In this survey, hidden surface removal algorithms are classified as operating on object-space or the image-space, and the degree of *coherence* they employ. Here coherence means the processing of geometrical units, such as areas or scan line segments, instead of single pixels.

There are currently two popular approaches to hidden surface removal: Z-buffer based systems and scan line based systems. Other approaches like area subdivision or depth-list schemes are not extensively used and they are only reserved for special-purpose applications like flight simulators.

The Z-buffer algorithm developed by Catmull [8], combined with the Phong reflection model represents the most popular rendering scheme. This algorithm, using Sutherland's classification scheme, works on image-space or screen-space.

Pixels in the interior of a polygon are shaded by an incremental shading technique and their depths are evaluated by interpolation from the z values of the polygon vertices. For each pixel the nearest visible point is buffered and compared to the next coming point, which is projected onto the same pixel (Figure 4.1).

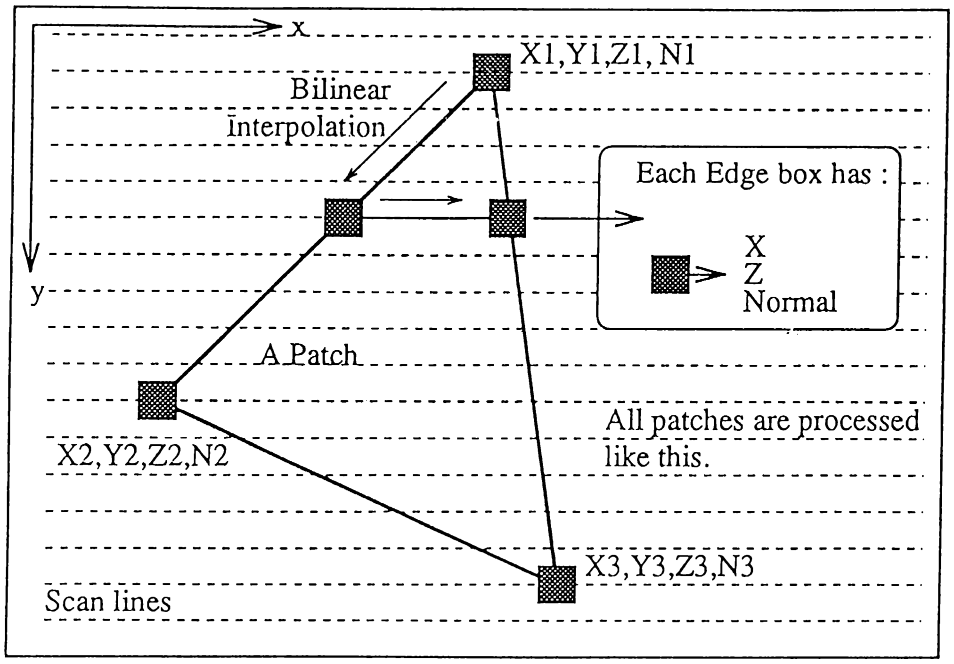


Figure 4.1. Formation of the Edge Boxes in the Z-buffer Algorithm

There is a variation of the Z-buffer algorithm for use with scan line based systems, which is called *scan line Z-buffer*. The rendering method, Mars uses is *scan line Z-buffer hidden surface removal algorithm* [26]. This algorithm consists of two phases. In the first phase, the algorithm goes through all the polygons in the scene to find and store the intersection points of each polygon with the scanlines of the image. Hence, the first phase effectively constructs a one-dimensional array of pointers *scanlines* where *scanlines(i)* points to the linked list that contains the edgeboxes on the i^{th} scanline (Figure 4.8). In the second phase, scanlines are processed one after another. In each scanline. the segments indicated by the edgeboxes in the corresponding linked list are rendered. All the pixels between two intersection points are shaded with Phong shading model and with an incremental shading technique. There are two approaches to calculate the intensities of the pixels, that lay between the two edgebox pixels. The first one, which is called Gouraud interpolation, is to calculate the intensities at the edges and then linearly interpolating those intensities for the inbetween pixels. The second approach, which is called Phong interpolation, is to interpolate the normals linearly for each inbetween pixel and calculate the intensity value afterwards. The second approach is apparently more expensive than the first one, but generally, more expensive methods generate realistic looking pictures. Phong interpolation generates highlights that look more realistic, while Gouraud interpolation results are narrower. Mars uses Phong interpolation, because realistic looking pictures form realistic animations after all.

We have preferred this algorithm mainly due to its very special nature that perfectly suits our tools of optimizing the rendering process. It first runs on the object-space and then the image space. Moreover it requires much less memory than conventional Z-buffer algorithms that holds all the screen space for the rendering. Scan line Z-buffer hidden surface removal algorithm is easy to implement, but as each pixel of each patch is visited, it is computationally expensive. The speed of this algorithm is the bottleneck of all the film-making process.

Of the three phases, rendering has attracted the most attention and research. More efficient techniques were needed to be developed to make the images look more realistic and to finish the overall process in a shorter amount of time.

If we think of rendering a picture as reducing a 3D scene to a 2D image, then the rendering of an animated film, i.e. a sequence of frames, is reducing a 4D scene (including time as the fourth dimension) to a 3D image (a series of frames, including time as the third dimension). Thus, rendering an animated sequence of frames must be thought differently than the rendering of a static scene. Hence, rendering a scene and a film are considerably different processes.

If we do the rendering of a sequence of animated frames separately, i.e. render each frame as totally irrelevant to each other, the result would be acceptable, but there are surely better ways to do this, as long as the sequence of frames has a very important characteristic that must be thought of. In terms of efficiency of processing, what makes a sequence of animated film frames different from a sequence of totally irrelevant frames is the concept of *temporal coherence* [26].

4.3 Temporal Coherence

Any object or joint in an animated film has a great degree of coherence between successive frames. That is to say, in consecutive frames, an object or a joint makes a relative translation or a rotation to its previous position and orientation [13].

Rendering each frame separately is of no sense. The optimal rendering algorithm

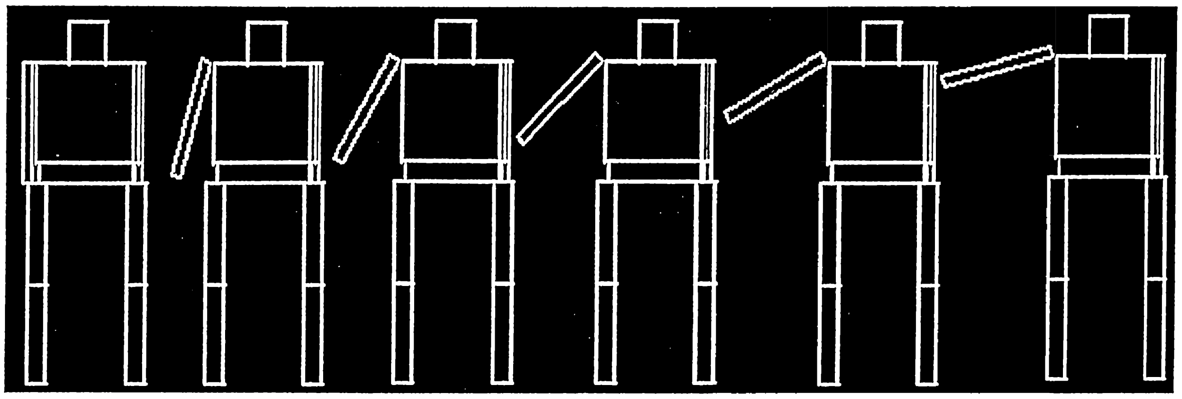


Figure 4.2. High degree of coherence in a film

should fully exploit the temporal coherence between successive frames in order to reduce the rendering job. It should avoid rendering the parts of the picture that do not change after the previous frame. Such an algorithm should have a *buffering mechanism* that buffers the parts of the picture that do not change and parts of the picture that will change in the next frame. After rendering a frame totally, creating the next frame can be done by simply rendering only those parts of data that have changed their place and orientation since the previous frame. The basis of such an algorithm is the coherence between successive frames of an animated film (for an example see Figure 4.2). Temporal coherence is one phenomena exploited fully to render animated film sequences more efficiently. As long as efficiency is our main academic goal, we must think also of optimizing our conventional sequential rendering algorithm. As will be seen in the *Algorithm* section, it is optimized to its best in terms of sequential processing but there is still something more we can do : *parallel processing.*

4.4 Parallel Processing

In literature, there are several works done on parallel rendering of a scene [28]. Most of the studies have a great dependency on the nature of the parallel architecture employed. The architecture we have implemented our algorithm on is Intel's iPSC/2 hypercube. iPSC/2 is a distributed memory and message-passing multicomputer. The iPSC/2 we are currently using has 32 processors. We have tried to improve an algorithm that would give good results on other parallel machines as well.

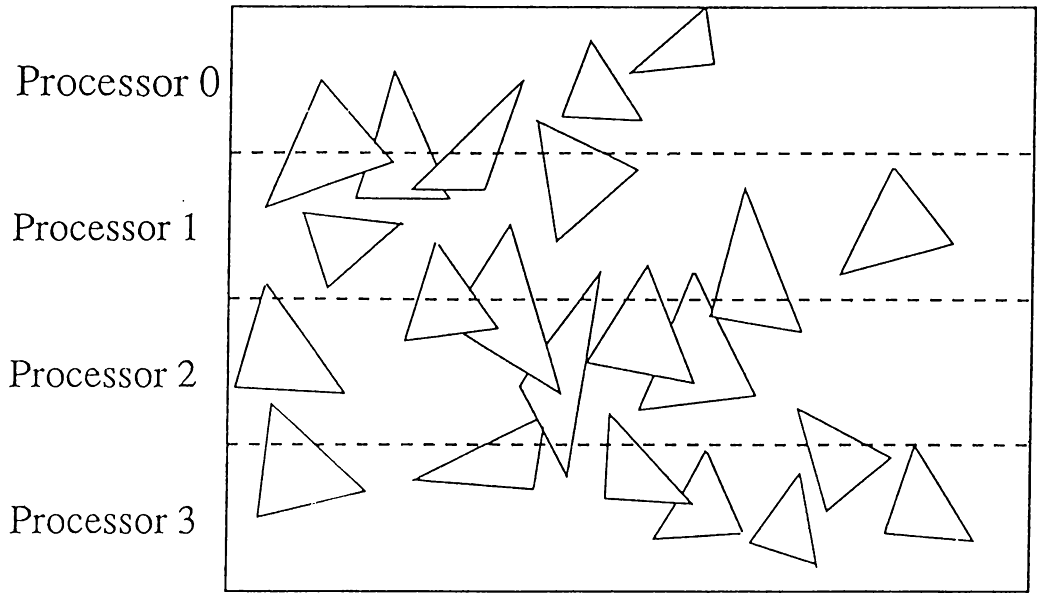


Figure 4.3. Screen space subdivision

4.5 The Algorithm

The algorithm to render the sequence of animated frames mainly depends on the temporal coherence and parallelism concepts and it is based on a modification of the conventional scan line Z-buffer hidden surface removal algorithm.

If we modify the Z-buffer algorithm such that, first all the patches are processed to form the edgeboxes, and then this heap of edgeboxes is processed to generate the intensity values, it becomes an algorithm that runs first in object space and then in the image (screen) space. This is the most important part of the parallel algorithm. The subdivision problem will be solved with the addressed concept.

Load balance is one of the main goals in parallel algorithm design process, but to achieve load balance in the rendering process of a scene, composed of 3D objects, the distribution of the objects that consist of patches to the processors is a critical job. Most of the time some assumptions and approximations are made.

There are mainly two approaches to the load distribution problem. One of them is *screen (image) space subdivision* and the other is *object space subdivision.*

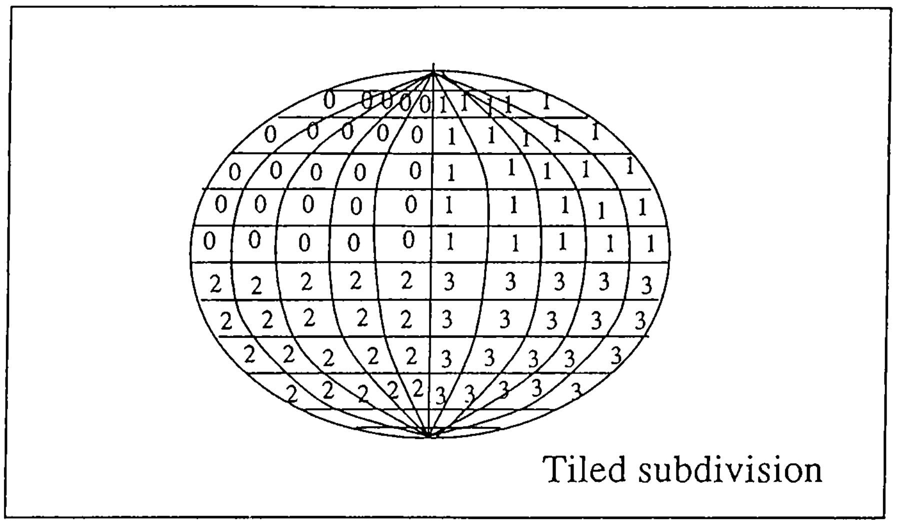
Screen space subdivision : In this distribution scheme, objects are distributed to the processors with respect to their locations in the projected view of the scene onto the screen (Figure 4.3). That is, a slice of the image is devoted to a single processor. This works fine with the second phase of the algorithm where edge-boxes are processed but to divide the screen such that equal loads of patches are distributed to the processors is very difficult. Usually, some preprocessing is done to make sure that equal number of patches exist in each of the slices, but this is also a difficult job, and still does not give satisfactory results, since shared patches may exist. Shared patches problem slows down the distribution phase.

Algorithms with image space subdivision are inefficient in achieving the load balance, but they are efficient in the second phase of the process where edge boxes are rendered and in the last part of the process where the contributions from each processor is merged to construct the final image. This merging is simply a concatenation.

Object space subdivision In this scheme, objects are treated as a list of patches, and they are mapped to the processors by using either tiled or scattered decomposition (Figure 4.4).

That is, to achieve the load balance, processors are given equal number of patches in any of the two division schemes. In the tiled division, *N* patches are divided to the *M* processors such that processor *i* receives patches from *i*(N/M)* to *(i+1)*(N/M) - 1*. The list is divided in a tiled way. In the scattered division, processor *i* gets every *(N/M)*th patch starting from the *i*th patch. Scattered division is more likely to achieve better load balance, but this heavily depends on the nature of the data.

Algorithms with object space subdivision work with a perfect load balance in the first phase, where edge-boxes are formed, but in the second phase and the last reconstruction phase, they have some deficiencies. The deficiency is that merging of the contributions from each processor to construct the final image is not straight forward. The objects devoted to different processors may occupy the same area, and a final Z-buffer checking has to be done in the merging phase. That is, sequential overheads are introduced in the last phase. Some algorithms try to overcome this overhead by also distributing the merging process, but still a big chunk of CPU time is wasted in this process.



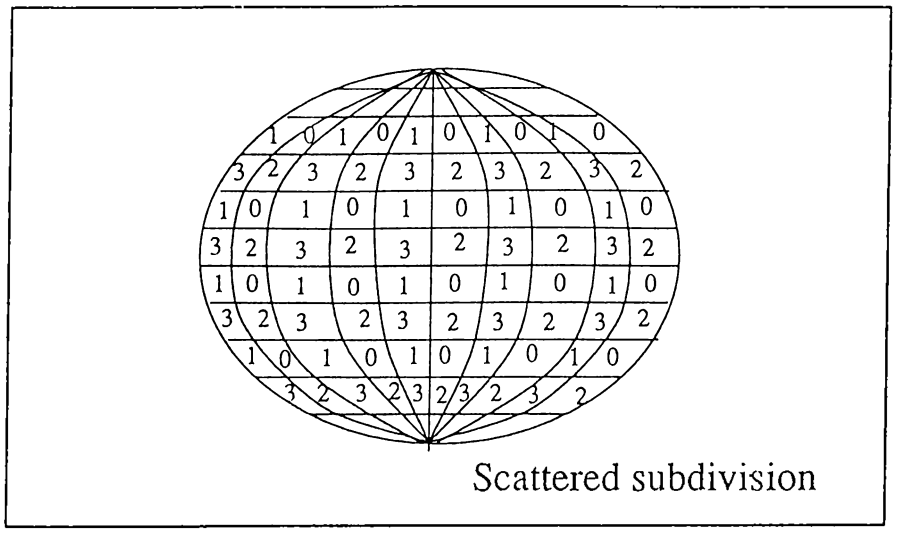


Figure 4.4. Object space subdivision (4 processors)

4.6 Model Data Distribution

After observing that algorithms with object space subdivision work more efficiently in the first part, and algorithms with image space subdivision work more efficiently in the second part of the rendering process, the problem would become to find an algorithm which works in the object space in the first part and in the image space in the second part.

After setting our goals, we have seen that to modify the conventional Z-buffer algorithm such that, at first all patches are processed to form the edge-boxes and then all the edge-boxes are processed to give the pixels, appropriate illumination values would work fine.

The data distribution scheme employed is such that the model data is uniformly divided to the processors in a scattered manner for the first part. Then, the edge-box data is divided to the processors with respect to their locations in the image space. Thus, in the first part object space subdivision, and in the second part image space subdivision is applied.

4.7 Film Data Distribution

As addressed above, our problem to be parallelized is a 4D problem. To achieve the data that will be used in the rendering process, model data and film data are needed. The model data which consists of 3D points in the space is multiplied with the transformation matrices of the film data for each frame.

We also have to distribute the film data to the processors. First, it is better to have a look at the nature of this data (Figure 4.5).

Our primary goal is to minimize the processor idle time by achieving load-balanced computations. The rendering of the keyframes is performed in parallel and this will be described in the next section. For the sake of simplicity, we assume that the number of inbetween frames between two keyframes are always multiples of the number of processors P. Hence, each processor can easily be assigned the rendering equal number of inbetween frames in scattered fashion. The scattered mapping of inbetween frames enables the realtime animation process.

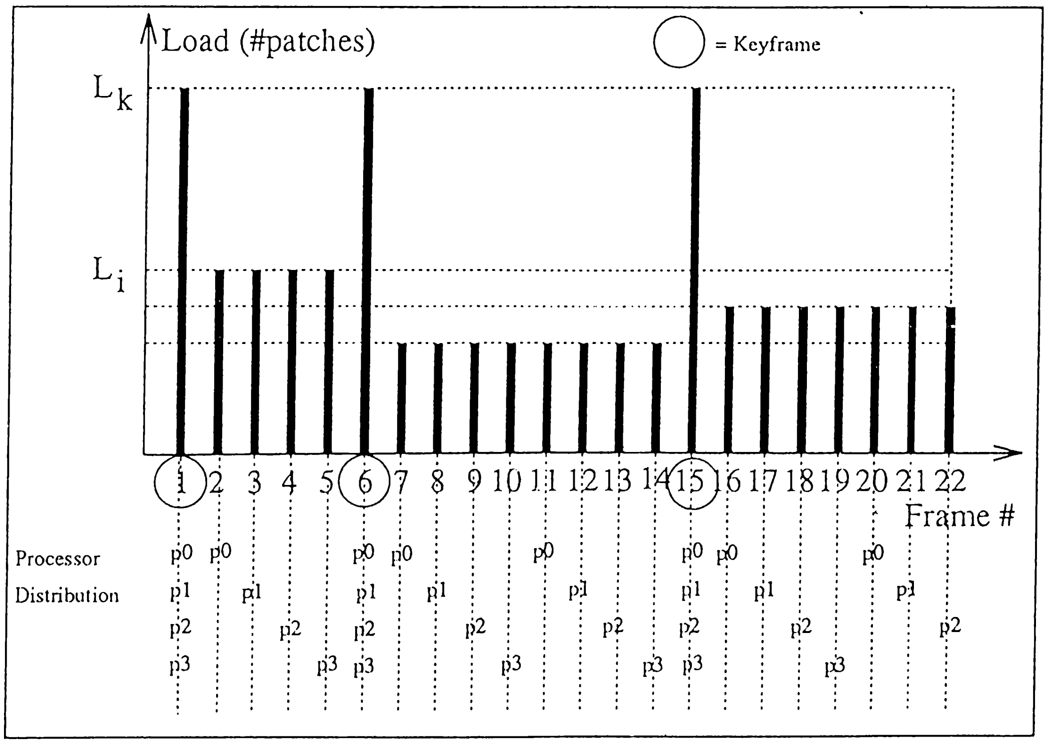


Figure 4.5. Load Distribution of the Frames (4 processors)

The dedication of inbetween frames to individual processors is because of the unpredictable computational load involved in the rendering of inbetween frames. If the granularity of rendering an inbetween frame is too small, parallel processing of that frame by P processors would even get longer than simple sequential rendering. Moreover, this way data is processed as if it is compressed between the keyframes. At the keyframes, we cannot compress the data because of the nature of the temporal coherence exploition mechanism that will be explained later.

4.8 Processing of a Keyframe

In the rendering phase, a keyframe is rendered by all processors concurrently because maximum load occurs in a keyframe. In a keyframe, all patches are to be re-rendered because stationary parts and moving parts may change completely or partially.

What is done in keyframe rendering is that, a constant frame buffer (CFB), a moving frame buffer (MFB), a constant Z-buffer (CZB) and a moving Z-buffer (MZB) are created (Figure 4.6).

The CFB keeps the image of the stationary parts that do not change in an *interval* (note that we use *interval* to refer to the frames between two consecutive keyframes). The CFB is constant throughout the interval and it is updated at every keyframe.

The MFB keeps the image of the moving parts that are actually moving in an interval and it is updated at every frame. The load of generating MFB depends on the number of joints or models that move in that interval.

The CZB keeps the *z-values* of the constant parts. Those values are used throughout the interval to determine visibility of each pixel. It is updated at every keyframe like the MFB.

The MZB keeps the *z-values* of the moving parts. This buffer is updated at every frame. Each calculated *z-value* is compared with both the CZB and the MZB to determine if the moving part is visible or not.

Recall that image space decomposition scheme is to be utilized for mapping rendering computations to the processors. Also recall that a scanline based Z-buffer

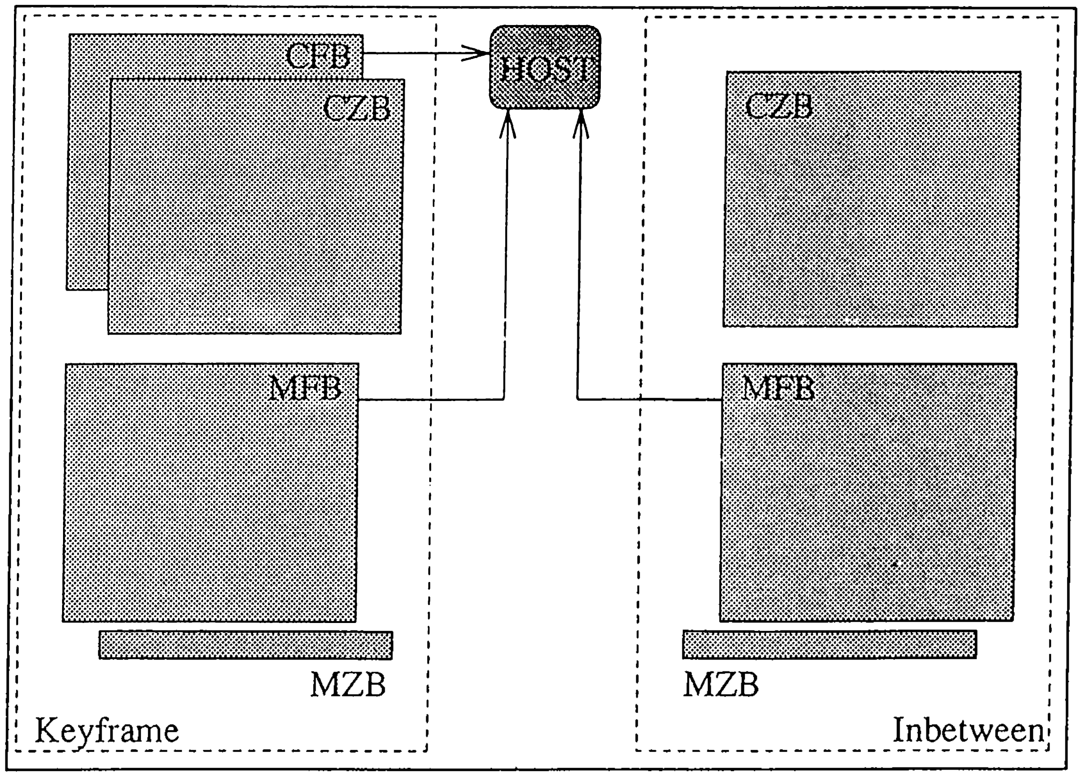


Figure 4.6. Buffers in a Frame

algorithm is selected for rendering. Hence the image space should be divided by scanlines. That is, a scanline is chosen as an atomic process to be performed sequentially by an individual processor. Otherwise, further division of individual scanlines may necessitate unacceptable computational overhead in order to maintain the spatial coherence. It is apparent that the amount of computations to be performed for each scanline is proportional to the number of edgeboxes on that scanline. Hence, a load-balanced mapping problem for the parallel rendering of keyframes can be modeled as follows:

Input Instance : Given n scanlines $(s_1, s_2, ..., s_n)$ with the corresponding computational weights $(w_1, w_2, ..., w_n)$. Here, w_i is an integer which denotes the number of edgeboxes on the scanline s_i and W is the sum of all w_i's.

Problem : Assignment of these n scanlines to P processors such that the sum of the weights of the scanlines mapped to each processor is close to optimal load $W_{average} = W/P$ as much as possible.

This problem is in fact the number partitioning problem which is *NP-hard.* Here, we propose a simple yet effective heuristic for the solution of this load-balanced mapping problem.

The steps for the rendering of the parallel rendering of the keyframes is given below :

Step 1: As discussed earlier, each processor is assigned almost equal number (either $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$) of patches using the scattered mapping scheme, where N denotes the total number of patches in the overall scene. Each processor performs the first phase of the scanline based Z-buffer algorithm for its local patches. That is, each processor constructs the edgelist for its local patches. At the same time, each processor also forms a local edgebox counter (EBC). EBC in each processor is a one-dimensional array such that EBC(i) holds the number of local edges in the i^{th} scanline.

Step 2: Each processor performs a prefix sum on its local EBC array so that EBC(i) holds the total number of local patches on the first i scanlines. Then, a duplicated global vector sum operation is performed on the local EBC vectors. At the end of this global operation, each processor holds a local copy of the EBC array where EBC(i) contains the total number of global patches on the first i scanlines.

```
Step = End = EBC(NO_LINES)/allnodes
i = 0
for node=0 to (allnodes-2) {

    while EBC(i)< End
      i = i+1

    if ( EBC(i)-End > End-EBC(i-1) )
      end(node)= i-1
    else
      end(node)= i

    End = End + Step
}

end(allnodes-1) = NO_LINES
```

Figure 4.7. Pseudo code of heuristic based mapping scheme

Step 3: Each processor runs the mapping-heuristic whose pseudo code is given in Figure 4.7.

Note that the proposed heuristic achieves the tiled decomposition of the scanlines. Hence, each processor determines the mapping information for all scanlines in the image. Then, each processor sends the edgeboxes of the non-local scanlines to their home processors according to the mapping information. Thus, each processor merges (by simple pointer operations) the received edgelist with its local edgelist (Figure 4.9).

Step 4: Each processor performs the rendering of its local scanlines in parallel.

First of all, each processor processes the stationary patches, and writes the resulting *z-values* to the CZB and the resulting *pixel values* to the CFB. This process is straight forward as it is the same as sequential processing.

Then, the moving parts are rendered. During this process, each edge-box is processed and the resulting *z-values* are compared to both the CZB and the MZB, and written to the MZB and generated *pixel values* are written to the MFB. The

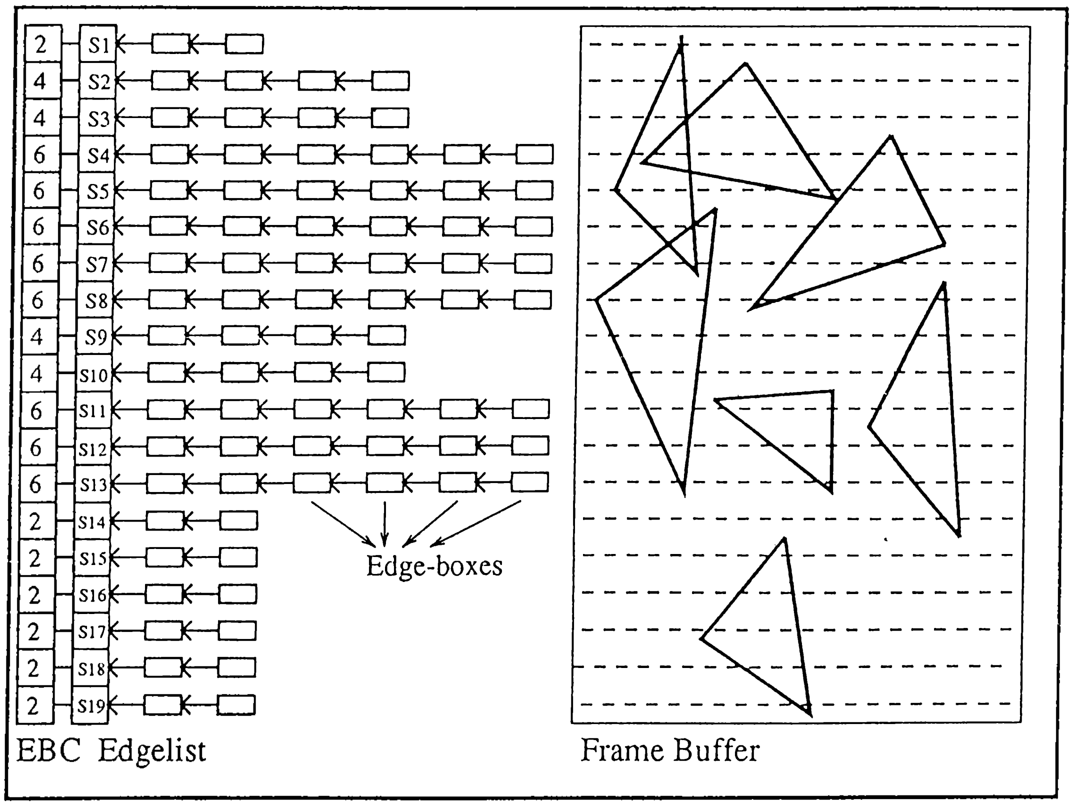


Figure 4.8. Edge list formation with counter for each line

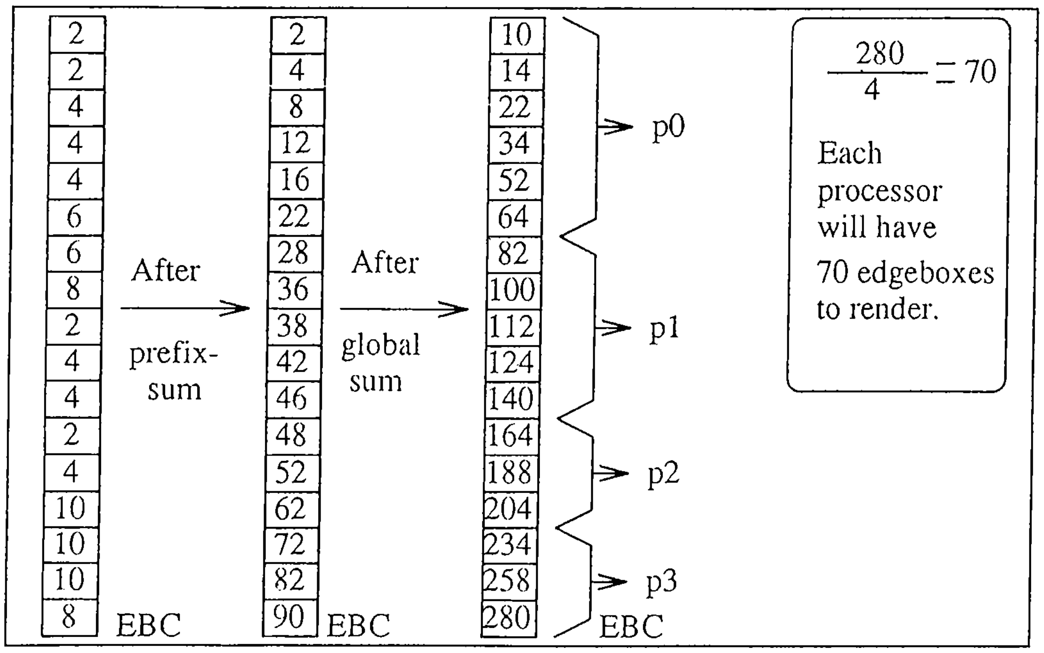


Figure 4.9. Prefix and global sums of EBC

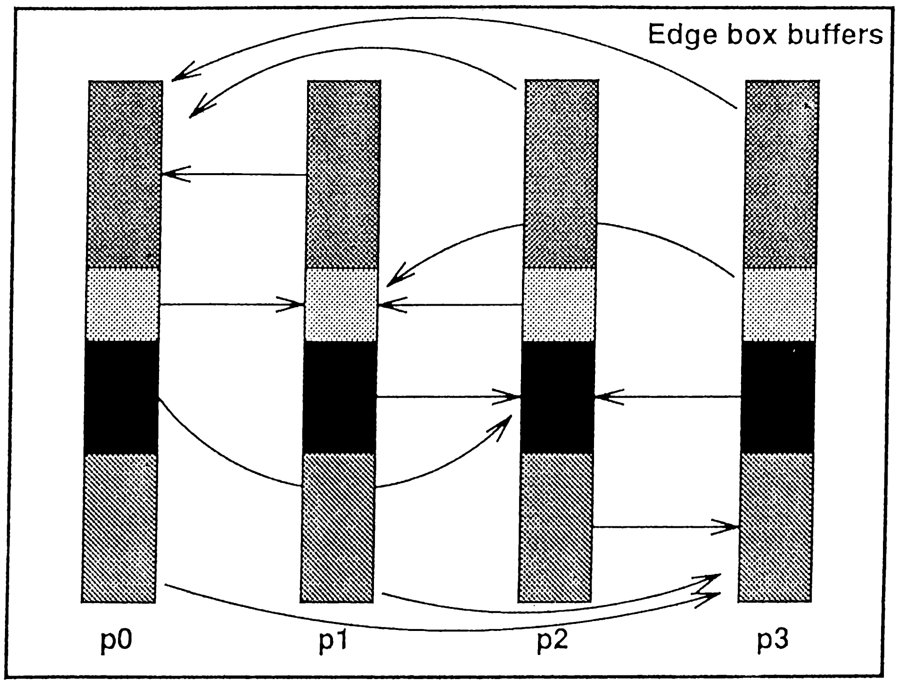


Figure 4.10. Exchange of Edge Box Data Between Processors

important thing about this process is that the *z-values* of the stationary parts are not lost and they will be used in the processing of the inbetweens of the next interval.

At the end of this process, the CFB and the MFB are sent to the host and the CZB's of the processors are inter-communicated because of the need to determine the visibility of the inbetween frames. The formation of the resulting frame buffer will be explained later.

4.9 Processing of an Inbetween Frame

After a keyframe is processed, each processor holds a CZB, which will be used in the creation of the inbetween frames. During the generation of the inbetween frame, recently computed *z-values* of the moving parts are compared to the MZB for local and the CZB for global visibility problems.

With the exception of this comparison to CZB, the processing of an inbetween frame is the same as sequential processing of the frame. Moreover, there are no communications between the processors except for the communication of the CZB at the end of keyframes.

4.10 How Host Interprets the Image Data

At each keyframe, the host receives a CFB. It writes this buffer on the screen. Then, it receives consecutive MFBs. To write a MFB on the screen, the host first has to copy the pixels that will be occupied by the MFB pixels. The background is saved because the host will write it back before writing the moving parts of the next frame.

Due to the nature of the problem, we need a *background saving* mechanism for the MFB writing.

The solution to this problem works very efficient. We hold the received CFB after writing it on the screen as a background buffer. We also hold a 2D binary array, where each bit denotes whether a block of pixels is overwritten or not. That is, as we write any pixel of the MFB on the screen, we set the corresponding bit to 1. Then, before writing a new MFB, the blocks of pixels with set flags are written on the screen.

As long as there is spatial coherence in the incoming data, the use of blocks instead of single pixels for flagging is efficient. The only issue in this mechanism is how large the blocks should be. If they are too large, unnecessary time will be spent in writing the background buffer back to the screen, if they are too small, overheads in comparison will be introduced.

4.11 Performance Results

To test the efficiency of the Mars algorithms, we made a number of processing time measurements.

To make the measurements, we first set the variables of the rendering process. The factors that affect the processing time of a scene is as follows :

- # of processors
- # of patches (data size)
- # of patches of moving data
- (very important) positioning of the patches.
- whether the frame is a keyframe or an inbetween

The number of processors is obviously one of the factors, since the load is distributed among the processors. As the number of processors increase, their individual loads decrease and a speedup is expected but later on we saw that in fact this is not the case because of granularity problems. When the number of patches is too small, increasing the number of processors beyond a certain value increases the processing time.

The number of patches factor is also obvious. When there are more patches to process, it takes longer to process!

When processing a keyframe or an inbetween, the percent of the data that is moving in the inbetween frames also directly affects the processing time. If this

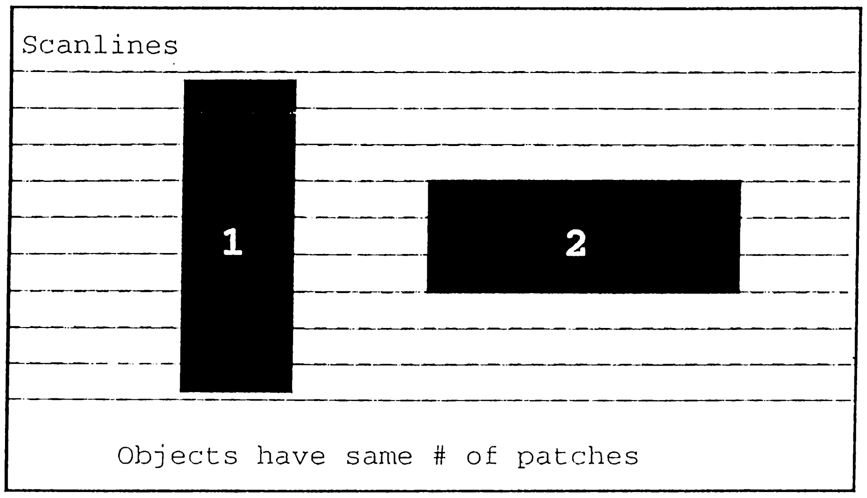


Figure 4.11. How positioning affects processing time

percent is close to zero, there is very little job to do in the inbetween frames and the keyframe is also easier to process. If this percent is close to hundred, things would not change much for the keyframe, but the inbetween frame processing time would be very large. In fact, there is an expected threshold value for this percent, at which the overall time is minimized, but another factor disturbs this intuitive expectation. It is the positioning of the patches.

The positioning of the patches of a frame, dominantly affects the processing time. This effect can be investigated in Figure 4.11. Objects 1 and 2 are identical, that is they have exactly the same shape and the number of their patches is the same. The only difference between them is their alignment. Object 1 is vertically placed, while the other is horizontally aligned. The first object is processed in a longer time than the second one because of the different number of scanlines they intersect. This directly affects the number of iterations of the algorithm, and the first object is processed in a longer time.

The fifth factor that affects the processing time is also obvious. We have two distinct algorithms for processing a keyframe and an inbetween. While processing a keyframe, the load of a single keyframe is distributed among the processing nodes. The nodes work concurrently and there is also communication between the nodes. At the last stage of the process, each processor sends the host its contribution of the image, and the host writes the final image on the screen. On the other hand, an inbetween frame is dedicated to a single processor with respect to its index number.

The processor does all the rendering of the inbetween and sends the resultant image to the host. So the issue of processing a keyframe or an inbetween frame obviously affects the processing time.

The important issues in the performance of the overall algorithm is as follows :

- There is a linear speedup in the inbetween frame processing. (2n processors process 2n inbetween frames in half time n processors process 2n inbetween frames). This is obvious since each processor processes a single inbetween frame independently of each other. So if number of processors is doubled, the number of frames that is processed in the same time is also doubled.
- For a keyframe, the percent of the moving data affects the processing time but not to a great extend (Tables 4.1 and 4.2). So we can simplify our discussion by dropping this factor from further inspection for keyframes.
- For an inbetween frame, the percent of the moving data is the main concept because the data to be processed is this moving data. As it increases, the processing time also increases.
- The positioning of the data to be processed is very important (discussed above). To obtain some results, we somehow tested the performance for some specific positioning. To show how positioning affects the processing time, we tested many films with same models. The models perform the same motion (that is the films have same number of patches and same percent of moving data). We tested only the keyframes as our aim is to show how positioning affects processing time. Each model is differently aligned in each of the films. The results show that positioning strongly affects the processing time (Table 4.4).
- Granularity problem becomes important for the processing of keyframes. For small data sizes, the maximum speedup is achieved with four processors (Table 4.1). For small data sizes, increasing the number of processing nodes does not decrease the processing time. This is called the granularity problem. As data size increases, this becomes eight (Table 4.2) and then it becomes sixteen (Table 4.3). Memory was not sufficient to test the algorithm for larger data sizes. This may be tested in future if memory becomes available.

We also plotted a 3D graph showing # of processors versus # of data versus processing time (Figure 4.12).

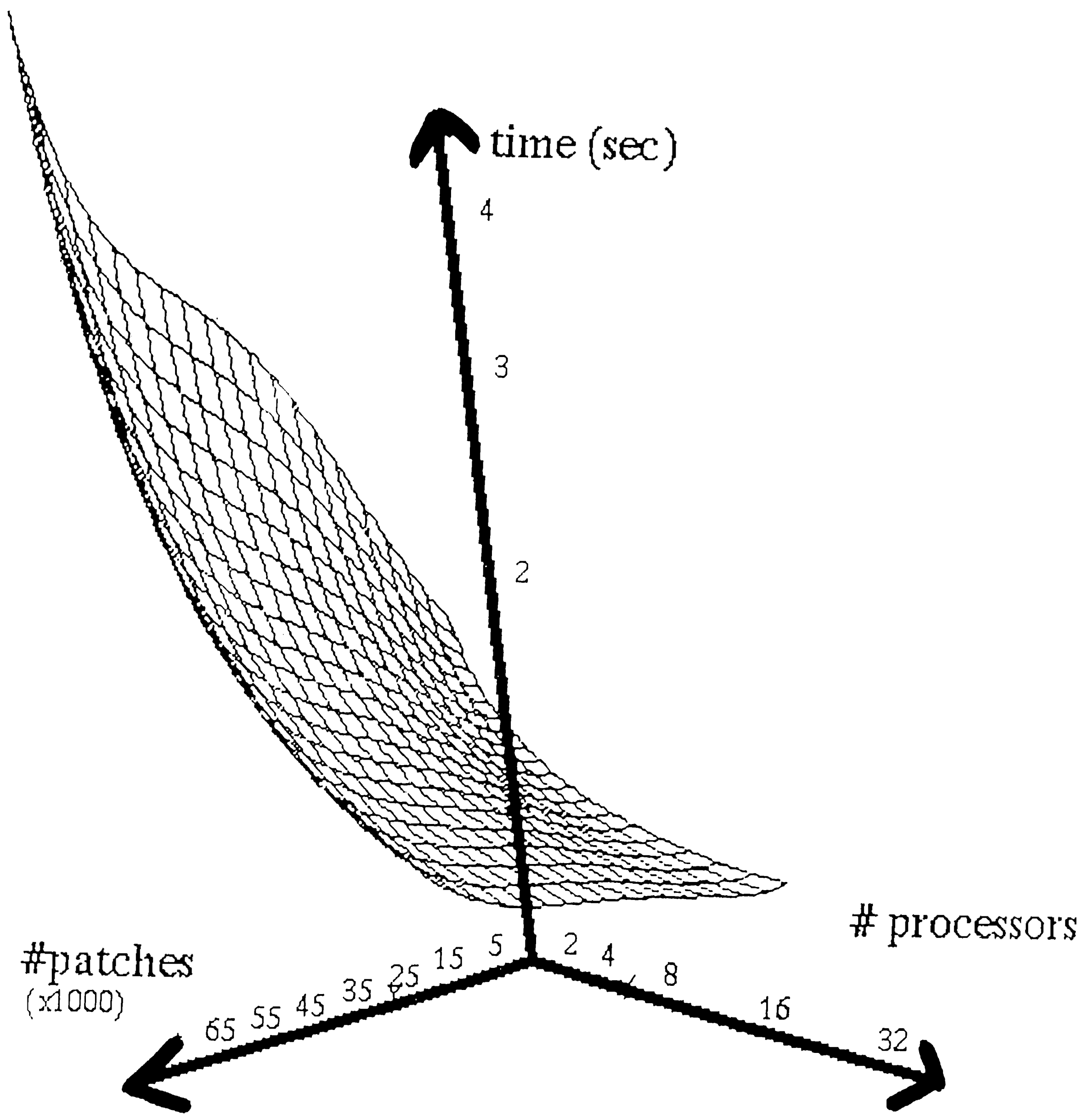


Figure 4.12. Plot of number of processors vs. number of patches vs time

Moving Part	Keyframe # of processors					Inbetween # of processors				
	2	4	8	16	32	2	4	8	16	32
1300	2449	2306	2486	2917	3711	403	269	164	101	58
2600	2450	2309	2476	2918	3739	492	317	186	113	64
3900	2455	2331	2468	2926	3721	583	353	207	123	68

Table 4.1. Results of the rendering algorithm (in ms) for data size = 3900 (3 segments)

Moving Part	Keyframe # of processors					Inbetween # of processors				
	2	4	8	16	32	2	4	8	16	32
1300		4320	3337	3503	4211	806	442	227	126	82
2600		4314	3321	3452	4195	1188	634	323	172	122
3900		4315	3446	3523	4184	1484	783	394	209	124
5200		4312	3335	3470	4203	2109	1101	563	290	163
6500		4323	3334	3468	4208	2298	1178	598	307	173
7800		4320	3212	3168	3625	2551	1293	662	339	188
9100		4290	3220	3157	3643	2691	1393	706	355	201
10400		4312	3218	3190	3615	3215	1693	850	432	236

Table 4.2. Results of the rendering algorithm (in ms) for data size = 10400 (8 segments)

Data size	Keyframe # of processors			
	2	4	8	16
5200	2308	1888	1917	1970
10392		2248	2031	2013
15568		2168	1866	1856
17296		2427	1961	1814
22488		2836	2279	1991
27664		2265	1851	1869
32856		4159	2946	2029
43224			2441	2066
55320		3562	2888	1919
65704		5041	3657	2527
70880			3007	2202
82976		4304	3026	2146
98536			3374	2325
110632		4984	3372	2673
138288			4121	2686
165944			3831	2821

Table 4.3. Results of the rendering algorithm (in ms) for different data sizes of a keyframe

Film	Keyframe # of processors				
	2	4	8	16	32
1	4817	3536	2950	3099	3922
2	5250	4018	3012	2891	3385
3	4949	3597	2841	2801	3266
4	3733	2860	2581	2669	3197
5	5134	3673	2810	2850	3329

Table 4.4. Results of the rendering algorithm (in ms) for same data sizes (27664) and percent of moving data (2 segments) but different topologies of a keyframe

Chapter 5

CONCLUSIONS

Mars is a testbed to implement some new thoughts and algorithms in modeling, animation and distributed rendering.

To implement our parallel animation rendering algorithm, we have implemented an animation system that would give appropriate animation data as input to our parallel renderer. But, as our system is tool based, any of the tools can be replaced with a more improved version in the future, and the integrity of the system can be kept.

Mars is still under development. In spite of the fact that the modeler and the animator are developed to make research on rendering of a film, they constitute a fairly complete system. It brings along some efficiencies, like multiple representation and animation and model creation flexibilities. Mars can create quite complex and real-looking scenes for animations.

The notions that exist in Mars are :

- The structure of the model can be of any n-ary tree structure, thus allowing all kinds of models with no closed chains.
- The segments of the model can be of any complexity. The animator can specify each segment of a model as a collection of very complex Bezier surface patches, so photo-reality in terms of modeling can be achieved.
- The keyframer uses a simpler (say skeleton) model while positioning the object,

so motion specification is performed easily and quickly.

- The previewing of the object can be done in real time no matter how complex the object is, as simpler representations are used. This leads to a quicker development of the keyframes and the animation.
- A matrix method is introduced in the interpolation part of the animator.
- The animation scene can be viewed in many directions in the motion specification part, to ease the job of the animator.
- Rendering of the frames is the longest and most expensive part of the computer animation generation process. To cut down the generation time, Mars uses distributed processing techniques on a multicomputer.
- The scenes and the models of an animation should have complex structures for a good feeling of reality. Admirable results come out only with expensive shading techniques (such as Phong). All those constraints imply that the processing time for rendering the frames will be quite long. In addition to distributed processing, Mars exploits the coherence that exist between the successive frames of a film. To exploit this coherence, Mars adopted the Multiplanes approach [14] used in traditional animation.
- Mars methods of distributed rendering optimizes the communication issues. This is very important because usually most of the distributed processing time is spent in communications between the processing nodes.
- Mars interacts with the user in a user-friendly environment (Sun Sparc workstations running XWindows) and gives output to a non-interactive but high-quality graphics engine (Intel personal super computer 2 with Targa graphics board).

Bibliography

[1] Norman I. Badler. Animating human figures: Perspectives and directions. In *Graphics Interface '86 & Vision Interface*, pages 115–20, 1986.

[2] Norman I. Badler and Kamran H. Manoochehri. Multi-dimensional input techniques and articulated figure positioning by multiple constraints. In *1986 Workshop on Interactive Computer Graphics*, pages 151–69, 1986.

[3] Norman I. Badler and Kamran H. Manoochehri. Articulated figure positioning by multiple constraints. *IEEE Computer Graphics & Applications*, 7(7):28–38, 1987.

[4] Phong Bui-Tuong. Illumination for computer generated pictures. *Communications of ACM*, 18:311–317, 1975.

[5] Danny G. Cachola and Gunther F. Schrack. Modeling and animating three-dimensional articulate figures. In *Graphics Interface 1986 & Vision Interface*, pages 152–57, 1986.

[6] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *Computers Graphics*, 15:307–316, 1982.

[7] Edwin Catmull. Computer display of curved surfaces. In *Proceedings IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pages 309–15, 1975.

[8] Gerald Ferin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press Inc., 1990.

[9] D.A. Field. Mathematical problems in solid modeling. In *Geometric Modeling : Algorithms and New Trends*, pages 91–108, 1987.

[10] S. Hewitt, G. Ridscale and T.W. Calvert. The interactive specification of human animation. In *Graphics Interface '86 & Vision Interface '86*, pages 121–30, 1986.

[11] Phillip Getto and David Breen. An object-oriented architecture for a computer animation system. *The Visual Computer*, 6:79–92, 1990.

[12] Don Heller. *XView Programming Manual.* O'Reilly & Associates, Inc., 2 edition.

[13] R.F. Sproull, I.E. Sutherland and R.A. Schumacker. A characterization of ten hidden surface removal algorithms. *ACM Computing Surveys*, 6:1–55, 1974.

[14] Alden W. Jackson and Joel M. Morris. Enhancement of diglib: Computer graphics software for animated computer-generated video movies. *Computers & Graphics*, 12:271–283, 1988.

[15] John Lasseter. Principles of traditional animation applied to 3d computer animation. *Computer Graphics*, 21:35–44, July 1987.

[16] Nadia Magnenat-Thalmann and Daniel Thalmann. *Computer Animation, Theory and Practice.* Springer-Verlag Tokyo, Berlin, Heidelberg,Newyork, 1985.

[17] S. Kamran Mahmud and Bülent Özgüç. Human body animation. In *Proceedings of the Fifth International Symposium on Computer and Information Science*, pages 885–894, 1990.

[18] S. Kamran Mahmud and Bülent Özgüç. Semi goal-directed animation : A new abstraction of motion specification in parametric key-frame animation of human motion. In *Proceedings of the Second Eurographics Workshop on Animation and Simulation Vienna*, pages 75–87, 1991.

[19] Brian Wyvill, Michael Chmilar and Chuck Herr. A software architecture for integrating modeling with kinematic and dynamic animation. *The Visual Computer*, 7:122–137, 1991.

[20] Marc R. Grosso, Norman I. Badler and Richard D. Quach. Anthropometry for computer graphics human figures. Technical report, University of Pennsylvania, 1989.

[21] Bülent Özgüç. Thoughts on user interface design for multi window environments. In *Proceedings of the Second International Symposium on Computer and Information Science Istanbul*, pages 477–488, 1988.

[22] David F. Rogers. *Mathematical Elements for Computer Graphics.* McGraw Hill, 1989.

[23] David F. Rogers. *Procedural Elements for Computer Graphics.* McGraw Hill, 1990.

[24] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of the SIGGRAPH*, pages 245–254, 1985.

[25] Yılmaz Tokad. *Analysis of Engineering Systems.* Bilkent University, 1990.

[26] Alan Watt. *Fundamentals of Three-dimensional Computer Graphics.* Addison-Wesley, 1989.

[27] Eben F. Ostby, William T. Reeves and Samuel J. Leffler. The menv modeling and animation environment. *The Journal of Visualization & Computer Animation.* 1:33–40, 1990.

[28] W.R. Franklin and M.S. Kankanhalli Parallel object-space hidden surface removal algorithm. *Computer Graphics*, 24, 1990.