3-D Animation and Morphing using RenderMan

A Thesis by: Somhairle Foley B.Sc.

Supervisor: Dr. Michael Scott Ph.D.

Submitted to the
School of Computer Applications
Dublin City University
for the degree of Master of Science

July 1996

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Sorcha Foley ID Number: 92700652 Date: 20th Sept 1996

Acknowledgments

I'd like to thank the School of Computer Applications in DCU for initially funding my research and for letting me complete it when it got delayed

The main person responsible for letting me do this thesis in the first place and for helping me get it finished is my supervisor Dr Michael Scott Thanks Mike

Thanks also to the technicians in the Computer Applications school, Jim Doyle and Eamonn McGonigle, and to Tony Hevey in Computer Services

This thesis would have been impossible to write without the help of a number of people from Pixar and other places via the Internet There are too many to name individually, but I'd like to thank them all too

Finally, I'd like to thank my family and friends, and the postgrads in the lab, past and present, for the more social side of being a postgrad

Table of Contents

Glossary

Composite Object	An object which consists of a number of primitives
CSG	Constructive Solid Geometry A modelling method which allows objects to be combined using set operators
DOF	Degree of Freedom An independent variable that controls the ability of an articulated object to move relative to another
FFD	Free Form Deformation A technique used to deform or warp objects independent of the object type
Keyframes	Frames of an animation that delimit a simple movement or action Using these the frames in-between can be created
Modelling	The process of describing objects and lights in a 3-D scene
Morphing	The process of transforming one object into another, usually by changing the surface representation of the object
NURBS	Non-Uniform Rational B-Splines A powerful type of object which is used to model smooth curved surface objects
Patch	A smooth curved surface modelling object defined by the combination of two splines Usually bi-cubic or bi-linear
Patchmesh	A set of connecting patches which can represent a surface
Primitive	Graphics object which cannot be split into component objects Examples Sphere, Cylinder, Polygon, Patchmesh, NURBS
PRMan (*prman*)	Photorealistic RenderMan Pixar's highest quality renderer It is the original RenderMan Interface compliant renderer
Rendering	The process of creating a computer generated image
RenderMan Interface	The public specification issued by Pixar for the description of 3-D scenes, separating the modelling and rendering phases
REYES	The underlying algorithm at the heart of PRMan
RIB	RenderMan Interface Bytestream A file or datastream with 3-D descriptions conforming to the RenderMan Interface
Splines	Mathematical representations of smooth curves defined by a set of points Can be used for interpolation or approximating surfaces Bézier, Catmull-Rom and NURBS are all types
Topology	The surface of an object or combination of objects

Abstract

an-I-ma-te~\an-e-,māt\vt
(1538)

- 1a to give life to
- b to give vigour and zest to
- 2 to give spirit and support to ENCOURAGE
- 3 to move to action
- 4a to make or design in such a way as to create apparently spontaneous lifelike movement preparation of animated cartoons
- 4b to produce in the form of an animated cartoon

meta-mor-pho-sis-\,met-e -'mo r-fə-səs\n, pl -pho-ses \,sê z\
[L, fr Gk metamorphō sis, fr metamorph *noun* to transform, fr meta- + morphê form]
(1533)

- 1a change of physical form, structure, or substance esp by supernatural means
- b a striking alteration in appearance, character, or circumstances
- 2 a marked and more or less abrupt developmental change in the form or structure of an animal (as a butterfly or a frog) occurring subsequent to birth or hatching

A Websters dictionary definition of the word ‘animate’ strikes to the heart of what animation is about - the illusion of life Giving life to a sequence of pictures is the purpose of animation This has been practised for over a century and nowadays computers are being used to create animations faster and more accurately than ever before Animations are no longer restricted to 'funnies', but can also be models of real-life situations These are based on data not images, and the final images are generated after the data has been processed for unambiguous visualization By representing the data in three dimensions it can be viewed in any number of ways according to the wishes of the 'end' user

RenderMan allows a scene to be viewed when defined in three dimensions This can then be viewed as an animated sequence where special effects - such as the metamorphosis of objects (morphing) - may take place to provide a photorealistic animation This thesis will examine how 3-D computer animation in general, and special effects such as morphing in particular, may be implemented using the RenderMan Interface specification and the RenderMan rendering program

Chapter One : Introduction

1.1 Thesis Outline

This thesis contains six chapters Chapter One is a general introduction to the area of computer generated images and concepts such as modelling and rendering are explained The RenderMan Interface specification and the REYES implementation of it are discussed and some of the common phraseology interpreted, before a quick overview of the subject topics of animation and morphing is be given

Chapter Two is a broad history of animation which highlights the significant stages in its development along the way to modern computer animation It notes that the important lessons learned from traditional animation are still relevant in today's high-tech productions

Chapter Three looks at animation and the current methods used to implement computer animation An important part of this are the methods (such as interpolation) used to control the movements of objects

Chapter Four is a discussion of morphing and the different approaches to it which have been attempted It also looks at the subject of complex objects and suggests methods for implementing a heuristic for morphing them

Chapter Five describes the process of implementing animation and morphing using RIB files Some of the methods suggested in previous chapters were implemented using two applications which were developed

Chapter Six contains the conclusions of the research which was carried out and identifies problems which arose during implementation A number of areas for future research are suggested and there is a brief look at the future

1.2 An Introduction to Rendering

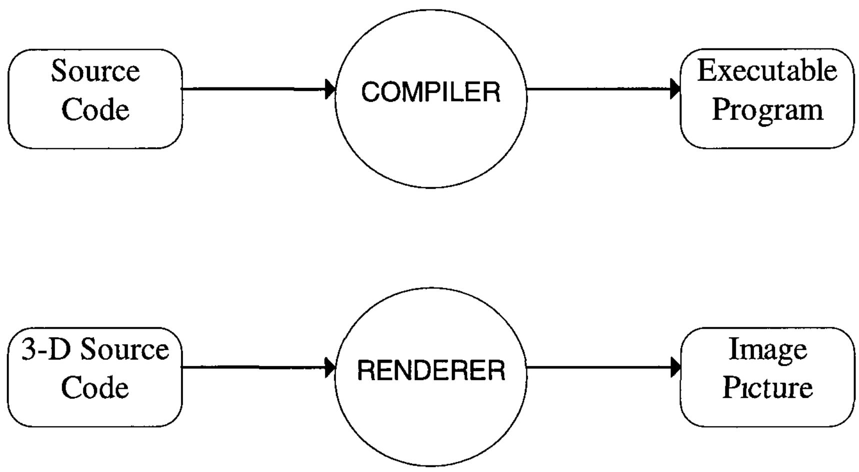


Figure 1-1 : Rendering is similar to compiling

This thesis is based on the creation of Computer Generated Images The process of creating an image from a three-dimensional description is called Rendering and a program that does this is called a Renderer In the same way that a compiler produces an executable program from source code, a renderer takes 3-D source code, processes it and outputs a 2-D image

Like a compiler, a renderer is a program that is generally supplied by a third party and which is unchangeable, although new versions and alternative suppliers' versions may provide new features To produce an image, the 3-D code is created by hand or, more commonly, by a program called a modeller Since rendering is usually a long and time-consuming process, most modellers can provide draft or wire-frame renderings in real-time or near real-time which allows a preview to be viewed without waiting for a full rendering

1.2.1 There are a number of types of renderer

There are an large number of renderers available these days - not all are compatible or work the same way Many do the same thing slightly differently but work on different formats or require different input sets The internal workings of a renderer can be completely different For example, four different methods of rendering are

- Z-Buffer
- Ray-Tracing
- Radiosity
- REYES (RenderMan)

- Z-Buffer Rendering

Z-Buffer rendering is where objects in World Co-ordinates (the world the scene is defined in) are transformed into Camera Co-ordinates This involves changing from a perspective view to a parallel view in Camera Co-ordinates The Camera Plane is usually implemented using a frame buffer which stores the colours that hit the plane at each pixel Objects are not processed in order (from far to near), so the buffer needs to store both the colour and the depth (the Z-coordinate) of the object The Z-coordinate is used to determine if an object is in front of or behind the object that is currently in view If it is in front, then the frame buffer for that pixel is updated and the new Z-coordmate is put in the depth-buffer (called the Z-buffer)

Once all objects are in camera co-ordinates, the objects need only have their Z-coordinate mapped to the buffer in order to project them onto the camera plane (the screen) The need for a depth-buffer can be eliminated by working from the farthest object from the camera plane to the nearest, all the objects will be correctly projected onto the frame-buffer as the human eye would see them The nearest objects will be 'in front' of the farther objects, giving the correct picture

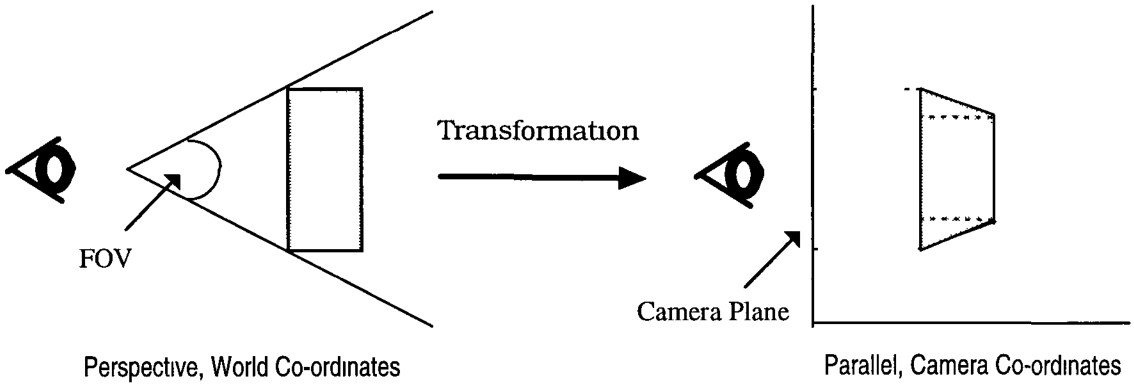


Figure 1-2 : Simple Z-Buffer Rendering

The Z-Buffer method is used in a surprising number of renderers (usually for 'draft' rendering) because of its simplicity to implement and hence its speed of execution On a number of systems, it has been implemented in hardware which allows real-time processing and viewing

- Ray-Tracing

Ray-Tracing is one of the most popular methods for generating photorealistic images Originally a hidden surface detection algorithm, it was developed into a full renderer Ray Tracing involves following a ray of light from a light source to the camera or vice versa

Forward Ray Tracing

Forward Ray Tracing is where each ray of light from a light source is followed until it is absorbed or hits the camera 'window' This is very inefficient since there may be millions of rays, none of which reach the camera but these would all have to be calculated and checked to see if they hit the camera window

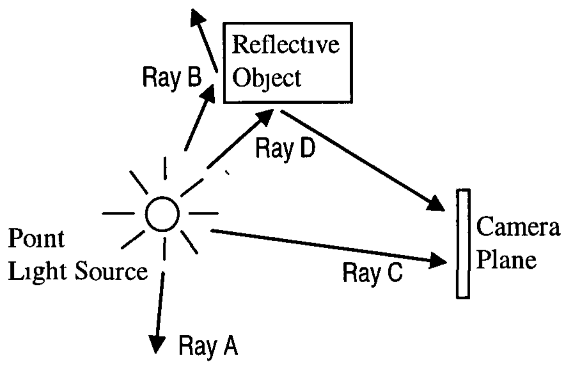


Figure 1-3 : Forward Ray Tracing

In the diagram, Ray A does not hit anything

Ray B hits the object and then reflected away

Ray C hits the camera plane

Ray D hits the object and is then reflected so it hits the camera plane

Backward Ray Tracing

To improve the speed, a method called Backward Ray Tracing was developed Backward Ray Tracing is where each pixel on the camera plane/window is considered to be a ray of light and it is followed until it hits a light source Rays can be reflected, refracted or transparency rays Also, illuminating and shadow rays can be followed

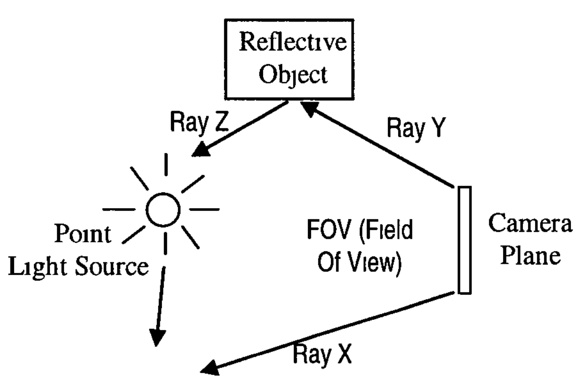


Figure 1-4 : Backward Ray Tracing

Ray X and Ray Y are the outer rays that are computed All the rays in-between them will be computed The reflection off the object is calculated by calculating the Ray Z which is traced back to the light source

Backward Ray Tracing is much more efficient than Forward Ray Tracing since only rays that hit the camera plane are computed However, proper reflection is quite difficult to achieve since the reflected light may come from a number of sources and depends on the texture of the surfaces that it is shining off Usually a recursive method is used to calculate the colour of a ray which works back from the ray hitting the camera plane until the ray hits a light source or has travelled so far that it has no significant effect on the colour of the ray at the camera When there are a large number of objects that reflect the rays, the ray tracer will be doing more recursive loops for each ray, so that the speed of rendering is slowed

Ray Tracing gives a very accurate picture since it requires every ray of light to be followed, giving better lighting and shading effects It is more computationally expensive than Z-Buffering, but it produces better results Examples of ray tracers are POV-Ray, PolyRay, Moral Ray and the *RendRIB* renderer from the RenderMan-Interface-compliant Blue Moon Rendering Tools

[FOLEY90][OREILL91] [MARRIO92]

- Radiosity

Radiosity is a method of rendering which works by tracing rays of energy rather than rays of light It was developed from the physics of thermodynamics to accurately simulate the way in which rays create various types of shadows As the energy is dispersed so the lighting/shading will change This is considered to be one of the most realistic ways to render light and shading since it allows the umbra and penumbra effects that other methods do not render accurately Radiosity rendering is more computationally expensive that other renderers and using it to render an entire scene is considered a waste since a lot of the scene will probably appear the same using a simpler renderer Radiosity renderers tend to be included with ray-tracing renderers to speed up rendering times The radiosity part is invoked only when required, leaving the ray-tracer to render the rest of the scene

- REYES (RenderMan)

The REYES algorithm was initially developed and used in 1981/82 by Loren Carpenter at Lucasfilm's Computer Animation Division for the computer simulation of 'The Genesis Effect' in the film *Star Trek II The Wrath of Khan* Its successful use caused more research and development (and the addition of Rob Cook and Ed Catmull to the team) to be invested in REYES The Computer Animation Division was purchased by Steve Jobs in 1986 and the company was named Pixar The REYES algorithm was developed and revised over a number of years during which time it was used in the creation of a number of landmark computer animations such as *Luxo Jr* and *Red's Dream*

In 1987, a paper on 'The REYES Image Rendering Architecture' was presented at the SIGGRAPH'87 conference This outlined the objectives of the REYES renderer and its implementation along with the advantages and disadvantages of using the new algorithm [COOK87]

The goal was to be able to produce high quality - 'photorealistic' - images within a reasonable time period for feature films To produce these images the models - scene descriptions - needed to have large numbers of complex objects All of these objects were reduced to a single type of object, micropolygons, upon which all functions were carried out By working on the micropolygon level, difficulties with different types of objects (geometric primitives, procedural models, fractals, etc) were eliminated and the same process applied to all objects by breaking them down into smaller objects recursively, and down to the micropolygon level which are smaller than the size of a pixel This solves a number of interpolation problems including clipping and shading

A programmable shader was included to allow for different possible surface characteristics - from different colours and reflection maps to bump maps, shadows and refraction A C-like shading language (which was to be defined later) allowed each point on a surface to be shaded/textured/coloured given the different types of light and their intensity that are touching that point

A number of design principles were laid out which were used in designing the algorithm

- Natural Co-ordinates should be used wherever possible to save on conversions to other co-ordinate systems
- Vectorization should be used to group similar calculations together
- Common Representation should be used - objects should be 'diced' into micropolygons
- Locality should cause primitives to be rendered without reference to other objects
- Linearity should cause rendering time to be linearly proportional to complexity of the objects being rendered
- A Back Door into the algorithm should allow other algorithms to have an input into the final image
- Texture Maps should be used to define complex shading patterns

It was noted early on that minimal ray-tracing was to be used in the algorithm It was decided that tracing rays of light or energy would be too time-consuming for complex models where a ray could be reflected and/or refracted any number of times Instead, global light sources were seen as the main method of illumination with environment and shadow maps used as surfaces of reflective and hidden objects respectively Programmable shaders would provide special reflection/refraction effects

Dicing is the term applied to the recursive sub-division of objects down to the micropolygon level After objects have been diced into micropolygons and these are shaded/texture mapped, they are sampled Micropolygons tend to have the approximate dimension in screen space of ¼ the area of a pixel However, they are not aligned with pixel boundaries so some form of sampling must be used to gain an accurate value for the pixel

In the REYES architecture, jittering (a type of stochastic sampling where a random displacement factor is used) is used to sample micropolygons for each pixel This is placed in a simple Z-Buffer where visibility is checked and if required amended

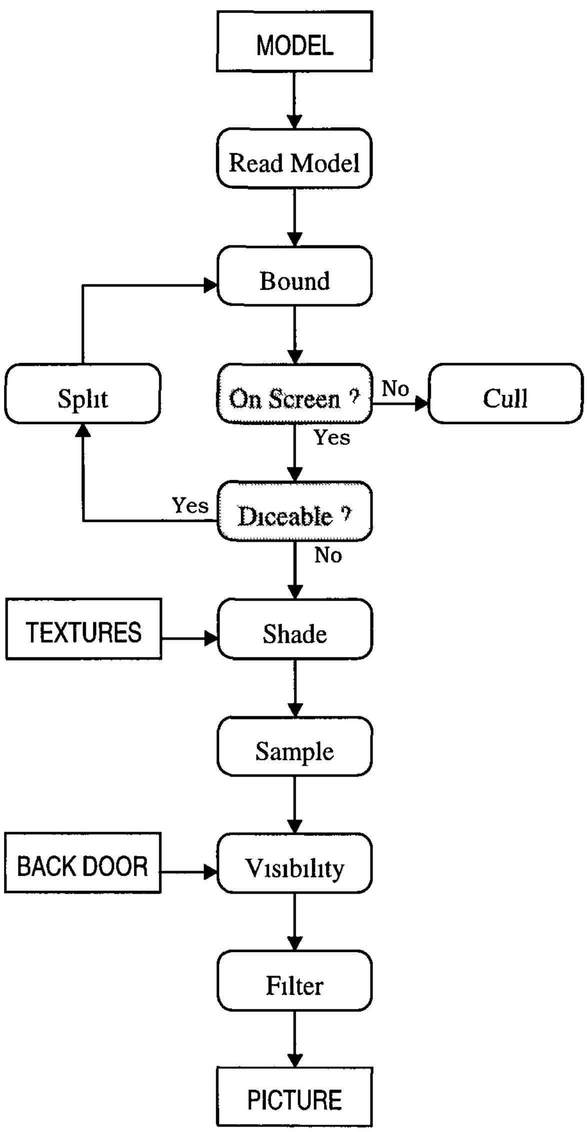


Figure 1-5 : Flowchart for REYES Algorithm

Once the model of the scene is read in, objects are checked to see if they are within the general bounding-box for the view and then the dice/cull/split occurs

If some part of an object is on-screen then that part must be diced into micropolygons By splitting up objects repeatedly, culling parts that don't appear and dicing the remaining (partial) objects, the object is effectively clipped

Micropolygons are texture mapped and sampled to get pixel values using jittering as described earlier The visibility of the sampled pixel is normally checked using a simple Z-Buffer However a 'Back Door' gives extra options that can modify the buffer if required

Features such as Field of View (Zooming), Motion Blur, Transparency and CSG (Constructive Solid Geometry) were added to the REYES architecture which now forms the core of the Photorealistic RenderMan (PRMan) renderer that Pixar sell

The REYES architecture does have problems with certain types of primitive such as particles and 'blobs' Particle rendering is the area related to the representation of small non-opaic objects which can distort images and is usually related to something such as the weather (mist, rain, fog) or fire While the RenderMan Interface does allow `Atmosphere` custom-shaders to be written and used, only some of the above have been implemented REYES also lacks a quick way of deciding if shading is to be constant across large surfaces and these can be needlessly broken down into micropolygons This has been rectified to a certain extent in the PRMan implementation by the ability to explicitly specify that constant shading is to be used It is difficult to optimise the dicing of texture-mapped polygons since they lack a natural co-ordinate system (polygons always use the current co-ordinate system) However, this is not a problem with most REYES/RenderMan models since they tend to use the more flexible bi-cubic patchmeshes

The REYES architecture was (and still is) a radically different method of rendering two-dimensional pictures from three-dimensional models The acronym REYES stands for 'Renders Everything You Ever Saw' which is actually quite a good description of how it works - it only concerns itself with viewable objects and renders those, ignoring ("culling") those that are not visible To a certain extent it does 'cheat' when posed some problems by using customised shaders to represent complex surfaces which are difficult to model - for example the screw threads on a light bulb as shown in chapter 5 - the light bulb's threads are modelled using a cylinder with a displacement shader However, it was designed on the basis of a number of principles and goals and the algorithm was designed for those principles whereas many other renderers are based on existing algorithms REYES removed a number of bottle-neck calculations that traditional approaches suffer from and provided a baseline against which other renderers are compared In keeping with the original goal of the architecture, PRMan has proved to be a favourite amongst animators and special effects companies when producing computer animations for feature films Photorealistic RenderMan has recently been used in such movies as *The Abyss*, *Terminator 2*, *Jurassic Park* and *Toy Story*

1.2.2 Photorealism

"Photography is truth The cinema is truth twenty-four times per second " - Jean-Luc Godard

Photorealistic renderers are a specific type of renderer that attempt to produce pictures of a quality that is indistinguishable from a photograph Renderers of this quality used to only be available to specialists with ultra-fast computers, but in recent years photorealistic renderers have been appearing which work on more popular hardware In some cases, a picture generated using a photorealistic renderer can be too 'perfect' - sometimes they have to be 'dirtied' or have motion blur added (the blur of an object moving fast enough to create multiple images of itself while the camera shutter is open) In visualisation, a photorealistic renderer may produce an accurate picture, but

e to the 'realism' of the image For example, an accurate
there is little light and many shadows may not be what is
ay be removed and the lights 'brightened'

lelling

ess in itself and is the precursor to any animation or special
rk with a scene you must describe the scene completely in
e is only as good as the scene that was described - and a
n the data that its given The amount of data required to
uite large and it is much more efficient to handle this data
d a modeller On some platforms - such as NeXT and SGI -
all applications and can be called from all programs, which
ion to create and manipulate a 3-D picture

e classified as three types of object

- Solid Objects
- Lights
- A Camera

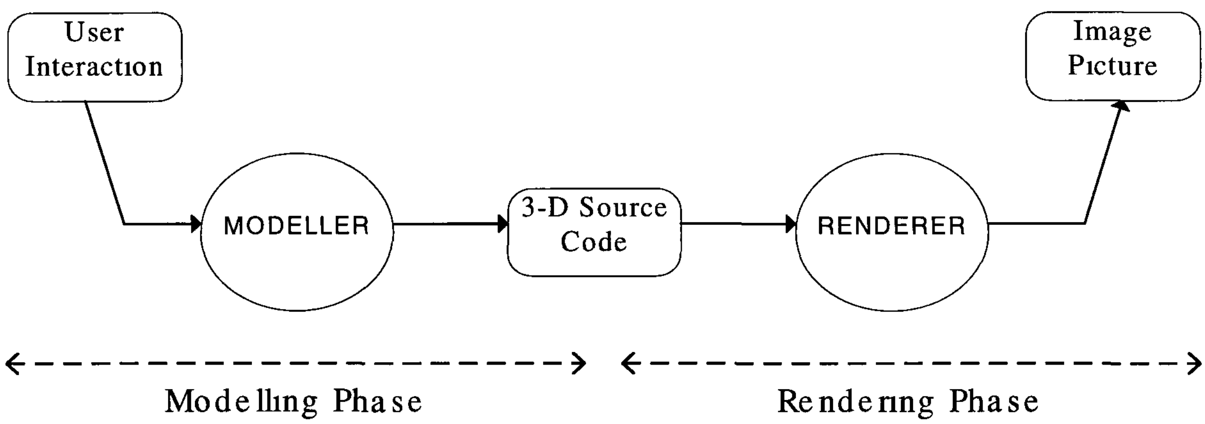


Figure 1-6 : The Modelling and Rendering Phases

1.3.1 Objects

Objects are usually defined by their surfaces For example, a cube is defined as six squares placed in the appropriate place There is nothing inside the cube All objects are surfaces which are infinitely thin, i e they exist only in two dimensions The most common building-block object is the polygon This is a two-dimensional object which has its boundaries defined by a series of points which are connected together

Most renderers allow other types of objects to be used such as quadratics (spheres, cylinders, cones, etc) and parametrics (splines, spline meshes, special spline types) Objects can be grouped together to create a single (composite) object One method of combining objects is CSG - Constructive Solid Geometry This allows set operations - union, intersection and difference - to be carried out on three dimensional objects So it is possible to create a composite object by the union of two or more individual objects It is also possible to define an object by defining 'what is not there' - for example, a bowling ball could be defined as a sphere less three cylinders (for the finger holes)

To declare where an object is situated in a scene (or in/on which part of another object), geometric transformations such as Scale, Rotate and Transform are used For example, to declare three lines which are perpendicular to each other (running along the x-, y- and z-axes), the following sequence would be appropriate

```
Instance Line (along z-axis)
Rotate X·90° Y 0° Z:0°
Instance Line (along z-axis)
Rotate X:0° Y:90° Z:0°
Instance Line (along z-axis)
```

Objects can have attributes such as colour and opacity Opacity allows objects to be solid, see-through or somewhere in between, This makes objects such as coloured glass much easier to model

Objects are usually defined in their own co-ordinate systems That means that the numbers assigned to an object's vertices are only valid as a representation of proportion and do not relate to other objects in the scene These are called local co-ordinates The co-ordinate system that represents the difference between different objects is called the world co-ordinate system (WC)

1.3.2 Lights

To describe a scene correctly, the light sources in it must be added to the scene description during the modelling phase There are usually four types of light source allowed ambient, distant, point and spot

Ambient Light

This is where the light shines equally on all surfaces irrespective of the angle

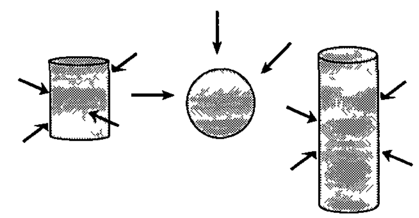


Figure 1-7 : Ambient Light

Distant Light

This is where the light shines equally on surfaces visible from a certain angle This could be viewed as a light coming from a point an infinite distance away - for example the Sun

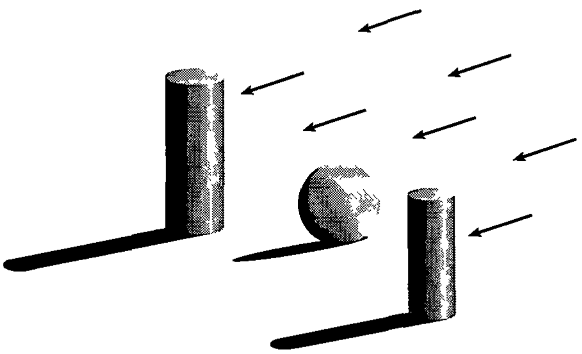


Figure 1-8 : Distant Light

Point Light

This is where light shines equally in all directions from a single point Surfaces are illuminated if they are visible from that point and within the falloff range of the light beam - just like a houselamp

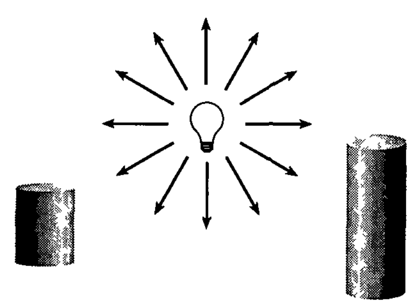


Figure 1-9 : Point Light

Spot Light

This is where light shines from a point in a given direction with a cone specifying the direction and distribution of the light beam Surfaces within the cone and within range of the beam will be illuminated if they are visible from the point

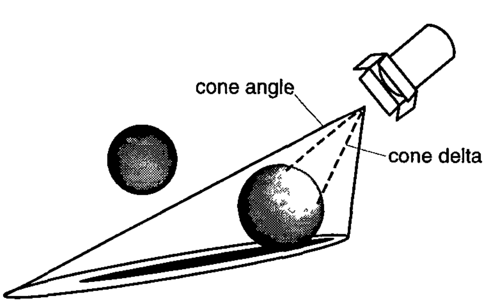


Figure 1-10 : Spot Light

1.3.3 Camera

The camera is a 'virtual camera' - it has no actual effect in the description of the scene, however it does describe how the scene is to be viewed It can be declared explicitly with a particular location and orientation or it can be defined implicitly by assuming all positions are relative to it (it is the origin) Virtual cameras can also allow features such as focusing, zoom, wide-angle lens and motion blur Like the spotlight, the camera is pointed in a given direction, but takes in all light within the cone The cone angle is called the field of view (FOV) and by changing this, the effect of 'zooming' is given

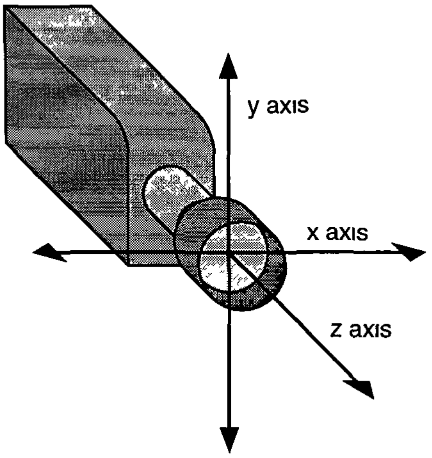


Figure 1-11 : Virtual Camera

1.4 Rendering Light and Shadows

Rendering can be a time-consuming process When creating an animation, a real-tine preview is a useful function However, a full quality rendering would take too long, so other, faster, renderers are used The simplest of these is a wireframe renderer This is where only the edges of the objects are rendered Usually the Z-buffer method is used, coupled with a hidden-surface removal algorithm

Wireframe graphics can be rendered quickly enough, but the lack of shading can make it difficult to comprehend Two of the most popular 'quick' shading algorithms are Gouraud and Phong shading Both of these methods are fast enough to work on desktop computers, preferably with hardware acceleration

1.4.1 Gouraud and Phong Shading

Gouraud shading was developed by Henri Gouraud in 1971 for shading flat (planar) surfaces He suggested that only a small number of points on a surface need actually be calculated for the surface to be reasonably shaded The light rays hitting the vertices of the surface are calculated, then the edges of the surface are linearly interpolated between the vertices The pixels on the surface are then also linearly interpolated from the edges

This gives a surface where the edges define the surface they bound This will give a faceted appearance to objects constructed from a number of Gouraud-shaded objects A 3-D version of this, called 'Smooth Gouraud' shading tries to overcome this problem with facets by averaging the surface normals for surfaces with shared vertices This causes shared edges to be shaded identically, giving the impression of a continuous surface over shared edges

While Gouraud shading is very fast and can be implemented using hardware acceleration for graphics workstations, it does have a number of flaws If a spotlight highlights an area entirely within the vertices, then there will be no highlighting at the vertices and hence the interpolation will cause the spotlight to be ignored It also assumes the surface is reflecting ambient and diffuse light only - the same amount of light is reflected in all directions Ambient light is the light that falls on a surface from any direction Diffuse light is light that falls on a surface from a specific direction While these two can produce a satisfactory result, they cannot produce accurate reflection of light for glossy or shiny surface This is where specular reflection is introduced

Phong shading takes account of all three reflection models - ambient, diffuse and specular Specular reflection differs from the other two in that it relates the position of the eye (observer) to the light that is reflected Only when the angles are correct will the light from a directional light be reflected off the object The shininess of an object will determine within what angle directional light will be reflected

In order to accurately render specular light, it is not sufficient to calculate the light at the vertices of a surface and interpolate from them - the specular component needs to be calculated for every pixel Computationally, this is much slower than Gouraud shading In order to speed this up, the surface normal vector for each pixel is interpolated from the surface normals at the vertices What this effectively does is say that the surface curves linearly between the vertices, allowing non-planar surfaces to be rendered The specular component is combined with the diffuse component and the pixel is coloured with the combined light colour

There are a number of optimisations for Phong shading involving reducing the number of pixels rendered and interpolating between them as well as numerical and geometric optimisations Both Gouraud and Phong shading tend to be used for draft rendering today, with the final image being generated by a photorealistic renderer like RenderMan

1.5 An Introduction to RenderMan

In 1987, Pixar examined the impact of the formalisation and publication of PostScript by Adobe Systems as a 2-D page description language had on the computer industry in general and the graphics community in particular By describing the appearance of a page without reference to what device it is to be represented on, page-creating applications were separated from the different sets of options and command languages available with different printers More importantly, the quality of the image was now limited only by the printers abilities This virtually caused the explosion in the desktop publishing industry

Pixar decided that the time would come when there would be a need for a similar device-independent interface for the 3-D graphics industry and in consultation with other 3-D graphics companies, they developed the formal specification for what is now called the RenderMan Interface

1.6 RenderMan and the RenderMan Interface

When referring to RenderMan, it is important to realise that there are two separate entities involved RenderMan is a program which takes in 3-D scene description data and produces an image (picture) as dictated by the input data The RenderMan Interface is a specification for the format of the 3-D description data This states how a RenderMan-compatible program expects a scene to be described The RenderMan Interface is a public specification which is available at a nominal cost Pixar hope to create a standard method of communication between modeller and renderer, The render need not be RenderMan - at least 4 other renderers are available which are RenderMan Compatible

The specification is bound to two formats function calls and bytestream This allows a compatible renderer to take the form of either a library of routines or a separate program working on a file/stream The RenderMan Interface Bytestream is the term given to the stream used for input to a compatible renderer and hence these files are known as RIB files

On NeXTSTEP platforms, two renderers are provided One is called Quick RenderMan (*qrman*) and is interfaced via object-oriented messages or calls to the 3DKit object library which is bundled with NeXTSTEP The other renderer is Photorealistic RenderMan (*prman*) which takes RIB files for input The two different renderers provide different outputs - *qrman* is a draft renderer which only has some features of the RenderMan Interface implemented and returns wireframe, faceted or draft images whereas *prman* implements almost all parts of the specification and outputs fully photorealistic image files with customised shading

The ability to use either renderer is an example of the flexibility that the RenderMan Interface brings to computer graphics The renderers are interchangeable depending on the quality required and the amount of time allowed for the pictures to be produced Any other renderer could be used in place of these once it complies with one of the bindings in the specification

On the modelling side, modellers are also able to use the specification to output to RenderMan (and compatible renderers) There are a number of formats for outputting data, but they suffer from the problem of being constantly updated for proprietary reasons This is where the renderer is changed (usually to include extra features) and the modeller has to be changed in order to access these new features and to output this new format This is the RenderMan Interface's strength - it is a public document Introducing a new version is a large and lengthy event, which will only happen to introduce a number of not-insignificant features People want to use RenderMan because it is widely regarded as the 'best' off-the-shelf renderer available and its C-like shading language makes it incredibly flexible

In the past, RenderMan (and the REYES algorithm before it) has been used to produce the images used in many films and television productions This has caused more investment and research to be carried out, making it better and used more often and so on It is available in a number of forms, from IBM PC and Apple Macintosh versions, to UNIX workstations such as HP, Sun, IBM RS/6000, NeXT, SGI and DECstations It even comes as standard with the NeXTSTEP operating system This allows RenderMan to operate in a multi-platform environment even using distributed processing across the different platforms

With the tumbling price of hardware, the cost of using a number of cheap machines as a ‘farm’ for rendering is more appealing than having one super-fast machine A number of special effects/animation companies have recently invested in a large number of PowerPC-based Apple Macintoshes and multiple copies of MacRenderMan with NetRenderMan This allows the Macs to be networked together to render animations as a ‘back-end’ to whatever modelling and animation software is being used on developers machine at a minimal cost

This typifies what is probably one of the greatest impacts the RenderMan Interface has had - the separation of the front-end modelling process from the back-end rendering process Previously, this was all done at once on specialist hardware, which cost a lot and limited what facilities were available By separating these actions, greater flexibility and speed of production have been achieved The modelling process can be completed using a draft renderer (faster but less detail) and the user-interaction then finished The output of the modeller can then be taken to the renderer This could be on a different platform/machine at a different time

An example of this setup is where a number of users are using workstations to model an animated sequence They use the workstations to do the modelling and then when finishing up, they send the modeller output to a server which can work in the background on creating the sequence When workstations become inactive (idle) they too can be employed to work on the sequence in the background This speeds up the process and is seamless to the users The machines need not all be the same The workstations could be a combination of Apple Macintoshes, NeXTs and Silicon Graphics machines while the server could be a Sun When you consider that a one minute animation can contain over 900 individual images with some images taking 30 minutes to create, this is a considerable advantage

Another advantage of RenderMan is its shading language After the modelling phase, the surfaces of an object in the scene can be changed by switching shaders This is the same idea as using different sets of object libraries when linking a program This provides flexibility and allows for third party shaders to be sold

There are five types of shaders defined in the RenderMan Surface, Displacement, Interior, Exterior and Atmosphere The main two are Surface, which describe how to evaluate the colour of a ray of light hitting a surface, and Displacement shaders which allow points on the surface of the object to be perturbed (moved), as shown later in Chapter 5

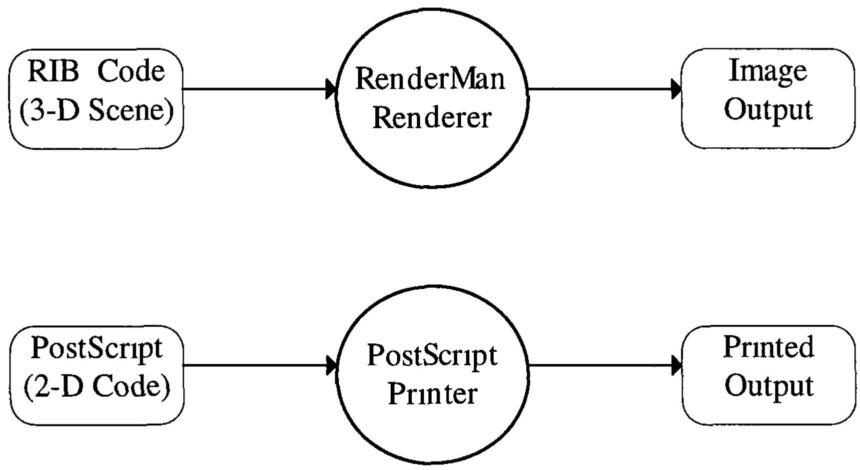


Figure 1-12 : RIB Processing vs. PostScript Processing

Another comparison can be made between RenderMan and Postscript In the same way that Postscript is a 2-D Page Description Language, the RenderMan Interface is a 3-D Scene Description Language RenderMan is not required to be known by the user of a 3-D modelling application, but when the scene is setup and a 'snapshot' taken (the description code is processed by the renderer), an image is produced on the computer in the same way that a page is printed once the PostScript code is processed by the printer The Postscript code is usually sent straight to a Postscript-compatible printer, which prints the page, but a RenderMan-Interface-compatible renderer usually creates a picture which is stored for viewing or printing later

The implementation of various theories in this thesis are based on 'intercepting' the RenderMan Interface Bytestream (RIB) output from the modeller and using it provide facilities - such as animation - without reverting to the modelling phase again

1.7 Graphical Terms explained

The vocabulary of the world of computer graphics is one that seems to change on a day-to-day basis This is a characteristic of the computing in general, given the rapid development of new areas, but in computer graphics it is difficult to distinguish between new fields and different ways of looking at more established fields This causes significant difficulty in the researching of previous work and indeed when trying to describe current work For example, the topic of morphing has been called a number of names in the past - “Shape Distortion”, “Fluid Objects”, “Warping”, “Deformable Models”, “Blended Surfaces”, “Topological Merging” and “Soft Objects”

The word “Morphing” is an abbreviation of the word “Metamorphosing” which is defined in the dictionary as “a change of physical form, structure or substance” In the computer graphics world, the two words can have separate meanings At Industrial Light and Magic, the custom has been developed to refer to two-dimensional changes with the term “morphing” and three-dimensional changes with “metamorphosing” When discussing both it is a useful way of differentiating between them, but it can be cumbersome when using them all the time In this thesis, when the word “morphing” is used, it is referring to three-dimensions not two, unless its specifically stated

It is important to note the difference between 2-D and 3-D Morphing (sometimes referred to as Image and Object Morphing respectively) Image Morphing has become extremely popular recently and it is a relatively simple operation in comparison to Object Morphing That is not to say that it is no good, in fact a lot of the morphing algorithms for 3-D have been derived from their 2-D counterparts A number of recent advertisements have used a motion-controlled camera to obtain two sets of images which are taken from the same viewpoint and then use image morphing to simulate a change between them This gives the appearance of 3-D morphing, but is actually a combination of special effects and 2-D morphing

Image Morphing works simply by taking one image (picture) and transforming it into another image There are a number of ways of doing this, the simplest being to linearly interpolate the colour of each pixel over the number of frames required for the transformation More imaginative image morphing is done by creating a control grid (mesh) over the original (source) picture and the final (target) picture By specifying where each point (node) on the source grid is to go on the target grid, effects such as one face transforming into another is given This would be done by specifying that the nodes around the features (eyes, nose, mouth, chin, ears and hairline) on the source move to the same features on the target

A certain amount of work has been done on this and a detailed paper on the making of the Michael Jackson video *Black or White* was presented to the SIGGRAPH'92 conference The video made extensive use of facial image morphing to change the faces between men and women and different races [BEIER92]

Chapter Two : History of Animation

2.1 Introduction

Since the earliest days of man on this planet, events have been visually described Cave paintings depict great battles and deeds - showing a sequence of pictures which relay a story When paper became available, we started to draw pictures on it, again showing a sequence of pictures to relate a story By showing a number of these pictures in short succession, the illusion of movement could be given Animation was born

Stick men could be animated by flicking between different pages and that is the basis for all animation - moving from one image to another image to simulate life/movement

More complex animation was achieved with the revolving carousel, called a zoetrope A cylinder with pictures on the inside had corresponding slits which allowed only one picture to be seen at a time When the carousel was spun, each picture would be visible for a small, but equal, amount of time When it was spinning fast enough, it gave the impression of seamless changes

The reason for this is that the human brain can only perceive about 25 different images every second A picture that changes more rapidly than that will no longer look like a series of individual images, but as a continuously changing image This is a principle that governs what we see - called the persistence of vision Any image updated slower than that loses its lifelikeness and becomes just another series of images This is how televisions and cinemas give the illusion of constant movement - although television actually works at a speed of 50 frames per second by displaying every second line of a picture every $\frac{1}{50}$th of a second and then displaying the alternate lines that it did not update in the next $\frac{1}{50}$th of a second and hence the entire picture is displayed in $\frac{1}{25}$th of a second using this method (called interlacing)

Cinema films are displayed at a rate of 72 frames per second, and this consists of the same image being displayed 3 times each (only 24 different frames are displayed) and a rotating disk blanking and un-blanking each actual frame 3 times every $\frac{1}{24}$th of a second

Animation has been studied specifically since 1824 when a paper on 'The Persistence of Vision with regard to Moving Objects' was presented to the Royal British Society From then on, until the turn of the century, there were a number of developments - including the zoetrope and phenakistoscope - which were the start of the film industry Eadweard Muybridge initiated his photographic collection of animals in motion (including humans) which is still used - and seen - today Thomas Edison created the kinetoscope which displayed 50 feet of film in 13 seconds George Eastman (founder of Kodak) created cellulose-based film and both the Lumière brothers and Thomas Armat (working on Edison's design) produced projectors which were the forerunners of today's projectors

The first animated film was created in 1906 and a number of animations were created following that Walt Disney's first breakthrough came when he produced the film *Alice's Wonderland* which combined live-action with cartoon characters, and again in 1937 when *Snow White* was released A year later, the first cartoon with synchronised sound was created It was called *Mickey Mouse*

Cartoons took over animation with hand-drawn 'funnies' until 1957 when John Whitney used mechanical devices to create analog computer graphics During the 60's, computers were starting to appear in numbers around the world By today's standards they were very slow and simple, but computer graphics was established as an area for further research In the 70's, many of the fundamental algorithms commonly used today in computer graphics were developed

In the 1980's, computers moved from large rooms in big institutions to the desktops of individuals at work and at home This brought computer graphics into the home since even the smallest and cheapest of computers could manage some form of graphics Computers like the Sinclair ZX81 managed black and white graphics with only 3 5K of RAM and this was followed by the Spectrum which brought colour computer graphics into the home (usually in the form of games) As the power of computers increased, graphics became less of an add-on and more of an essential Computers started being used in engineering, architecture, design and layout (DTP) New terms such as CAD and CAM appeared and computers began to be used for animation

Computer graphics were still always recognisable as being just that It couldn't be mistaken for anything else until photorealistic rendering was developed This meant that the computer generated image was indistinguishable from the real thing For this to happen, settings such as lighting, surface texture and shape must be exactly the same as in real life, Photorealism has only come about in the past 10 years or so, and has only really been available 'on the desktop' in the 90's But it has made great impact Incredible special effects and unbelievably lifelike animations have been generated recently, and the higher quality quickly becomes the norm

On television, in science programs and news reports, it is common to see a 3-D animation to explain/visualise some point or a geographic location Title sequences, advertisements as well as programmes and films usually contain some sort of animation In science-fiction, the use of photorealistic graphics has taken over, almost entirely from physical model based animation For example, *seaQuest DSV* - a $16 million Speilberg-backed show based on the adventures of a submarine - is modelled and created entirely on computer There is no submarine and no miniature model of the submarine

This has changed the way that film-makers approach special effects George Lucas recently said that he had to wait for the special effects technology to improve before he could make the sequel to the *Star Wars* trilogy made in the late 70's and early 80's He has now started work on this

2.2 Documented Research

2.2.1 Animation is well researched and documented

A large amount of research work has been done on animation, however high-quality object morphing is a 'new area' and research is only starting to be freely published since the initial work was mostly carried out for commercial reasons The amount of published research work available in these areas represents this, with research on morphing consisting mostly of commercial work (films, music videos and advertising)

2.2.2 Research on morphing tends to be very specific

Most papers on morphing are specific to particular objects formats or specific problems There are not many 'general' morphing algorithms, mostly because the algorithms tend to be dependant on the models It is possible to transfer the ideas from one model to another but it is not always successful For example, the Wyvills use objects called Soft Objects which have a variable 'field' projected around them which is their surface This is a quite radical method compared with conventional modelling tools and there is no clear-cut solution to transferring their ideas to work on common object types [WYVILL86][WYVILL89]

2.2.3 A lot of work on Facial Animation has been done

Facial Animation is an area where a significant amount of research and implementation has been done It is significant to this thesis not only because of its historical value, but because it is/was one of the main reasons that people started looking into morphing

When constructing an animation of someone's face, the main requirements are quality of detail and quality of expression A computer-generated face must look like a face and act like a face, but it must also convey the meaning of a facial expression For example, to give the impression of surprise or shock, a face's eyebrow would go up, but it must move like a human eyebrow, otherwise the animation won't be 'real' [WATERS87][MAGNEN89][REEVES90]

2.2.4 Collecting 3-D object data

This is a good time to look at how the model of a 3-D object is created in the computer There are no 'definite' methods of inputting a 3-D model and, almost always, the model will required tweaking to suit its purpose, but inputting data by hand is a very long and complex process so the function has been automated A number of strategies have been used when getting 3-D data about a subject

The simplest of these is to take photographs using cameras with tripods set up at different angles to the subject The subject will usually have a grid of numbered points drawn on its surface so that the points can be correlated later The cameras should have a long focal length so that perspective does not distort the size of the grid The photographs can then be scanned in and the points digitised (This is basically an automation of a process done by Inter Cert students in Mechanical Drawing - given a plan, front elevation and end elevation, to draw the object as viewed from a given angle in three dimensions) This was originally conceived by Parke in 1975 [PARKE75b] and an example of its implementation is given in an Australian University's technical report from which the face that appears in this thesis is generated [MARRIO92]

There are a number of drawbacks to this method These can come from aberrations of the camera lenses, incorrect digitising, improper physical setting of the camera height or angle as well as movement of the subject which changes the grid - something which can happen when modelling a human face The main drawback is the length of time and energy used to get the 3-D model Not surprisingly, faster and more accurate methods have been designed

The method preferred by most facial animators is to use the Cyberware 3D Scanner This is a laser-scanning digitizer which rotates around its subject emitting a vertical line of laser light Where this line hits the subject is measured by a camera which moves with the scanner yielding a 3-D mesh of points which covers about a 250° sweep of the object The quality of definition is better than is required for most subjects For human actors, the resolution has to be decreased so that the small lines and wrinkles on a face don't show up [SHAY87]

During the making of *The Abyss* a major scene involving about 70 seconds of photorealistic computer graphics was required The scene was that of an alien - a pseudopod consisting of seawater - which explores an underwater station and meets with some of the human inhabitants The final part of this had the creature taking the shape of a human face and responding to real actors in the scene The actors were scanned using the Cyberware scanner and this data was then used for the facial mimicking sequence

Since this had never been done before, the special effects company Industrial Light and Magic (ILM) adapted a method they had used before Image morphing had been pioneered in the film *Willow* in 1984 and the method they used for that was changed to work for 3-D data as well as adding in the rippling effect of water [ANDERS90] This was based on using the scientific principle developed independently by Schmitt, Barskey and Du that was presented at SIGGRAPH'86 [SCHMIT86]

It is not just the surface structure of real-life objects that can be captured Trying to animate an object so that it moves like something real is very difficult When Pixar were making *Tin Toy*, they had to watch about ten babies very closely over a period of time to ensure that they got the movements of their computer generated baby correct It is possible to capture the motion of a real object (usually a human) using movement sensors and a package such as SoftImage's *Channels* The sensors are strapped to the subject in a number of locations and provide telemetry on location, speed and direction These can then be applied to a computer generated model using Free Form Deformations (FFD) as mentioned in chapter four

2.3 Computer Animation has been implemented in various forms

2.3.1 First use of computer animation in feature films

During the early '80s, a full length feature called *Tron* was produced by Disney Studios This film combined computer animation with hand animation and live-action The computer animation techniques were quite basic - mostly Evans and Sutherland algorithms and Gouraud shading were used on the movie, which was state of the art then It took five months to make just 55 seconds of animation

2.3.2 European Work

While most of the research in computer animation is carried out in the United States, there are a growing number of 'pockets' working in the area around the world In Geneva, Daniel Thalmann and Nadia Magnenat-Thalmann have done a lot of pioneering work on computer generated 'synthetic actors' They are not only concerned with the quality of the images and photorealism of surfaces, but with the actions of the characters within their animations Their work, using models of Marilyn Monroe and Humphrey Bogart, is to model human characteristics and be able to reproduce them on demand - such as a woman blushing or a man drinking If completely successful, we would be faced with the possibility of movies without human actors, created entirely on computer While their worth is impressive - *Rendezvous À Montréal* won a number of awards - the perfect 'synthetic actor' is still sometime away [MAGNEN90]

London has a number of commercial companies working on 3-D graphics projects - mostly for the games market The power of games consoles is ever increasing and this allows very realistic images to be used Usually the images are rendered at full photorealistic quality and are then reduced to suit the power of a specific console, but the newer consoles can handle the highest quality images

2.3.3 The Growth of Special Effects in Movies and Videos

Companies are falling over themselves to produce special effects for movies RenderMan was used for the award-winning effects in *The Abyss*, *Terminator 2*, *Jurassic Park* and *The Mask* At ILM, most of their 3-D work is done using RenderMan at some stage RenderMan isn't the only rendering system being used, many different packages now exist and provide different cost-quality relationships suited to different customers A quick look at MTV will demonstrate the enormous use of computer graphics in the music video industry Different methods are used for different types of videos and music Some feature only computer graphics since there are no artists to star in the videos - the music is created by computer sampling in the first place

Recently, SGI have announced that they will co-produce a computer generated movie with Steven Spielberg's Dreamworks company for 1998 However, by then a number of computer generated movies will already have been produced

2.3.4 Pixar's new computer generated movie

Pixar recently finished work on the first feature-length computer-animated movie, called *Toy Story*, which was released in November 1995 in the US It was considered an qualified sucess and after three months, it had made $150m at the box office This figure is even before its European release in March 1996 It is a co-production with Disney and stars the voices of Tom Hanks and Tim Allen with songs by Randy Newman The story is of the adventures of two toys, a cowboy and a space superhero, who vie for the attention of their owner, a boy called Andy

The film is 78 minutes long and comprises 112,000 frames covering 1,700 scenes Over 1,000 gigabytes were needed to store the animation which took half a million hours to render (spread over a number of computers) It differs from all other movies, in the way that the computer animation is not used for special effects but for complete character animation

Pixar have been working with The Walt Disney Company since 1987, when they created a 2-D computer animation package called CAPS - Computer Animated Paint System This was first used in *The Little Mermaid,* and then in later Disney films *Beautwy and the Beast, The Lion King* and *Pocohantas* CAPS won a Technical Oscar in 1992 Pixar have won a number of Oscars, John Lasseter and Bill Reeves were nominated for Best Animated Short Film in 1986 for *Luxo Jr* and they won that category in 1988 with *Tin Toy* In 1993 nine of Pixar's staff were awarded Scientific and Technical Achievement Oscars for RenderMan "in recognition that computer animation had come of age"

2.3.5 Advertising

Advertisements too have been increasingly using computer animation to get their message across Computer animation allows washing powders to be seen working on dirty clothes at the microscopic level and bacteria being killed by detergents While the advertising message may not have changed, the method of conveying it has Using computer graphics, companies can make their products move, jump, sing and dance in order to get, and keep, the viewers attention They can also generate scenes that would be too difficult or expensive to replicate in real life One advertising agency created a commercial for washing powder that seemed to have a cast of hundreds and repeated the famous British Airways ad that involved an enormous number of people forming a face The washing powder ad was created mostly on computer and had a cast of twenty These were duplicated, to give the effect of hundreds and the objects they were supposedly carrying - socks and underwear - were super-imposed on top after being deformed to give the ripple effect of being carried

Morphing has also been used successfully a number of times in this field - there have been at least two car ads that used morphing (the one with the horse turning into a Volvo and the Nissan ad that runs through the various models) Advertisements, however, with smaller budgets and less output required, would normally use much-simpler 2D morphing (i e shoot a video of the cars turning with a motion-controlled camera and then interpolate the shots)

2.3.6 Desktop Animation

'Desktop animation' is the term given to the phenomenon of being able to create production quality animations on a fairly standard desktop computer This is a result of the explosion of the ability of hardware coupled with software packages such as 3D-Studio, Swivel Pro, TrueSpace, Simply 3D, Autodesk Animator and many others which are flooding onto the market, making animation available to a much bigger user base For a few hundred dollars now a system is available which would have cost ten's of thousands Output to video is very easy now with video boards being commonplace Even so-called 'home computers' are now used with to produce animations The Amiga, with the addition of a 'Video Toaster' (a set of IBM RISC chips) is as powerful as a lot of expensive animation systems and it is used in *seaQuest DSV* and *Babylon 5* where little or no difference can be seen from physical models

Recently, a new development means that even a standard PC can now be turned into a video editing machine with the installation of a PAR (Personal Animation Recorder) card This consists of video compression and decompression hardware and a very large and fast hard disk The hard disk is formatted so that each sector is the exact size of a frame, usually stored in TGA (Targa) format The compression hardware works in real-time so it is possible to record and playback video directly from the hard disk The only limit on the length of the video segment is the size of the hard disk The PAR hard disk appears as just another drive on the computer to other applications so they can input and output to video easily

2.4 Copyright and Ownership Problems

There is still a lot of discussion as to the legal problems that arise with using a standard format such as RIB The output of a program belongs to the owner so the RIB file is copyright the owner of the package However, since RIB files are usually hierarchically ordered, parts of the hierarchy can be 'grabbed' and used in other hierarchies And with a few minor changes the file is no longer the same one that is copyrighted While this has, for the most part, been solved in the case of source code for programs using the intellectual property laws, RIB files may not fall under this category since they are usually created by modelling programs which can add a significant amount to the work and may therefore not be seen as property of the owner of the modelling program

The net result of all this is that most commercial organisations refuse to publicly distribute RIB files When it comes to demonstrations, they tend to distribute images rather than a 3-D scene, which would allow full 'interactivity' for prospective customers While this is only really a problem in the 3-D graphics community right now, it may grow as the technology appears that allows a more interactive relationship with commercial companies The Geomview group at the University of Minnesota have developed a 3-D browser tool called Cyberview-X which will allow 3-D manipulation of objects over the World-Wide Web Recently, another product called WebSpace has been released which aims to be a Virtual Reality Web Browser This has the backing of SGI and is being released on a number of platforms This will have the same problem as RIB in that 3-D data will have to be made freely available from Internet sites The VRML - Virtual Reality Manipulation Language - file format used by WebSpace is a cut-down version of the SGI Inventor format which only allows polygonal shapes While this is adequate for simple objects, it does mean that complex objects require more polygons and hence VRML files tend to be very large - a drawback when the entire file has to be downloaded The use of only polygons does limit the scope for using VRML, and is not very popular with companies within the graphics area

A similar problem existed with PostScript where 'font-hackers' were able to take special fonts from a PostScript file and re-use them at will Companies that produce special fonts were caught out by this and lost business as a result Shortly after this, the Encapsulated PostScript format appeared What this does is embed the graphic bitmap of the font into the PostScript document And since its only a bitmap, the font cannot be smoothly scaled for different typefaces, thereby making it useless to a potential font hacker

It has been suggested that some form of Encapsulated RIB format should be created, but it is difficult to see how it would be implemented as an open interface One solution for these 'demo' scenes to be freely available would be to have a rendering/browsing application available which has private key decryption built-in The RIB files could then be encrypted by public key by anyone and made available The decrypted RIB file would only exist within the application, allowing the authors to control access to the RIB file without hindering the viewing of the scene

2.5 A word or two about Visualization

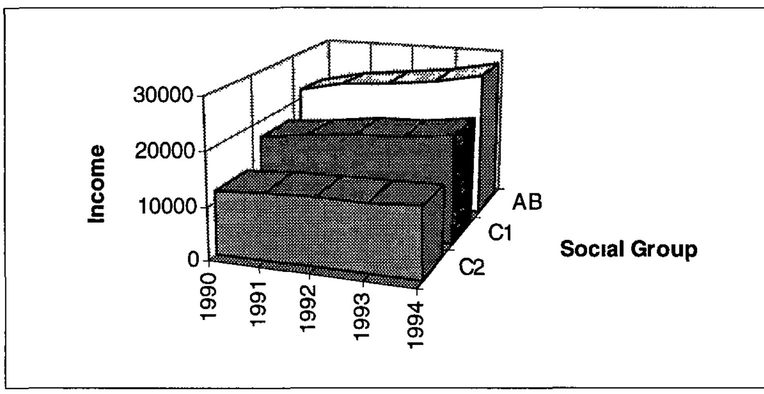
Visualization has become a new buzzword in the computer graphics field recently, however, it has been around since computer graphics were first produced - and even before then What has happened recently though, is that the hardware and software available have improved dramatically and the cost of these has also fallen dramatically Visualization is basically where some concept or situation which is difficult to describe is presented in a visual format to aide and ease understanding Scientific Visualization is where some data is represented visually to demonstrate the inter-relationships of the data and (preferably) the relationship that the data has to the context from which it was created

Until recently the visual representation of data would have been quite limited due to the power of the computer that it is running on, however, computer processing power and abilities have increased dramatically over the past few years in terms of graphics capabilities Where a simulation which would have been created on the most powerful computers in the world over a number of weeks in the past, this simulation can now be done in a matter of hours using commonly available equipment and applications

Historically, visualization cannot be said to have been invented, but it grew out of peoples need to use their eyes to interpret some problem or give them some insight into something they didn't understand The earliest examples of this could be said to be astronomy and cartography In 1603 an amateur astronomer called Johann Beyer printed the first modern set of star charts - '*Uranometria*' - which were actually based on the observations of a Danish astronomer Tycho Brahe which were particularly accurate Edmund Halley (as in Halley's Comet) published the first meteorological chart in 1686 and undertook the first ever purely-scientific-research-sea-voyage to the South Atlantic to take magnetic compass readings which he translated into the first magnetic charts and published them in 1701 National surveys of England, France and Switzerland were taken during the mid-18th century and from these maps were drawn showing the height of land in places using contour-lines, then different shades of grey and eventually colour These contour lines allowed the slopes of hills and mountains to be visualised [COLLIN93]

To some people the entire purpose of computer graphics is visualization, but it can be equally argued that visualization is an application of computer graphics This is a circular argument given that the two are so heavily inter-linked Visualization is dependant on the quality and abilities of computer graphics, but it is not the entire process - the question of which data should be displayed It is a matter of interpretation and is very important, quite often statistical methods determine the final outcome

The power of visualisation is the speed and clarity that it provides when it comes to understanding numerical data In this case, the small tables are not too difficult to interpret but imagine if the data was going back fifteen years not five and there were six or seven social groups instead of three - the task of interpreting them would be much more difficult, yet a graph would still be able to show the trends and differences from over the fifteen years



Graph 2-1 : Simple Visualisation of tabular data

Income for Social Groups over 1990-1994					
Social Group	1990	1991	1992	1993	1994
C2	12000	12500	12750	13000	13500
C1	17000	18000	19000	19500	21000
AB	22000	24000	25000	26000	28000

Table 2-1 : Data for Graph 2-1

2.6 Traditional Techniques are still relevant

"Its the spectators that make the pictures" - Marcel Duchamp

While much of Computer animation is related to mathematical functions, recursive sub-division and massive amounts of processing power, there is also the other side of it all to be considered - what does the end-product represent ? How will the viewer react to the animation ? These questions are not new ones - they have been considered in many animations since 1920's and 1930's Modern animators can use the lessons learned back then by applying some of the old techniques to new computer animations

The purpose of an animation can be varied - it can be to entertain, inform or enlighten Computer animation is still relatively new and there are no pre-defined methods established - almost every major production has had a paper discussing the production presented to scientific journals or conferences since there is usually some aspect of the production that is new or unusual While 3-D animation is new, it is not without any foundations - many of the ideas that are/were used in 2-D animation (also called Cel animation) hold true in some form for modern 3-D animation

John Lasseter, who is one of the best-known computer animators, analysed the traditional methods of animation and pointed out how they can be applied to computer animation in a paper presented at SIGGRAPH'87 In particular, he pointed out how these principles had been used in the design and production of *Luxo Jr* and *The Adventures of Andre and Wally B* - award-winning computer animations produced at Pixar [LASSET87]

The principles were taken from a book that is widely regarded as the 'bible' of animation - *Disney Animation - The Illusion of Life* by Thomas and Johnston [THOMAS81] This describes animation as practised by Disney Studios and is almost a training manual for Disney animators They identified twelve principles that should be used in the preparation of an animation

2.7 The Twelve Principles of Animation

1 Squash and Stretch - Defining the rigidity and mass of an object by distorting its shape during an action

2 Timing - Spacing actions to define the weight and size of objects and the personality of characters

3 Anticipation - The preparation for an action

4 Staging - Presenting an idea so that it is unmistakably clear and keeping the audience focused on that

5 Follow Through and Overlapping Action - The termination of an action and establishing its relationship to the next action

6 Straight Ahead Action and Pose-To-Pose Action - The two contrasting approaches to the creation of movement

7 Slow In and Out - The spacing of the in-between frames to achieve subtlety of timing and movement

8 Arcs - The visual path of action for natural movement

9 Exaggeration - Accentuating the essence of an idea via the design and the action

10 Secondary action -The action of an object resulting from another action

11 Appeal - Creating a design or an action that the audience enjoys watching

12 Solid drawing - Areas should be drawn/shaded equally by hand (this does not really apply to computer animation)

These principles which have been worked on over the past seventy years provide a guide for anyone creating an animation today Staging is probably the most important of the principles since it is concerned with directing the viewers attention A simple stick animation can be more effective than an animation with the highest quality of pictures if it is staged correctly Following the principles does not guarantee an animation success, but it should ensure that its message is communicated to the viewer

Chapter Three : Animation

The word 'animation' comes from both Roman (*anima*) and Greek (*animos*) both relating to the bringing to life of something Animation is - literally - to bring to life Probably the singly most important fact about life is that it is changing over time - if something doesn't change over time it is not alive In this thesis, the concern is more with the simulation of life rather than the more medical aspect of animating things The type of animation that is of concern here is where a sequence of images are shown in succession giving an observer the impression of movement These images can be sets of dots, lines, shaded drawings or the most detailed of photographs These images will each be similar, but will contain slight changes from one to another

While current techniques are quite different from the past, the actual process involved in creating an animation has remained mostly the same The older methods of animation are still valid today whether using modern technology or not - which just stresses a fundamental of animation Its the story that you are telling that matters How you do the animation is secondary to that

In computer generated animation, interest is centred on the changes that occur between frames and how to create and control them For computer generated animation, there are generally considered to be three types of animation

1) Image-based Keyframe Animation (point-based)

2) Parametric Keyframe Animation

3) Algorithmic Animation

[THALMA89]

It is also important to look at different methods of animation to gain an insight into what sort of actions and where computer animation can be used to improve it

3.1 Frame-by-Frame

To create an animation requires each frame to contain a slight change from the previous frame In traditional animation, this is called stop-frame animation Examples of this would be *The Wombles, Thunderbirds* and Nick Park's Oscar-winning *The Wrong Trousers* These were made by physically creating the situation using models and modelling clay and then photographed A small change to the situation (including movement of the camera) is made and then another photograph is taken This will be repeated for up to 25 times per second of film-time This obviously is a lengthy process and requires that the models be made out of some material that is suitably malleable and yet stable enough to allow it to hold its shape and position after minute changes have been made

3.1.1 Modern Stop-Frame animation

For the most part, stop-frame animation is no longer the most popular method of animation, although recently Nick Park's work and Tim Burton's film *The Nightmare before Christmas* have caused stop-frame animation to be given new attention In the case of *The Nightmare before Christmas*, computer technology was used in the opposite way it is usually used in computer animation

The main character 'Jack' had over 1800 heads with different facial expressions, which was too much to be able to decide which heads to use to change an expression A computer was used to choose the appropriate heads for a facial change given the initial and final heads, and provided the director with a number of different ways to get from initial to the final head The computer could then tell the director exactly which of the 1800 heads to use and in what order

Stop-frame animation is the physical alternative to what is done with RenderMan, and the animation could have been done with this, involving scanning in the heads in three-dimensions and then morphing the initial head into the final head, but the director chose the physical way

3.2 Keyframes and Interpolation

3.2.1 What are keyframes ?

When an animation is developed, it is usually from a storyboard This is a sequence of rough drawings of what will happen at different points in time during the animation It gives a guide to what the camera should be looking at over these times From these, the scenes can be set up and the views from the storyboards created These will be the *key*frames - i e they are the most important More and more keyframes are developed until the difference between frames is trivial (e g a simple movement) Then the frames *in-between* can be created by a sub-ordinate or computer

This is a simple mathematical process that is quite vital when it comes to animation, it is usually referred to as *tweening* when used in context with animation The word '*tweening*' is a shortened version of 'in-betweening' which refers to the process of creating the frames that come <u>in-between</u> keyframes To interpolate, according to Webster, means to insert between other things or parts, or to estimate values of a function between two known values

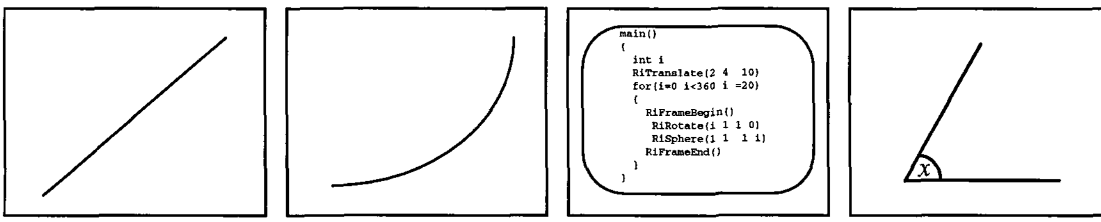
In computer animation, keyframes are used to specify starting and ending pictures/scenes The in-betweens are then worked out by the computer Different methods of interpolation can be used to create different effects, both in 2-D and 3-D

3.2.2 Different methods of interpolating between keyframes

Interpolation is where values in the range between two known values are estimated Counting to ten is an interpolation between zero and ten In animation, interpolation is where values are obtained for a parameter with a range between two known values

For example, given the starting and finishing locations for an object and a requirement to produce a one-second animation of its movement, values for 25 frames would be required The starting and finishing positions would be used for the first and last frames, so 23 frames would have to be made which interpolate positions between the starting and finishing locations

There are a number of approaches to this estimation process The choice of one approach over another will determine the complexity and flexibility of the animation



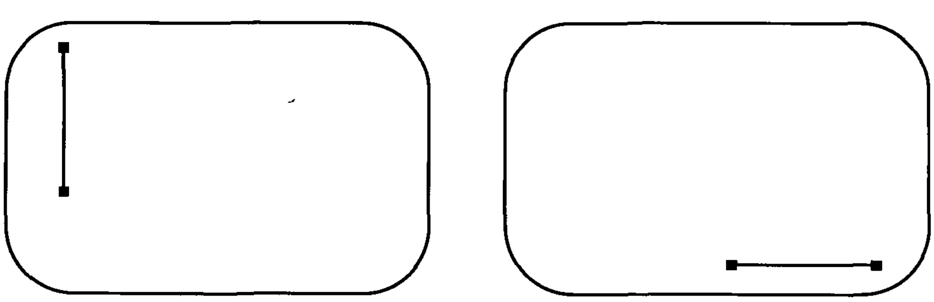
linear spline procedural parametric

Figure 3-1 : Different types of interpolation

<u>Linear Interpolation</u>

Linear interpolation is where the points between the two known values are equi-distant This will cause objects to move at the same speed per frame This may not be very life-like, but it can work well over short intervals and can give a good idea of what is moving and where it is going in a scene

Linear interpolation is the simplest (although by no means the only) way to interpolate and it is called *linear* because the solutions are all on a *line* A good example of this is to take two simple 2-D frames, take the starting keyframe as the horizontal line in frame one, and the ending keyframe to be a vertical line as in frame two



Frame 1 Frame 2

Figure 3-2 : Simple 2-D Linear Interpolation Keyframes

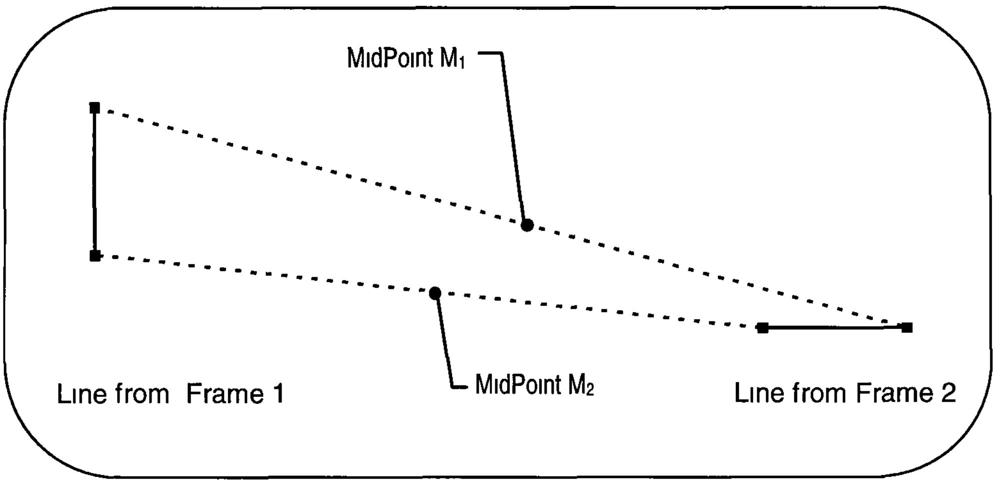


Figure 3-3 : Midpoints of Interpolation between lines

Using linear interpolation, the two endpoints of the line segment take a straight path from frame one to two, as shown by the dotted lines in Figure 3-3 Halfway between the two, joining the midpoints of these paths (M_1 and M_2), is the line that is produced halfway through the interpolation It is a diagonal line which is oriented at precisely 45° to both of the lines This assumes that we only want three frames in the animation (Frame 1 at the start, followed by the interpolated frame and finishing with Frame 2), but in most cases more than one interpolated frame would be required

In order to create the other frames, the above procedure can be repeated by sub-dividing between the interpolated line and the other lines on either side of it and getting the line joined by their midpoints and so on until the required number of frames have been created However, this does have a number of problems

This recursive-binary-linear-interpolation has the property of being very easy to implement, however it can lead to discrepancies - when not requiring a power-of-two number of frames (i e 1,2,4,8,16) to be interpolated the animation will seem uneven with the line moving further in some frames than in others

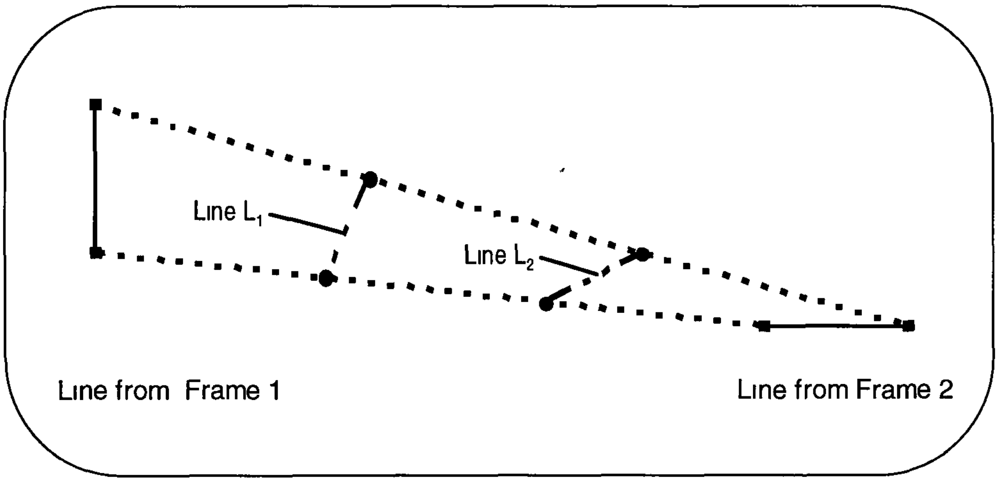


Figure 3-4 : Interpolation for Four-Frame Animation

A way of improving this method is to divide the path between the two lines to give equi-distant points, one point for each of the frames that is to be interpolated instead of recursively sub-dividing For example, if a total animation of four frames is required, the starting and ending keyframes form two of the frames so only two more frames are required to be interpolated This requires that the paths between the lines be divided into *three* equal segments as in Figure 3-4 Note that the number of segments is one less than the number of frames required

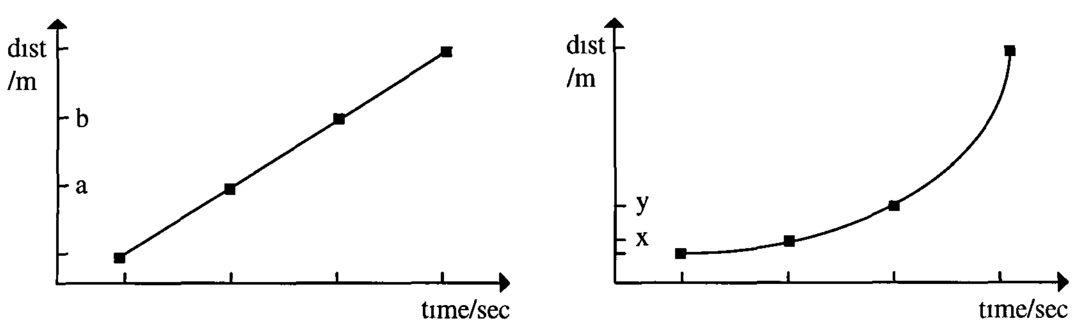
It should be noted that the lines in Figure 3-4 are not all the same length The interpolation method outlined does not preserve the lengths of non-parallel lines There are other interpolation methods which allow greater flexibility and control of the interpolation process, such as spline interpolation

Spline Interpolation

Interpolation using splines allows non-linear movement Splines are curves that provide a smooth non-linear method to move objects over a number of frames This allows much more natural movement to be created Phenomenon like acceleration and deceleration can be easily represented and this usually suffices for most types of movement

Splines are very important in computer graphics, not just for interpolation, but for modelling too It is important to understand some of the fundamentals of spline curves because they are useful at so many different levels

3.3 Linear vs. Spline Interpolation



Linear Interpolation　　　Spline Interpolation

Figure 3-5 : Comparison of Linear and Spline Interpolation

Spline Interpolation is necessary because objects do not generally observe a linear motion For example, when a car accelerates, it does so in a non-linear fashion If a linear interpolation method was being used, then the resulting (four frame) animation could look like this

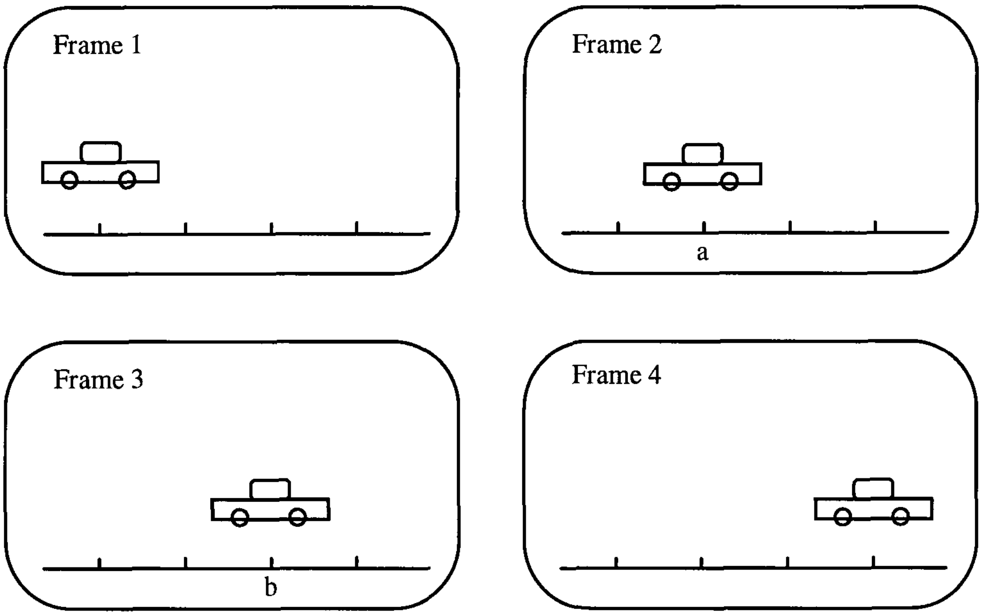


Figure 3-6 : Linear Interpolation of a car moving from rest

Using the spline interpolation method, the car can exhibit more natural behaviour such as constant acceleration shown in Figure 3-7 rather than the constant speed displayed using linear interpolation shown in Figure 3-6

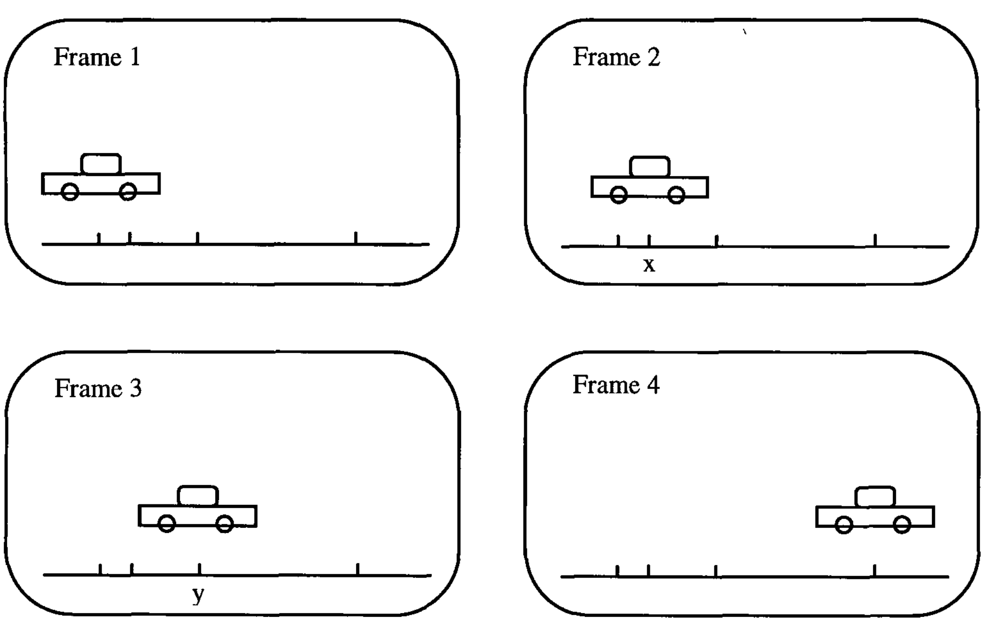


Figure 3-7 : Spline Interpolation of a car moving from rest

3.4 A word about Splines...

Splines were originally used by draughtsmen to draw smooth curves A flexible piece of metal had weights (called ducks) attached at various intervals which bent the metal, providing a repeatable process which gave a smooth curve The term *spline* was applied to the mathematical version during the Second World War when aeroplane blueprints replaced models which were liable to damage during transit

Splines are parametric representations of curves Parametric representations are desirable in computer graphics due to their ability to represent a surface using discrete points rather than an implicit representation which would require defining and solving quadratic, cubic and non-linear equations The use of parametric representation is therefore much more flexible (and stable) than implicit representation and it tends to be used for most complex surfaces in computer graphics

A parametric curve is usually some form of polynomial A polynomial of degree *k+1* can be written as $Q(u) = p_0 + p_1 u + p_2 u^2 + \dots + p_k u^k$

It would be difficult to manipulate the coefficients p_i in the above equation to represent the curve, so the polynomial form is re-arranged into control points and basis functions to provide a more 'human' approach to forming the curve The basis functions are independent polynomials termed as $b_i(u) = u^i \quad 0 \le i \le k$

With the co-ordinates p_i called control points, $Q(u)$ can now be defined as

$$Q(u) = \sum_{i=0}^{k} p_i b_i(u)$$

This gives a curve which can be manipulated by changing the location of the control points The basis functions are also important in defining the spline If the basis functions are non-negative and sum to 1, then the spline curve will be within the bounds of the control polygon made from joining the control points because any point on the curve will be a weighted average of the control points This is a very useful property for computer graphics rendering, allowing bounds to be checked without actually calculating the curve

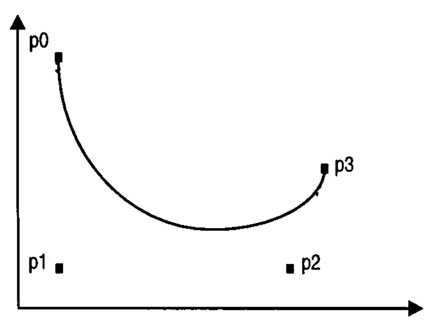


Figure 3-8 : A Bézier curve

In computer graphics, the most common type of spline is a Bézier curve as show in Figure 3-8 The curve is considered to be cubic ($k = 3$) because it is defined by three line segments These are in turn defined by four control points (p0,p1,p2,p3) When the control points are connected together the shape that results is called the control hull as illustrated with the doted lines

For Bézier curves, the basis functions are represented by $Q(u) = \sum_{i=0}^{3} p_i B_{i,3}(u)$

The basis functions $B_{i,3}(u)$ are shown in Figure 3-9

The functions are

$$B_{0,3}(u) = (1-u)^3$$
$$B_{1,3}(u) = 3u(1-u)^2$$
$$B_{2,3}(u) = 3u^2(1-u)$$
$$B_{3,3}(u) = u^3$$

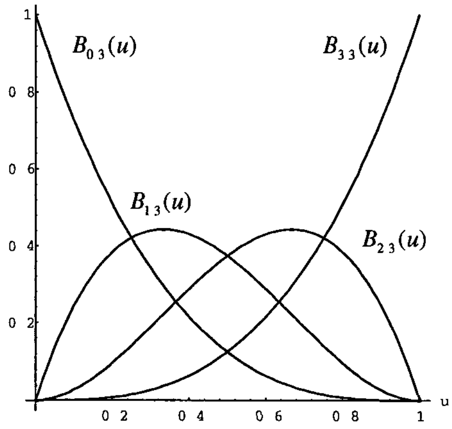


Figure 3-9 : The Bézier basis functions

What these functions do is determine the amount of 'influence' that a control point has on the curve From Figure 3-9, it can be seen that the first control point will have complete control over the start of the curve since $B_{0,3}(u)$ is at 1 and all the other functions are a 0 As u approaches 1, the influence of the first control point goes to 0 Likewise the second control point has most influence at the peak of $B_{1,3}(u)$, around $u = 1/3$, the third control point has most influence around $u = 2/3$ and the last control point influences the end of the curve The basis functions of a Bézier curve cause all control points to have some (even if minimal) effect on the curve at every point Because of this, they are sometimes referred to as blending functions

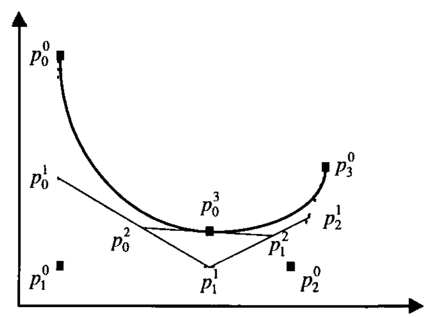


Figure 3-10 : The de Casteljau representation of a Bézier curve

One of the advantages of using Bézier curves in computer graphics is their ability to be implemented using a recursive linear interpolation algorithm - the de Casteljau representation Linear interpolation is quite simple on a computer and this allows the curve to be drawn quickly with a variable level of detail depending on the required quality/speed trade-off

The de Casteljau representation generates points on the curve by repeating a linear interpolation The control points of the curve are $p_0, p_1, \ldots, p_n$ and the curve can be defined recursively as

$$p_i^r(u) = (1-u)p_i^{r-1}(u) + up_{i+1}^{r-1}(u) \quad \text{where} \quad r = 1, \ldots, n \quad i = 0, \ldots, n-r \quad p_i^0(u) = p_i$$

In Figure 3-10 above, the point p_0^3 is calculated for $u = 0.6$

A point on the curve is now given by $p_0^n(u)$ where u is between 0 and 1 A sequence of points on the curve can now be obtained by evaluating u at a range of values This is the ratio for division of the lines formed by the control points Recursion brings it down to a single line segment which is then sub-divided and the resulting point is a point on the curve By evaluating u at an appropriate step size, the points will form a continuous curve

For example, in Figure 3-10 the three line segments made by connecting the control points is divided in the ratio for sub-division of 0 6 This yields three points on those line segments which, when connected, form two line segments When these are sub-divided, and the points connected and sub-divided again, will give a single point on the curve By repeating this process for $u = 0,0 1, \ldots ,0 9,1$ a total of 11 points on the curve will be calculated

Bézier splines are not the only type of splines Other types include B-splines, Beta(ß) Splines, Catmull-Rom and Hermite splines Of these, B-splines and Catmull-Rom splines are probably used the most in computer graphics It is possible to convert between these different forms of spline, using matrices Once the different forms are expressed in matrix form, it is relatively simple to convert between spline curve types using matrix multiplication While the modelling and animation may be done with any of these types, they are usually converted to Bézier form at the rendering stage because it has useful properties (such as the convex hull test) which allow for better efficiency in a renderer

Bézier splines are known as approximating splines because only the first and last control points are on the curve - all the other points are approximated The opposite of this is an interpolating spline, like a Catmull-Rom spline This is where the spline curve intersects all of the control points except the first and last points This is very useful in computer graphics for controlling animations Specific points, called waypoints, can be set so that the curve (and hence any parameter the curve controls) will intersect that point, causing that parameter to have a specific value at a specific time

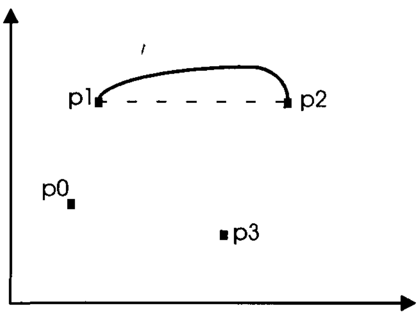


Figure 3-11 : A Catmull-Rom Interpolating Spline

The first and last points are important for specifying the tension and bias of the spline at the interpolated points A Catmull-Rom curve is constructed by making the curve at p_n parallel to a line drawn between p_{n-1} and p_{n+1} The lines adjoining p_n can be thought of as vectors - their scalar values denote tension and direction values denote bias Tension defines the 'sharpness' of a curve and bias affects where p_n is on the curve

An example of how tension and bias are used to control a Catmull-Rom spline which is being used to control parameters over time can be seen in the example of parameter tracking shown in Figure 3-13

The third main type of splme used in computer graphics is the B-spline They are a lot more complex and more powerful than either Bézier or Catmull-Rom splines and have a number of useful qualities for modelling One of the most important of these features is that not all control points influence the curve at every point At each parameter evaluation - called a 'knot' (in Figure 3-10 the knots were at $u = 0,01, \ ,09,1$), the control points required to influence the spline can be specified The knot interval need not even be uniform - the spline could be evaluated at $u = 0\ 35,0.51,0\,88$ In this case the spline is said to be non-uniform These are the most powerful of spline modelling tools and are called NURBS - Non Uniform Rational B-Splines A full explanation of B-splines and NURBS is not necessary for the scope of this thesis, but they will be used later when implementing morphing, where they will be used to approximate the surface of quadratic objects such as spheres and hyperboloids

[CATMUL74][TILLER83][BARSKEY87][PEIGL85][PEIGL87][FARIN90] [UPSTILL90][VINCE92][WATT92]

3.5 Procedural Interpolation

Procedural interpolation (also known as Algorithmic interpolation) is where a sequence of commands is used to interpolate between values Situations where procedural interpolation is suited are ones where complex control is required For example, in an animation of a car driving along a road with bumps, the wheel of the car must be kept in contact with the road Some algorithm must calculate the rotation and movement of the wheel given the roads details and speed of the car, so that the wheel moves to keep in contact with the road and it rotates to cover the distance that the car has travelled in a frame In this example, it is obviously not just a matter of interpolating between two points, but this too could be controlled algorithmically

The ability to use recursion or external routines makes procedural animation very useful When used as in the above example, it is not simply a matter of simple interpolation, but one of control of a number of objects There are a number of graphical languages and animation scripting systems that allow this sort of procedural control In order to control objects in a scene, there must be some sort of 'higher knowledge' about the objects and how they relate to each other This 'knowledge' is usually in the form of how far and in what directions an object can move, what objects it effects and in what way does it effect them This is still quite high-level in terms of control, so the use of parameters is introduced so that low-level transformations and rotations can be referred to as a simple parameter

For example, to describe an object in orbit around another static object, its exact location could be used Instead the situation can be described using two parameters - the distance from the static object and the angle of rotation These two parameters provide a much more useful form of control and allow some understanding of the behaviour of the objects Procedural animation usually will be implemented with objects controlled by parameters rather than low-level commands, and is commonly used as a form of control

3.6 Parametric Control

Parametric control allows objects to be controlled in a manner consistent with what the objects represent rather than how they are modelled When an object is modelled, 3-D building-block objects are used, such as spheres, cylinders, cones, polygons and patchmeshes While these may represent an object, they do not actually describe its behaviour Parameters are used to describe the behaviour of an object

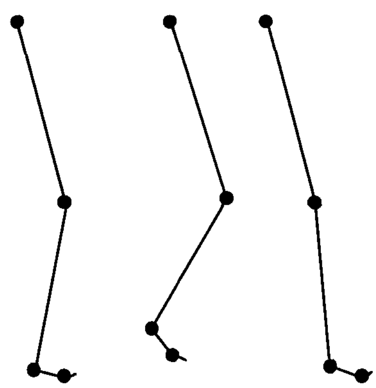


Figure 3-12 : Parameters for representing the joints on a leg

Parameters are usually used at the lowest level of abstraction from the modelling specification This allows maximum control without resorting to the model and hence the details that it entails An excellent example of this is when animating the movement of the human leg By using parameters for the joints at the top of the leg, the knee, the ankle and the toes, all aspects of the movement can be animated The parameters themselves can be controlled by linear, spline or algorithmical interpolation [BURTNY76]

It is obvious that having a parameter for each individual joint allows great control, however it should be noted that not all parameters are completely independent of each other For example, in Figure 3-12 some part of the 'foot' should always be in contact with the 'ground', which is not true in the case of the middle leg These issues have led to a lot of effort being put into areas like kinematics which allow inter-dependent control of complex objects

Having to control a large number of individual parameters can be confusing for an animator and it is here that an object-oriented approach can be applied This would allow complex commands such as 'track' (rotate in order to face another object) to be implemented without the individual parameters having to be specified

3.7 Kinematics

Kinematics is the term given to the area of study of motion independent of the force(s) that produced the motion It is concerned with movement and energy and how objects that are linked will react Kinematics is used for modelling the movement of articulated objects such as the human body and other complex objects that have a certain freedom of movement while remaining connected to other objects Usually, this is represented using state vectors with each element of the vector representing a degree of freedom (DOF)

A DOF is an independent position variable which specifies the state of a structure The number of DOF required is the number of independent variables required to completely describe the position of the structure - there is usually one for every 'joint' For a completely free object, there are six degrees of freedom - three for translation and three for rotation For a more constrained object there will be less DOF but there will be a minimum of one - otherwise, the object will not be independent and will just be considered part of another object

Forward and Inverse Kinematics

Forward kinematics is where the movements of all joints - all position variables - are specified explicitly While this may seem cumbersome, it does allow motion to accumulate so that movements of are implicitly calculated For example, the transformation applied to a foot is the accumulation of the transformations applied to the hip, knee and ankle

Inverse kinematics is where the movements of joints are computed after the required end-movement ("put the hand on the table") is specified This form of goal-directed motion allows relatively high-level commands to create a sequence of movements The movements required to carry this operation out are calculated by the computer Hence it is a matter of working backwards

3.8 Tracking

When a number of parameters are being used, it can be difficult to keep track of their values from frame to frame If a parameter has a number of key values (derived from keyframes) then it is important that the changes in certain related parameters are kept in synchronisation A simple graphical way of representing the parameter values over time is called tracking This is where the values are graphed side by side, representing the changes over time

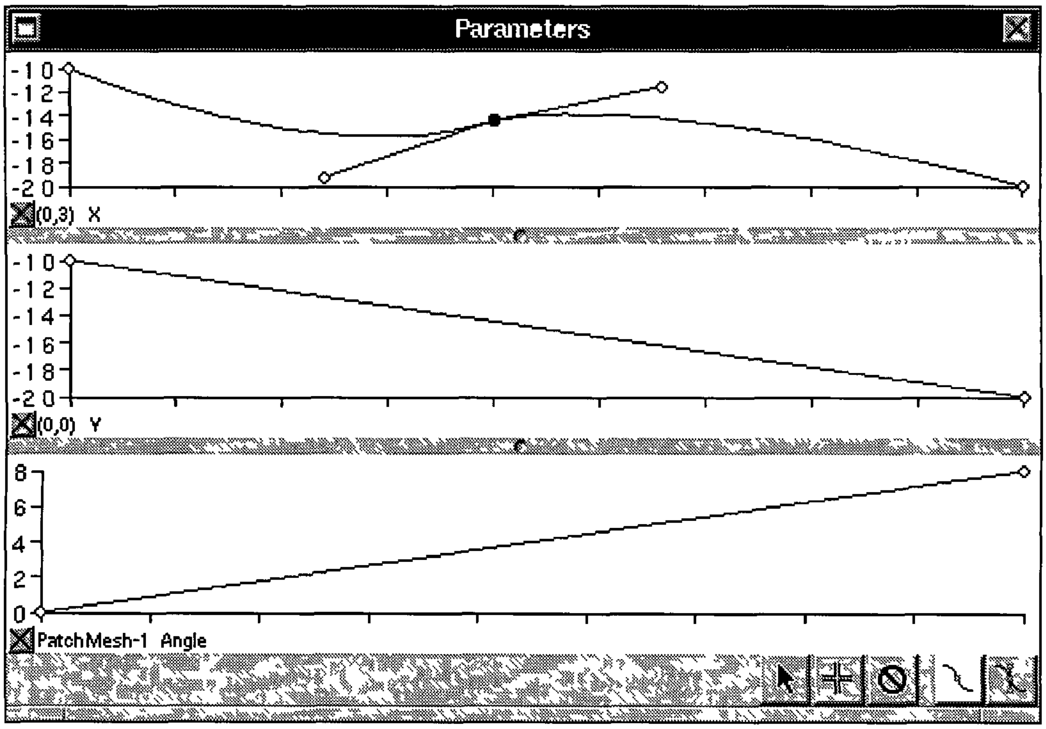


Figure 3-13 : Parameter tracking using spline interpolation

In Figure 3-13 the key values are represented with white circles at the start and end of the ten-frame animation The top parameter is the X-axis component of a splines control point (referred to as *(0,3) X* above) This has an additional key value as represented by the black circle The points indicated by the extra lines around this key point control the tension and bias of the Catmull-Rom spline interpolation Since both of the lines are relatively short, the curve of the spline is smooth The direction of the lines causes the curve to dip before the key point and to rise slightly just after the key point

Chapter Four : Morphing

4.1 Introduction

"As Gregor Samsa awoke one morning from uneasy dreams he found himself transformed in his bed into a gigantic insect" - Franz Kafka, Die Verwandlung

This quotation from *Die Verwandlung (Metamorphosis)* has been the topic of many a philosophical discussion and poses the question of whether things are what they seem A metamorphosis is where a transformation between two states occurs and the idea of showing such a transformation of something on film is not a new one In early horror films such as *The Wolfman* and *Dr Jekyll and Mr Hyde* transformations had a very important role Though implemented with effects that would be considered quite basic by today's standards, they instilled a great sense of fear where a man turns into a monster This theme is timeless - a direct modern day version of this is the cyborg from the future in *Terminator 2* which can impersonate anyone or anything

This chapter is concerned with the three-dimensional metamorphosis of graphics objects, which is called morphing It can be difficult to separate between animating an object and morphing an object - especially in cases such as kinematics However, there is a difference Morphing is where the object concerned is having its surface structure changed in some way, whereas animation is where the relationship with other objects is changing Morphing and animating an object are not mutually exclusive operations, an object's structure can be changing while it is interacting with other objects A specific case where the object's interaction causes its surface to change is called deformation

In much the same way that keyframes in an animation specify the starting and finishing locations and orientations of objects, an object that is being morphed usually has two states its beginning state (source) and its final state (target) The goal is then to transform the source object into the target object while fulfilling any criteria required by the context This usually entails the object keeping some cohesive form during the transformation

The process of morphing the source object to the target object can usually be divided into two steps the correspondence or mapping step and the interpolation step

The mapping step is where the structure of the source object are mapped onto the structure of the target object This is probably the more important step of the two since the effectiveness of the interpolation step will depend entirely on the mapping correspondences created during this step In the simplest situation, where the objects are defined using equal-sized sets of polygons, the mapping step would establish a one-to-one relationship between each individual point

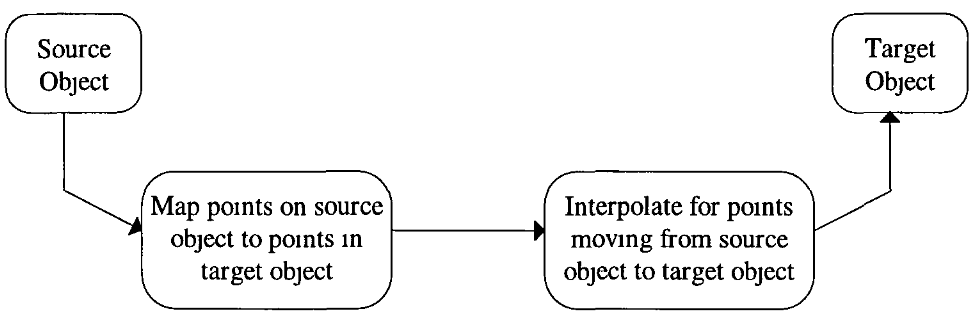


Figure 4-1 : Morphing as a two-step process

The second step is that of interpolating between the source and target objects using the correspondences established in the previous step There are a number of methods of interpolation as outlined in the previous chapter, and the principles used to make a convincing animation can be used here When talking about interpolation, the most important principle would be 'slow in and out' During morphing, the object will usually change from a recognisable source object to an unrecognisable 'in-transit' object and then to a recognisable target object Since the 'in-transit' object can be unpredictable, the most important parts of the morphing scene are the initial frames as the object starts to morph, and the final frames as the morph ends Usually, the interpolation will not be linear and will speed up or slow down depending on how close it is to start or end of the transformation

A number of strategies have been developed for morphing They are mostly based on different proprietary formats for objects which the designers are working with So while there are various strategies that can be used, they tend to suit a particular type of object, which makes it more difficult to implement with other types of objects This is one reason why RenderMan can be so valuable in this area - it is a standard format encompassing a large number of methods for defining objects

4.2 Topological approach

The topological approach is where the object is considered in terms of its surface An example of this could be where a sphere is represented with hundreds of small polygons Implicit surfaces are based on the idea of an object with a skeleton frame which has a field surrounding it forming its surface An example of an implicit surface could be a sphere defined by its location and radius - no points on the surface are actually used (and all objects in computer graphics are, by default, hollow) Implicit surfaces will be examined in more detail later in this chapter

4.2.1 Original Morphing Methods

It is difficult to say exactly where morphing first started because in the course of animating most objects, some deformation will usually be carried out The first time morphing emerged as a separate phase was when ILM created a number of transformations for the film *Willow* This was still only 2-D image morphing, but in 1986, ILM were working on *Star Trek IV The Voyage Home* when a special sequence was required The heads of the crew of the Enterprise were to be transformed between each other during a time-warp and to give this scene a completely different look, computer generated imagery was required The heads of the crew had to rotate while they were being transformed, so the 2-D approach would not work The special effects team decided to try something new

The heads of the actors were scanned in three dimensions using the Cyberware scanner which yielded a 256x512 grid of 3-D points representing the surface of the actors heads This resolution was coarse enough so that the creases and wrinkles of the actors were not visible However, the grid data was of the actors entire head including their hair Since hair is not a solid surface and some of the actors had radically different hairstyles, this caused problems when mapping In the end, hair was treated like a solid surface and the heads resembled sculptures The mappings were relatively straightforward since they all used the same 256x512 grid, but getting an aesthetically pleasing look required some extra work - some tweaking was required to stop Mr Spock's nose from sticking out! While the entire scene only lasted about thirty seconds on screen and would be considered standard today, it was revolutionary then and was the first time 3-D data had been used for morphing It took over a month of design and rendering time [SHAY87]

In 1989, ILM was again working on another film which required computer generated imagery, *The Abyss* Set on a deep-sea mining plant, one of the themes of the film is the discovery by the crew that there are aliens living on the sea bed The aliens can manipulate water to form any shape they want which allows them to take a corporeal form and in the most memorable scene, an alien explores the underwater vessel in the form of a pseudopod - a long snake-like column of water The scene lasted about five minutes and contained 70 seconds of computer animation

The pseudopod was created on computers using Alias and RenderMan and it was later merged with the live-action film where the actors were interacting with the pseudopod Since the pseudopod was made of water, it had to be partially transparent giving the effect of refracting the scenery behind where it travelled, even hidden beams in the ceiling which were not visible to the camera were refracted The rippling effect of water was generated by assigning a number of sine wave generators to calculate the surface displacement

During the pseudopod's exploration, it meets two members of the crew, a man and a woman, who it closely inspects and then mimics their faces like a 3-D mirror This required two facial morphs, from the initial 'head' of the pseudopod to the woman's face and from the woman's face to the man's The heads of the actors were scanned in as before using the Cyberware scanner The ILM team then used a method they termed the *hybrid* method This involved using 2-D parametric interpolation in a 3-D keyframe interpolation sequence

The 3-D data collected from the scanner formed a cylinder which went right around the head The front 180° of the 3-D data (the front of the face) was mapped to a 2-D depth map which allowed the features on the faces to be viewed as an image The morphing was then implemented using the *morf* program for the same 2-D parametric interpolation that had been used in *Willow* The image morphing program allows a parametric grid to be placed over the two images (in this case depth maps) Each region in the grid on the source image is mapped to the corresponding region on the target image, allowing regions to be grouped together or resized depending on the image This method, as shown in Figure 4-2 is sometimes referred to as mesh warping

Using 2-D image morphing on the depth maps allowed the features on a face to be identified and interpolated separately Different features were visually identified in the depth map by the differences in their colour - the eyes were dark, the tip of the nose was bright and the mouth was slightly darker than the nose These features were separated using different grid regions The scanning process took about 30 seconds and hence was unable to capture data for actors blinking Instead, keyframe scans were taken with the actors smiling, frowning and with their eyes shut These keyframes were converted to depth maps and then interpolated using *morf* The blinking eyes depth maps were then cut and pasted in to the appropriate regions of the grid for all the sequences where it was required This meant that the smiling, blinking and other facial expressions (including sticking out its tongue) could be implemented separately, in synchronisation with the actors facial movements on the live-action film

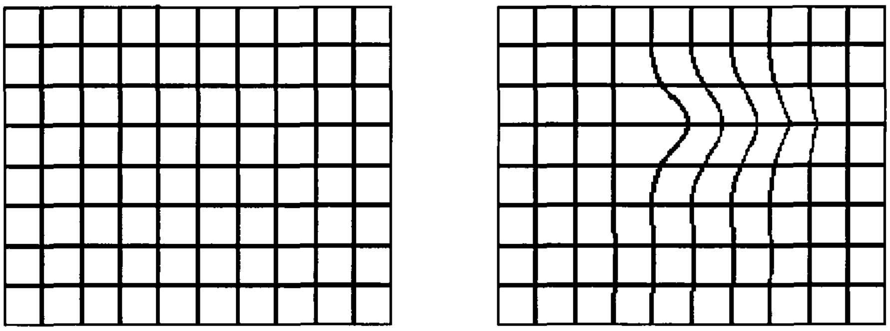


Figure 4-2 : Mesh Warping in Two Dimensions

The hybrid method works very well for facial animation and facial morphing Its main appeal lies in the ability to use simple 2-D images to control complex 3-D models It was devised when there was neither the hardware nor the software available to manage all the complexities required in three dimensions, so it is limited to objects which can be represented with depth maps This can be partially overcome by having more than one depth map or splitting the object up into different sections, but problems can occur where the objects and depth maps overlap

The director of *The Abyss*, James Cameron, was pleased with the effects (and the fact that they came in under budget) and this helped to convince him to go ahead with his next project - *Terminator 2* - in 1991 The movie was based on a cyborg from the future which could transform itself into anything Naturally, the special effects involved morphing the cyborg almost continually throughout the movie This became the most technologically advanced movie of the time and remained so for a number of years Over an hour of computer generated imagery was used in the film with the human actor, playing the evil cyborg, appearing undoctored by computers only half the time The special effects were once again created at ILM using the various methods they pioneered The movie was an instant success and set the standards for computerised special effects by which films and animations are still judged today

[WOLBER90][ANDERS90][ANDERS93]

4.2.2 Advanced Topological Morphing Methods

When a section of SIGGRAPH'92 was dedicated to the area, it signalled that morphing had become a mainstream topic of discussion for the graphics community Three papers were presented at the conference on feature-based image morphing, Fourier volume morphing and shape transformation for polyhedral objects

The feature-based image morphing paper documented the methods used to create the video for Michael Jackson's *Black or White* The problem was that conventional mesh warping algorithms did not work very well for faces that were moving and the video required the actors to be dancing The solution involved modifying the mesh warping algorithm to use line segments drawn to indicate specific features (eyes, nose, mouth, ears and hairline) as the basis for the transformation rather than deformed meshes The results were the undetectable changes between the images which make up the video

One of the main advantages of this method was the ease of use - all that was required was to draw the line segments over the appropriate features in the keyframes However, it is much more difficult to convert this method to 3-D since it is not a trivial matter drawing a line in three dimensions It would be very difficult to draw an arbitrary line in 3-D which relates to the objects in its proximity since this would depend of the viewing angle While it is possible, all the advantages of simplicity and ease of use would be lost to a complex and difficult process [BEIER92]

The second paper, on scheduled Fourier volume morphing, is related to the topic of implicit surfaces which are radically different to the topological approach Volume morphing is a specific type of morphing where the primary concern is to keep the enclosed space (volume) of an object constant during the morphing process This is useful in areas such as CAD and visualization of liquids where keeping a constant volume is important

The final paper on morphing from SIGGRAPH'92 is very relevant for 3-D topological morphing A previous paper by the authors had already outlined a method for star-shaped, genus zero objects* This stated that the morphing process can be divided up into two steps as shown in Figure 4-1 Firstly, all points are mapped from the source to the target and then the interpolation for each of these points is executed, with frames being generated during this interpolation phase The reason for using star-shaped objects was that they allowed the mapping of all points on the surface from a single point From such a point (usually the centre point), every point on the surface of the source object is projected onto the inside of the unit sphere which encompasses the object and has its centre at the same point of the object This effectively 'blows-up' the object like a balloon until it is spherical The same process is applied to the target object and the points are matched on the spheres All points on the source object sphere are tagged to go to the nearest point on the target object sphere and this is used to map points from the source object to the target object

By restricting the object range to star-shaped objects, the algorithm was quite limited, so in the paper presented at SIGGRAPH'92 the authors outlined an updated version of the algorithm which allowed complex polyhedral objects to be morphed Since the original algorithm worked quite well, but only for star-shaped objects, they focused their attention on some way to convert non-star-shaped objects into star-shaped ones They did this by 'snapping' the object to a 'convex hull' What this meant was that complex objects were treated as simple ones during the mapping phase by using a recursive dividing/projecting method In one example they give, this allows a wine glass to be mapped to a cylinder which is then projected onto the unit sphere This does not cause the glass to be turned into a cylinder, since it is only treated as such for the purpose of finding a mapping for the points between the source and target objects

[KENT92]

---

* A star-shaped object is one which has all points visible from one point (usually the centre point) A genus zero object is one which has no 'holes' - for example a donut (torus) is genus one

4.3 Implicit Surfaces

All of the methods discussed up to now use surface topologies as their basis This means that the surface of an object is the only aspect that is considered when morphing While spline-based patches do cause the surface to be interpolated from a series of control points, the individual patch is usually a small part of the object and is considered to be part of the surface Surface topologies are also widely used because of the ability to represent real-world objects, whether through 3-D digitizers, laser scanners or CAD programs

There is another method of implementing morphmg that uses a different type of object, called an implicit surface object This type of object does not have a surface defined by a series of points, but by an underlying structure and a set of rules governing how a surface is calculated based on the structure Implicit surface objects make morphing easier and allow more creative effects to be applied when morphing, but they tend not to have analogies for solid objects They are of use in describing phenomenon which are otherwise difficult to represent using surface topologies such as liquids, gases, rain, fog, clouds and smoke These phenomenon can cause real problems when modelling and animating, since they cannot be treated as a single object, but rather a collection of individual objects that can split or join with others during motion Implicit surface objects have been designed and used over the years under a number of different names such as blobby objects, soft objects, skeletal keyframes and blending iso-surfaces

Most of the work on implicit structures has been carried out by Brian and Geoff Wyvill They started out in the mid-'80s by designing a new type of object called a soft object which could be mixed with other common objects such as polygons and fractals They defined a soft object as one whose shape changes constantly due to the forces exerted on it by its surroundings Some research had been done with 'blobby objects' by James Blinn in 1982 and fuzzy objects (particles) by William Reeves in 1983, but these were concerned with the modelling and rendering of such objects and neither of these dealt with animating interaction with other objects

The main feature of a soft object is that its surface, called an iso-surface, is a field projected around the key points that make up the base shape of the object There can be thousands of key points used in constructing each object If these points had their field/surfaces created separately then it would just appear as a large number of spheres in close proximity to each other and there would be no sense of a continuous surface

Using the Wyvills' method allows all of these individual surfaces to be joined together to make a smooth surface (hence 'iso-surface') The key points form an underlying skeleton which is 'covered' by the iso-surface To create the iso-surface, key points must be able to combine with their neighbours to keep the surface continuous

A function is defined to allow neighbouring key points to contribute to the field/surface The field value at any point is a function of the distance from that point to nearby key points Using a function, like that in Figure 4-3, allows a seamless field to be constructed at different resolutions It can be seen that *c* (contribution to the field/surface) decreases as *t* (distance from key point) increases up to the radius of influence R This allows key points to contribute to any surfaces within their radii of influence

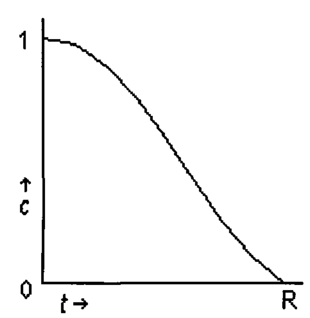


Figure 4-3 : Function for Field Contributions for an Iso-Surface

It is important to choose a field function which goes to zero beyond a certain distance (i e having a radius of influence for each control point) The field will have to be calculated at a large number of points in space, so the number of calculations required for each point must be minimised Only key points that are within their radius of influence need have their contribution to the field calculated This ensures that key points do not have an undue influence on the iso-surface - each key point will only influence the iso-surface in its locality.

In practice, building a model with individual (key) points is a labour intensive proposition and even simple shapes will require a large number of points Instead of using points as the key to constructing the iso-surface, line segments can be used These can be instanced into a set of points when calculating the field, but more commonly, they will be used as whole line segments since a number of them will be interacting with each other to contribute to an iso-surface When working with line segments rather than points, it is necessary use a different equation for describing the field The field/surface created by a single key point is a sphere, but the iso-surface created by a line segment is an ellipsoid, so the field function will describe an ellipsoid

The actual iso-surface is defined by picking a field contribution value *c* for the field function and plotting the iso-surface by connecting all points whose field value equals the chosen value This is calculated by working backwards though the field function For example, defining a field due to a single key point will result in an iso-surface that is a sphere around that point By increasing the field contribution value required, the radius of the sphere will decrease and by decreasing the field value, the radius of the sphere will increase up to a maximum of R

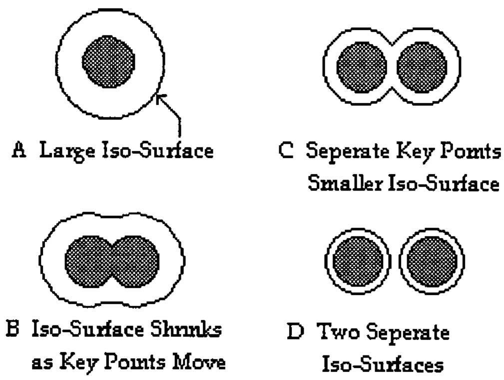


Figure 4-4 : A Droplet Splits Up

If two key points are in exactly the same place The contribution to the field for both points is added together, so the radius of the iso-surface for the same field contribution value *c* is increased An example of how the iso-surface changes is given in Figure 4-4 As the key points separate, the iso-surface shrinks, remaining joined until one point is too distant to influence the other

The field function for this example is suitable for modelling liquids - when two droplets combine into one, the volume of the droplet is increased Other choices of field function will suit other types of objects [BLINN82][REEVES83][WYVILL86a][WYVILL86b] [WYVILL90]

4.3.1 Blending Surfaces

Another type of object which combines aspects of both topological models and implicit surface models is a blended surface object This tends to be used to model solid objects which are created using CSG (Constructive Solid Geometry) operations like union, intersection and difference The objects involved in the operations have implicit functions assigned to them, which defines how their surface re-acts when involved in a CSG operation

The blended surface forms a smooth transition between different but intersecting surfaces and hence smoothing out corners, kinks and creases where the surfaces meet Among other things, blending surfaces are used in mechanical design to diminish stress concentrations, enhance liquid flow and improve aerodynamics Also, manufacturing processes such as high pressure moulding have great difficulty in producing precise sharp edges and corners Blending surfaces in terms of computer animation are usually used for giving smooth edges to objects constructed with CSG

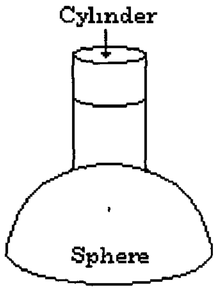


Figure 4-5 : Un-blended surfaces

For example, the union of a cylinder and a sphere using CSG will appear as in Figure 4-5 if no blending is applied There is no smoothing where they intersect When blending is applied to the surfaces, a smooth surface will appear to join them as in Figure 4-6, giving a stronger, more realistic join of the two objects

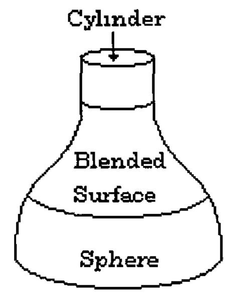


Figure 4-6 : A blended surface

Blended surfaces are not generally used in computer animation and special effects because of the overheads required to design suitable functions and compute the new surface. However, in the area of design, there is considerably more re-use of objects than in animation. Once a suitable function has been established for an object, the object can be used repeatedly in CSG work.

[ROCKWO89]

4.3.2 Morphing soft objects

During just about any motion of a soft object in an animation, it will be changing its structure. As defined earlier - in relation to topologically modelled objects - this constitutes morphing. However, the surface of soft objects are, by their nature, deformable and the deformation of soft objects happens automatically when they interact with other soft objects. So the previous definition of morphing is not really accurate for soft objects.

When talking about soft objects, morphing can be considered to be where one underlying model transforms into a different model, while maintaining a smooth continuous iso-surface. Morphing using iso-surfaces has the advantage over topological methods since the surface will never break up into unmatched primitives.

When morphing the underlying objects, many of the same problems of matching and sorting that occur with topological objects occur with implicit surface objects. These problems will be examined later. The only advantage is that there tend to be less primitives in the underlying skeleton of an implicit surface object than there are in a topologically modelled object.

One method of morphing which is unique to soft objects is surface in-betweening. This involves changing the field contribution value for the source and target objects. The field values are weighted so that the source object will have a weighting of 1 at the start and 0 at the end. Likewise the target object's weighting go from 0 to 1. This causes the source object to 'deflate' while the target smoothly 'inflates' from inside it.

Linearly interpolating the weights tends to over-emphasise the inflating/deflating process A solution to this is to use a cosine function for interpolating the source weighting and a sine function for the target as shown in Figure 4-7 The field now changes more slowly and smoothly as the source object dissipates and the target object emerges

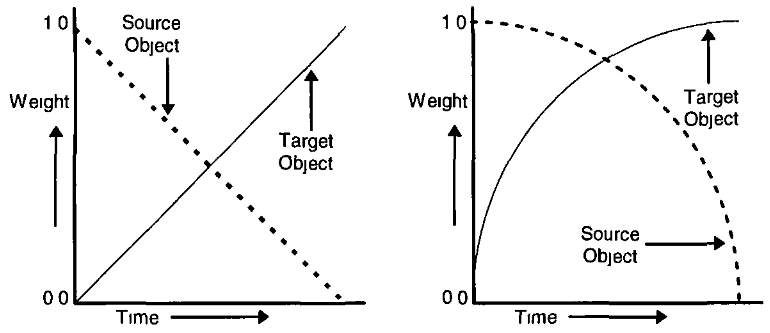


Figure 4-7 : Alternative methods for interpolating field contributions during morphing

Using sine and cosine functions as a method of interpolating weightings is an elegant evasion of the problem when attempting surface in-betweening While the procedure provides a relatively quick method of morphing implicit surface objects, it is not perfect It is best suited to objects which have similar underlying shapes Since there are no actual changes in position of the underlying key points during the process, it can leave spheres and ellipses unattached (in 'mid-air') if the source and target objects have key points or line segments which are not in the same position or hidden by the expanding or contracting iso-surfaces

4.3.3 Using skeletal keyframes for animation

When creating the iso-surfaces on soft objects, polygons can be used, but polygon meshes are a poor choice for representing curved surfaces Spline-based patches and patchmeshes are an much better way of representing the curved iso-surfaces Patches are controlled by a matrix of control points called the control hull However, when animating a soft object, great care must be taken with the control hull to ensure that patches remain a closed surface and do not intersect each other

Like the spline-based patches, a soft object is defined by a set of key points, but unlike patches, the soft object's control points form a skeleton of the model The skeleton provides a much more intuitive idea of the shape of the object than the control hulls of patches In most cases, the number of control points required for a model will be considerably less for an iso-surface based model than a spline-based patch model

Using the skeleton to animate objects gives the animator the ability to control complex animations with simple tools - the underlying motion of the object will always be shown by the skeleton of control points Specifying the status of the skeleton in two different positions allows them to be used as keyframes and the movement can be interpolated Skeletal keyframes allow special animation rules such as hierarchies and inverse kinematics to be applied to add reality to the movements Using a skeleton also allows an animation with a number of moving objects to be created and displayed in real-time using wireframe graphics Almost all animations involving human or animal movement will have been initially planned using skeletal keyframes

It is not always necessary to use implicit surface objects with skeletal keyframes When making the pseudopod sequence for *The Abyss*, skeletal keyframes were used, but a cylinder was modelled around the skeleton The smooth watery look was then obtained by applying a customised RenderMan displacement shader This method of attaching a customised shader to a simple object in order to make it appear as something different is a common occurrence when creating animations and effects - its not how its constructed, its what it looks like in the end This will be looked at in greater detail later on

4.4 Morphing complex objects

In most cases of morphing, a non-simple object is involved This is an object which is created using any number of different smaller objects of potentially different types and usually referred to as a complex or composite object These objects will usually be grouped together for reasons of modelling (using CSG) or animation These groupings are also useful for implementing morphing

4.4.1 Grouping Objects using Hierarchies

Objects are rarely constructed with only one simple object (called a primitive) Usually it will take a number of different primitives of different types to made up one object It is difficult for the computer to decide on which primitives make up which objects without the help of a human who can look at the screen and see the groupings Since this may not always be possible and is usually quite time-consuming, the primitives are usually grouped together in hierarchies

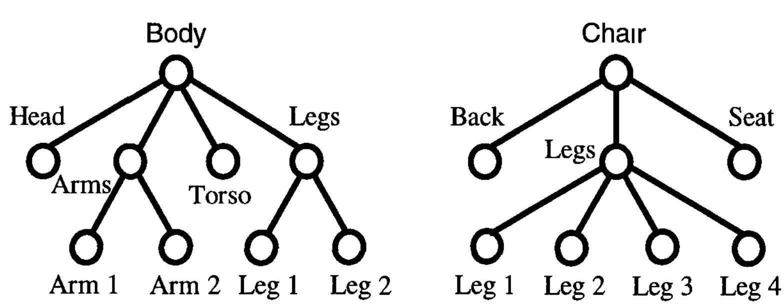


Figure 4-8 : Object hierarchies for a body and a chair

A composite object's hierarchical system can be used to determine mappings for component parts of one object onto parts of another

For example, as in Figure 4-8, hierarchies can be used to map a human onto a chair The arms and legs of the human could map to the legs of the chair, the torso could map to the seat and the head could map to the back of the chair Treating the arms and legs in a similar way may cause problems - they may be detached from the main body

In general, each node in a hierarchy will have a number of sibling nodes (of the same level) and possibly some child nodes There are two main rules followed for matching nodes on the source hierarchy to nodes on the target hierarchy

- Nodes on the source are matched to nodes of the same level on the target
- On each level, nodes are matched according to the number of child nodes they have

These rules usually operate in ascending order, so that they start with terminal nodes (those nodes that have zero child nodes - i e primitives) These are matched to those with the least (if any) child nodes Then those nodes with one child node are matched to remaining target nodes with the least child nodes and so on

Hierarchical matching tends to be quite useful in cases where the objects are similar - between animals, it is a relatively simple process However, the complexity and structure of the hierarchy are the keys to using it and these tend to be defined by the modeller (human or computer program) There is no guarantee that the hierarchies will be suitable for hierarchical matching and they may require a certain amount of adjustment For the above heuristic rules to function, there must be the same number of levels in the hierarchies, while this is relatively easy to ensure when a small number of nodes are concerned it can be time consuming for a large number

4.4.2 Cellular Matching

Assuming correctly adjusted hierarchies, hierarchical matching can still create unwanted effects since it has no concept of where objects are located Objects can be required to move over a considerable distance in order to morph into a hierarchically assigned object For example, if the arms of the body in the previous example were in the air, they would have to travel 'through' the body to get in place as the chairs' legs

While such a transformation is possible, any concept of object coherence would be lost during the transformation Object coherence refers to the state of the object during the morphing process - objects should look like *something*, even if its not identifiable

Another method for matching objects which does take account of their location is called cellular matching In this technique, the primitives are matched corresponding to the space they occupy The space that the composite objects are in is divided up into a 3-D grid of cells This is done by finding the extents of each composite object and creating a corresponding rectangular bounding box Each bounding box is then divided along the x-,y- and z-axes by some user-defined amount into cells

The bounding boxes of the source and target composite objects are different shapes, but they have been divided into an equal number of cells This gives a one-to-one correlation between the cells in the source and the cells in the target Each primitive is then evaluated (solving for different co-ordinate systems and transformations) to establish which cell it is located in Primitives are then mapped from the source cell to primitives in the corresponding target cell

While the composite objects can be of radically different sizes and shapes, this method of matching primitives maintains some position coherence between the source and target objects This method also has the advantage of not requiring intricate hierarchies to be established before morphing is initiated It also allows 'rough' versions of the morph to be previewed by allowing the number of cells to be created to be set dynamically by the user

It is possible to use cellular matching in combination with hierarchies Hierarchical matching can provide a number of terminal nodes (primitives) to be matched using the cellular method Using the previous example, the hierarchical method would resolve that the arms and legs of the body are to be matched with the legs of the chair The cellular method would then decide which arms and legs become which chair legs

In the example, the arms and legs are considered to be primitives, but in actual use they would be defined as composite objects Cellular matching could be applied to them recursively, which would determine which source primitives should map to which target primitives However, problems can occur when source cells contain no primitives and the target cells do.

4.4.3 Morphing different size composite objects

When looking at the problem of morphing composite objects, it is obvious that there will be a problem when there are a different number of primitives in each of the composite source and target objects The problem can generally be addressed in two ways, which derive from old concepts initially done in the early days of video effects for television [BURTNY71]

The first and simplest way to deal with the problem, is to use 'invisible objects' These are objects that can be infinitely small and allow a composite object to have more objects than it really has for accounting purposes For example, if the source composite consists of three objects and the target composite has five objects, two extra invisible zero-sized objects can be added to the source composite There can now be a one-to-one relationship between the objects inside the source and target composites

The second way to deal with the problem, which is usually more desirable, is to split the existing objects in a composite object up until there are an equal number of objects in the source and target Applying this method to the previous example, the first object in the source composite will split into two objects using some pre-defined method Then the second object in the source composite will split and this will leave the source composite with the same number of objects as the target

While the examples given are only concerned with the case where there are more objects in a target composite that a source composite, the two methods can be applied in reverse if the number of objects in a composite needs to be reduced there can be invisible objects in the target composite and the objects in the source composite can merge together to form fewer objects It is important to note that it would not be correct to split the objects in the target composite since the target is always fixed - only the source can 'change' since the composite is supposed to be changing during the morph

4.4.4 Using Different Primitives

Morphing an object requires changing the set of 3-D primitives that describe an object to a new set of primitives so that they describe a new object After the mapping or correspondence stage in morphing, all source object primitives should have a target assigned While it can be relatively simple if there is only one primitive and it is of the same type, complications occur when interpolating between different types of primitives and indeed different numbers of primitives

Morphing primitives of the same type is quite simple in terms of the interpolation stage since the format will be the same, only the points and/or parameters used need be interpolated using some of the interpolation techniques outlined previously However, assuming that all primitives will be of the same type is quite restrictive and inflexible For example, a sphere can be divided into wedges, quarters and pie slices using clipping planes and sweep angles, but it still can't be made to model something inherently different, say, a cylinder

To address the problem of different types of primitive being assigned to transform into one another, there are two solutions one is to have multiple converters from every type to every other type, the other is to have converters for every object type to and from just one specific type

The second option is more elegant and adaptable, so a specific object type will be required which can represent objects of all types of objects Obviously a sphere can't represent all types of object so some other type will be needed The same problem will occur with other quadratic objects, so some other type is required

Using polygons, an approximation of the surface of such quadratics could be obtained The surface of objects can be instanced by solving the appropriate equation at discrete intervals However, this format will have to be able to model the surface of a sphere, a cylinder and other curved objects, a polygon will not provide a completely smooth surface

Polygons are planar (flat) and an object will resemble a number of flat surfaces rather than a single smooth surface This can be solved by reducing the size of the polygons to a point where they are smaller (when rendered) than a pixel But this will require an enormous number of polygons to be modelled which is undesirable because of the increase in size and time that is needed to render and the fact that moving closer to the object will require even more polygons to be used

The use of polygons is obviously not ideal so some other way of representing the surface of an object is required A good solution to this is to use spline-based patches and patchmeshes A patchmesh is where a sequence of touching patches is grouped together to reduce storage space and provide smooth inter-patch interpolation This guarantees that the edges of a patch in the mesh meet the edges of neighbouring patches smoothly providing a constant surface Using spline interpolation, patches and patchmeshes provide smoothly interpolated curved 3-D surfaces For objects which are particularly difficult to model using patchmeshes - such as spheres and complex hyperbolic objects - NURBS (Non-Uniform Rational B-Splines) can be used, but for most objects a patchmesh or group of patchmeshes can be used

Choosing a bicubic patchmesh as the primitive object type to convert all other objects to when morphing, means that the topological approach is the being used and the implicit surface approach is abandoned This means that the surface in-betweening method is no longer available, however hierarchical mapping and cellular matching are both still operational since they operate at higher (object structure) and lower (world co-ordinates) levels, respectively

All that is required now is a number of converters for each type of primitive object that is likely to occur, to convert that object into a patchmesh or set of patchmeshes which represents the object This will be looked at in more detail in Chapter 5

4.5 Covering the seams

Coherence has been mentioned a number of times when discussing morphing There are two aspects to this, object coherence and surface coherence Object coherence refers to the need to keep the object that is undergoing the transformation in one solid form so that it doesn't split up into multiple objects or unwanted shapes This is generally controlled by the correspondence heuristic chosen, such as hierarchical mapping or cellular matching

Surface coherence refers to the need to keep the surface constant and unbroken during the morphing process When using topological objects, it is possible that the underlying primitives, which have been evenly matched in the source and target object descriptions, will become unmatched during the morphing process Using a patchmesh ensures that all the patches in the mesh are evenly matched, but not every object can be described using a single patchmesh There can be problems where objects join each other as with blending surfaces and some objects (like a sphere) cannot be represented using a bicubic patchmesh

The surface of an object being morphed may need to change in order to cover for 'mistakes' like those mentioned above It can also change the 'look' of the object, so it highlights the fact that the object is undergoing a transformation Usually, objects being changed will take on a liquid-like look, becoming an unrecognisable rippling body Examples of this are the pseudopod in *The Abyss* and the shape-shifting character 'Odo' in the television series *Star Trek Deep Space Nine*

One of the main features of RenderMan is the customisable shading language This allows small C-like routines, called shaders, to be written These shaders control how objects in a scene are to be shaded One type of shader is a displacement shader, which allows the surface of an object to be perturbed slightly Using a displacement shader, any surface can be made to look as if it is turning into a liquid with a rippling effect added to it A displacement shader and its effect on a smooth sphere can be seen in Figure 4-9

The RenderMan Shading Language is based on C Shaders take the form of functions which allow parameters to be passed from the RIB file, overriding any default values specified for the parameters Local variables are allowed to be declared There are also a number of global variables and global functions available to the shader and in the end, it is these global values that control the shading

The main global variables are the colours *Cs*, *Os*, *Cl*, *Ci* and the points *I*, *P*, *N*, *Ng*, *E* The colours represent the input surface colour, surface opacity, light colour and output surface colour, respectively The points represent the viewing direction, the surface position, the surface shading normal, the surface geometric normal and the camera position

Some of the main global functions are **noise**(), **transform**(), **faceforward**(), **normalize**() and **calculatenormal**() as well as the common maths functions like **sin**(), **cos**(), **abs**() and **pow**() The **transform**() function returns the value of a given point in the coordinate system specified The **noise**() function returns semi-random numbers based on the values passed to it The **normalize**() function returns a vector for the given point The **faceforward**() function returns a vector which points in the opposite direction to the specified vector and at the point specified This is usually used to ensure that a surface normal points at the viewpoint (camera) The **calculatenormal**() function returns the normal to a surface for a specified point

The sphere with the 'bumpy' surface that is shown in Figure 4-9 was created using the displacement shader shown in Figure 4-10 This shader works by moving the point on the surface being shaded (*P*) a variable distance each time the shader is called The surface normal (*N*) is then adjusted to reflect the new position of the surface Once the displacement shader finishes, the normal colouring/shading process continues and any other customised shaders are run, using the new (displaced) surface position This effectively changes the topology of the object without changing the underlying structure - a form of morphing in itself

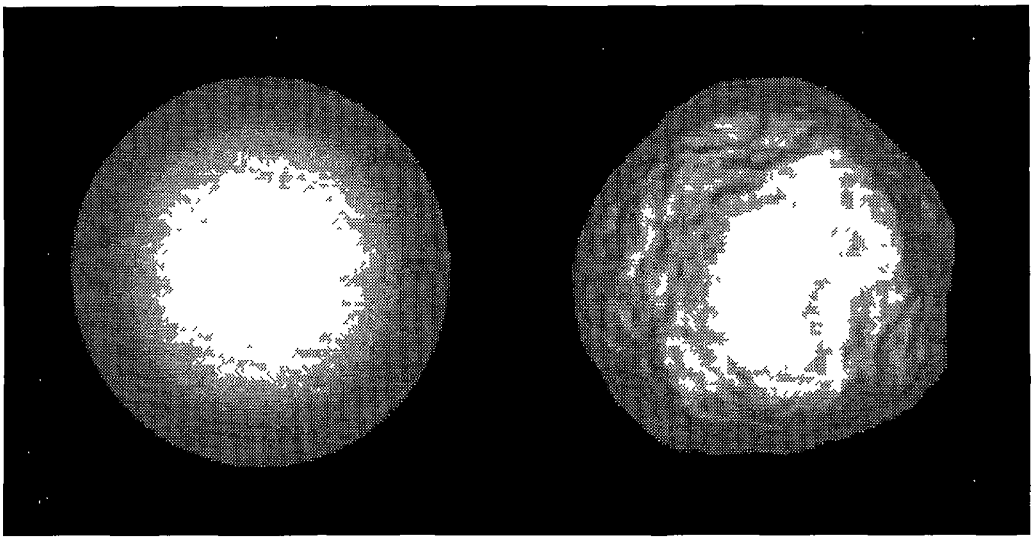


Figure 4-9 : Spheres with smooth and displaced surfaces

```
/*
 * waterbump sl
 *
 * Fractal Brownian Noise displacement shader
 */
displacement
waterbump(
      float freq = 1,   /* initial noise frequency */
      float octaves = 8,      /* # octaves of noise */
      float lacunarity = 2,   /* noise freq shift factor */
      float freq_exp = 1,     /* freq exponent */
      float Kscale = 1,)
{
      point PtShade,
      float frequency,
      float var,
      float i,

      PtShade = transform("shader",P),
      frequency = freq,
      var = 0,
      for (i=0, i<octaves, i+=1)
      {
            var += ((noise(PtShade)*2)-1) *
                        pow(frequency,-freq_exp),
            PtShade *= 2,
            frequency *= lacunarity,
      }

      P += Kscale * var * faceforward(normalize(N),I),
      N = calculatenormal(P),
}
```

Figure 4-10 : Waterbump displacement shader

Chapter Five: Implementation

5.1 Overview

The RenderMan Interface describes two bindings for 3-D scene description data - one in the form of function calls in the C language and the other in the RenderMan Interface Bytestream (RIB) format The C binding is a useful way of building up a set of objects and allows programming fundamentals such as iteration, recursion and condition checking to be used Unfortunately, it has a number of drawbacks too These are caused by the need to recompile every time there are any changes made, which makes the process inflexible and potentially incompatible with different systems

Using the RIB binding allows much more flexibility since it is an ASCII text file/stream which is processed by the renderer The obvious disadvantage of the RIB binding is that it does not allow the use of programming constructs - every object or attribute is explicitly instanced However, this does mean that every part in the scene can be changed and re-rendered by simply editing ASCII text It is this process of manipulating the ASCII text in a RIB file that forms the basis for the implementation of the concepts examined in this thesis

The difference between the C binding and the RIB binding can be seen below

C Binding

```
RtColor aColour={0 1,0 1,0 9},
RiAttributeBegin(),
RiSurface("plastic",RI_NULL),
RiColor(aColour),
RiScale(1,1,1 75),
RiSphere(0 6,0,0 6,360,RI_NULL),
RiAttributeEnd(),
```

RIB Binding

```
#No variables in RIB
AttributeBegin
Surface "plastic"
Color [0 1 0 1 0 9]
Scale 1 1 1 75
Sphere 0 6 0 0 6 360
AttributeEnd
```

RIB output requires all calls and variables to be instanced individually, effectively ‘unrolling’ any iterative loops, conditional statements or recursive functions RIB files are usually larger than the C-code required to produce the same scene

```
for(i=1,i<=3,i++)
{
    RiTranslate(1,0,0),
    RiCone(i,0 5,360,RI_NULL),
}
```

```
Translate 1 0 0
Cone 1 0 5 360
Translate 1 0 0
Cone 2 0 5 360
Translate 1 0 0
Cone 3 0 5 360
```

5.2 Using a two-pronged approach

There were two angles of approach for the practical implementation of methods discussed in this thesis This was due to the constraints of hardware and software available and the time required to develop and test the differing aspects of computer generated imagery It was apparent that all operations would be carried out on RIB files rather than the C-language binding Initially, the NeXT hardware (and built-in renderer) was not available, so a set of ANSI C routines for RIB files was developed

To create an animated scene using RIB files, a RIB file is created for each frame This requires that the 'invasive' method of editing a file be repeated for every frame in an animation With each frame requiring a slight difference in the values specified in the file and no way to tell if they correctly produced the desired visual effect without waiting for a complete rendering, it became evident that some method for quickly previewing a sequence of RIB files without having to render them would greatly speed up the process of development and testing From this previewing application, extra features were added so that it allowed non-linear previewing and control over some of the animation such as camera positions and attributes and thus became a second area for development and testing

The back-end RIB file processing program was known as *Morphit* This was written in ANSI C so that it could be ported and run on either UNIX (any type) or PC systems The front-end previewing and interface program was adapted from the NeXTSTEP 3 0 3DKit demonstration program *Simple*, and this was known as *SomhSimple* It was written in ANSI standard Objective-C and runs on any computer running NeXTSTEP 3 0 or higher This allows the use of the 3DKit class libraries to access the built-in Quick RenderMan renderer which allows near-realtime draft renderings in the form of point-cloud, wireframe, faceted or smooth surfaced graphics It also allows photorealistic rendering using PhotoRealistic RenderMan All development in NeXTSTEP was carried out on a monochrome NeXTstation 'pizza box' with a Motorola 68040 processor running at 25Mhz and 32Mb of RAM

5.3 Initial results with procedural animation

The most basic requirement for an animation is where the object or objects in a scene move from one location in the scene to another This requires a number of frames be created with the same object(s) definition but with a slowly changing location In a properly structured RIB file the location of objects is set by placing a `Translate` command before the objects' definition details Therefore, to create an animation requires a set of RIB files, each with a slightly modified `Translate` command

The *Morphit* application was developed gradually over the course of this thesis This allowed simple methods such as procedural animation to be tested and refined before moving on to the more complicated methods such as keyframe interpolation as outlined in Chapter 3 Initially *Morphit* allowed RIB files to be modified to allow values to vary over a sequence of frames The values varied according to the number of the frame - in effect, procedural animation

Initial versions of *Morphit* used an existing RIB file which was edited to have the characters `%F` in the output file name and `%T` in the parameters of a `Translate` command The percent symbol was chosen because it does not normally appear in RIB files The *Morphit* application was then run with the number of frames required specified at the command line This copied the RIB file, character by character, to sequentially numbered RIB files replacing only the `%F` with the frame number and `%T` with a value based on the frame number This resulted in numbered RIB files which were then rendered and the image files viewed sequentially

The RIB command parameters that could be varied were extended to include the `Rotate` and `Scale` commands and to allow features such as including other RIB files and specifying offsets for the values being used These allowed the control of small details such as the propeller rotating on an aeroplane but they were not particularly convincing for creating life-like animation

An example of the type of animation possible with these controls is shown in Figure 5-1 Note the slowly rotating propeller which was animated procedurally

In order to create a very simple animation, - for example a 360° rotation of some object like the propeller shown in Figure 5-1 - it is only necessary to change one lines before the actual instancing of the object In this case it would be a matter of `Rotate %R 0 0 1` where `%R` is the number of degrees to rotate, which depends on the number of frames required in the animation

In order to have an animation with 20 frames, a rotation step of `%R` = 18° would represent a 360° rotation Then 20 frames, identical except for `%R` which varies from 0 to 342 These frames, when rendered, will display the object and rotate it 360° around the z-axis It is important to note that the z-axis may not actually be what it should be - there may be any number of rotate commands before the one used for the animation and what is currently the z-axis may be in a completely different axis - i e the coordinates are not expressed in camera coordinates and possibly not even in world coordinates

During the implementation of basic animation, two problems arose which did not have simple solutions The first problem was that the structuring conventions used in sample RIB files differed, depending on the source of the RIB file For example, some of the RIB files used the `ObjectBegin ObjectEnd` block declaration outside of the hierarchies which allowed a constant set of object statements to be instanced a number of times within the hierarchy However, some of the files used the `MacroBegin MacroEnd` block declaration instead which allowed varying parameters inside the block, and is not part of the RenderMan Interface specification, but which was implemented in the NeXT versions of PRMan

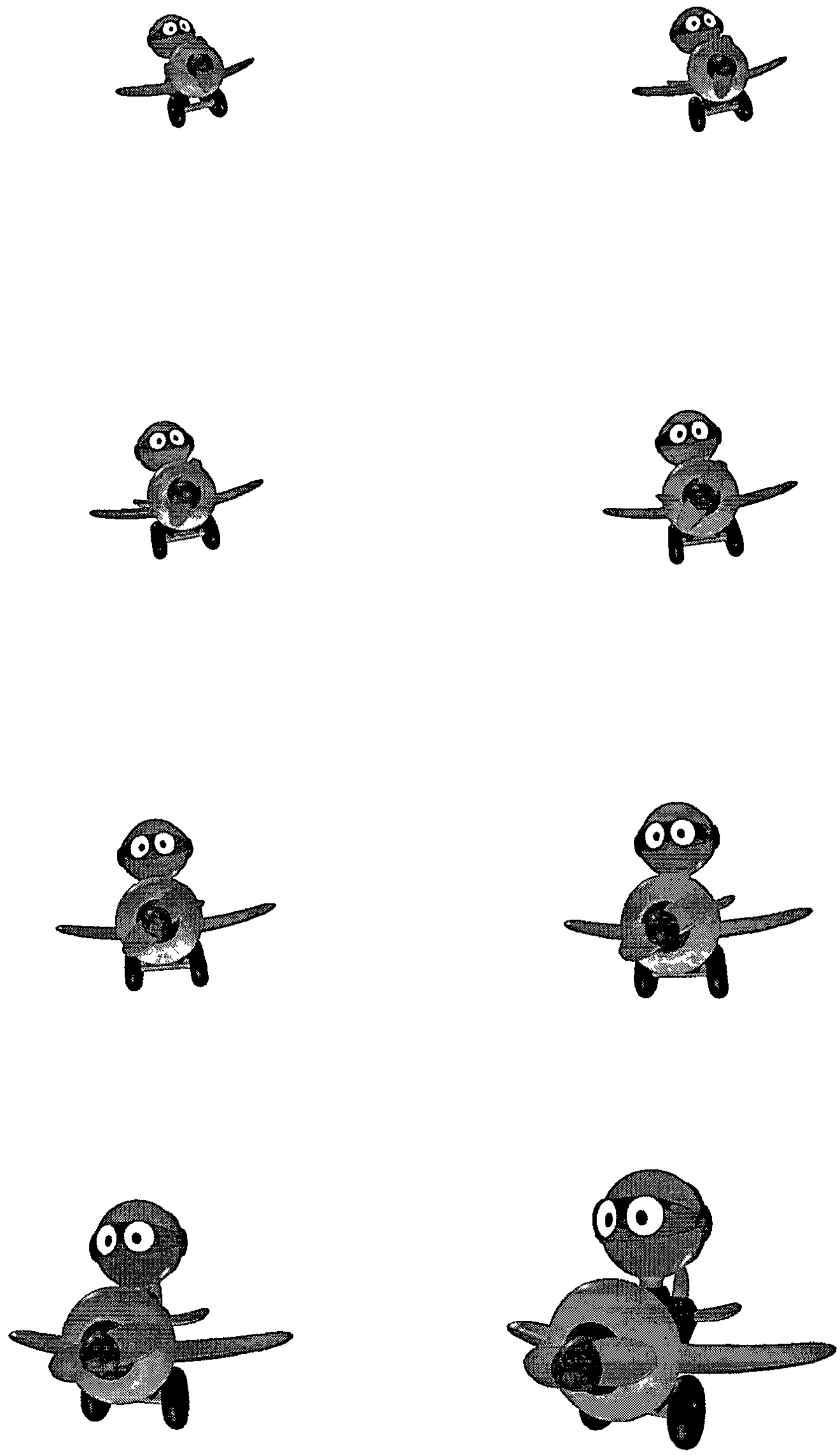


Figure 5-1 : A simple animation generated by *Morphit*

Sample RIB files came from a number of different sources including Pixar's MacRenderMan/Showplace, NeXT's 3DKit sample files, Stone Design's 3DReality and BMRT sample files as well as those created manually The applications used were written over a number of years starting in 1990, and the RenderMan Interface specification was not well established during their creation The specification has a number of guidelines on the structuring conventions used for RIB files, but the PRMan implementation has backward-compatibility with older versions so it does not enforce these conventions RIB files from commercial modelling packages were not available due to the expense of these packages and the lack of operating systems and hardware to run them RIB files are also quite difficult to obtain freely due to the copyright problems that were mentioned in Chapter 1

The second major difficulty encountered during implementation was the difficulty in determining the direction of the axes of objects in RIB files These depend on the coordinate systems being used and the method in which the objects were modelled While many applications use coordinate systems similar to those in RenderMan, a number (such as Autodesk's 3DStudio) use different (and possibly non-hierarchical) co-ordinate systems, for reasons just as having their own built-in renderer Even in modellers with compatible coordinate systems, problems can occur where the person creates an object which can be interpreted in a number of ways For example, an ovoid (egg shape) can be seen as a squashed sphere from one axis and as a stretched sphere from another RenderMan does allow names and details to be assigned to objects, so it is possible to have some customised method of identifying difficult objects, but generally, trial and error are required

5.3.1 Structuring RIB objects

Within the RenderMan Interface specification, not only is the Bytestream binding declared, there are guidelines on creating a properly structured RIB file A properly structured RIB file has hierarchies laid out within it and while it is not exactly necessary, it does allow for more efficient processing While this is not always implemented in modelling programs, it is getting better, with more and more applications structuring the RIB files For example, if modelling a table and chair in a room, the topmost node should be the room, with two leaves of table and chair The table could then be made up from a combination of a polygon for the top and four cylinders for the legs

This allows objects to be manipulated as a whole, rather than as individual parts This saves on the number of manipulations required - one command will work on an entire sub-hierarchy and keeps objects coherent It also provides for greater clarity and understanding in the file An object's location and orientation can be changed by modifying the structure within the which the object will be instanced

Assuming a properly structured file, it is possible to 'parse' the RIB file and reduce it to a number of positioning statements and objects By noting the occurrence of `AttributeBegin` and `AttributeEnd` statements, the topmost hierarchy can be obtained and as the statements become nested, the lower parts of the hierarchy can be obtained After each `AttributeBegin` statement, the RenderMan Interface specification guidelines recommend the statement `Attribute "identifier" "name" "objectname"` which declares the name of the object about to be specified The objects' names should correspond to the objects visible on screen and their constituent parts, allowing the RIB file to be more easily read

5.3.2 Coordinate Systems in RenderMan

The RenderMan Interface uses a number of coordinate systems which allow objects to be grouped together in hierarchical order It uses a type of stack system where graphics transformation commands are pushed on top of a stack, with markers placed to denote where an *xx*`Begin` block starts and when a matching *xx*`End` is found, the commands issued within that block are popped off the stack This is how graphics states are preserved throughout complex definitions

In RenderMan, there are five predefined coordinate systems - raster, screen, camera, world and object Raster coordinates are in the output image which start at 0,0 in the top left-hand corner of the image and pixels lie at non-negative integer locations Screen coordinates are on the viewing plane, where the output image usually is taken from the range [-1,+1] in screen space, but this is customisable as is the type of projection used to transform camera coordinates onto the viewing plane

Camera coordinates are the coordinates used in the three dimensional space where the viewpoint is considered the origin and the direction of the view is along the positive side of the z-axis Camera coordinates are stated before the `WorldBegin` `WorldEnd` block, allowing some transformations between the location of the world and the location of the camera World coordinates are the ‘global’ coordinate system that objects are declared in and are independent of the camera World coordinates are the “top-level” coordinates inside the `WorldBegin` `WorldEnd` block

Objects declared will have their parameters (declared in object coordinates) translated into world coordinates by applying the transformation that is current inside that block There is no limit on the number of object coordinate transforms preceding any given object instance Each time an `AttributeBegin` or `TransformBegin` statement is reached, a new object coordinate system is defined And inside each of these blocks there may be multiple `Rotate`, `Scale` and `Translate` commands which transform the current coordinate system These transformations remain part of the current coordinate system until they are popped off the stack when an `AttributeEnd` or `TransformEnd` statement is reached

When creating a properly structured RIB file, this means that even the simplest of objects should really be nested inside a `TransformBegin` `TransformEnd` block or more correctly inside an `AttributeBegin` `AttributeEnd` block with an `Attribute "identifier"` statement giving the object's name Unfortunately, very few modellers go into such detail, making the identification of individual objects and coordinate transformation systems difficult

Using two coordinate systems (Camera and World) at the highest 3-D levels means that the virtual camera can have characteristics such as orientation, zoom and focus which are completely unrelated to the objects characteristics in a scene Hence an animation which is solely about camera moves and changes will have the exact same contents of the `WorldBegin` `WorldEnd` block throughout all frames

5.4 Moving the goalposts

At the initial stages of this thesis, the possibility was explored of using two RIB files and a command-line program which produced a sequence of frames containing the in-between movements of all the objects in the files and morphing objects where required After initial experimentation, it became clear that this was not really possible except for the simplest of scenes The uncertainty created by using multiple coordinate systems and the varying structures of RIB files make it too difficult to make any decisions blindly - a human pair of eyes is really required to make the decisions

The trial and error method of editing the base RIB file on a 'lets see what this does' basis was clumsy and slow - a user interface was really required NeXTSTEP was chosen to implement this interface because it allowed the *qrman* renderer to be used to provide near-realtime draft rendering of RIB files An added bonus was the ability to use the object-oriented N3DKit class hierarchies to manipulate the scenes created by the RIB files The NeXTSTEP class hierarchies allow rapid development of user interfaces and provide considerable time-saving when compared to other systems The disadvantage is that the user interface will only run on NeXTSTEP/OpenStep systems

At the time, the only RenderMan Interface compatible renderers available were on the (black and white) NeXT computers and very slow Apple Macintoshes The speed and ease of development of the user interface were deemed to outweigh the disadvantages of using the NeXTSTEP system Development of the user interface started by examining the demonstration programs for the 3DKit hierarchies to see if they could be modified to allow the previewing of multiple RIB files in sequence The *Simple* program allowed the user to view and rotate the famous Utah teapot in three dimensions Using the source code from this program's window interface, it was possible to create a user-interface program called *SomhSimple* which allowed multiple RIB files to be viewed This program became the main previewing tool for the research in this thesis

The *SomhSimple* program allows the objects in a RIB file to be viewed in draft mode All commands outside the `WorldBegin` `WorldEnd` block are ignored in this mode, so all camera operations are controlled interactively Usually all camera transformations are embedded in a single `ConcatTransform` statement before the `WorldBegin` statement which makes the actual combination of commands difficult to recreate For this reason, *SomhSimple* always has no pre-applied camera transformations The camera transformations are controlled either by clicking and dragging with the mouse or using various sliders in the panels which surround the preview image as shown in Figure 5-3 These can be changed for each frame and saved if a new view is required, but more commonly, only the starting and finishing camera transformations are obtained and then placed in the base RIB file and marked for interpolation

Effectively this means that the entire animation is based on a single base RIB file which has all the values required for animating the scene embedded in it All that is required is a program that will expand the base RIB file into the appropriate number of frames - the back-end RIB processing file Extra features for keyframe interpolation were added to *Morphit* so that values need not be strictly related to the frame number This allowed `%Kx y` to specify interpolation between `x` and `y`

5.5 *SomhSimple* - an interface for viewing and animating objects

As mentioned previously *SomhSimple* is a (heavily) modified version of a demonstration program which allows users to view a sequence of RIB files Camera positions and attributes can be saved and applied to the sequence for photorealistic rendering Objects in the RIB files are displayed and the display can be updated for any changes made to the RIB files All the code for this program is in two modules The modules relate directly to the classes available in the 3DKit - there a number of classes, but the basic two are N3DCamera and N3DShape These provide the ability to define and view 3-D objects The screen window is of class N3DCameraView, a subclass of N3DCamera This class allows the use of single Objective-C messages to change aspects of the current camera settings independent of the shape's definition

The second module is an instance of the N3DShape class This allows RenderMan primitives to be specified in C binding and imported from a stream using the non-standard `RiResource` function There can be multiple instances and hierarchies of N3DShape, but when importing a RIB file directly and there are no objects other than that, it is only necessary to have one N3DShape instance

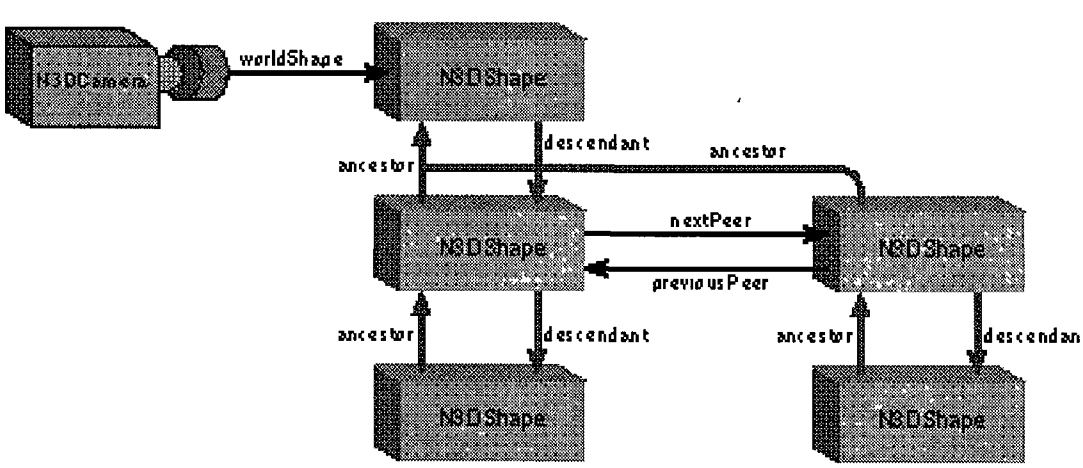


Figure 5-2 : The NeXT 3DKit Objects (N3DCamera and N3DShape) Hierarchies

There are five panels surrounding the main camera window, four of which allow different aspects of the previewed scene to be controlled, the final panel allows different frames (RIB files) to be chosen individually or played in sequence The camera control panels are for axes control (x-axis and/or y-axis and/or z-axis), quality control (point cloud, wireframe, faceted or smooth), camera attributes (from-to vector, camera roll, field of view) and transformation control (rotate, scale and translate)

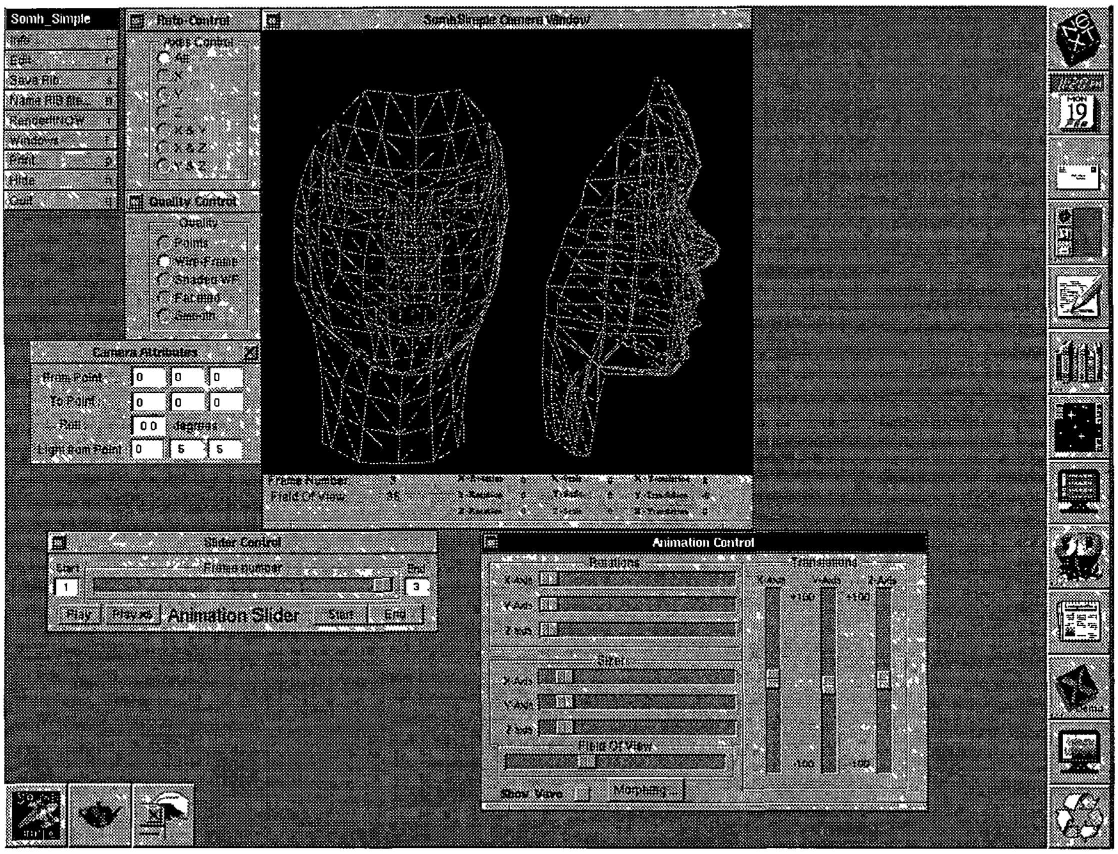


Figure 5-3 : Screen Shot of *SomhSimple.app*

As shown in Figure 5-5, the current details are displayed at the bottom of the camera window - the frame number, the field of view and the current transformations Other options are provided in the menu in the top left-hand corner The 3-D data for the face displayed came originally from a face-scanning project in Australia [MARRIO92] A converter was written to transform the triangular polygons into RenderMan `Polygon` format

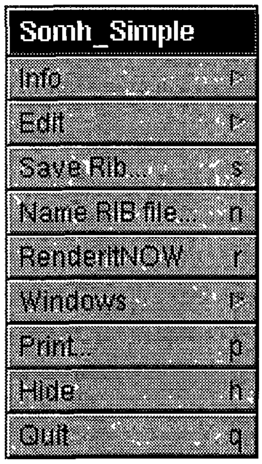


Figure 5-4 : Menu

Apart from the normal NeXTSTEP application options, three other main menu options are provided *Name RIB File* allows a sequence of RIB Files to be chosen, *Save RIB* allows the current camera position and settings to be written to a RIB file The final option, *Render It Now*, will photorealistically render the objects in the camera window at a chosen quality level and update the camera window with the photorealistic image when it has finished rendering, also allowing the image to be saved

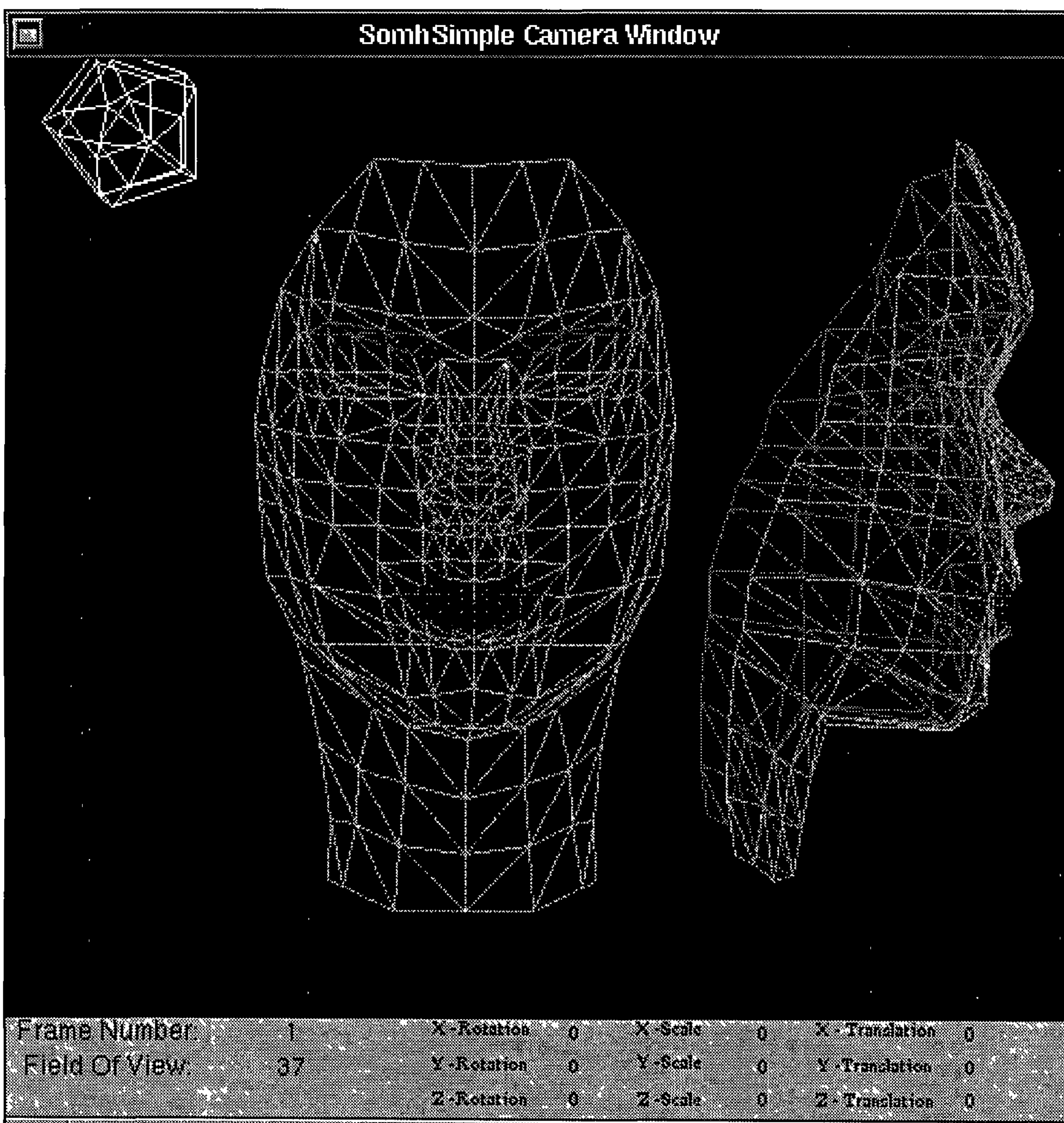


Figure 5-5 : Screen shot of *SomhSimple.app* camera window

An interface to the RIB file(s) when animating is quite important The difficulties in visualising the effect of even the simplest changes make it essential to have some method of previewing animations What *SomhSimple* does is provide a basic non-linear previewer for RIB files and allows different camera angles and settings to be viewed For simple (linear) previews of a sequence of RIB files, the wireframe renderer *rendribv* (part of the BMRT) was used When an SGI machine was available, the BMRT renderer *rgb* allowed Gouraud shaded previews

SomhSimple became a very useful tool in assessing the effectiveness of various morphing routines when they were being attempted During the development of morphing functions, bizarre and unexpected effects can occur, creating complex shapes which would require considerable amounts of processing time to render photorealistically Since *SomhSimple* allows the user to control the camera, it was possible to view the effects of morphing on different sections of objects which would normally be hidden to the camera

5.6 Implementing Morphing

The RenderMan Interface has bindings for a total of 17 object instancing statements These can be broken down into different types four polygon, three spline, seven quadratic and three special objects The four polygon types are for a convex polygon, an irregular polygon, a group of convex polygons and a group of irregular polygons The three spline types are for a uniform spline patch, a group of uniform spline patches and a non-uniform spline patch These seven object statements allow the object to be defined using a variable-length list of points

The seven quadratics (sphere, cone, cylinder, hyperboloid, paraboloid, disk and torus) all take a small list of parameters which differs for each object type The final three special type of objects are related to previously defined objects or objects built into the renderer and are not able to be accessed in any way

This leaves three different types of object which will have to be morphed into each other polygons, splines and quadratics There is also the problem of morphing groups of these and groups of all types into each other

Leaving the issue of groups of objects to one side for the moment, it can be seen that the three types of objects will require some method of being translated between their different forms if they are required to morph into a different type Its is possible to define a specific method for each object to transform into each different object type Simple calculations show that this would require a minimum of 14×13 (182) methods

While creating this many methods is quite possible, it is beyond the breadth of this thesis, and is also a rather inefficient way of dealing with the problem Usually, RIB files contain instances of quadratics and bi-cubic `Patches` or `Patchmeshes` (a `Patchmesh` is a set of `Patches` defined and grouped together so their touching edges are smooth, giving the effect of a continuous surface) Some occasionally contain polygons (usually these have been converted over from other file formats) This meant that only eleven object types needed to be considered

In order to further cut down on the number of types of objects that methods needed to be provided for, it was decided that `Patchmeshes` would become a 'base' format into which all the other object types would be converted so that morphing operations would only need to be carried out on one object type

This meant that there had to be ten conversion methods to convert objects into `Patchmeshes` For (convex) `Polygons` this was trivial, since a `Patchmesh` requires only that the same points for the polygon be instanced For groups of `Polygons` (called a `PointsPolygon`) the conversion was less trivial While the polygons in a `PointsPolygon` are instanced in any order, a `Patchmesh` requires a grid-like formation in the control hull If the polygons do not fit into a grid-like formation, it may be necessary to split the group up into a number of separate `Patchmesh` instances

Convertıng a `Patch` ınto a `Patchmesh` ıs exceedıngly sımple sınce a `Patch` ıs just a `Patchmesh` of grıd sıze 1×1, wıth the same poınt lıst followıng Sınce both splıne objects and polygon objects are ınstanced wıth poıntlısts, representıng them wıth `Patchmeshes` ıs a matter of ınstancıng poınts ın theır pomtlısts ın the appropnate order However, thıs ıs not the case wıth quadratıc objects whıch have no poınt lısts

The ınstancıng commands for the seven quadratıc objects all take dıfferent parameters These parameters vary from beıng the heıght of a cone or the outer radıus of a torus to beıng the sweep angle for the entıre object In order to represent the quadratıcs as `Patchmeshes`, ıt ıs ımportant to remember that the `Patch` and `Patchmesh` can be thought of as pıeces of paper They can be eıther bı-lınear or bı-cubıc, ı e they can represent a surface defıned usıng two lınear or cubıc equatıons Thıs accurately defınes a `Hyperbolo1d` and hence ıts derıvatıves `Cones` and `Cyl1nders`

Thıs leaves the `Sphere`, `Parabolo1d`, `D1sk` and `Torus` It ıs clear that none of these can be represented 100% accurately by eıther a bı-lınear or bı-cubıc functıons Most of the research ınto representıng quadratıcs wıth splıne-based objects has been carrıed out on spheres The upshot of thıs ıs that a sphere can be represented usıng a bı-quartıc splıne-based patch such as a NURBS (an `NuPatch` ın RenderMan)

[TILLER83][PIEGL85][PIEGL86][PIEGL87][COBB88][FARIN88][FARIN90]

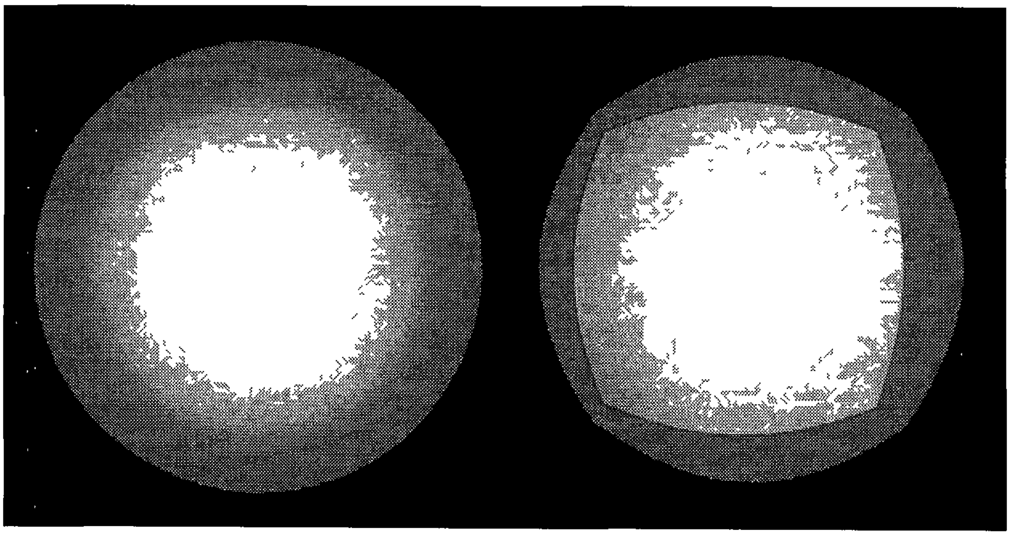


Figure 5-6 : A sphere and an approximated sphere

5.7 Practical Implementation of Morphing

The practical implementation of morphing was executed by the *Morphit* program As with the animation process, a base RIB file was used and the morphing commands were accessed in a separate file which was specified by a `%Mtarget rib` command in the base RIB file The object(s) following the `%M` were considered the source object(s) and those in the external file were the target object(s) The objects were parsed and their details recorded (mesh size, sweep angle, height, etc)

At this stage, the objects were matched and ordered into source and target objects Where there were more source objects than target objects, some of the source objects were allocated the same target object This matching process was the subject of experimentation with hierarchical and cellular matching as mentioned later Once they were matched, the source objects were mapped to the target objects (in the case of `Patchmeshes` this meant making the mesh sizes equal) and then interpolated as described in Chapter 4 Due to time limitations, it was decided to concentrate mostly on one particular type of object For flexibility, the choice had to be one of the spline-based patches (either a `Patchmesh`, a `Patch` or a NURBS)

Working with NURBS instead of `Patchmeshes` will require that all patches are individually declared instead of being declared en masse as a mesh This will actually create a number of problems by breaking up one `Patchmesh` statement into a large number of smaller statements For the ease of implementation in this thesis, it was decided to leave the base object type as `Patchmesh` and to approximate the `Sphere`, `Paraboloid`, `Disk` and `Torus` objects using `Patchmesh` statements

This can be achieved by splitting the object up into smaller sections (usually eight pieces will do) and then adding a custom shader which will disguise the imperfections in the approximation

The translations and approximations will be executed on both the source and target objects which will now consist solely of `Patchmesh` objects These will still be all of different sizes and (hopefully) grouped in hierarchies Now some methodology must be used to determine which groups in the source are to morph into which groups in the target The hierarchical and cellular matching procedures mentioned in Chapter 4 are used The hierarchies can be very complicated so no automatic matching is carried out Using an interface, the user can specify which named groups morph into each other To simplify matters these groups are split up and separated into different files

Once separated into different files, it can be assumed that there will now be two files for each group to be morphed - a source and a target Each will contain a number of `Patchmesh` statements In order to decide which `Patchmesh` statements in the source are to map to which `Patchmesh` statements in the target, cellular matching is used A bounding box for each of the `Patchmesh` statements in the target is created and the `Patchmesh` statements in the source are matched to the nearest corresponding box However, all the `Patchmesh` statements in the target must get at least one `Patchmesh` from the source, so the nearest `Patchmesh` in the source is mapped to a target `Patchmesh` which has no source `Patchmeshes` already assigned

This leaves a situation where every `Patchmesh` statement in the target has at least one `Patchmesh` in the source assigned to it It is not yet at the final stage where the `Patchmeshes` pointlists can be interpolated because the `Patchmeshes` can be of different size meshes A `Patchmesh` is generally of the form

```
Patchmesh "bicubic" 10 "nonperiodic" 12 "nonperiodic" "P" [   ]
```

What this states is that a mesh of bicubic patches, whose end points in $\vec{u}$ or $\vec{v}$ do not meet, approximated over a grid of 10×12 of points is required So for one `Patchmesh` to morph into another in a sensible fashion, the size of the grid/mesh must be the same

The implementation of the changing of the mesh size proved to be one of the more complex aspects of the code written in the course of this thesis A quick overview seems to indicate that the process is just a matter of interpolating values in a 2-D array to approximate the values in a different size 2-D array

However, it is not that simple - the values in the arrays are actually specifying the control hulls for a series of spline-based `Patches` Any changes made (such as adding an extra row of points) can cause between two and six `Patches` to be changed for *each point* on the row A number of different approaches were tried to find a solution to this problem While it is possible to have one `Patchmesh` approximate another, it requires complex recalculation of every single point on the control hull, and even then it is unlikely to be an exact replica - if it has less control points it may be impossible

It must be remembered that this change of mesh size is all part of the correspondence phase of morphing (as mentioned in Chapter 4) The (source) `Patchmesh` with a different mesh size only exists to create a one-to-one mapping with the target for the interpolation phase This mesh will never be used in place of the original source object, only for calculation purposes and so it is not vital that it is an exact replica

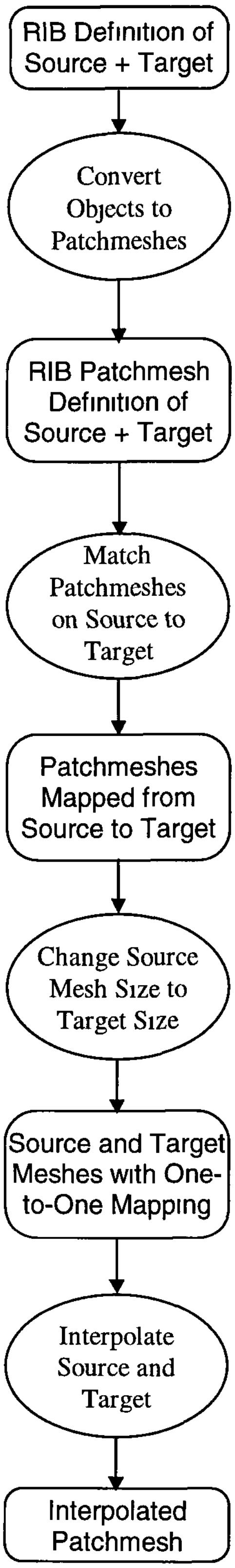


Figure 5-7 : The morphing pipeline using RenderMan

With this in mind, a simpler approach to the mesh size problem was adopted Through experimentation, it was discovered that mesh sizes could be changed relatively seamlessly by simply adding or removing rows or columns of points at the edges of the mesh Adding rows/columns was implemented by duplicating the values in the last row/column as many times as was required Removing rows/columns was implemented slightly differently, because existing rows/columns were removed and replaced by one row/column with values which contained an average of the values in the removed rows/columns This can be seen in Figure 5-8, which shows the transformation between a five element mesh and a nine element mesh

This leaves two `Patchmeshes` of the same size, requiring only the interpolation of their values for each frame This can be carried out using the various interpolation methods outlined in Chapter 3 The entire process is repeat for every frame in the animated sequence This produces a number of RIB files containing the 3-D object descriptions and camera positions and settings for the entire scene These can then be rendered at the appropriate level of quality by any RenderMan Interface compatible renderer

5.8 Examples of morphing implemented with RenderMan

<u>Initial Mesh</u> <u>Approximation using *Morphit*</u>

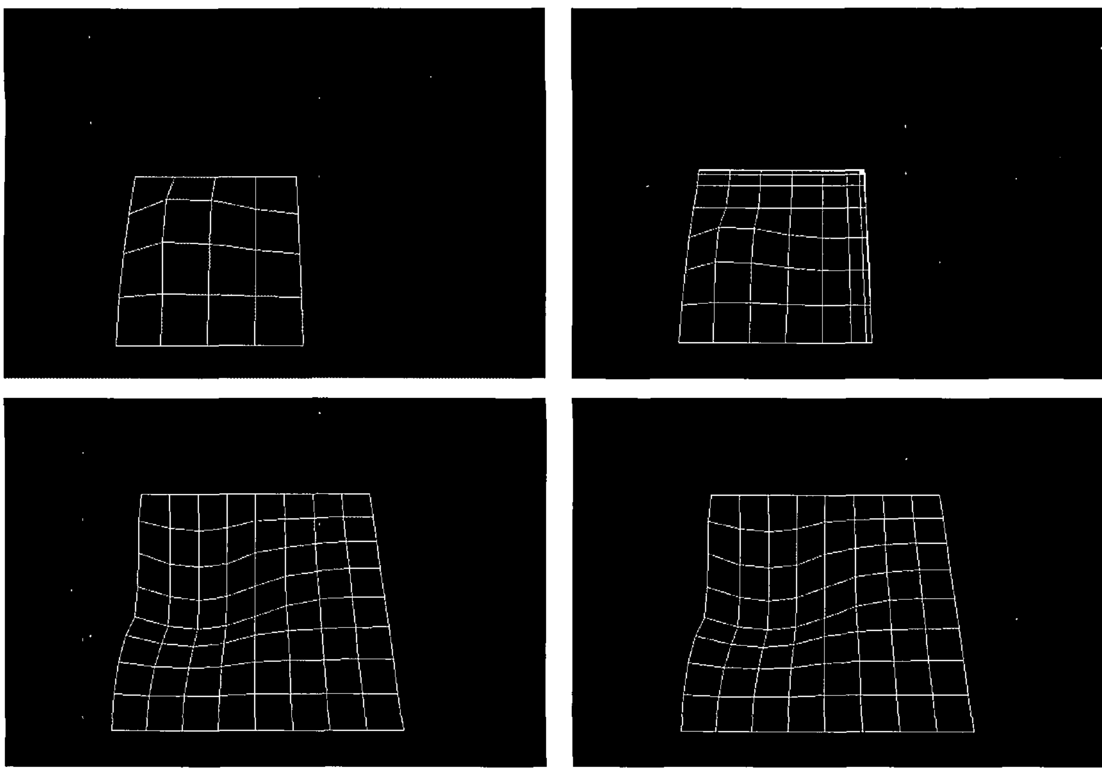


Figure 5-8 : Morphing the underlying meshes

The source five-element mesh (above) is not approximated as well as the target nine-element mesh (below) because *Morphit* uses the target mesh size for all of the instances during the transformation While the difference between the top left and top right images is noticeable, this is a wireframe representation of the mesh and will be less obvious when rendered as a solid surface If the discrepancies are still visible they can be disguised in a number of ways such as speeding up that part of the transformation, motion blurring or using a displacement shader

The mesh sizes of five and nine were chosen for demonstration purposes Normal mesh sizes vary, but most are in the range of 10-20 They are also usually different sizes in the $\vec{u}$ and $\vec{v}$ directions (i e 14×11 mesh sizes are common)

As mentioned in the previous section, objects need to be transformed into `Patchmeshes` before any morphing may be done There are a number of different approaches to transforming an object into a set of `Patchmeshes` One of the methods used for splitting a sphere up is the cubic method This involves placing a cube inside a unit sphere so that its vertices touch the sphere and dividing the sphere into six separate segments as shown on the right of Figure 5-9 These segments can be approximated using identical `Patchmeshes`

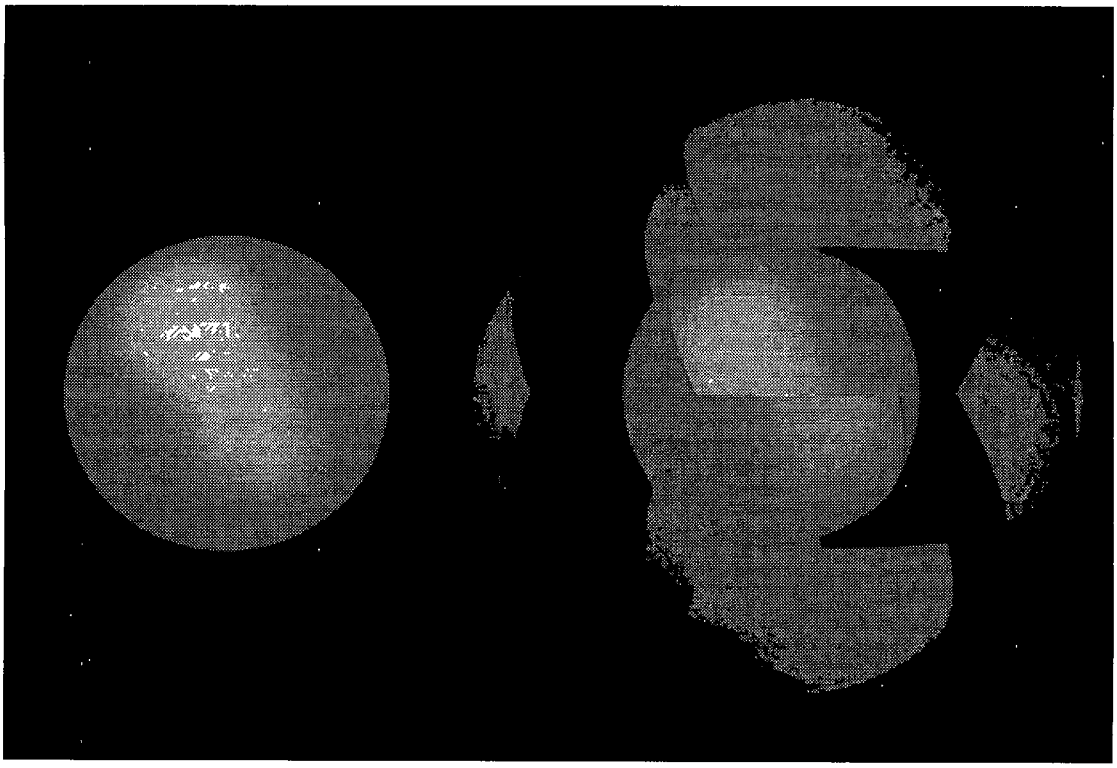


Figure 5-9 : A sphere and a cubic-replica using six `Patchmeshes`

The `Patchmeshes` can then be used to morph from another object, such as a cube, into a sphere While the representation may not be exact, using a shader can 'cover the seams' and in the case of solid objects, the true target object may be hidden inside the `Patchmesh` representation and scaled up to its true size and the representation scaled down at the end of the transformation

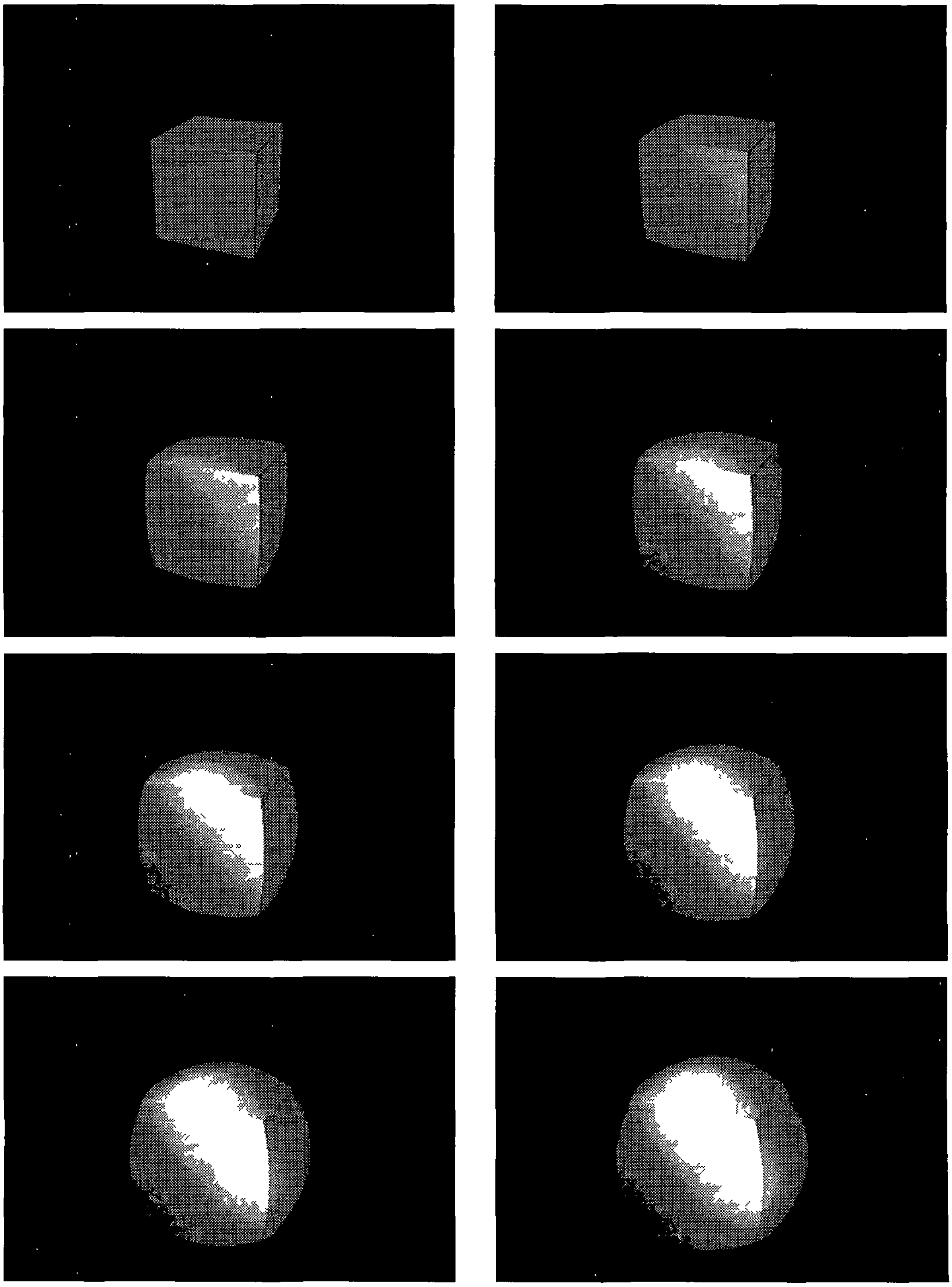


Figure 5-10 : A cube morphing into a sphere (no shader)

The transformation seen above in Figure 5-10 can be also be seen in Colour Plate 1 with the *waterbump* shader (from Chapter 4) used to hide the facets

5.9 PhotoRealistic RenderMan and the Blue Moon Rending Tools

For this thesis, all rendering was carried out using Pixar's Photorealistic RenderMan (referred to as *PRMan*) and a renderer called *RendRIB* which is part of the Blue Moon Rendering Tools (BMRT) package written by Larry Gritz They are both RenderMan Interface compatible renderers and hence use RIB files as their main input Both are compatible with the RenderMan Shading Language Interface which is a part of the RenderMan Interface specification that deals with writing customisable shaders

The main difference between them is that *PRMan* uses a version of the REYES algorithm for rendering and *RendRIB* uses a ray-tracing with radiosity algorithm Using *RendRIB* allows shadow casting and reflections without having to use shadow or environment maps The RIB files that each accept are different in that one will not process shadow map commands and the other will not process extra lighting attributes However, the files are still parseable and acceptable and will be rendered in full except that shadows will not be processed

This is the idea of the RenderMan Interface's ability to deal with future options and upgrades - the RIB binding is parseable even if individual commands are not implemented or understood This gives it a certain amount of flexibility when it comes to new methods and commands - older renderers will be able to process new RIB files to the best of their abilities

Chapter Six : Conclusions and The Future

6.1 Topical Conclusions

The topics of this thesis cover a broad range of subjects modelling objects in three dimensions, animation and computer animation, morphing and implementing these with the RenderMan Interface While the first two topics (3-D and animation) are quite general and are relatively 'old' concepts which will continue to be refined, the final two (morphing and RenderMan) are more recent developments and as such it is difficult to predict their final status

After being the favourite special effect for some time, morphing has become just one of a number of effects available to animators It is still a trying problem for implementors because of the need for radically different approaches to the same problem to provide different effects The key to simplifying morphing is to ensure that all the objects are modelled using the same building blocks (object types and structures) Once the problem has been reduced to one of interpolation, the implementation of different effects is significantly eased

The RenderMan Interface was initially designed to provide a common interface between modellers and renderers, usually in the RIB binding RIB has become one of many file formats used in 3-D graphics as can be seen in Appendix B The explosion in computer graphics and animation has seen the number of packages explode, but as yet only the very high quality sector requires file format compatibility The area is still so new that the best selling packages such as *SoftImage* and *3-D Studio* provide sufficient renderers that most customers don't need (or indeed want to bother with) other renderers

It is difficult to see exactly what the future is for RenderMan Pixar have changed their focus from software to producing animations and have decided to stop producing software except for the SGI platform While this may seem difficult to comprehend at a time of such expansion in the area, it is typical of the computer industry as a whole The market leaders in the computer animation area are Microsoft-owned SoftImage, SGI-owned Alias Research and Autodesk (makers of *AutoCAD* and *3-D Studio*) The room for a small independent company is getting smaller and smaller as these three giants compete, so Pixar's focusing on computer animated feature films is understandable However, after the success of *Toy Story*, this area too will start to be filled with competition in the next five years

6.2 Implementation Conclusions

The purpose of the programs implemented in the course of this thesis was to discover if and how the concepts discussed could be applied In general, the concepts were implementable using RenderMan, but a few were found to require lengthy coding and were difficult to implement

6.2.1 Implementation Difficulties

Animation in general is a very interactive type of process and it was not really suited to the initial back-end implementation As mentioned in Chapter 5 problems arose with multiple coordinate systems and the 'human understanding' of an object While the coordinate system problems could be overcome with an extremely intelligent AI parsing program, it is difficult to see how a program could understand what an object 'is' and how it should behave Animation is ultimately for viewing by humans and attempting to entirely remove the human element from the animation process is never likely to succeed

After this initial experimentation, it became clear that a user interface would be required to allow anything but the simplest of animations as outlined in Chapter 5 The more powerful the front-end, the better, was the conclusion of experimentation The best type of interface allowed all 3-D commands and objects to be viewed both visually and in RIB format, allowing them to be edited and controlled by using a mouse or graphics tablet

Morphing complex objects (objects consisting of groups of smaller objects) was much more difficult than initially thought The simple AI heuristic for hierarchical matching that was described in Chapter 4 was not able to handle large hierarchies Obviously, it was dependent on the hierarchies being similarly arranged on the source and target objects, but even then bizarre effects occurred For example, the handles of a shopping bag would have been suited for morphing into the (single) handle of a coffee mug But designing a heuristic that says 'similar objects should be matched' is almost impossible to program in a generic manner In the end, the best way was to let the heuristic display its results in an advisory capacity and allow a human to make the ultimate decisions on matching issues

As mentioned in Chapter 5, the choice of the `Patchmesh` command as the standard format for all objects did exclude the exact replication of some of the quadratic objects The most flexible spline-based patch available in the RenderMan Interface is the `NuPatch` - a NURBS However, this would have required the splitting up of all `Patchmesh`es and hence increasing the number of individual objects almost one hundred times on average

The ease of morphing allowed by using the soft objects (from chapter 4) was counter-balanced by the difficulty in implementing them with a 'hard object' rendering system like RenderMan While the flexibility soft objects provide is amazing, it would require another application to control and animate the soft objects which would then translate the object shapes into RIB format and ensure their correct location within a scene To a certain extent, the flexibility provided by soft objects are available in most modern animation packages as free-form deformations

It is possible that the RenderMan Interface and the RIB binding are not the best formats for the operations attempted in the course of this thesis It is important to remember that RIB is not intended to be a scripting language for animation The RenderMan Interface and RIB are designed for a much lower level of operation, but in the course of this thesis, it has been demonstrated that properly structured, hierarchically ordered RIB files can allow almost all the functionality required for higher-level animation systems while including all the detail for lower level access

6.3 Further areas of research.

A number of areas that were only briefly mentioned in this thesis bare up to a deeper analysis In addition, some parts of the implementation for this thesis were cut short for time reasons

6.3.1 Padding out the blanks

There were a number of areas under discussion in this thesis and it was not possible to implement them due a number of reasons The prime factors in deciding on coding an implementation were the time required to create any such programs, the results produced by the programs and the overall knowledge accruing from the implementation For these reasons, parts of the code for the two main applications that were implemented - *Morphit* and *SomhSimple* - have blank functions or blank areas in functions These blanks were created because the actual programs were designed as templates Functions and methods that were required to carry out the most necessary operations were coded, while other functions (usually dealing with the quadratic objects) were left It would be possible to extend the abilities of the applications by coding the blanks

6.3.2 Transforming objects into different formats

The large number of file formats (see Appendix B) which can actually represent the same object is astonishing As the number of packages (and hence file formats) increases and without the adoption of a common standard, the need to transform objects between file formats will increase

A common problem is that many packages will only output to standard formats in polygonal form An interesting area of research would be to investigate methods for transforming polygonal objects into smooth curved surfaces (e g NURBS)

6.3.3 A 3-D World-Wide Web Browser

The World-Wide Web is one of the popular formats for Internet communication these days Its successor is seen by some as a 3-D browser and eventually a VR/Simulation environment Whatever about the later, the 3-D browser idea is already available in the form of SGI's WebSpace program This is based on VRML, a polygon-only version of SGI's Inventor format

A number of other companies are pushing their own formats and browsers, but since the amount of data required for a decent 3-D scene requires a large amount of bandwidth, the 3-D WWW browser market is still embryonic The issues involved in a 3-D World-Wide Web cover a broad range of topics from networking and data compression to 3-D modelling and desktop Virtual Reality

6.3.4 An object oriented animation system

Object orientation is a key term these days and in 3-D animation it is literally applicable While implementing the *SomhSimple* application on the NeXT using the object oriented 3DKit, it became apparent that a hierarchically structured object oriented animation system would be flexible enough to allow even non-programmers to write 'macros' to allow the powerful manipulation of objects in a scene The ideal system for animation would have all the abilities and features that have been mentioned A high level scripting system controlled by a powerful user interface for objects hierarchically modelled Frames and objects could 'opt-out' (using subclasses) or get a derogation from the overall system/hierarchy and be tinkered with individually By treating (graphics) objects as (OOP) objects, they can be controlled by being connected to a controller-class object - be that a script, a macro or a user interface control The first generation of commercial applications using object orientation have already started to appear, with *3-D Studio MAX* leading the way, and others quickly following in their path

6.4 Final Comments

Observing the computer graphics industry over the past four years has been quite like watching a rocket lift off - you can see its moving very fast, but the dust cloud obscures just about everything else The frontiers in computer graphics are being pushed back all the time - with films like *Jurassic Park* and *Toy Story* - but how exactly this impacts more common applications is not exactly clear

Many people liken the explosion in computer graphics and animation with the desktop publishing revolution If this analogy holds it could have a number of consequences the top-quality animations will excel and become more popular with producers and audiences, large-sized companies and agencies will create visualisation previews and VR simulations for many projects with these becoming as common as mission statements are today, the entertainment industry will be creating more and more life-like games and special effects and finally, the quality and range of animation products available to the average (amateur) animator will be dramatically increased, allowing natural talent to be revealed

From a technical viewpoint, computer graphics used to be an area where a lot of specialised knowledge was required to produce even minimal results and now this has been reversed At Pixar, they have redefined themselves as Pixar Animation Studios In creating animations, now traditional animators work on the initial stages of creating an animation - storyboarding, layout, blocking and animation - and the later stages are worked more on by the technical people - mapping surfaces and textures and adding lighting and reflections In effect, the old movie breakdown between artist (writer/director/actor) and technician (cameraman/editor/stuntman) has been transferred into the new computer animation industry where the animators are the actors providing the action, emotion and effects and the computer specialists provide the realism and the detail Pixar have found the best results come from a marriage of both the animators and the technical specialists

The old sayıng 'the camera never lıes' has become redundant The qualıty of computer generated graphıcs has ıncreased dramatıcally over the past few years What was once deemed ımpossıble ıs now commonplace - for example, whıle everyone knew that the dınosaurs ın *Jurassıc Park* were computer generated, few were able to detect that the jeeps that were beıng crushed by a dınosaur were actually computer generated too Photorealısm may not just yet be commonplace, but ın many fılms, ıf not specıfıcally lookıng for computer generated effects ıt ıs quıte easy to accept them as real Sır Davıd Putnam's recent comment best summed thıs up

"Its twenty-four whoppers per second "

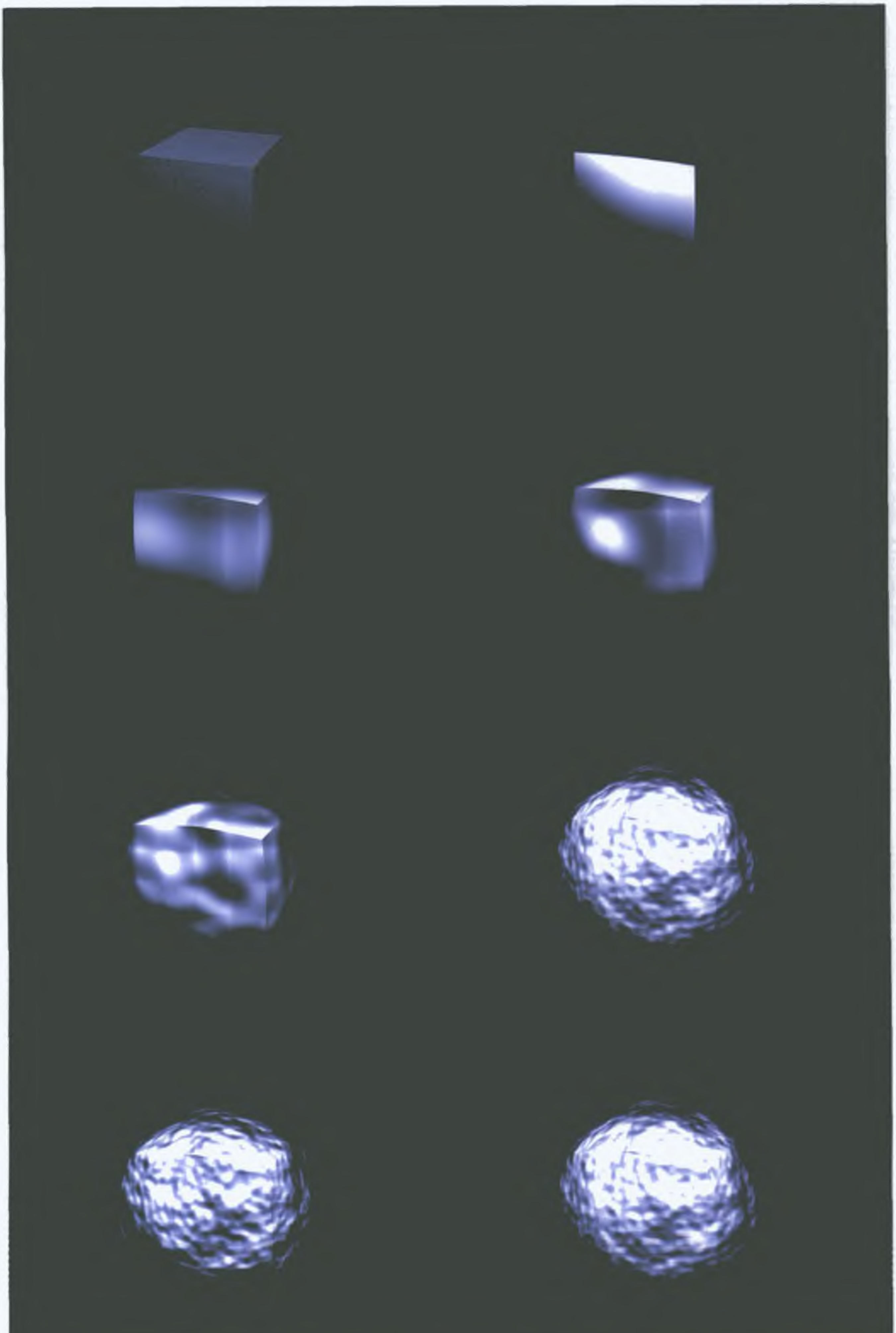


PLATE 1 : MORPHING A CUBE INTO A SPHERE USING A DISPLACEMENT SHADER

Appendix A: Bibliography

[AMMERA88] Ammeraal, Leendert *"Interactive 3D Computer Graphics"* Chichester Wiley 1988

[ANDERS90] Anderson, Scott E *"Making a Pseudopod An Application of Computer Graphics Imagery"* Proceedings of AUSGRAPH '90 pp 303-311 Melbourne, Australia 1990

[ANDERS93] Anderson, Scott *"Morphing Magic"* Carmel, Indiana Sams 1993

[APODAC87] Apodaca, Anthony A , and Mantle, William *"RenderMan Pursuing the Future of Graphics"* IEEE Computer Graphics and Applications vol 10, no 4, pp 44-49 Jul 1987

[BARSKY87] Barsky, Brian *"Computer Graphics and Geometric Modeling using Beta-Splines"* Tokyo, New York Springer-Verlag 1987

[BEIER92] Beier, Thadius and Neely, Shaun *"Feature-Based Image Metamorphosis"* Proceedings of SIGGRAPH '92 - Computer Graphics vol 26, no 2, pp 36-42 Jul 1992

[BETHEL89] Bethel, E Wesley and Uselton, Samuel P *"Shape Distortion in Computer-Assisted Keyframe Animation"* Proceedings of Computer Animation '89 Jun 1989

[BLINN82] Blinn, James *"A Generalization of Algebraic Surface Drawing"* ACM Transactions on Graphics vol 1, pp 235-256 1993

[BREENE93] Breene, Bryant *"Image Warping by Scanline Operations"* Computers & Graphics vol 17, no 2, pp 127-130 Mar 1993

[BURTNY71] Burtnyk, N and Wein, M *"Computer-Generated Key-Frame Animation"* Journal of the SMPTE vol 80, pp 149-153 Mar 1971

[BURTNY76] Burtnyk, N and Wein, M *"Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key-Frame Animation"* CACM vol 19 no 10, pp 564-569 Oct 1976

[CATMUL74] Catmull, Ed *"Subdivision Algorithm for the Display of Curved Surfaces"* PhD Thesis, University of Utah 1974

[CGINTE89] CG International '89 with Earnshaw, Rae and Wyvill, Brian (Editors) *"New Advances in Computer Graphics - proceedings of CG International '89"* Tokyo Springer-Verlag 1989

[COBB88] Cobb, J *"Tiling the Sphere with Rational Bezier Patches"* Technical Report UUCS-88-009, Computer Science, U of Utah 1988

[COLLIN93] Collins, Brian M *"Data Visualisation - Has It All Been Seen Before ?"* 'Animation and Scientific Visualisation - Tools and Applications' pp 3-28 1993

[COOK87] Cook, Robert; Carpenter, Loren and Catmull, Ed. *"The Reyes Imaging Rendering Architecture."* Proceedings of SIGGRAPH '87 - Computer Graphics. pp. 95-102. Anaheim, California, USA. Jul 1987

[CORRIE93] Corrie, Brian and MacKerras, Paul. *"Data Shaders."* Proceedings of Visualization '93. pp. 275-282. San Jose, California, USA. Oct 1993

[EARNSH93] Earnshaw, Rae A. and Watson, David F. (Editors). *"Animation and Scientific Visualization - Tools and Applications."* London : Academic. 1993

[ENCARN93] Encarnacao, Jose et al. *"Advanced Research and Development Topics in Animation and Scientific Visualization."* book 'Animation and Scientific Visualization - Tools and Applications'. pp. 37-73. 1993

[FARIN88] Farin, Gerald; Piper, B. and Worsey, A.J. *"The Octant of a Sphere as a Non-Degenerate Triangular Bezier Patch."* Computer Aided Geometric Design. vol. 4, no. 4, pp. 329-332. 1988

[FARIN90] Farin, Gerald. *"Rational B-splines."* Geometric Modelling Methods and Applications'. 1990

[FOLEY90] Foley, James D.; vanDam, Andries; Feiner, Steven K. and Hughes, John F. *"Computer Graphics - Principles and Practice."* Reading, Massachusetts : Addison-Wesley. 1990

[FRIEDH89] Friedhoff, Richard Mark and Benzon, William. *"Visualization - The Second Computer Revolution."* New York : Abrams. 1989

[GARFIN93] Garfinkel, Simson L. and Mahoney, Michael K. *"NeXTSTEP Programming - Step One : Object-Oriented Applications."* New York : Springer-Verlag. 1993

[GONZAL93] Gonzalez, R. E. *"An Object-Oriented Library for Hierarchical Animation Sequences."* Computer Graphics, on-line issue . May 1993

[GUDUKB93] Gudukbay, Ozguc. *"An Animation System for Rigid and Deformable Models."* Computers & Graphics. vol. 17, no. 1, pp. 71-77. 1993

[GUO93] Guo,Shang; Robergé, James and Grace,Thom. *"Controlling Movement using Parametric Frame Space Interpolation."* Proceedings of Computer Amination '93. Geneva, Switzerland. pp. 216-227. Jun 1993

[HUGHES92] Hughes, John F. *"Scheduled Fourier Volume Morphing."* Proceedings of SIGGRAPH '92 - Computer Graphics. vol. 26, no. 2, pp. 43-46. Jul 1992

[KAUFMA91] Kaufman, Arie. *"Volume Visualization."* Washington: IEEE Computer Society Press. 1991

[KAUL91] Kaul, and Rosssignac. *"Solid-Interpolating Deformations: Construction and Animation of PIPs."* Proceedings of Eurographics '91. Sep 1991

[KENT91] Kent, Parent, and Carlson *"Establishing Correspondences by Topological Merging A new approach to 3-D Shape Transformation "* Proceedings of Graphics Interface 91 Calgary, Alberta, Canada Jun 1991

[KENT92] Kent, Wayne, Richard *"Shape Transformation for Polyhedral Objects "* Proceedings of SIGGRAPH '92 - Computer Graphics vol 26, no 2, pp 47-54 Jul 1992

[LASSET87] Lasseter, John *"Principles of Traditional Animation applied to 3-D Computer Animation "* Proceedings of SIGGRAPH '87 - Computer Graphics pp 35-44 Anaheim, California, USA Jul 1987

[MAGNEN89] Magnenat-Thalmann, Nadia *"The Problematics of Facial Animation "* Proceedings of Computer Animation '89 Jun 1989

[MAGNEN90] Magnenat-Thalmann, Nadia and Thalmann, Daniel *"Synthetic Actors in Computer-Generated 3D Films "* Berlin, New York Springer-Verlag 1990

[MAGNEN93] Magnenat-Thalmann, Nadia and Thalmann, Daniel *"Models and Techniques in Computer Animation "* Tokyo Springer-Verlag 1993

[MARRIO92] Marriot, Andrew *"Face Mapping for Character Animation "* Technical Report no 14, Curtin University of Technology, Perth, Australia 1992

[MEALIN94] Mealing, Stuart *"Three Dimensional Modelling and Rendering on the Macintosh "* Oxford Intellect 1994

[OREILL91] O'Reilly, Derek *"Computer Generation of Photorealistic Images using Ray Tracing "* M Sc in Computer Applications Thesis, Dublin City University, Dublin, Ireland 1991

[OSTBY89] Ostby, Eben F *"Simplified Control of Complex Animation "* Proceedings of Computer Animation '89 Jun 1989

[PARKE75a] Parke, Frederic I *"A Model for Human Faces that allows speech synchronized animation "* Computers & Graphics vol 1, pp 3-4 1975

[PARKE75b] Parke, Frederic I *" Measuring Three-Dimensional Surfaces with a Two-Dimensional Tablet "* Computers & Graphics vol 1, pp 5-7 1975

[PIEGL85] Piegl, Laszlo *"Representation of Quadratic Primitives by Rational Polynomials "* Computer Aided Geometric Design vol 2, pp 151-155 1985

[PIEGL86] Piegl, Leslie *"The Sphere as a Rational Bezier Surface "* Computer Aided Geometric Design vol 3, no 1, pp 45-52 1986

[PIEGL87] Piegl, Leslie and Tiller, Wayne. *"Curve and Surface Constructions using Rational B-Splines."* Computer Aided Design. vol. 19, no. 9, pp. 485-498. 1987

[PIXAR89] Pixar. *"RenderMan Interface Specification v. 3.1."* California : Pixar Animation Studios. 1989

[REVEES83] Revees, William T. *"Particle Systems - A Technique for Modelling a Class of Fuzzy Objects."* ACM Transactions on Graphics. vol. 2, pp. 91-108. 1990

[REVEES90] Revees, William T. *"Simple and Complex Facial Animation: Case Studies."* Proceedings of AUSGRAPH '90. Melbourne, Australia. 1990

[ROCKWO89] Rockwood, Alyn P. *"The Displacement Method for Implicit Blending Surfaces in Solid Models."* ACM Transactions on Graphics. vol. 8, no. 4, pp. 279-297. Oct 1989

[SACKS93] Sacks, and Joskwicz. *"Automated Modelling and Kinematic Simulation of Mechanisms."* Computer-Aided Design. vol. 25, no. 2, pp. 106-118. 1993

[SCHMIT86] Schmitt; Barskey, Brian; and Du. *"An Adaptive Method for Surface-Fitting from Sampled Data."* Computer Graphics. vol. 20, no. 4, pp. 179-188. Aug 1986

[SEGAL92] Segal; Korobkin; vanWidenfelt; Foran; and Hoaerberli. *"Fast Shadows and Lighting Effects using Texture Mapping."* Proceedings of SIGGRAPH '92 - Computer Graphics. vol. 26, no. 2, pp. 249-252. Jul 1992

[SHAY87] Shay, J.D. *"Humpback to the Future."* CINEFEX. 29. Feb 1987

[SIGGRA92] SIGGRAPH. *"3D Graphics Standards Debate: PEX versus OpenGL."* Proceedings of SIGGRAPH '92 - Computer Graphics. vol. 26, no. 2, pp. 408-409. Jul 1992

[THALMA89] Thalmann, Daniel. *"Motion Control: From Keyframe to Task-Level Animation."* Proceedings of Computer Animation '89. Jun 1989

[THOMAS81] Thomas and Johnston. *"Disney Animation - The Illusion of Life."* New York : Abbeville Press. 1981

[TILLER83] Tiller, Wayne. *"Rational B-Splines for Curve and Surface Representation."* IEEE Computer. Society Computer Graphics and Applications. vol. 3, no. 6. Sep 1983

[UPSTIL90] Upstill, Steve. *"The RenderMan Companion."* Reading, Massachusetts : Addison-Wesley. 1990

[VINCE92] Vince, John. *"3-D Computer Animation."* Wokingham, England; Reading, Massachusetts : Addison-Wesley. 1992

[VVEDEN92] Vvedensky, Dimitri D and Holloway, Stephen *"Graphics and Animation in Surface Science "* Bristol ,England A Hilger 1992

[WATERS87] Waters, Keith *"A Muscle Model for Animating Three-Dimensional Facial Expressions "* Proceedings of SIGGRAPH '87 - Computer Graphics pp 17-24 Jul 1987

[WATT92] Watt, Alan and Watt, Mark *"Advanced Animation and Rendering Techniques - Theory and Practice "* New York ACM Press , Wokingham, England Addison-Wesley 1992

[WOLBER90] Wolberg, G *"Digital Image Warping"* IEEE Computer Society Press 1990

[WYVILL86a] Wyvill, Geoff, McPheeters, Craig and Wyvill, Brian *"Data Structure for Soft Objects "* Visual Computer vol 2, no 4, pp 227-234 1986

[WYVILL86b] Wyvill, Brian, McPheeters, Craig and Wyvill, Geoff *"Animating Soft Objects "* Visual Computer vol 2, no 4, pp 235-242 1986

[WYVILL89] Wyvill, Brian and Wyvill, Geoff *"Using Soft Objects in Computer Generated Character Animation "* Computers in Art, Design and Animation pp 283-297 1989

[WYVILL90] Wyvill, Brian *"Metamorphosis of Implicit Surfaces "* Notes from SIGGRAPH '90, Course 23 - Modelling and Animation Dallas, Texas, USA Aug 1990

[WYVILL92] Wyvill, Brian *"Warping Implicit Surfaces for Animation Effects "* Proceedings of 1992 Western Computer Graphics Symposium Apr 1992

Appendix B : 3-D Object File Formats

This is a list of some of the file formats that can define 3-D objects It shows the difficulties involved in working with 3-D objects since objects can be in just about any format and the formats are not static - they change with newer releases of software

Center for Innovative Computer Applications 3D Object File Formats

This document contains information on various 3D object file formats and how to view them from Mosaic Some of the formats have detailed format specifications available by clicking on them If you know of any other 3D object file formats, or have descriptions or pointers to any of the formats that are here, please send email to *brian@cica indiana edu* and let us know

Note: Some object file formats are being revised constantly Where possible, links to the original sites of the file formats are given to allow you to view the most recent version You are more likely to find the most recent version of a format by following the link to the original site rather than viewing the local copy

3D Object File Formats

ART (Another Ray Tracer)
Used by the *ART* ray tracer which comes with the public domain *VORT* package for Unix
Extensions art

AVS (Application Visualization System)
Used by the *AVS* commercial high-end visualization environment
Extensions geom, prop, scr

BYU
Used by the *Movie BYU* program
Extensions byu

DKB
Used by the public domain *DKB-Trace* ray tracer The *POV-Ray* ray tracer (see below) is an extension of DKB-Trace, however they use somewhat different object file formats
Extensions dkb

DXF (Drawing Interchange File)
Used by *AutoCAD* and other CAD packages There is also a **minimal format specification** to help you create DXF 3D object files
Extensions dxf

IGES (Initial Graphics Exchange Standard)
Format used by many commercial programs including *Autocad* and *Alias*
Extensions iges

Infini-D
Used by the *Infini-D* package on the Macintosh
Extensions ???

Inventor
Used by SGI's *Inventor* graphics programming package There is a very nice program called *ivview* that is available on SGI machines and is written using Inventor libraries
Extensions iv

LightWave
Used by the *LightWave* on the Amiga
Extensions ???

MGF (Materials and Geometry Format)
MGF documentation and examples are available at
ftp //hobbes lbl gov/www/mgf/HOME html
Extensions mgf

MSDL (Manchester Scene Description Language)
An intro to MSDL is available from the Computer Graphics Unit at the University of Manchester Libraries and filters are also available via ftp at *ftp mcc ac uk pub/cgu/MSDL*
Extensions msdl

NFF and **ENFF** (Neutral File Format)
Used by a variety of programs including several public domain raytracers There is at least one NFF *previewer* program written using the *VOGLE* graphics library, but it doesn't allow you to rotate or move the object, and only displays the scene as a wireframe ENFF is the Extended Neutral File Format
Extensions nff, enff

NURBS (Non-linear Uniform Rational B-Splines)
Spline surface format These models were created using the *Alpha_1* geometric modeling system at the Computer Science Department, University of Utah
Extensions txt, nurbs

OBJ
Used by the *Wavefront* suite of commercial high-end animation packages
Extensions obj (for ASCII), mod (for binary)

OFF (Object File Format)
Mesh format used by several programs including some public domain raytracers
Extensions off

OOGL (Object Oriented Graphics Library)
These files can be displayed using the public domain *Geomview* program on SGI There is now a Beta version using standard X for a variety of platforms, and a version running under NeXTSTEP A **tutorial** of the OOGL format is also available An **extension of the MESH and OFF files** are used by the *Meshview* program developed at Indiana University
Extensions oogl, off, list, tlist, grp, quad, mesh, inst, bez, vect

PLY
Used by the *ZipPack* polygon mesh "zippering" package on the SGI The *zipper* program and code for reading and writing the PLY format can be found at
ftp //graphics stanford edu/pub/zippack More information about *ZipPack* can be found at
http //www-graphics stanford edu/software/zippack
Extensions ply

POV (Persistence of Vision)
Used by the *POV-Ray* ray tracer for the Mac, PC, Amiga, and Unix A **tutorial** is also available
Extensions pov

Radiance
Used by the *Radiance* public domain radiosity renderer for Unix The ASCII files are converted into an octree format for rendering Documentation of the scene description files and the *Randiance* package are available at
ftp //hobbes lbl gov/www/radiance/radiance html
Extensions rad, oct

Rayshade
Used by the *Rayshade* public domain ray tracer for Unix
Extensions ray,.shade

RIB (RenderMan Interface Bytestream)
Used by the *RenderMan* commercial renderer by Pixar
Extensions rib

RWX (MEME Shape file)
Used by the MEME commenrical virtual reality system for the IBM PC by Immersive Systems
Extensions rwx

SCENE
The SCENE format is for the storage and interchange of 3D geometric information The format is under construction Comments are being called for from interested parties, these can be emaied to *pdbourke@ccu1 auckland ac nz* The draft document is available through Mosaic from *http //archpropplan auckland ac nz/graphics/scene/scene html* and is located in the "Computer Graphics" directory
Extensions scene

SCN (SCeNe)
This format was designed to replace a very simple format called SFF used by the *RTrace* raytracer There is a filter that converts SFF to SCN Many other converters are available to convert to and from this format available via anonymous ftp from
asterix inescn pt pub/RTrace For more information about the *RTrace* package and information about its author, see *http //diana inescn pt/acc/acc html*
Extensions scn

Sculpt
Used by *Sculpt3D* on the Amiga
Extensions scene

SDL (Scene Description Language)
Used by the *Alias* suite of commercial high-end animation packages SDL is actually a language, and as such is very tricky to convert to other formats
Extensions sdl

SDML (Spacial Data Modeling Language)
Used by the CLRMosaic package for Silicon Graphics workstations. The updated version of the SDML format can be obtained from *http://www.clr.toronto.edu:1080/CLRMOSAIC/SDML.html*. More information about CLRMosaic can be found at *http://www.clr.toronto.edu:1080/CLRMOSAIC/help-about.html*.
Extensions: sdml

SGO and **Flip File** (Silicon Graphics Object)
Used by the *IRIS Showcase* package for Silicon Graphics workstations. The Flip File format is a very simple format which supports only quadrilaterals (four-sided polygons).
Extensions: sgo

Strata
Used by the *StrataVision* package on the Macintosh.
Extensions: ???

TDDD (3D Data Description)
Used by the Impulse's *Imagine* and *Turbo Silver 3.0* raytracers for the Amiga.
Extensions: tddd

TPoly (Triangulated Polygon)
Triangulated polygon files.
Extensions: tpoly, tnpoly

VID
Amiga *VideoScape* format.
Extensions: vid

YAODL (Yet Another Object Description Language)
Used by Silicon Graphics *Powerflip* program.
Extensions: ydl, yaodl

X3D
Used by *x3d 2.0* and the *xdart* renderer, both available via ftp from *dpls.dacc.wisc.edu:/graphics*.
Extensions: x3d, obj

3DMF (3D Metafile)
Used by the Quickdraw 3D package from Apple. Additional documentation of the 3DMF format can be found at *http://www.info.apple.com/qd3d/3DMFspec.HTML*.
Extensions

3DS
Used by the *AutoDesk 3D-Studio* package on the Macintosh.
Extensions: 3ds

3D2
Used by the *Stereo CAD-3D 2.0* package for the Atari ST.
Extensions: 3d2

Virtual Reality Modeling Language Object File Formats
Virtual Reality Modeling Language (VRML) is a platform-independent language for virtual reality scene design A standard format is being devised which will allow Web users to share and link 3D objects and scenes with each other, in much the same way that HTML documents can now be linked together Clicking on an object in a virtual world could jump you to another virtual world on the Web Several proposals for such a format have been submitted This section lists some of those formats

CDF (Cyberspace Description Format)
The purpose of the CDF is to provide a standard framework to store, retrieve, modify and exchange descriptions of cyberspace objects These descriptions encompass object initialization, object state and object scheduling within a cyberspace simulation The original source for this document can be found at *http //vrml wired com/proposals/cdf/cdf html*
Extensions cdf

FFIVW (File Format for the Interchange of Virtual Worlds)
A proposal for a standard file format for storing descriptions of both individual objects and entire worlds The original source for this document can be found at
http //vrml wired com/proposals/ffivw html
Extensions ffivw

IV-VRML (Inventor VRML Format)
A proposal for a **VRML** standard using SGI Inventor format as a basis This proposal evolved into the VRML specification The original source for this document can be found at
http //www sgi com/tech/Inventor/VRML/VRMLDesign html
Extensions iv

Labyrinth-VRML (Labyrinth Virtual Reality Markup Language Format)
A proposal for a VRML standard for distributing virtual reality worlds over the World Wide Web The original source for this document can be found at
http //vrml wired com/proposals/labspec html
Extensions vrml

SDML (Spacial Data Modeling Language)
Used by the CLRMosaic package for Silicon Graphics workstations The updated version of the SDML format can be obtained from
http //www clr toronto edu 1080/CLRMOSAIC/SDML html More information about CLRMosaic can be found at *http //www clr toronto edu 1080/CLRMOSAIC/help-about html*
Extensions sdml

VRML (VRML Format)
A proposal for a VRML standard based on SGI Inventor format The original source for this document can be found at *http //www eit com/vrml/vrmlspec html*
Extensions vrml

WebOOGL (Web Object Oriented Graphics Library Format)
WebOOGL is an extension of the OOGL format which allows URL links to be imbedded within 3D objects, and allows multiple WebOOGL objects from different locations on the Web to be combined into a single scene The original source for this document can be found at *http //www geom umn edu/docs/weboogl/weboogl html* More information about how OOGL can be used as a geometry format for VRML can be found at
http //vrml wired com/proposals/oogl html
Extensions oogl

Appendix C : Table of Figures

Appendix D: Program Listings

Morphit

copyit c
morphit c

SomhSimple app

SimpleCamera h
SimpleCamera m
SimpleShape h
SimpleShape m

Other Programs

joinribs c

```
/* copyit c  -- the program that will make a number of copies of a given RIB
    file inserting different values in the file at each instance

    Somhairle Foley    14th May 1993 - 31 Oct 1995

        example

    >morphit 145 rball -72

    this will take the rball rib file and make 145 copies of it
    with file names from rball001 rib     to rball145 rib

    the -72 part tells the program to start using the coordinates
    at -72  i e  values are to be treated as if from -72 to +72
    a total of 145 frames ( remember that 0 is a value too )

    the -72 part is optional  If it is left out, then the initial
    value is taken to be 1

         16/06/93

        Current assignments are
              %I     -   include another file
              %Ja b -   Interpolate(ints) a+((b-a)/num_files)
              %Ka b -   Interpolate(floats) a+((b-a)/num_files)
              %M     -   morphing the following Patchmes
              %R     -   the number of the file 1-num_files
              %S     -   the count +/- the offset
              %T     -   %S * 5
              %U     -   %S * 10
              %V     -   %S / 16
              %W     -   %S * 2 5
              %X     -   %S / 2
              %Y     -   %S / 5

*/
#include <stdio h>
#include <stdlib h>
#include <string h>

#define TRUE 0
#define FALSE 1

/* For adding morphing */
extern int  morphit( char*, FILE*, FILE*, int, int ),

/*extern char in_char, cur_word[40],
extern int  letter,
*/

char keyword[10] = "%S\0" ,

FILE *in_fp, *copy_fp,
FILE *save_fp,



int main(int argc, char* argv[])
{
  char  in_char, in_filename[50], copy_filename[50], copy_file_number[50],
  int   file_count, by_ten_count,
  int   num_frames, start_num,
  int   file_name_count = 0, including_file = FALSE,
  char  include_file[50], morph_file[50],
  float start_value, end_value, cur_value, scale_factor,
  char  string_value[32],
  int   string_value_count,

  if ( argc != 3 && argc != 4 )
  {
   fprintf(stderr, "Usage  morphit <number of frames>  <RIB filename> +/-[start number] \n"),
   exit(1),
  }

  if ( argc == 4 && argv[3][0] != '-' && argv[3][0] != '+'  )
  {
   fprintf(stderr, "Usage  morphit <number of frames>  <RIB filename> +/-[start number] \n"),
   exit(1),
  }

  num_frames = atoi( argv[1] ),
  if ( num_frames < 1 || num_frames > 199 )
  {
   fprintf(stderr, "morphit  number of frames must be between 1 and 199 \n"),
   exit(1),
  }

  strcpy( in_filename, argv[2] ),
```

```
  strcat( in_filename, " rib" ),
  if ((in_fp = fopen(in_filename, "r")) == NULL)
  {
   fprintf( stderr, "morphit  Cannot open input file %s\n",in_filename ),
   exit(1),
  }

  if ( argc == 4 )
  {
  start_num = atoi(argv[3]),
  if ( ( start_num < 1 && start_num < (-num_frames/2) ) || ( start_num > 1 ) )
    fprintf( stderr, "morphit  Warning  Number of frames is unbalanced  It will run from %03d to %3d \n",start
_num, start_num+num_frames-1),
  }
  else
  start_num = 1,


  for ( file_count = 1, file_count <= num_frames , file_count++ )
  {
  rewind( in_fp ),
  sprintf( copy_filename, "%s%03d rib\0", argv[2], file_count ),

  if ((copy_fp = fopen( copy_filename, "w")) == NULL )
   {
     fprintf( stderr, "morphit  Cannot open output file %s\n",copy_filename ),
     exit(1),
   }

  printf( "%03d   Making Frame File %s   current translations are %03d\n", file_count, copy_filename, start_nu
m+file_count-1 ),

  while ( 'feof(in_fp) || including_file == TRUE )
  {


    /* Check if we are including a file and it has come to the eof */

  if ( including_file == TRUE && feof(in_fp) )
  {
    fclose( in_fp ),
    in_fp = save_fp,
    including_file = FALSE,
  }

  in_char = fgetc(in_fp),
  if ( in_char == '%' )
  {
    in_char = fgetc(in_fp),

      if ( in_char == 'I' )
      {
      file_name_count = 0,
      strncpy( include_file, "\0", 30 ),
      while ( (in_char = fgetc(in_fp)) '= '\n' && file_name_count < 30 )
      {
         include_file[file_name_count++] = in_char,
      }

      save_fp = in_fp,
      if ( (in_fp = fopen( include_file, "r" )) == NULL )
      {
        fprintf(stderr,"morphit  Warning - cannot open include file %s \n",include_file),
        in_fp = save_fp,
        including_file = FALSE,  /* this line shouldn't be necessary */
      }
      else
        including_file = TRUE,

      }
    else if ( in_char == 'J' )
      {
      /* For integer key-value interpolation, read in the rest of line
         as start value, colon, end value
      */

      start_value = -999, end_value = -999,
      /* Read in Start Value */

      string_value_count = 0,
      strncpy( string_value, "\0", 30 ),
      while ( (in_char = fgetc(in_fp)) '= '\n' && in_char '= ' ' && in_char '= ' ' && string_value_count < 30
)
      {
         string_value[string_value_count++] = in_char,
      }

      if ( string_value_count >= 30 )
```

```
    {
      fprintf(stderr,"Error reading \045K Keyframe start value\n"),
      break,
    }
    start_value = atof(string_value),

    if ( in_char == '\n' )
    {
      fprintf(stderr,"Error reading \045K Keyframe no colon or end value\n"),
      break,
    }

    /* Read in End Value */
    string_value_count = 0,
    strncpy( string_value, "\0", 30 ),
    while ( (in_char = fgetc(in_fp)) != '\n' && in_char != ' ' && string_value_count < 30 )
    {
       string_value[string_value_count++] = in_char,
    }

    if ( string_value_count >= 30 )
    {
      fprintf(stderr,"Error reading \045K Keyframe end value\n"),
      break,
    }
    end_value = atof(string_value),

    printf("Read Keyframe values start %f end %f ", start_value, end_value),
    /* DO SOMETHING */

    scale_factor= ((float)file_count-1)/((float)num_frames-1),
    cur_value = start_value + ((end_value-start_value)*scale_factor),

    printf("cur_value (integer) is %d \n",(int)cur_value),
    fprintf(copy_fp," %d ", (int)cur_value),

    /* If %J is last value on line, output a newline character */
    if (in_char == '\n' )
      fputc('\n',copy_fp),

    }
  else if ( in_char == 'K' )
    {
    /* For floating-point key-value interpolation, read in the rest
       of line as start value, colon, end value  Values are floats
    */

    start_value = -999, end_value = -999,
    /* Read in Start Value */

    string_value_count = 0,
    strncpy( string_value, "\0", 30 ),
    while ( (in_char = fgetc(in_fp)) != '\n' && in_char != ' ' && in_char != ' ' && string_value_count < 30
)
    {
       string_value[string_value_count++] = in_char,
    }

    if ( string_value_count >= 30 )
    {
      fprintf(stderr,"Error reading \045K Keyframe start value\n"),
      break,
    }
    start_value = atof(string_value),

    if ( in_char == '\n' )
    {
      fprintf(stderr,"Error reading \045K Keyframe no colon or end value\n"),
      break,
    }

    /* Read in End Value */
    string_value_count = 0,
    strncpy( string_value, "\0", 30 ),
    while ( (in_char = fgetc(in_fp)) != '\n' && in_char != ' ' && string_value_count < 30 )
    {
       string_value[string_value_count++] = in_char,
    }

    if ( string_value_count >= 30 )
    {
      fprintf(stderr,"Error reading \045K Keyframe end value\n"),
      break,
    }
    end_value = atof(string_value),

    printf("Read Keyframe values start %f end %f ", start_value, end_value),
    /* DO SOMETHING */
```

```
  scale_factor= ((float)file_count-1)/((float)num_frames-1),
  cur_value = start_value + ((end_value-start_value)*scale_factor),

  printf("cur_value is %f \n",cur_value),
  fprintf(copy_fp," %f ", cur_value),

  /* If %K is last value on line, output a newline character */
  if (in_char == '\n' )
    fputc('\n',copy_fp),

  }
else if ( in_char == 'M' )
  {
  /* For morphing, read in the rest of line as morph file name
     and send it to morphit()  This should leave the fp back
     at the start of the line following the next RIB primative
  */

  file_name_count = 0,
  strncpy( morph_file, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 30 ),
  while ( (in_char = fgetc(in_fp)) != '\n' && file_name_count < 30 )
  {
     morph_file[file_name_count++] = in_char,
  }

  /* Call the morphit() function in morphit c file */
  morphit( morph_file, in_fp, copy_fp, file_count, num_frames ),

  }
else if ( in_char == 'R' )
  {
  sprintf( copy_file_number, "%03d\0", file_count ),
  fputc( copy_file_number[0], copy_fp ),
  fputc( copy_file_number[1], copy_fp ),
  fputc( copy_file_number[2], copy_fp ),
      }
  else if ( in_char == 'S' )
  {
    sprintf( copy_file_number, "%03d\0", start_num+file_count-1 ),
fputc( copy_file_number[0], copy_fp ),
    fputc( copy_file_number[1], copy_fp ),
        fputc( copy_file_number[2], copy_fp ),
  }
  else if ( in_char == 'T' )
      {
        sprintf( copy_file_number, "%04d\0", (start_num+file_count-1)*5 ),
        fputc( copy_file_number[0], copy_fp ),
        fputc( copy_file_number[1], copy_fp ),
        fputc( copy_file_number[2], copy_fp ),
  fputc( copy_file_number[3], copy_fp ),
  }
  else if ( in_char == 'U' )
  {
  by_ten_count = (start_num+file_count-1) * 10,
  while ( by_ten_count > 360 )
     by_ten_count -= 360,
  sprintf( copy_file_number, "%04d\0", by_ten_count ),
  fputc( copy_file_number[0], copy_fp ),
  fputc( copy_file_number[1], copy_fp ),
  fputc( copy_file_number[2], copy_fp ),
  fputc( copy_file_number[3], copy_fp ),
  }
  else if ( in_char == 'V' )
  {
   strncpy( copy_file_number, "\0", 10 ),
   sprintf( copy_file_number, "%0 4f\0", (start_num+file_count-1) / 16 0 ),
   if ( copy_file_number[0] != '\0' ) fputc( copy_file_number[0], copy_fp ),
   if ( copy_file_number[1] != '\0' ) fputc( copy_file_number[1], copy_fp ),
   if ( copy_file_number[2] != '\0' ) fputc( copy_file_number[2], copy_fp ),
   if ( copy_file_number[3] != '\0' ) fputc( copy_file_number[3], copy_fp ),
   if ( copy_file_number[4] != '\0' ) fputc( copy_file_number[4], copy_fp ),
   if ( copy_file_number[5] != '\0' ) fputc( copy_file_number[5], copy_fp ),
   if ( copy_file_number[6] != '\0' ) fputc( copy_file_number[6], copy_fp ),
   if ( copy_file_number[7] != '\0' ) fputc( copy_file_number[7], copy_fp ),
  }
else if ( in_char == 'W' )
  {
  sprintf( copy_file_number,  %4 1f\0"  (start_num+file_count-1)*2 5 ),
  fputc( copy_file_number[0]  copy_fp ),
  fputc( copy_file_number[1], copy_fp ),
  fputc( copy_file_number[2], copy_fp ),
  fputc( copy_file_number[3], copy_fp ),
  }
  else if ( in_char == 'X' )
  {
   strncpy( copy_file_number, "\0", 10 ),
   sprintf( copy_file_number, "%0 4f\0", (start_num+file_count-1) / 2 0 ),
```

```
      if ( copy_file_number[0] '= '\0' ) fputc( copy_file_number[0], copy_fp ),
      if ( copy_file_number[1] '= '\0' ) fputc( copy_file_number[1], copy_fp ),
      if ( copy_file_number[2] '= '\0' ) fputc( copy_file_number[2], copy_fp ),
      if ( copy_file_number[3] '= '\0' ) fputc( copy_file_number[3], copy_fp ),
      if ( copy_file_number[4] '= '\0' ) fputc( copy_file_number[4], copy_fp ),
      if ( copy_file_number[5] '= '\0' ) fputc( copy_file_number[5], copy_fp ),
      if ( copy_file_number[6] '= '\0' ) fputc( copy_file_number[6], copy_fp ),
      if ( copy_file_number[7] '= '\0' ) fputc( copy_file_number[7], copy_fp ),
     }
     else if ( in_char == 'Y' )
     {
      strncpy( copy_file_number, "\0", 10 ),
      sprintf( copy_file_number, "%0 4f\0", (start_num+file_count-1) / 5 0 ),
      if ( copy_file_number[0] '= '\0' ) fputc( copy_file_number[0], copy_fp ),
      if ( copy_file_number[1] '= '\0' ) fputc( copy_file_number[1], copy_fp ),
      if ( copy_file_number[2] '= '\0' ) fputc( copy_file_number[2], copy_fp ),
      if ( copy_file_number[3] '= '\0' ) fputc( copy_file_number[3], copy_fp ),
      if ( copy_file_number[4] '= '\0' ) fputc( copy_file_number[4], copy_fp ),
      if ( copy_file_number[5] '= '\0' ) fputc( copy_file_number[5], copy_fp ),
      if ( copy_file_number[6] '= '\0' ) fputc( copy_file_number[6], copy_fp ),
      if ( copy_file_number[7] '= '\0' ) fputc( copy_file_number[7], copy_fp ),
     }
     else
    {

    fputc( '%', copy_fp ),
    fputc( in_char, copy_fp ),
    }
  }
    else if ( 'feof(in_fp) )             /* Stops the end-of-file markers being written */
         fputc( in_char, copy_fp ),       /* write out the char */

    }                          /* end while 'feof(in_fp) */
    fclose( copy_fp ),
  }            /* end for */

   printf("=================================================================\n"),
   printf("morphit  Finished Ok   %s rib  %d frames\n\n" ,argv[2],num_frames),

  return( fclose( in_fp ) ),

}
```

```
/***** This file must be linked with copyit c to run properly *****/
/*                                                                */
/* morphit c - does the object analysis/morphing when required    */
/*                                                                */
/* Somhairle Foley  Created June 1994                             */
/*          Modified up to July 1996  for bugs  and upgrades      */
/******************************************************************/

#include <stdio h>
#include <stdlib h>
#include <ctype h>
#include <string h>

#define TRUE                1
#define FALSE               0

#define Copy3DPoints(dest,src) (dest) point[0]=(src) point[0], (dest) point[1]=(src) point[1], (dest) point[2]
=(src) point[2],

#define NUM_KEYWORDS        21

#define MAX_3D_POINTS       130
/*  This is set to 130 or 400 beecause the PC can't handle anything bigger
  It needs to be reset for UNIX  Meshes of 41x11 are too big for PC
*/

#define NOTHING            -1

#define SPHERE              0
#define CYLINDER            1
#define CONE                2
#define DISK                3
#define POLYGON             4
#define GEN_POLYGON         5
#define TORUS               6
#define HYPERBOLOID         7
#define PATCH               8
#define PATCHMESH           9
#define PARABOLOID          10
#define ROTATE              11
#define SCALE               12
#define TRANSLATE           13
#define ATTRIBUTEBEGIN      14
#define ATTRIBUTEEND        15
#define TRANSFORMBEGIN      16
#define TRANSFORMEND        17
#define COLOR               18
#define ATTRIBUTE           19
#define GEOMETRIC_REP       20



/* for storing a whole object as a list of points for interpolate */

typedef struct
    {
/* Temporarily treating this as a float   double point[3], */
    float point[3],
    } pointlist,


/* for storing the details at the start of a Patchmesh description */
typedef struct
    {
    char meshtype[14],              /* 'bicubic' or 'bilinear' */
    int  num1,
    char period1[16],               /* 'periodic' or 'nonperiodic' */
    int  num2,
    char period2[16],
    char pointtype[4],              /* Always "P" */
    } patchmesh_record,

/* for storing the details at the start of a Patch description */
typedef struct
    {
    int meshtype,
    int pointtype,
    } patch_record,

/* for storing the details of a Sphere */
typedef struct
    {
    float radius,               /* radius of Sphere */
    float zplus,                /* pos  distance for z-axis 'slice'   */
    float zminus,               /* neg  distance for z-axis 'slice'   */
    float z_sweep,              /* degree of sweep of object  <= 360 */
    } sphere_record,
```

```
typedef union
    {
    patchmesh_record mesh,
    patch_record     patch,
    sphere_record    sphere,
    } object_record,


char keywords[NUM_KEYWORDS][24] = {
        "Sphere", "Cylinder", "Cone", "Disk", "Polygon","GeneralPolygon",
        "Torus", "Hyperboloid", "Patch", "PatchMesh",
        "Paraboloid", "Rotate", "Scale", "Translate",
        "AttributeBegin", "AttributeEnd",
        "TransformBegin", "TransformEnd",
        "Color", "Attribute", "GeometricRepresentation"
        },


char cont,      /* used to allow users to press 'QQQ' once */

/* These 'anal' variables are to allow someone to decide not to see any more
   analysis of Attribute, Patch and Patchmesh statements respectively
   They have to be global because the functions that they are used in would
   not keep track of their values ( as local vars )
*/

int name_anal = TRUE,
int patch_anal = TRUE,
int mesh_anal = TRUE,


int read_primative( FILE* fp, int* primative_type, pointlist* the_pointlist, object_record* details ),
int write_primative( FILE* fp, int primative_type, pointlist* the_pointlist, object_record* details ),
int addextrapoints( object_record* source_obj, pointlist* start_points, object_record* target_obj ),


/*  You can't win' When someone says they want 5 frames  That means that
  they want to have the start points in Frame 1 and the end points in
  Frame 5  This doesn't add up  This leaves only Frames 2,3 and 4
  A total of 3 frames  Since we are actually only interested in changes
  in this function, this means that we have 4 different sets
  1->2, 2->3, 3->4, 4->5

  This means that we don't use mult/div ( the current frame / total number
  of frames )  We use (mult-1)/(div-1)
*/

void interpolate( pointlist *start, pointlist *end, pointlist* interp, int mult, int div )
{
  int   cur_point = 0,
  float scale_factor,

  scale_factor = mult-1,
  scale_factor /= (div-1),

  /* Interpolation is carried out until the same point in the pointlists
    are equal to 0 - this may have to be changed to get the size of a list
        from the object_record of the objects
  */

  while('(start[cur_point] point[0] == 0 && end[cur_point] point[0] == 0 &&
      start[cur_point] point[1] == 0 && end[cur_point] point[1] == 0 &&
      start[cur_point] point[2] == 0 && end[cur_point] point[2] == 0) )
  {
    interp[cur_point] point[0] = start[cur_point] point[0] + ((end[cur_point] point[0]-start[cur_point] point[
0])*scale_factor),
    interp[cur_point] point[1] = start[cur_point] point[1] + ((end[cur_point] point[1]-start[cur_point] point[
1])*scale_factor),
    interp[cur_point] point[2] = start[cur_point] point[2] + ((end[cur_point] point[2]-start[cur_point] point[
2])*scale_factor),
    cur_point++,
  }

/*  printf(""),
*/
}


  /* This all assumes there are more target points than source points */
  /* We want to have this function take in the pointlist and find out how
    much the different is between the source and target is when it comes to
    points per row/colomn

    The initial algorithm will be
    mutliply source num1*num2 and take that away from target num1*num2
    Decide how many times the difference can cover the source
    (i e  double,triple,etc  the source points )
```

```
    Add the left over points to the start points
    Double, Triple, Quadruple, etc., all the points
        Reset the last point to 0,0,0

        This means that points will only be duplicated - no interpolation
    happens
    11/08/94
    This is a brute-force answer to the problem. But duplicating
    the points is a complex problem. You must be sensitive to what
    points you are changing

    To change a 13x10 patchmesh to a 14*10 patchmesh,
        duplicate every 13th point
    To change a 14x10 patchmesh to a 14*11 patchmesh,
        duplicate the last 14 points

    => To change a 13x10 to a 41x11
                putting them all at the start/end points
      duplicate every 13th point 28 times and
      duplicate the last 41 points 1 time

    this gives us    duplicate every source num1 point
                (target num1-source num1) times
          and  duplicate the last target num1 points
                (target num2-source num2) times

    and we do this by
      copying the first source num1 points over
      duplicating the source num1th point diff_num1 times
      repeating the above procedure source num2-1 times
      on the source num2th time, record the values(target num1 of them)
      duplicate the target num1 values diff_num2 times
  */


int addextracolumns( object_record* source_obj, pointlist* start_points, object_record* target_obj )
{
  int     diff_num1, src_offset;
  pointlist temp_points[(2*MAX_3D_POINTS)];
  pointlist dup_point;
  int     temp_count, src_count, row_count;

  /* Remember  NUM1 is columns, NUM2 is rows  */

  diff_num1 = ((target_obj->mesh).num1 - (source_obj->mesh).num1 );

  temp_count = 0;        /* This is the counter for temp_points */
  row_count = 0;       /* This counts how many rows are processed */

  /* Repeat for all the rows in the source mesh */

  while ( row_count < (source_obj->mesh).num2 )
  {
    /* First copy the source 'row' over to temp */
    for ( src_count=0; src_count < (source_obj->mesh).num1; src_count++ )
    {
      src_offset = (row_count * (source_obj->mesh).num1) + src_count;

      Copy3DPoints(temp_points[temp_count],start_points[src_offset]);
      temp_count++;
    }


    /* Save the last point in the row */
    Copy3DPoints(dup_point,start_points[src_offset]);

    /* Now add the points to the end of the row */
    for ( src_count=0; src_count < diff_num1; src_count++ )
    {
      Copy3DPoints(temp_points[temp_count],dup_point);
      temp_count++;
    }

    row_count++;
  }

  /* Check that the right number of points are generated */
  if ( temp_count != (target_obj->mesh).num1 * (source_obj->mesh).num2 )
  {
    fprintf(stderr, "\7\7\7morphit: addextracolumns count error \n\t temp_count is %d, and ",temp_count);
    fprintf(stderr,"target for points is %d\n",(target_obj->mesh).num1 * (source_obj->mesh).num2);
    return(0);
  }

  /* Copy temp_points into start_points */
  for ( src_count=0; src_count<temp_count; src_count++ )
  {
    Copy3DPoints(start_points[src_count],temp_points[src_count]);
```

```
  }

  /* A Point {0,0,0} marks the end of the pointlist */
  start_points[src_count] point[0] = 0,
  start_points[src_count] point[1] = 0,
  start_points[src_count] point[2] = 0,

  /* Update mesh size in object record */
  (source_obj->mesh) num1 = (target_obj->mesh) num1,

  return(1),
}

int addextrarows( object_record* source_obj, pointlist* start_points, object_record* target_obj )
{
  int      diff_num2, src_offset,
  pointlist temp_points[(2*MAX_3D_POINTS)],
  pointlist dup_line[MAX_3D_POINTS],
  int      temp_count, src_count, loc_last_row, dup_count, row_count,

  /* Remember  NUM1 is columns, NUM2 is rows  */

  diff_num2 = ((target_obj->mesh) num2 - (source_obj->mesh) num2 ),
  temp_count = 0,        /* This is the counter for temp_points */
  row_count = 0,       /* This counts how many rows are processed */

  /* Repeat for all the rows in the source mesh */

  while ( row_count < (source_obj->mesh) num2 )
  {
    /* First copy the source 'row' over to temp */
    for ( src_count=0, src_count < (source_obj->mesh) num1, src_count++ )
    {
      src_offset = (row_count * (source_obj->mesh) num1) + src_count,

      Copy3DPoints(temp_points[temp_count],start_points[src_offset]),
      temp_count++,
    }

    row_count++,

    /* Check if this is starting the last row */
    if ( row_count == (source_obj->mesh) num2 - 1 )
      loc_last_row = temp_count,
  }

  /* Now save the entire last row of the modified source */
  for ( dup_count=0, dup_count<(source_obj->mesh) num1, dup_count++ )
  {
    Copy3DPoints(dup_line[dup_count],temp_points[loc_last_row+dup_count]),
  }

  /* Duplicate the last row diff_num2 times */
  for ( row_count=0, row_count < diff_num2, row_count++ )
    for ( dup_count=0, dup_count<(source_obj->mesh) num1, dup_count++ )
    {
      Copy3DPoints(temp_points[temp_count],dup_line[dup_count]),
      temp_count++,
    }

  /* Check that the right number of points are generated */
  if ( temp_count != (source_obj->mesh) num1 * (target_obj->mesh) num2 )
  {
    fprintf(stderr,"\7\7\7morphit  addextrarows count error \n\t temp_count is %d, and ",temp_count),
    fprintf(stderr,"target for points is %d\n",(source_obj->mesh) num1 * (target_obj->mesh) num2),
    return(0),
  }

  /* Copy temp_points into start_points */
  for ( src_count=0, src_count<temp_count, src_count++ )
  {
    Copy3DPoints(start_points[src_count],temp_points[src_count]),
  }

  /* A Point {0,0,0} marks the end of the pointlist */
  start_points[src_count] point[0] = 0,
  start_points[src_count] point[1] = 0,
  start_points[src_count] point[2] = 0,

  /* Update mesh size in object record */
  (source_obj->mesh) num2 = (target_obj->mesh) num2,

  return(1),

}

/*  REMOVEEXTRAPOINTS is a function which takes the details of two objects and a list of
  3D points as its parameters  The list of 3D points is in the form specified in the
```

```
  object record CURRENT  Its goal is to remove extra rows and columns of 3D points so
  that the 3D point list is in the form specified in the object record REQUIRED

  The point list is a 2D array of 3D points which specify a patchmesh  The dimensions
  of the 2D array are given in an object record (patchmesh record) as NUM1 and NUM2
  NUM1 refers to the U vector and NUM2 refers to the V vector  For the purposes of this
  code, NUM1 is columns and NUM2 is rows

  This function has a number of large problems

  It assumes that the required mesh has both less rows and less columns than the current
  mesh  i e  a 41x11 -> 13x10 is fine, but a 13x10 -> 10x13 causes problems

  It removes rows and columns by duplicating all but the last row and column in the
  required mesh with the indentical points from the row/columns in the current  The
  last point on each row of the required mesh is set to be an average of all the
  additional points on that row in the current mesh  Similarly, the last row of the
  required mesh is set to be an average of all the remaining rows on the current

  current mesh   41x11      ->  remove extra columns  13x11

  1,1  2,1  3,1           40,1  41,1  ->  1,1  2,1  3,1          12,1  avg(13,1-41,1)
  1,2  2,2  3,2           40,2  41,2  ->  1,2  2,2  3,2          12,2  avg(13,2-41,2)
  1,3  2,3  3,3           40,3  41,3  ->  1,3  2,3  3,3          12,3  avg(13,3-41,3)
                                      ->
                                      ->
                                      ->
  1,10 2,10 3,10          40,10 41,10 ->  1,10 2,10 3,10         12,10  avg(13,10-41,10)
  1,11 2,11 3,11          40,11 41,11 ->  1,11 2,11 3,11         12,11  avg(13,11-41,11)

  remove extra rows  13x7

      1,1           2,1           3,1                 12,1                avg(13,1-41,1)
      1,2           2,2           3,2                 12,2                avg(13,2-41,2)
      1,3           2,3           3,3                 12,3                avg(13,3-41,3)


      1,6           2,6           3,6                 12,6                avg(13,6-41,6)
  avg(1,7-1,11) avg(2,7-2,11) avg(3,7-3,11)       avg(12,7-12,11) avg(avg(13,7-41,7)-avg(13,11-41,11))

  It is obvious that the reduced size mesh will almost definitely not look like the original
  except for meshes whose last rows and columns of control points are positioned very close
  to eachother  Some method of approximating a larger mesh with a smaller mesh is required
  OR how about always using the larger mesh ? But what about morphing ?

  This why this method can be used  The mesh size should not "jump"  It, like everything
  else should be linked to the interpolation phase of morphing  The mesh size should
  be different for each frame and hence each mesh size would only change slightly
  Which is what this method is good at !!!!
  But will this work  ???
*/


int removeextracolumns( object_record* current, object_record* required, pointlist* the_mesh )
{
  int                  diff_cols, row_offset,
  int                  row_count, col_count, temp_count, src_count,
  pointlist      temp_points[(2*MAX_3D_POINTS)],
  pointlist      last_point,

  /* Remember  NUM1 is columns, NUM2 is rows  */
  diff_cols =  (current->mesh) num1 - (required->mesh) num1 + 1,

  /* Do the columns first */
  row_count = 0,       /* This counts how many rows are processed */
  temp_count = 0,

  /* Repeat for all the rows in the source mesh */

  while ( row_count < (current->mesh) num2 )
  {
    /* First copy all the points on each row except the last point thats required */
    for ( col_count=0, col_count < ((required->mesh) num1 - 1), col_count++ )
    {
      row_offset = row_count * ((current->mesh) num1 ),
      Copy3DPoints(temp_points[temp_count],
              the_mesh[row_offset+col_count]),
      temp_count++,
    }

    /* Now calculate the average of the remaining points on this row */

    last_point point[0] = last_point point[1] = last_point point[2] = 0 0,

    for ( col_count = ((required->mesh) num1-1), col_count < (current->mesh) num1, col_count++ )
    {
/*      Copy3DPoints(last_point,the_mesh[row_offset+col_count]),
*/      last_point point[0] += the_mesh[row_offset+col_count] point[0],
```

```
      last_point point[1] += the_mesh[row_offset+col_count] point[1],
      last_point point[2] += the_mesh[row_offset+col_count] point[2],
    }

    if ( diff_cols != 0 )       /* VIP - In case of division by zero (NAN) */
    {
      last_point point[0] = last_point point[0] / (float)diff_cols,
      last_point point[1] = last_point point[1] / (float)diff_cols,
      last_point point[2] = last_point point[2] / (float)diff_cols,
    }

    /* Set the last point on the row to the average of all the others */

    Copy3DPoints(temp_points[temp_count], last_point),
    temp_count++,
/*    Copy3DPoints(temp_points[row_offset+(required->mesh) num1-1],
            last_point),
*/
    /* Thats this row completed  Now continue while loop for each row */
    row_count++,
  }

  /* Check that the right number of points are generated */
  if ( temp_count != (required->mesh) num1 * (current->mesh) num2 )
  {
    fprintf(stderr,"\7\7\7morphit  removeextracolumnss count error \n\t temp_count is %d, and ",temp_count),
    fprintf(stderr,"target for points is %d\n",(required->mesh) num1 * (current->mesh) num2),
    return(0),
  }

  /* Copy temp_points back into the_mesh */
  for ( src_count=0, src_count<temp_count, src_count++ )
  {
    Copy3DPoints(the_mesh[src_count],temp_points[src_count]),
  }

  /* A Point {0,0,0} marks the end of the pointlist */
  the_mesh[src_count] point[0] = 0 0,
  the_mesh[src_count] point[1] = 0 0,
  the_mesh[src_count] point[2] = 0 0,

  /* Update mesh size in object record */
  (current->mesh) num1 = (required->mesh) num1,

  return(1),

}


int removeextrarows( object_record* current, object_record* required, pointlist* the_mesh )
{
  int                 diff_rows, row_offset,
  int                 row_count, col_count, src_count, temp_count,
  pointlist     temp_points[(2*MAX_3D_POINTS)],
  pointlist     last_row[MAX_3D_POINTS],

  /* Remember  NUM1 is columns, NUM2 is rows  */
  diff_rows =  (current->mesh) num2 - (required->mesh) num2 + 1,

  /* Do the columns first */
  row_count = 0,      /* This counts how many rows are processed */
  temp_count = 0

  /* Repeat for all the rows in the target mesh except the last row */

  while ( row_count < ((required->mesh) num2 - 1) )
  {
    /* First copy all the points on each row */
    for ( col_count=0, col_count < (current->mesh) num1, col_count++ )
    {
      row_offset = row_count * ((current->mesh) num1 ),
      Copy3DPoints(temp_points[temp_count], the_mesh[row_offset+col_count]),
      temp_count++,
    }

    /* Thats this row completed  Now continue while loop for next row */
    row_count++,
  }

  /* Now the TEMP_POINTS array is filled, the extra rows need to be
    eliminated and averaged out in the last row
  */

  /* Clear the LAST_ROW array before calculating averages */
  for ( col_count = 0, col_count < (required->mesh) num1, col_count++ )
  {
    last_row[col_count] point[0] = 0 0,
    last_row[col_count] point[1] = 0 0,
```

```
    last_row[col_count] point[2] = 0 0,
  }

  /* Now get the sum per column of the points on the remaining rows */

  for ( row_count = ((required->mesh) num2-1), row_count < (current->mesh) num2, row_count++ )
  {
    row_offset = row_count * (required->mesh) num1,
/* This causes a floating point error - calculating the not-needed bit maybe ??
*/
    for ( col_count = 0, col_count < (required->mesh) num1, col_count++ )
    {
      last_row[col_count] point[0] += the_mesh[row_offset+col_count] point[0],
      last_row[col_count] point[1] += the_mesh[row_offset+col_count] point[1],
      last_row[col_count] point[2] += the_mesh[row_offset+col_count] point[2],
    }
  }

  /* Put the average of the remaining rows into last row of SECOND_TEMP_POINTS */
  for ( col_count = 0, col_count < (required->mesh) num1, col_count++ )
  {
/*    temp_points[(((required->mesh) num2-1)*(required->mesh) num1)+col_count] point[0] = last_row[col_count]
point[0] / (float)diff_rows,
    temp_points[(((required->mesh) num2-1)*(required->mesh) num1)+col_count] point[1] = last_row[col_count] po
int[1] / (float)diff_rows,
    temp_points[(((required->mesh) num2-1)*(required->mesh) num1)+col_count] point[2] = last_row[col_count] po
int[2] / (float)diff_rows,
*/
    if (diff_rows != 0)    /* VIP to check for divide by zero = NAN errors */
    {
      temp_points[temp_count] point[0] = last_row[col_count] point[0] / (float)diff_rows,
      temp_points[temp_count] point[1] = last_row[col_count] point[1] / (float)diff_rows,
      temp_points[temp_count] point[2] = last_row[col_count] point[2] / (float)diff_rows,
      temp_count++,
    }
  }

  /* Check that the right number of points are generated */
  if ( temp_count != (required->mesh) num2 * (current->mesh) num1 )
  {
    fprintf(stderr,"\7\7\7morphit  removeextrarows count error \n\t temp_count is %d, and ",temp_count),
    fprintf(stderr,"target for points is %d\n",(required->mesh) num2 * (current->mesh) num1),
    return(0),
  }

  /* Temp_Points is now a mesh of the appropriate size  It needs to
    be copied back to The_Mesh, so it can be accessed from outside
  */

  for ( src_count=0, src_count<temp_count, src_count++ )
  {
    Copy3DPoints(the_mesh[src_count],temp_points[src_count]),
  }

  /* A Point {0,0,0} marks the end of the pointlist */
  the_mesh[src_count] point[0] = 0 0,
  the_mesh[src_count] point[1] = 0 0,
  the_mesh[src_count] point[2] = 0 0,

  /* Update mesh size in object record */
  (current->mesh) num2 = (required->mesh) num2,

  return(1),
}



int read_patchmesh( FILE* fp, pointlist* the_pointlist, object_record* the_details )
{
  unsigned char ch,
  int meshopen = FALSE,
  int point_count = 0, num_points = 0,
  char detail_string[20], num_string[20],
  int str_index, detail_count = 1,
  int num1, num2,
  double the_point[3],


/*  int pl_index = 0,              /* index for point_list */


  if ( mesh_anal == TRUE )
   {
/*     printf("Found a PatchMesh\n"),
*/     ch = fgetc(fp), num1 = 0, num2 = 0, str_index = 0,

    while( meshopen != TRUE )       /* Repeat  Until a '[' is found */
    {
```

```
/*     printf( "%c", ch ),
*/     if ( ch == '[' )
      {
        meshopen = TRUE,
/*        printf("\n"),
*/      }
     else if ( ch == ' ' )
      {
        /* Assumes that is always like "bicubic 10 periodic 13 periodic P" */
        detail_string[str_index] = '\0',
        switch ( detail_count )
        {
        case 1  strcpy((the_details->mesh) meshtype, detail_string ),
            break,
        case 2  num1 = (the_details->mesh) num1 = atoi( detail_string ),
            break,
        case 3  strcpy((the_details->mesh) period1, detail_string ),
            break,
        case 4  num2 = (the_details->mesh) num2 = atoi( detail_string ),
            break,
        case 5  strcpy((the_details->mesh) period2, detail_string ),
            break,
        case 6  strcpy((the_details->mesh) pointtype, detail_string ),
            /*(the_details->mesh) pointtype = '\0',*/
            break,
        }

        detail_count++,
        str_index = 0,
      }
     else
      detail_string[str_index++] = ch,         /* End of Repeat until '[' */

     ch = fgetc( fp ),
    }

/*     printf("\n Num1 is %d, Num2 is %d \n",num1,num2),
*/

    num_string[0] = '\0', str_index = 0,
    while( meshopen == TRUE )          /* Repeat  Until a ']' is found */
    {
      if ( ch != '\n' && ch != '\t' && ch != ']' && ch != ' ' )
        num_string[str_index++] = ch,

      if ( ch == ' ' || ch == '\t' || ch == '\n' || ch == ']' )
      {
        if ( str_index != 0 )
        {
          point_count++,
          num_string[str_index] = '\0',
          the_point[point_count-1] = atof(num_string),
          str_index = 0,
          num_string[str_index] = '\0',

          if ( point_count >= 3 )        /* We've read in one 3D point */
            {
             if ( num_points > MAX_3D_POINTS )
              {
                printf("\nToo many 3D Points encountered\n"),
                exit(2),
              }

             for ( point_count = 0, point_count < 3, point_count++ )
             {
              the_pointlist[num_points] point[point_count] = the_point[point_count],
             }

             point_count = 0,
             num_points++,
             the_point[0] = the_point[1] = the_point[2] = 0,
            }

        }

        if ( ch == ']' )
        {
          meshopen = FALSE,
          the_pointlist[num_points] point[0] = 0 0,
          the_pointlist[num_points] point[1] = 0 0,
          the_pointlist[num_points] point[2] = 0 0,
        }
       }

    ch = fgetc(fp),
    }

/*     for ( pl_index = 0, pl_index < num_points, pl_index++ )
```

```
    {
     printf(" %0.8f %0.8f %0.8f\n ", the_pointlist[pl_index].point[0], the_pointlist[pl_index].point[1], the_p
ointlist[pl_index].point[2]);
    }

*/
    /* We've read in the entire PatchMesh */
/*     printf("]       ------ Number of 3D Points counted: %d\n",num_points);
*/     if ( num_points != ( num1*num2 ) )
      {
       printf("\7\7\7\7 Error: incorrect number of 3D Points counted.\n");
       printf("   it should be %d * %d = %d \n", num1, num2, num1*num2 );
       return(-1);
      }
/*     if ( (cont = getchar()) == 'q' || cont == 'Q' )
       mesh_anal = FALSE;
*/    }

   return( num_points );
}




write_patchmesh( FILE* fp, pointlist* the_pointlist, object_record* the_details )
{
  int num_points, point_count;

  fprintf( fp, "%s %d %s %d %s %s [ \n",
       (the_details->mesh).meshtype, (the_details->mesh).num1,
       (the_details->mesh).period1, (the_details->mesh).num2,
       (the_details->mesh).period2, (the_details->mesh).pointtype );


  num_points = (the_details->mesh).num1 * (the_details->mesh).num2;

  for ( point_count = 0; point_count < num_points; point_count++ )
  {
  fprintf( fp, "%0.8f %0.8f %0.8f ", the_pointlist[point_count].point[0],
                the_pointlist[point_count].point[1],
                the_pointlist[point_count].point[2] );

  if ( ((point_count/4) == (point_count/4.0)) && point_count != num_points-1)
    fprintf( fp, "\n");
  }

  fprintf( fp, "]\n");

  return(1);
}

/* morphit() is required to leave the file pointer in the input file at the
   start of the line following the primative that starts at the current
   file pointer. i.e. it must scan in the primative

   It must also write the primative out to the output file, making the
   diifferences in the primatives points for the appropriate frame number
*/



int read_primative( FILE* fp, int* primative_type, pointlist* the_pointlist, object_record* the_details )
{
  char in_char;
  int  cur_let = 0, count;
  char cur_word[40];

  in_char = fgetc(fp);
  *primative_type = NOTHING;

  /* Read in a word from current line */
  while( in_char != ' ' && in_char != '\n' && !feof(fp) && cur_let < 30 )
  {
  cur_word[cur_let++] = in_char;
  in_char = fgetc(fp);
  }

  /* Check if word matches a keyword */
  if ( cur_let < 30 )
  {
   cur_word[cur_let] = '\0';
   for ( count=0 ; count< NUM_KEYWORDS; count++ )
   {
    if ( !(strcmp( cur_word, keywords[count])) )  /* compare word with keyword*/
    {
      *primative_type = count;         /* count is the code for a primative */
      count = NUM_KEYWORDS;            /* break out of for loop */
```

```
    }
   }
  }

  switch ( *primative_type )
  {
   case PATCHMESH   read_patchmesh( fp, the_pointlist, the_details ),
             break,
/*   case PATCH       patch_points( the_pointlist ),
             break,
   case POLYGON     polygon_points( the_pointlist ),
             break,
*/   default          return(0),
  }

  return(1),
}

write_primative( FILE* fp, int primative_type, pointlist* the_pointlist, object_record* the_details )
{
  char primative_name[24],
  int  cur_let = 0,

  /* Print out the name of the primative and a space after it */

/*  strncpy( primative_name, "\0", 24),
  strcpy( primative_name, keywords[primative_type] ),
*/
  sprintf( primative_name, "%s \0\0", keywords[primative_type] ),
  while( primative_name[cur_let] != '\0' )
  {
    fputc( primative_name[cur_let++], fp ),
  }

  /* Print out any details */

  switch (primative_type)
  {
    case SPHERE       break,
    case PATCH        break,
    case PATCHMESH    write_patchmesh( fp, the_pointlist, the_details ),
               break,
    case CYLINDER     break,
    case POLYGON      break,
  }

  return(1),
}



int morphit( char* morph_filename, FILE* in_fp, FILE* copy_fp, int frame_no, int max_frames )
{
/* Gets as parameters
    morph_filename    name of file containing target(s) Patchmesh statements
    in_fp             fp for start of source primitives
    copy_fp       fp for current point of output file
    frame_no      the number of the current frame 1-max_frames
    max_frames      the total number of frames

   Purpose
    To read in as many primitives in the source (in_fp) until an end_of_morph
    block marker is found  For the moment, lets just make it in another file
    called SRC_PRIM RIB  It just also read in the primitives in the target
    file (morph_filename)
    Each primitive in the target must have at least one primitive from the
    source mapped to it  Other source primitives are mapped to the nearest
    target primitive
    Every source primitive is then morphed to the appropriate target primitve
    even though this means duplicating the target primitives a number of times
    These can be elimitated when frame_no == max_frames

/* These were all defined as static */
  FILE            *morph_fp, *src_fp,
  int               prim_type, src_count, targ_count, src_prim, targ_prim,
  static pointlist      targ_points[10][MAX_3D_POINTS],
  static pointlist      start_points[10][MAX_3D_POINTS],
  static pointlist    morphed_points[MAX_3D_POINTS],
  static object_record  start_obj_details[10], targ_obj_details[10],
  int            src_targ_map[10],
  char           in_char,

  prim_type = NOTHING,

  if ( (morph_fp = fopen(morph_filename,"r")) == NULL )
  {
    fprintf(stderr,"copyit  Warning - cannot open morph target file %s \n",morph_filename),
    return(0),
```

```
  }

  if ( (src_fp = fopen("SRC_PRIM.RIB","r")) == NULL )
  {
    fprintf(stderr,"copyit: Warning - cannot open morph target file SRC_PRIM.RIB\n");
    return(0);
  }

  rewind(morph_fp);
  rewind(src_fp);

  /* Read in the Target Primatives from the external 'morph' file */
  for (targ_count=0;targ_count<10;targ_count++)
  {
    if ( !read_primative( morph_fp, &prim_type, targ_points[targ_count], &targ_obj_details[targ_count] ))
    {
      fprintf(stderr,"morphit: Warning - problem reading Target morph primative number %d in file %s.\n",targ_
count+1,morph_filename);
      return(0);
    }
    in_char = fgetc(morph_fp);
    if ( feof(morph_fp) )
      break;
    else
      ungetc(in_char,morph_fp);
  }
  if (targ_count >= 9 )
    fprintf(stderr,"morphit: 9 or more target primitives counted for morphing... continuing\n");

  targ_prim = targ_count;

  /* Read in the Source Primitive from the current file */
  for (src_count=0;src_count<10;src_count++)
  {
    if ( !read_primative( src_fp, &prim_type, start_points[src_count], &start_obj_details[src_count] ))
    {
      fprintf(stderr,"copyit: Warning - cannot read Source morph primative number %d.\n",src_count+1);
      return(0);
    }
    in_char = fgetc(src_fp);
    if ( feof(src_fp) )
      break;
    else
      ungetc(in_char,src_fp);
  }
  if (src_count >= 9 )
    fprintf(stderr,"morphit: 9 or more source primitives counted for morphing... continuing\n");


  src_prim = src_count;

  /* There should now be up to 9 source and target Patchmeshes with details
     and pointlists in memory.
  */

  for (targ_count=0;targ_count<=targ_prim;targ_count++)
  {
    /* Assign best-match source patchmeshes to targets ensuring at
     at least one Patchmesh for every target */
     src_targ_map[targ_count] = targ_count;
  }

  for (src_count=targ_prim;src_count<=src_prim;src_count++)
  {
    /* Assign remaining source patchmeshes to targets */
     src_targ_map[src_count] = src_count;
  }

  if (src_count<=9)
    src_targ_map[src_count] = -1;

  /* Just to test things out. Assign diff primitives here */


  /* Now ensure that the grid sizes are the same for each src primitive */

  for (src_count=0;src_count<=src_prim;src_count++)
  {

    /* Check if there are too many or too few columns */
    if ( (start_obj_details[src_count].mesh.num1) < (targ_obj_details[src_targ_map[src_count]].mesh.num1) )
      addextracolumns( &start_obj_details[src_count], start_points[src_count], &targ_obj_details[src_targ_map[
src_count]] );
    else if ( (start_obj_details[src_count].mesh.num1) > (targ_obj_details[src_targ_map[src_count]].mesh.num1)
 )
      removeextracolumns( &start_obj_details[src_count], &targ_obj_details[src_targ_map[src_count]], start_poi
nts[src_count] );
```

```
    /* Check if there are too many or too few rows */
    if ( (start_obj_details[src_count] mesh num2) < (targ_obj_details[src_targ_map[src_count]] mesh num2) )
      addextrarows( &start_obj_details[src_count], start_points[src_count], &targ_obj_details[src_targ_map[src
_count]] );
    else if ( (start_obj_details[src_count] mesh num2) > (targ_obj_details[src_targ_map[src_count]] mesh num2)
 )
      removeextrarows( &start_obj_details[src_count], &targ_obj_details[src_targ_map[src_count]], start_points
[src_count] );


/*     {
      /*  Expand the number of points on the source to the number on the target */
/*      addextrapoints( &start_obj_details[src_count], start_points[src_count], &targ_obj_details[src_targ_map
[src_count]] );
    }
    else if ( (start_obj_details[src_count] mesh num1 * start_obj_details[src_count] mesh num2) > (targ_obj_de
tails[src_targ_map[src_count]] mesh num1 * targ_obj_details[src_targ_map[src_count]] mesh num2) )
    {
      /* Remove extra points on the source to the number on the target */
  /*    removeextrapoints( &end_obj_details, &start_obj_details, end_points );
  */     /*XX removeextrapoints( &start_obj_details[src_count], &targ_obj_details[src_targ_map[src_count]], st
art_points[src_count] );
    }

    /* Gives a linear interpolation between two same-length pointlists
      and returns it in morphed_points  */
    interpolate( start_points[src_count], targ_points[src_targ_map[src_count]], morphed_points, frame_no, max_
frames );

/*    strcpy( (&targ_obj_details[src_targ_map[src_count]] mesh)->meshtype,  (&start_obj_details[src_count] mes
h)->meshtype );
*/
    write_primative( copy_fp, prim_type, morphed_points, &targ_obj_details[src_targ_map[src_count]] );

  }

  fclose( src_fp );
  fclose( morph_fp );
  return(TRUE);


}
```

```
#import <3Dkit/3Dkit h>

#define TO_CAMERA 0
#define TO_WORLD  1

@interface SimpleCamera N3DCamera
{
  id   theRotator,
  id   rotoMatrix,

  id   qualityMatrix,
  id   frameSlider,
  id   minFrameBox,
  id   maxFrameBox,

  id   infoPanel,
  id   frameDisp,

  id   rotDisp,
  id   scaleDisp,
  id   transDisp,

  id   rotationSlider,
  id   fovSlider,
  id   translateSlider,
  id   scalerSlider,

  id   tWaveCheckBox,

  id   formatBox,
  id   formatMatrix,

  id   cameraRollBox,
}

- worldBegin (RtToken)context,
- dumpRib sender,
- setQuality sender,
- changeFrameNumber sender,
- setNewFrameNumber sender,
- showInfo sender,
- playStepOne sender,
- playAllStepFive sender,
- rotateObject sender,
- setCameraRoll sender,
- setFieldOfView sender,
- renderPic sender,
- camera sender didRenderStream (NXStream *)s tag (int)atag frameNumber (int)n,
- moveTowards sender,
- reScale sender,
- startAnimationKey sender,
- endAnimationKey sender,
- showTWave sender,
- print sender,
- setFormat sender,
- nameRIBFile sender,
- setStartNumberOfFrames sender,
- setEndNumberOfFrames sender,

@end
```

```
#import <appkit/appkit h>
#import <appkit/Control h>
#import "SimpleCamera h"
#import "SimpleShape h"


/* Arsed around with from May 1993 by Somhairle Foley */

/* SimpleCamera -- by Bill Bumgarner 6/1/92
 *           with assistance from Dave Springer
 *
 * SimpleCamera demonstrates the creation of a very simple 3Dkit scene
 * that has mouse control via the N3DRotator class, supports dumping RIB
 * code to a file, contains light sources (ambient light and a point light),
 * has a surface shader, supports both WireFrame and SmoothSolid rendering,
 * and has a single custom N3DShape that generates a Torus (or teapot)
 *
 * Simple app was created as an example of using the 3Dkit   Parts of it
 * come from Teapot app by Dave Springer (see SimpleShape m)
 *
 *  You may freely copy, distribute and reuse the code in this example
 *  NeXT disclaims any warranty of any kind, expressed or implied,
 *  as to its fitness for any particular use
 */

/****** These are variables used in when displaying in SimpleShape m ******/

int   theFrameNumber,
float xRotation, yRotation, zRotation,
float xTranslate, yTranslate, zTranslate,
float xScale = 1 0, yScale = 1 0, zScale = 1 0,
int   theFOV,
int   showTWaveFlag = FALSE,
float   cameraRollAngle,

float start_xRotation, start_yRotation, start_zRotation,
float start_xTranslate, start_yTranslate, start_zTranslate,
float start_xScale, start_yScale, start_zScale,
int   start_theFOV,

/**************/

@implementation SimpleCamera
- initFrame (const NXRect *) theRect
{
  // camera position points
  RtPoint fromP = {0,0,5 0}, toP = {0,0,0},

  // light position point
  RtPoint lFromP = {0 5,0 5,0 75},

  // the various 3Dkit object id''s that we will initialize here
  id ambientLight,
  id aLight,
  id aShader,
  id aShape,

  // initialize camera and put it at (0,0,5 0) looking at the origin (0,0,0)
  // roll specifies the roll angle of the camera
  [super initFrame theRect],
  [self setEyeAt fromP toward toP roll 0 0],

  // create a shader that will shade surfaces with a simple matte surface
  aShader=[[N3DShader alloc] init],
  // uncomment the following lines to generate a blue matte surface
  // This is slow on a monochrome system
   [aShader setUseColor NO],      // SF 21/4/94
  // [aShader setUseColor YES],
  // [aShader setColor NX_COLORBLUE],
  [(N3DShader *)aShader setShader "matte"],

  // initialize the world shape and set its shader to be aShader
  aShape=[[SimpleShape alloc] init],
  [(N3DShape *) aShape  setShader aShader],
  [[self setWorldShape aShape] free], // free the default world shape

  // create an ambientlight source
  ambientLight=[[N3DLight alloc] init],
  [ambientLight makeAmbientWithIntensity 0 1],
  [self addLight ambientLight],

  // create a Point light and put it at (0 5, 0 5, 0 75) at
  // full intensity (1 0)
  aLight=[[N3DLight alloc] init],
  [aLight makePointFrom lFromP intensity 1 0],
  [self addLight aLight],

  // set the surface type to generate smooth solids   The mouseDown
```

```
  // method automatically drops to N3D_WireFrame whenever the user manipulates
  // the scene via the mouse (see the mouseDown  implementation below)
  // This must be done after the setWorldShape  method (or after any new shape
  // is added to the hierarchy)
  [self setSurfaceTypeForAll N3D_SmoothSolids chooseHider YES],

  // allocate and initialize the N3DRotator object that governs
  // rotational control via the mouseDown  method
  theRotator=[[N3DRotator alloc] initWithCamera self],

  return self,
}

- worldBegin (RtToken)context
{
  static RtInt clipon = 1,     //, clipoff = 0,

  // RiDepthOfField(myFstop, myFocalLength, myFocalDistance),

  /* select clip object mode and read a RIB file */
  RiOption(RI_ARCHIVE, "clipobject", &clipon, RI_NULL),
  [super worldBegin context],

  // RiOption(RI_ARCHIVE, "clipobject", &clipoff, RI_NULL),

  return self,
}

- dumpRib sender
{
  static id savePanel=nil,
  NXStream *ts,
  char buf[MAXPATHLEN+1],

  // initialize the savePanel, if it hasn''t been done so previously
  if ('savePanel) {
    savePanel=[SavePanel new],
    [savePanel setRequiredFileType "rib"],
  }

  // run the savepanel
  if([savePanel runModal]){
    // returned w/pathname, open a stream and
    ts=NXOpenMemory(NULL, 0, NX_WRITEONLY),
    // process the file name for a custom display line such that
    // "prman <<filename>> rib" will put the resulting image somewhere
    // predictably useful
    strcpy(buf, [savePanel filename]),
    // remove the  rib extension from the path returned by the SavePanel
    strrchr(buf,' ')[0]='\0',
    // feed to NXPrintf to put in the custom Display command
    NXPrintf(ts, "Display \"%s tiff\" \"file\" \"rgba\"\n", buf),
    // then feed the rib code to the stream and
    [self copyRIBCode ts],
    // save the stream to the file selected in the savepanel
    NXSaveToFile(ts, [savePanel filename]),
    // and close the stream (which also flushes it), also making sure
    // that the allocated memory is freed
    NXCloseMemory(ts,NX_FREEBUFFER),
  }
  return self,
}

#define ACTIVEBUTTONMASK (NX_MOUSEUPMASK|NX_MOUSEDRAGGEDMASK)
#define POINTS, break,
#define WIREFRAME, break,
#define SHADEDWIRE, break,
#define FACETED, break,
#define SMOOTH, break,


- mouseDown (NXEvent *)theEvent
{
  int                    oldMask,
  NXPoint   oldMouse, newMouse, dMouse,
  RtMatrix    rmat, irmat,

  // find out what axis of rotation the rotator should be constrained to
  switch([rotoMatrix selectedRow]){
  case 0  [theRotator setRotationAxis N3D_AllAxes], break,
  case 1  [theRotator setRotationAxis N3D_XAxis], break,
  case 2  [theRotator setRotationAxis N3D_YAxis], break,
  case 3  [theRotator setRotationAxis N3D_ZAxis], break,
  case 4  [theRotator setRotationAxis N3D_XYAxes], break,
  case 5  [theRotator setRotationAxis N3D_XZAxes], break,
  case 6  [theRotator setRotationAxis N3D_YZAxes], break,
  }
```

```
  // track the mouse until a mouseUp event occurs, updating the display
  // as tracking happens

  [self lockFocus],
  oldMask = [window addToEventMask ACTIVEBUTTONMASK],

  // switch to the N3D_WireFrame surface type
 // [self setSurfaceTypeForAll surfaceType chooseHider YES],

  oldMouse = theEvent->location,
  [self convertPoint &oldMouse fromView nil],
  while (1)
  {
    newMouse = theEvent->location,
    [self convertPoint &newMouse fromView nil],
    dMouse x = newMouse x - oldMouse x,
    dMouse y = newMouse y - oldMouse y,
    if (dMouse x '= 0 0 || dMouse y '= 0 0) {
      [theRotator trackMouseFrom &oldMouse to &newMouse
       rotationMatrix rmat andInverse irmat],
      [worldShape concatTransformMatrix rmat premultiply NO],
      [self display],
    }
    theEvent = [NXApp getNextEvent ACTIVEBUTTONMASK],
    if (theEvent->type == NX_MOUSEUP)
      break,
    oldMouse = newMouse,
  }
  // switch back to the N3D_SmoothSolids surface type
 //  [self setSurfaceTypeForAll N3D_SmoothSolids chooseHider YES],
  [self display],
  [self unlockFocus],

  [window setEventMask oldMask],
  return self,
}

-setQuality sender
{
  int     surfaceType = N3D_WireFrame,

  printf("setQuality Selected    %d\n", [[qualityMatrix selectedCell] tag]),
  switch([[qualityMatrix selectedCell] tag])
  {
    case 0  printf("0 Selected\n"),surfaceType = N3D_PointCloud, break,
    case 1  printf("1 Selected\n"),surfaceType = N3D_WireFrame, break,
    case 2  printf("2 Selected\n"),surfaceType = N3D_ShadedWireFrame, break,
    case 3  printf("3 Selected\n"),surfaceType = N3D_FacetedSolids, break,
    case 4  printf("4 Selected\n"),surfaceType = N3D_SmoothSolids, break,
  }

  // switch to the N3D_WireFrame surface type
  [self setSurfaceTypeForAll surfaceType chooseHider YES],
  [self display],

  return self,
}

// probably dont need this*******
- changeFrameNumber sender
{
   int frameNumber,

   frameNumber = [frameSlider intValue],
/*   printf("In getFrameNumber method, frameNumber is %d\n", frameNumber),
*/   return self,
}

- setNewFrameNumber sender
{
   int nextFrameNumber,

   nextFrameNumber = [frameSlider intValue],
   if ( nextFrameNumber '= theFrameNumber )
       {
           theFrameNumber = nextFrameNumber,
 /*        printf("The Frame Number is %d\n", theFrameNumber ),
 */        [frameDisp setIntValue theFrameNumber],
           [self display],
       }

   return self,
}

- showInfo sender
{
  if (infoPanel == nil)
```

```
    if ( ! [NXApp loadNibSection: "info.nib"  owner:self withNames:NO ] )
     return nil;
   [infoPanel makeKeyAndOrderFront:self];
  return self;
}

- playStepOne:sender
{
  int count, maxval;

  maxval = [maxFrameBox intValue];

  for ( count=[frameSlider intValue]; count<=maxval; count++ )
  {
    [frameSlider setIntValue:count];
    [self setNewFrameNumber:self];
  }
  return self;
}

- playAllStepFive:sender
{
  int count, maxval;

  maxval = [maxFrameBox intValue];

  for ( count=[minFrameBox intValue]; count<=maxval; count+=5 )
  {
    [frameSlider setIntValue:count];
    [self setNewFrameNumber:self];
  }
  return self;
}

- rotateObject:sender
{
 /*  printf("rotateObject: Slider No: %d\n", [[rotationSlider selectedCell] tag]);
 */

  switch([[rotationSlider selectedCell] tag])
  {
    case 0: xRotation = [[rotationSlider selectedCell] floatValue]; break;
    case 1: yRotation = [[rotationSlider selectedCell] floatValue]; break;
    case 2: zRotation = [[rotationSlider selectedCell] floatValue]; break;
  }


  [  [rotDisp selectCellWithTag:[[rotationSlider selectedCell] tag]]  setFloatValue:[[rotationSlider selectedC
ell] floatValue]];

 /*  [[rotDisp selectCellWithTag:0] setFloatValue:xRotation];
  [[rotDisp selectCellWithTag:1] setFloatValue:yRotation];
  [[rotDisp selectCellWithTag:2] setFloatValue:zRotation];
 */  [self display];
  return self;
}

-setCameraRoll:sender
{
  // camera position points
  RtPoint fromP = {0,0,5.0}, toP = {0,0,0};

  cameraRollAngle = [cameraRollBox floatValue];
  printf("cameraRollAngle is %f\n",cameraRollAngle);
  [self setEyeAt:fromP toward:toP roll:cameraRollAngle];
  [self display];
  return self;
}

- setFieldOfView:sender
{
  theFOV = [fovSlider intValue];
  [self setFieldOfViewByAngle:theFOV];
  [self display];
  return self;
}

- renderPic:sender    // INVOKED BY A MENU ITEM
{
  [self setDelegate:self];  // SET THE CAMERA'S DELEGATE
  [self renderAsTIFF];    // INVOKE THE RENDER PANEL
//[self renderAsEPS];   // INVOKE THE RENDER PANEL
  return self;
}

- camera:sender didRenderStream:(NXStream *)s tag:(int)atag frameNumber:(int)n
{
  NXImage *renderPic;
```

```
  NXPoint photoPicPos={0 0, 0 0},
  NXStream *theStream,
//  int fd,
  id  save = [SavePanel new],

  renderPic=[[NXImage alloc] initFromStream s],

  [self setFormat formatMatrix],
  [save setAccessoryView formatBox],
  if ( [save runModal] == 1 )
  {
    theStream = NXOpenMemory( 0,0,NX_WRITEONLY ),
    [renderPic writeTIFF theStream],
    NXSaveToFile( theStream, [save filename] ),
    NXCloseMemory( theStream, NX_FREEBUFFER ),
  }

//  fd = open( "testqqq ps", O_CREAT | O_WRONLY | O_TRUNC, 0666 ),
//  theStream = NXOpenFile( fd , NX_WRITEONLY ),
//  [renderPic  writeTIFF theStream],
//  NXClose(theStream),
//  close(fd),

  [sender lockFocus],
  [renderPic composite NX_COPY toPoint &photoPicPos],
  [sender unlockFocus],
  [[sender window] flushWindow],
  return self,
}

- moveTowards sender,
{
  xTranslate = [[translateSlider selectCellWithTag 0] intValue],
  yTranslate = [[translateSlider selectCellWithTag 1] intValue],
  zTranslate = [[translateSlider selectCellWithTag 2] intValue],

  [[transDisp selectCellWithTag 0]  setFloatValue xTranslate],
  [[transDisp selectCellWithTag 1]  setFloatValue yTranslate],
  [[transDisp selectCellWithTag 2]  setFloatValue zTranslate],

  [self display],
  return self,
}

- reScale sender,
{
  switch([[scalerSlider selectedCell] tag])
  {
    case 0  xScale = [[scalerSlider selectedCell] floatValue], break,
    case 1  yScale = [[scalerSlider selectedCell] floatValue], break,
    case 2  zScale = [[scalerSlider selectedCell] floatValue], break,
  }

  [  [scaleDisp selectCellWithTag [[scalerSlider selectedCell] tag]]  setFloatValue [[scalerSlider selectedCel
l] floatValue]],
  [self display],
  return self,
}

- startAnimationKey sender
{
  start_xRotation = xRotation,
  start_yRotation = yRotation,
  start_zRotation = zRotation,
  start_xTranslate = xTranslate,
  start_yTranslate = yTranslate,
  start_zTranslate = zTranslate,
  start_xScale = xScale,
  start_yScale = yScale,
  start_zScale = zScale,
  start_theFOV = theFOV,

  [frameSlider setIntValue [minFrameBox intValue]],
  [self display],

  return self,
}

- endAnimationKey sender
{
/*  float end_xRotation, end_yRotation, end_zRotation,
  float end_xTranslate, end_yTranslate, end_zTranslate,
  float end_xScale, end_yScale  end_zScale,
  int   end_theFOV,

  end_xRotation = xRotation,
  end_yRotation = yRotation,
  end_zRotation = zRotation,
```

```
  end_xTranslate = xTranslate,
  end_yTranslate = yTranslate,
  end_zTranslate = zTranslate,
  end_xScale = xScale,
  end_yScale = yScale,
  end_zScale = zScale,
  end_theFOV = theFOV,
*/
  xRotation = start_xRotation,
  yRotation = start_yRotation,
  zRotation = start_zRotation,
  xTranslate = start_xTranslate,
  yTranslate = start_yTranslate,
  zTranslate = start_zTranslate,
  xScale = start_xScale,
  yScale = start_yScale,
  zScale = start_zScale,
  theFOV = start_theFOV,

  [frameSlider setIntValue [maxFrameBox intValue]],
  [self display],
  return self,
}

- showTWave sender
{
  if ([tWaveCheckBox intValue] == 0)
    showTWaveFlag = FALSE,
  else
    showTWaveFlag = TRUE,

  [self display],
  return self,
}

- print sender
{
    return [self printPSCode sender],
}

- setFormat sender,
{
  char *format,
  char *cc,

  format = NXCopyStringBuffer( [ [sender selectedCell] title] ),
  for ( cc = format, *cc, cc++ )
  {
    *cc = NXToLower(*cc),
  }
  [ [SavePanel new] setRequiredFileType format],
  free( format ),
  return self,
}


/* action method, called when the user chooses open in the menu */
- nameRIBFile sender
{
    const char *const *files,
    static const char *const fileType[2] = {"rib", NULL},
    OpenPanel *openPanel,
    char fullName[MAXPATHLEN], nameNoExt[MAXPATHLEN],
    char *ptrExtension,
  FILE* fp,

  /*
   * Declare that the user can select multiple files to be opened in the
   * Open Panel   All apps should do this, since its so easy
   */
    openPanel = [[OpenPanel new] allowMultipleFiles NO],

  /* run the open panel, filtering for out types of our documents */
    if ([openPanel runModalForTypes fileType])
  {
      /* open all the files returned by the open panel */
    files = [openPanel filenames],
    for (files = [openPanel filenames], files && *files, files++)
    {
      strcpy(fullName, [openPanel directory]),
      strcat(fullName, "/"),
      strcat(fullName, *files),
        ptrExtension = strchr(fullName,    ')
      strncpy(nameNoExt,fullName,(ptrExtension-fullName)),
      nameNoExt[ptrExtension-fullName]='\0',

      if ( (fp=fopen("THENAME TXT","w"))==NULL )
        printf("error opening THENAME TXT"),
```

```
      else
      {
        fprintf(fp,"%s", nameNoExt),
        fclose(fp),
      }
    }
    }
    return self,
}

- setStartNumberOfFrames sender
{
  int minval,

  minval = [minFrameBox intValue],
  [frameSlider setMinValue minval],
    return self,
}

- setEndNumberOfFrames sender
{
  int maxval,

  maxval = [maxFrameBox intValue],

  [frameSlider setMaxValue maxval],
    return self,
}


@end
```

```
#import <3Dkit/3Dkit h>

@interface SimpleShape   N3DShape
{
}
- readInRIBFile (char*)RIBFilename  returnMemoryLocationTo (char**)fileInMemLocation,
- renderSelf (RtToken)context,

@end
```

```
#import <appkit/appkit h>
#import "SimpleCamera h"
#import "SimpleShape h"
#import <ri/ri h>
#import <sys/param h>
#import <stdio h>

/*
 *  You may freely copy, distribute and reuse the code in this example
 *  NeXT disclaims any warranty of any kind, expressed or implied,
 *  as to its fitness for any particular use
 */

extern void Go(void),
extern void BowlingBall(void),

extern int theFrameNumber,
extern float xRotation, yRotation, zRotation,
extern float xTranslate, yTranslate, zTranslate,
extern float xScale, yScale, zScale,
extern int theFOV,
extern int showTWaveFlag,

extern float  start_xRotation, start_yRotation, start_zRotation,
extern float  start_xTranslate, start_yTranslate, start_zTranslate,
extern float  start_xScale, start_yScale, start_zScale,
extern int    start_theFOV,

//#define NOTFOUND 1
//#define FOUND 0

/*
char* getStartOfWorld( char** fileInMemLocation, int ribFileSize )
{
  int letter, offset=0, found = NOTFOUND,
  char *next_char,
  char the_char,
  char cur_word[30],

  next_char = *fileInMemLocation,

  while ( offset <= ribFileSize && found == NOTFOUND )
  {
    letter = 0,
    the_char = *next_char,
    next_char++,


    while( the_char != ' ' && the_char != '\n' && offset <= ribFileSize && letter < 30 )
    {
        cur_word[letter++] = the_char,
        the_char = *next_char++,
    }

    if ( letter < 30 )
    {
      cur_word[letter] = '\0',
      if ( strcmp( cur_word, "WorldBegin") == 0 )
        found = FOUND,
    }
    offset++,
  }

  if ( found == FOUND )
    return( next_char ),
  else
    return( NULL ),

},
*/

@implementation SimpleShape N3DShape


- readInRIBFile (char*)RIBFilename  returnMemoryLocationTo (char**)fileInMemLocation
{
/*   get the filesize, malloc it, read in the file into ASCIIZ,
    use the form for ReadArchive for using memory
*/

  int    error_check=0,
  int    mem_loc=0,
  FILE* RIBFile_fp,
  char   CurrentDirectoryFilename[255],
  long   filesize,
  static int *test=NULL,

  printf( "error check=%d ,  mem_loc = %d\n",error_check,mem_loc ),
```

```
  RIBFile_fp = fopen( RIBFilename, "r" ),
  if ( RIBFile_fp == NULL )
  {
    if ( 'getwd( CurrentDirectoryFilename ))
      exit(1),
    printf( "Error  Can't find %s and the current dir is %s\n", RIBFilename, CurrentDirectoryFilename ),
    return(-1),
  }

  fseek( RIBFile_fp, 0L, SEEK_END ),
  filesize = ftell( RIBFile_fp ),
  rewind( RIBFile_fp ),


  test = (int*)malloc(sizeof(int)),
  free(test),

  *fileInMemLocation
 = (char*)calloc(1,(size_t)filesize+1),

  error_check = fread( *fileInMemLocation, 1, filesize, RIBFile_fp ),
  printf( "error check=%d ,  mem_loc = %d\n",error_check,mem_loc ),

  fclose( RIBFile_fp ),

//  *fileInMemLocation = getStartOfWorld( fileInMemLocation, error_check ),


/*  for( mem_loc = 0, mem_loc <= filesize, mem_loc++ )
  {
    if ( *(*fileInMemLocation+mem_loc)== '\n' )
      *(*fileInMemLocation+mem_loc) = 0,
  }
*/
  return self,

}


/************************************************************************/


- renderSelf (RtToken)context
{

  char inFilename[255] = "\0",
  FILE* fn_fp,
  int  count = 0,
  char ch = 'a',
  char*  fileInMemLoc=NULL,
  RtToken myname,
  char *myrib = "Cylinder  5  2 1 360",
  char *ribFilename,

  ribFilename = (char*)malloc((size_t)255),

  // generate a Torus
  //   RiTorus(0 8, 0 3, 0 0, 360 0, 360 0, RI_NULL),

  // comment out the above and uncomment the following lines to render a Teapot
  // RiScale(0 4, 0 4, 0 4),
  // RiGeometry("teapot", RI_NULL),


  //ShowQuads(),
  //BowlingBall(),

  // RiResource( "myres", RI_ARCHIVE, RI_FILEPATH, "/NextDeveloper/Examples/RenderMan/Airplane rib", RI_NULL )
,

  //RiGeometricRepresentation("primitive"),

  count = 0,
  fn_fp = fopen( "THENAME TXT", "r" ),

  if ( fn_fp == NULL )
  {
    if ( 'getwd( inFilename ))
        exit(1),
    printf( "Error  Can't find THENAME TXT and he cuurent dir is %s\n", inFilename ),
  }

  ch = fgetc(fn_fp),
  while( ch '= ' ' && ch '= '\n' && 'feof(fn_fp) && count < 255 )
  {
    inFilename[count++] = ch,
    ch = fgetc(fn_fp),
```

```
  }

  inFilename[count] = '\0',
/*  printf( "The IN file name is %s \n", inFilename ),
*/  sprintf( ribFilename, "%s%03d rib", inFilename, theFrameNumber),

  printf( "The RIB file name is %s \n", ribFilename ),


/* To set the file up so the bits outside of WorldBegin/WorldEnd are
   ignored

*/

/* Take out becuase of 'rfWriteParameters  bad type' error */
//[self readInRIBFile ribFilename  returnMemoryLocationTo &fileInMemLoc],


printf("the field of view is %d\n", theFOV ),
if ( theFOV < 55 )
  theFOV = 55,


//RiProjection( "perspective", "fov", theFOV, RI_NULL),

RiTranslate( xTranslate, yTranslate, zTranslate ),
RiRotate( xRotation, yRotation, zRotation, 1 0 ),
RiScale( xScale, yScale, zScale ),

RiTranslate( 1,1,1 ),
RiSphere(  2,  2, - 2, 360, RI_NULL ),


// Check if the Toridal Wave is to be shown
if ( showTWaveFlag )
{
  Go(),
/*   RtPoint hyperpt1,hyperpt2,

   hyperpt1[0] = 0,

   hyperpt1[1] = 0,

   hyperpt1[2] = 5,

   hyperpt2[0] = 0 3,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = 4 7,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 0 3,

   hyperpt1[1] = 0,

   hyperpt1[2] = 4 7,

   hyperpt2[0] = 0 7,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = 4,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 0 7,

   hyperpt1[1] = 0,

   hyperpt1[2] = 4,

   hyperpt2[0] = 0 5,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = 2,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 0 5,

   hyperpt1[1] = 0,

   hyperpt1[2] = 2,

   hyperpt2[0] = 1 5,

   hyperpt2[1] = 0 ,
```

```
   hyperpt2[2] =  5,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 1 5,

   hyperpt1[1] = 0,

   hyperpt1[2] =  5,

   hyperpt2[0] = 1 4,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = -3,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 1 4,

   hyperpt1[1] = 0,

   hyperpt1[2] = -3,

   hyperpt2[0] = 0 3,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = -4,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
   hyperpt1[0] = 0 3,

   hyperpt1[1] = 0,

   hyperpt1[2] = -4,

   hyperpt2[0] = 0,

   hyperpt2[1] = 0 ,

   hyperpt2[2] = -4 2,

   PolyBoid( hyperpt1, hyperpt2, 5, 1 ),
*/
}
RiTransformBegin(),

//just use filenames until 'rfWriteParameters  bad type' error is fixed
//myname = RiResource("myres", RI_ARCHIVE,
//                    RI_ADDRESS, &fileInMemLoc, RI_NULL),
/*myname = RiResource("myres", RI_ARCHIVE,
                    RI_ADDRESS, &myrib, RI_NULL),
*/


RiTranslate(0,0,-20),
myname = RiResource("myres", RI_ARCHIVE,
                    RI_FILEPATH, &ribFilename, RI_NULL),
RiReadArchive( myname, NULL, RI_NULL ),

RiTransformEnd(),

//free( fileInMemLoc ),
fclose(fn_fp),


  return self,
}

@end
```

```
/* joinribs c  --   the program that will concatenate a number of RIB files
          to make one "BIG" RIB files that uses FrameBegin and
          FrameEnd to seperate different Frames

    Somhairle Foley    18th February 1995

                example

    >joinribs 145 rball

    this will take the files called rballXXX rib file where the
    XXX is from 001 to 145 and make one file called BIGrball rib
    which contains all of the frames in it

    Note  the  rib file extension is not specified

               16/06/93

                Current assignments are

                          %I    -    include another file
                          %M    -    morph the following object with a file
*/

#include <stdio h>
#include <stdlib h>
#include <string h>
#include <ctype h>

#define TRUE 0
#define FALSE 1

FILE *out_fp, *little_fp,

int main(int argc, char* argv[])
{
  char out_filename[50], little_filename[50],
  char the_char,
  int  file_count,
  int  num_frames, start_num,

  if ( argc != 3 )
  {
   fprintf(stderr, "Usage  joinribs <number of frames>  <RIB filenames>\n"),
   exit(1),
  }

  num_frames = atoi( argv[1] ),
  if ( num_frames < 1 || num_frames > 199 )
  {
     fprintf(stderr, "joinribs  number of frames must be between 1 and 199 \n"),
     exit(1),
  }

  sprintf( out_filename, "BIG%s rib", argv[2] ),
  if ((out_fp = fopen(out_filename, "w")) == NULL)
  {
     fprintf( stderr, "joinribs  Cannot open output file %s\n",out_filename ),
     exit(1),
  }

  start_num = 1,
  rewind( out_fp ),

  for ( file_count = 1, file_count <= num_frames , file_count++ )
  {

  fprintf( out_fp, "FrameBegin %03d\n", file_count ),

    /* Put this instance of the filename in the little_filename string */
    sprintf( little_filename, "%s%03d rib\0", argv[2], file_count ),

    if ((little_fp = fopen( little_filename, "r")) == NULL )
     {
     fprintf( stderr, "joinribs  Cannot open input file %s\n",little_filename ),
     exit(1),
   }

  printf( "Making Joined Frame File %s   current translations are %03d\n",
       out_filename, start_num+file_count-1 ),

  while ( !feof(little_fp) )
  {
    the_char = fgetc(little_fp),
    if isascii(the_char)
      fputc( the_char, out_fp ),
  }
```

```
  fclose( little_fp ),

  fprintf( out_fp, "\nFrameEnd \n#\n"),

  }              /* end for */

   printf("\nFinished Joining a RIB called `%s  I Think \n" ,argv[2] ),
   printf("=============================================================\n\n"),

 return( fclose( out_fp ) ),

}
```