Computing Science

Ph.D Thesis

Ray Tracing Displacement Mapped Surfaces

James R Logie

Submitted for the degree of

Doctor of Philosophy

Thesis
10219
Copy 1

Ray Tracing Displacement Mapped Surfaces

by
James R Logie

Submitted to the Department of Computing Science
on November 1, 1995
for the degree of
Doctor of Philosophy

Abstract

Displacement mapping is a technique in computer graphics which allows a simple base surface to be deformed into a more complex surface by applying a texture to change the geometry. This is achieved by applying to each point on the surface a displacement, specified by a displacement function, with a magnitude specified by a height field. This technique makes it possible to transform the simple primitives used in computer graphics today into visually rich and geometrically complex surfaces. Displacement mapping is a texture mapping technique in which the texture is the height field defining the displacement magnitudes. But, unlike any other form of texture mapping, displacement mapping alters the surface geometry. This has many implications for the rendering of displacement mapped surfaces. It must be considered early in the rendering process during the visibility calculations (since it defines the actual geometry of the surface). This is in contrast to other forms of texture mapping which are applied after the visibility of the surface is known. This fact accounts for much of the power and complexity involved in rendering displacement mapped surfaces

This thesis provides an investigation into ways to render such surfaces by the use of ray tracing. It is commonly believed that displacement mapped surfaces are too complex to be ray-traced due to the complex nature of the geometry they define. This myth is disproved by the algorithms contained herein. Three algorithms are presented which tackle the ray-surface intersection problem for displacement mapped surfaces (this being the core calculation in a ray tracer). The first algorithm tackles the problem geometrically by analysing the geometry of the intersection calculation. This approach provides a fast algorithm but with limited applicability. It is only suitable for simple base surfaces where the underlying geometry can be easily analysed. The second algorithm reduced the intersection calculation to a system of non-linear equations and applies existing numerical techniques to solve these. This approach, although very general, proves to unsuccessful due to the enormous amount of computation involved. The third approach polygonalises the displacement-mapped surface as it is rendered and calculates the intersections with the generated polygons. This, combined with a system to allow the efficient generation, storage and processing of the generated polygons, provides the first practical system for ray tracing displacement mapped surfaces

Thesis Supervisor: Dr. J. W. Patterson
Title: Ray Tracing Displacement Mapped Surfaces

Acknowledgements

I would like to gratefully thank my supervisor Dr. John W. Patterson for his help, encouragement, guidance and patience in the research which has led to this thesis. My introduction to computer graphics stemmed from an Honours project which I was forced to do as a fifth choice but, through John's patience and enthusiasm, become my thesis topic. Without that odd turn of fate, I would never have contemplated this area of work.

I am deeply indebted to my wife Trish who has put up with me during my research. Without her support and love my life would not be a complete as it is today.

I would also like to thank my colleagues in the Department of Computing Science whose friendship and help provided a environment which stimulated research and made my task all the more enjoyable and easy. I would especially like to thank Deryck Brown who prepared a Glasgow University thesis style for LaTeX which made the preparation of this document much simpler.

Finally, I would like to acknowledge SERC and the Charles and Barbara Tyre Trust for there financial support during my studentship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Within the field of computer graphics the generation of high quality images remains a constant challenge. Many techniques have been used to produce ever more complex and realistic images. This thesis extends the range of tools available for this purpose by developing techniques to ray trace displacement-mapped surfaces. Displacement mapping is a complex form of texture mapping in which the texture defines a height field. This, when mapped onto a surface, displaces the geometry of the original surface to create a radically new surface with (potentially) a far richer and more complex shape. In this way, the simple standard objects in existing rendering systems can be transformed to create new and useful shapes.

In the early days of image synthesis it was realized that a major problem with computer generated images was their perfect, smooth mathematical appearance, whereas in real life surfaces exhibit a myriad of detail. This detail appears in many forms from colours, roughness and reflectivity (from perfect mirrors to diffuse cloth) as well as geometric detail of small bumps and scratches. The realism of computer generated images can be greatly enhanced by simulating this detail. Texture mapping[Hec86] is a technique which has been used in many forms to simulate this. Texture mapping proceeds by relating each point on a surface to a value defined by a texture function. The value of the texture function is then used to define some material property of the surface, such as its colour. The texture function is commonly defined by a rectangular array of values (such as an image). This array is referred to as the "texture map" and its individual elements called "texels".

The earliest forms of texture mapping[Cat74] mapped images onto the surface of objects. This was only partially successful. Patterns[BN76] can be successfully mapped but scanned images presented many problems. Photographs of wood patterns and bark looked, as they were, like photographs wrapped onto surfaces and not like real wood. This arises from inconsistent shading between the object and the shading in the scanned image. Later uses of texture mapping used the texture map to define surface properties such as roughness and reflectivity[Bli78a]. With these techniques the appearance of the surface can be varied by altering parameters passed to the lighting model. A further use of texture mapping is "bump-mapping"[Bli78b], which simulates the effects of small changes in surface geometry (e.g. bumps). With this technique the normal to the surface is perturbed, altering the way light reflects from the surface, to give the impression of small changes in surface geometry. This gives rise to a seemingly complex surface geometry. It must be noted that bump mapping does not change the actual geometry of the surface and can only be used to simulate small features. Since these early uses of texture mapping, it has been extended further to handle other surface parameters. Environment mapping[BN76, Gre86] places a texture not onto an individual object but around the entire scene. This places the scene in a context which can be seen by reflections on other objects. An example of this might be to wrap an environment map of an office around a scene of a desk. The texture can also be used to store parameters of the surface used to calculation the illumination. These can range from specular, diffuse and glossiness coefficients as used in the Phong lighting model[Pho75] to physically derived values to model the scattering of light from a surface[Kaj85, PF90].

Texture mapping, although very powerful, can introduce artifacts if naively implemented. These problems occur when the samples taken from the texture map do not sample it accurately and gives rise to aliasing. This problem has received considerable attention by computer graphics practitioners and a number of solutions exist. The idea behind these is to calculate the area of the texture covered by the screen pixel and then filter this area to find the correct texture value. This may be done directly for each pixel[FLC80] or the texture may be stored pre-filtered at different resolutions and the correct resolution chosen to find the texture value[Cro77, Wil83, Gla86, Hec86]. For displacement maps the different resolutions correspond to different levels of visible detail and must be handled by different techniques. This issue is beyond the scope this thesis..

The purpose of this thesis is to present a number of new algorithms to handle displacement mapping[Coo84]. This is the form of texture mapping in which the texture defines height values which are used to alter the surface geometry. Displacement mapping, unlike other forms of texture mapping, alters the surface geometry not just its lit appearance. This fact makes displacement mapping unique and powerful. Unfortunately, it also adds a far larger computational burden to rendering than other forms of texture mapping. Solutions to these computational problems form the body of this thesis.

The techniques of texture mapping have previously been classified into two types[CG85]. The first deals with microscopic detail of the surface such as roughness, specular or diffuse reflections. These are characterised by using statistical approximations to model very fine detail (of the order of the wavelength of visible light) on the surface. The quantities cannot be modelled directly since they are too small to measure accurately and the resolution used in the rendering is vastly greater than the geometry of the effects. These effects are generally accounted for by terms in the lighting model. Examples of this are the micro-facet distribution and orientation terms in the Torrance-Sparrow[Bli78a, CT81] shading model or the coefficients of a physical model simulating anisotropic scattering[Kaj85, PF90, War92, WAT92]. The second type of texture mapping techniques deal with the macroscopic details of the surface geometry. This can be small bumps[Bli78b] on the surface (as modelled by bump-mapping) or it can be visible detail such as the weave pattern on cloth[YYiT92]. Within this classification displacement mapping is a macroscopic technique.

This classification, though, misses many essential properties of displacement mapping. It must be handled completely differently from all other forms of texture mapping and at a different time in the rendering process. Displacement mapping alters the geometry, not just the appearance, of the surface to which it is applied, thus it must be considered early in the rendering process. Other forms of texture mapping apply after the visible surface calculations have been performed but displacement mapping, since it alters the geometry of an object, must be considered as part of the visible surface calculation. This fact moves displacement mapping away from being solely a rendering issue towards the realm of modelling. It also accounts for the power and complexity involved.

Ray tracing[Gla89] has been chosen as the method of rendering displacement-mapped surfaces. Ray tracing works by simulating the paths of rays of light through the scene giving a unified frame work for reflections, refractions, shadows and other optical effects. The key calculation involved is the intersection of a ray and a surface. The major results of this thesis are methods to perform this calculation for displacement-mapped surfaces. Amongst rendering techniques, ray tracing is renowned for producing very high quality images, but at a large computational cost. This cost is closely related to the complexity of the surfaces used. Displacement-mapped surfaces are commonly regarded as too complex to be ray traced but, as shall be shown, they can be handled efficiently within the ray tracing model.

1.2 Displacement mapping

Displacement mapping provides a conceptually simple yet extremely powerful form of texture mapping, its main strength being that it provides a compact way to express very complex geometries. There are many surface types commonly used in computer graphics today. The simplest of these are polygons, quadrics, swept surfaces and surfaces of revolution. Other more complex surface types are also used including generalised cylinders, superquadrics and many forms of spline patches. With these primitives, complex objects and scenes can be designed. Unfortunately, this is a difficult and time-consuming task. The rough shape of an object can be specified reasonably easily but the major task is fine tuning the geometry and adding fine geometric detail. The many existing forms of texture mapping can do much to disguise a lack of geometric detail but can only help up to a point. The models used often betray their mathematical origin by appearing with smooth and regular surfaces. This problem can be overcome with existing modelling primitives but at the expense of the large amount of time and effort it takes to model complex details. Displacement mapping provides a solution to this problem. It adds detail directly to the geometry of the model producing more realistic and attractive surfaces.

A displacement-mapped surface is defined by the general equation in 1.1.

$$\underline{S}(u,v) = \underline{f}(u,v) + t(u,v).\underline{d}(u,v) \tag{1.1}$$

where

$\underline{f}(u,v)$ = underlying or base surface,

$t(u,v)$ = the texture function,

$\underline{d}(u,v)$ = the displacement function.

The underlying surface, $\underline{f}$, can be any parametrically defined surface. Thus displacement mapping, in its most general form, places few restrictions on the type of surface to which it can be applied. All of the standard surfaces used in computer graphics today (except fractals) can be displacement-mapped. The main research for this thesis has concentrated on the standard modelling primitives described earlier, with a particular emphasis on spline patches.

The actual displacements applied to the surface are controlled by two functions t and $\underline{d}$. The first of these, t, just defines the height of the texture function at a given point. The most common forms for this function are as bilinear or bicubic interpolations of an array of height values(i.e. a height field). These height values are defined in the texture map. The function, $\underline{d}$, defines the direction in which the surface is displaced. This will usually be the direction of the (positive) surface normal but can in principle be any function. It will commonly be the case that the displacement vector will be normalised to unit length as this gives a consistent meaning to the texture height over the complete surface. The specification of the displacement function in two parts has many advantages. It allows textures to be designed independently of the surface. This promotes the reuse of textures allowing the same texture to be used with many displacement functions. Also, the definition of the displacement function in two parts as above has the advantage that a texture height of zero corresponds to a zero displacement.

The previous discussion of displacement mapping has described how a displacement-mapped surface can be modelled. This shows that with displacement mapping a new geometric model is formed. This is in contrast to texture mapping techniques in general which are regarded as being in the rendering domain. The question arises of whether displacement mapping is a rendering or modelling technique. Other texture mapping techniques fall squarely into the rendering category as they describe surface characteristics, such as colour, of an existing surface, the surface itself having been produced by the modelling process. Displacement mapping sits uneasily in either the rendering or the modelling category. It is applied after the traditional modelling step as a surface must exist before displacements can be applied to it. Although both the displacement map and the object are modelled, the result of displacing one with the other need never have been created before rendering. On the other hand, displacement mapping must be considered as part of the visible surface calculation during rendering. This occurs well before any other texture mapping is performed.

When applied to surfaces the scale of the displacements used is very important. They must be large enough to visibly change the geometry of the surface. The size of the displacements can be classified into 3 types[WAT92]. The first is micro-scale. This describes displacements which are far smaller than the extent of the surface. The effect of these is too small to alter the visible geometry of the surface but they will alter the way light is reflected and absorbed by the surface. Displacements of this size must be handled within the lighting model (such as the micro-facets in the Torrance-Sparrow model) and will not be considered within the realm of displacement mapping. The other two categories of displacement size are milli-scale and object scale. Milli-scale displacements are those which alter the surface by small visible amounts but do not radically alter the geometry of the surface. The scale of these is such that they cannot be considered part of the lighting model. These displacements are commonly rendered by using "bump mapping". In general bump mapping provides a good approximation to the desired surface, but as can be seen in figures 1.1 and 1.2, displacement mapping provides more accurate images. The final type of displacement scale is object scale. This is a scale at which the displacements radically alter the surface and cannot be handled except by using the true displacement-mapped surface.

The milli-scale and object scale displacements are the ones considered in this thesis. This does not preclude the use of micro-scale effects to simulate roughness on the surfaces.

To date little research has been carried out into displacement mapping. Bump mapping [Bli78b], although not a true displacement mapping technique, was the first attempt to tackle the issue of texture mapping to alter geometry (or at least fake it). Since then displacement mapping has been mentioned in connection with the REYES[CCC87] system (but not properly investigated). The only work on ray tracing displacement-mapped

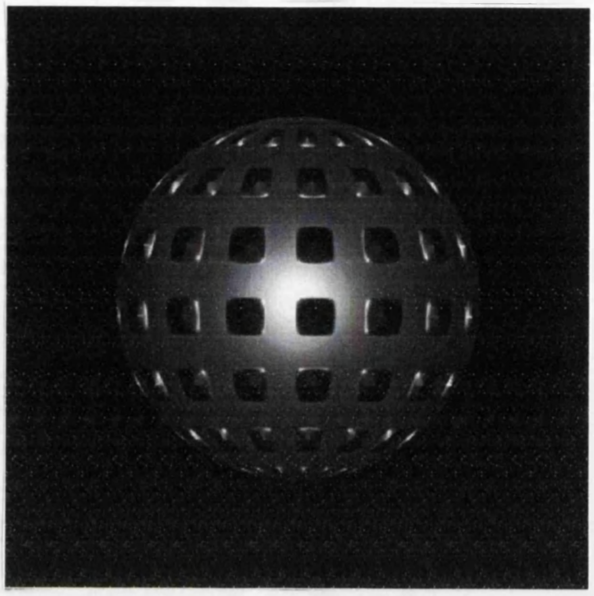


Figure 1.1: Bump-mapped sphere

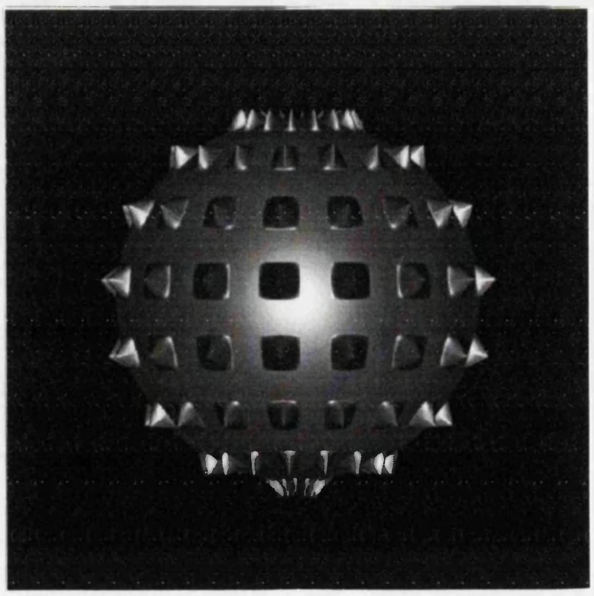


Figure 1.2: Displacement-mapped sphere

surfaces is the technique of "Inverse Displacement Mapping"[PHL91, LP95]. A discussion of this is postponed until Chapter 2.

1.2.1 Bump mapping

Bump mapping is a technique which simulates the effects of displacement mapping in the case where $\underline{d}$ is the unit normal to the surface. The idea behind bump mapping is that for textures whose heights are small compared with the spatial extent of the surface, the effect of the displacements can be accounted for by altering the way the light reflects off the surface without altering the geometry of the surface.

Since the displacement-mapped surface is only simulated, bump mapping needs only to find a point on the base surface (which is comparatively easy to find) and not the true point on the displacement-mapped surface. Given this point we can find the normal to the surface, as if it had the displacements applied, from

$$N = n + \frac{\left[n \times \frac{\partial f}{\partial u} - n \times \frac{\partial f}{\partial v}\right]}{|n|} \tag{1.2}$$

For the derivation of this equation see[Bli78b].

This approach is quick and easily implemented but has a number of major drawbacks. The bumps rendered must have heights which are small compared with the extent of the surface. If the bumps are too large then they should noticeably change the silhouette of the object. But, since bump-mapping calculates intersections only with the base surface then the silhouette can only be that of the base surface. This is clearly seen by comparing the figures 1.1 and 1.2. The spikes on the two spheres are still fairly small at one tenth of the radius of the sphere but the effect of true displacement mapping is dramatically different. With the original form of bump-mapping the shadows produced were formed solely from the geometry of the base surface. This problem has been tackled by a number of authors. As was shown in[Max88] it is possible to produce the effect of self-shadowing with bump-mapping. This is achieved by creating tables to hold approximations for the shadows from one bump to another and using this information to create self-shadowing. Although this technique can solve the problem of self-shadowing it has high memory requirements and appears fraught with problems. Further, the shadows from other bump-mapped surfaces onto other objects can be improved by starting the shadow rays from displaced points[NS94]. This can also help improve the realism of the bumped-mapped surfaces by giving a shadow with a true displacement-mapped shape. Although giving visually improved shadows, the resulting shadows are still incorrect as the shadow rays start from a point displaced from the intersection with the base surface and not from the true intersection with the displacement-mapped surface. These techniques, although useful, can not remove the inherent limitations of bump-mapping.

Thus, bump mapping provides an adequate solution to displacement mapping only in the case of small texture heights[1]. If it is used with large texture heights then the approximations it involves become too severe and a method which does alter the underlying surface must be used instead.

1.2.2 REYES image rendering architecture

The REYES (Render Everything You Ever Saw) architecture[CCC87] was developed at Pixar to provide a system for rendering very complex scenes, in a reasonable time and to photo-realistic[2] quality. This architecture was designed to be used in the production of feature length animated films. The crucial ideas behind the architecture are the use of a Z-buffer[AWW85, RSC87] to perform the visible surface calculations and the reduction of all primitives to sub-pixel sized quadrilaterals, called micro-polygons, before rendering. Using this architecture displacement-mapped surfaces have been rendered. An overview of the REYES algorithm is given in figure 1.3.

As is clear from figure 1.3 each primitive is read in and rendered before the next one is considered. From the rendering as opposed to the shading point of view four procedures must be given for each class of object.

Bound this function returns a bound for the extent of the given object and the object is then guaranteed to lie within that bound.

Dice this function reduces the object to a list of micro-polygons.

Split if an object can't be diced for some reason then this function reduces the object to a number of other objects of the same or different types.

[1]Bump mapping was never proposed as a general solution to displacement mapping.
[2]A photo-realistic image is one which is indistinguishable from a live action motion picture photograph.

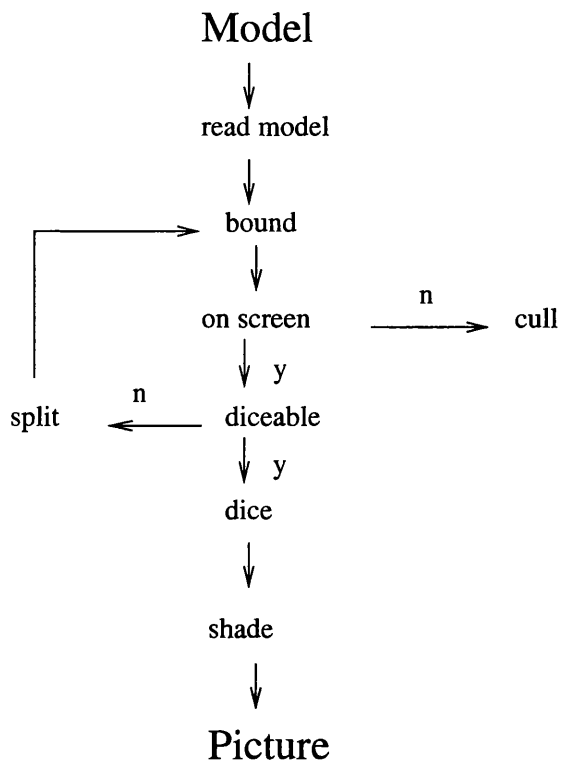


Figure 1.3: The REYES architecture

Diceable this function determines whether or not a object can be diced to micro-polygons or whether it must be split.

The published description of the architecture states that this architecture can be used to perform displacement mapping but gives no details of how this can be done. It also states that no research has been done on the use of large displacements.

If true displacement mapping is to be performed with the REYES architecture then a method for dicing, splitting and bounding displacement-mapped objects must be found. Also, it would be useful if a method for doing this depended only on the split, bound and dice routines of the base surface, the texture interpolation scheme and the displacement function. If this was the case then it would be easy to add new primitives which could be displacement-mapped and to add new displacement functions. The ability to extend the architecture easily was one of its design principles.

If such a scheme exists then it would be reasonable to assume that the displacement-mapped surface was diceable only if the base surface was diceable. The ability to dice the displacement-mapped primitive when the primitive itself cannot be diced leads to a

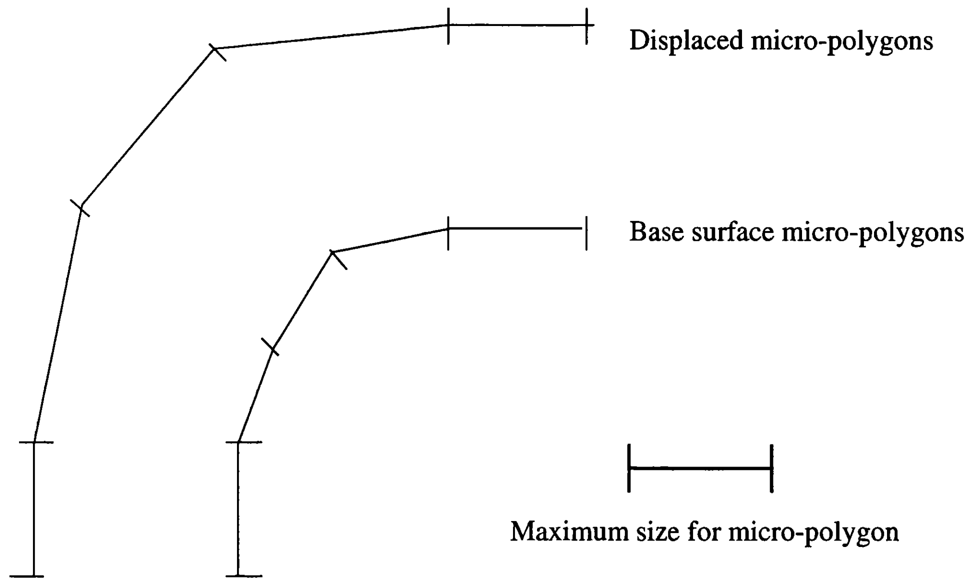


Figure 1.4: The displaced micro-polygons.

contradiction if a displacement of zero magnitude is used. If the surface had to be split it would be into primitives of the same or simpler types and, in the implementation, the displacement mapping functions could be assumed to exist for simpler types as they would have been implemented first.

It is at this point that the first problem arises. It is not sufficient to dice the base surface and to displace these micro-polygons as this may give much larger polygons than required, as is shown in figure 1.4.

This is a major problem as it implies that the dice routine for the primitive does not give enough information about the primitive to allow a displacement-mapped version to be diced. Thus, special code would have to be written to handle displacement mapping for each type of primitive. A discussion of this point took place on the Usenet news group `comp.graphics` in which a PIXAR employee stated the solution they used was to dice displacement-mapped objects "*a bit finer*"!

There are a number of other problems with the REYES architecture which make it unsuitable as a platform to develop displacement mapping:

- effects such as reflection, refraction and motion blur must be approximated because there exists no means to calculate them exactly with a Z-buffer.
- the method suggested for specifying displacements is shade trees[Coo84] and no

details exist about how this can be done.

- the desired speed for rendering (3 minutes per frame) has not been achieved for the types of scenes the architecture was designed to render and it is an open question as to whether this is possible. For problems of this complexity the only method of achieving this speed with the technology of the foreseeable future is to use parallel machines and this architecture has no clear parallel implementation.

Thus the REYES Architecture, although it can handle displacement mapping, is not overly suitable for the job because of its inherent limitations. Even so, some of the few published images[Ups90] using displacement mapping have been produced by an implementation of this system called RenderMan and to date this is the most complete photo-realistic rendering system in general use today.

1.3 Ray tracing

Ray tracing[Gla89] is a powerful rendering technique which combines a solution to the visible surface problem with a global lighting model. The core idea behind ray tracing is to trace beams of light through the scene and see what they hit. In this thesis ray tracing refers to backward ray tracing. This term arises since the rays of light are traced backwards from the eye point through the scene to the light source(s). In reality light will travel in the opposite direction but it is highly inefficient to work forwards from the light source, since the majority of the light from a source will miss the eye point.

Ray tracing works by taking a single (primary) ray from the eye point through each point on the screen (image) and seeing which object it hits first. This object will be the visible one. If a ray is traced from the intersection point on the object towards each of the light sources then shadows can be calculated. If the ray hits any object before it hits the light source then there is an object blocking the light and the object must be in shadow. By tracing the path of the light through the scene, techniques from geometric optics can be used to simulate reflection and refraction. In these cases, when a ray hits an object, the ray will split into two parts to give reflected and refracted rays. The colour of a given point can be calculated as the combination of all the sub-rays. This process is referred to as recursive ray tracing as show in figure 1.5.

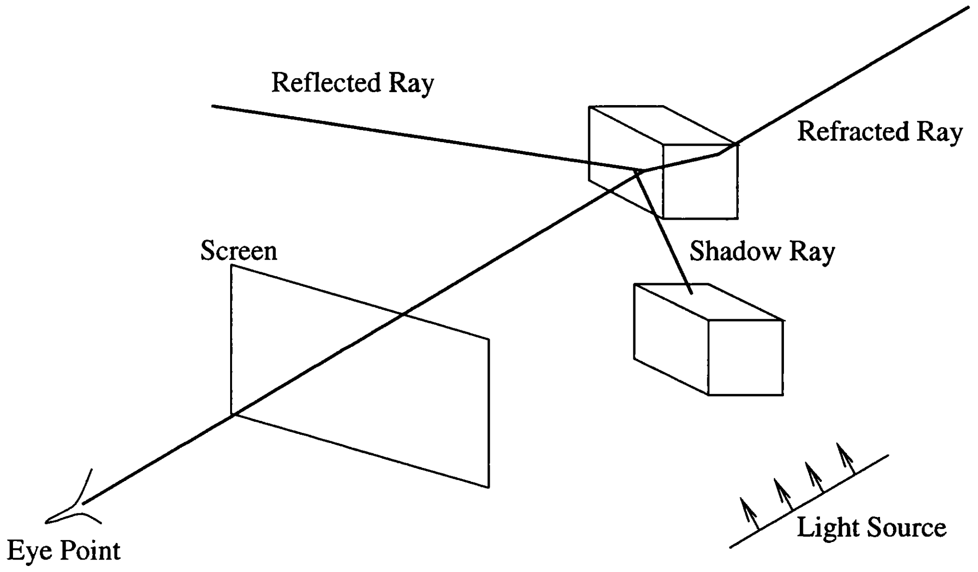


Figure 1.5: Recursive ray tracing

The core computation in the ray tracing algorithm is the intersection of a ray and an object. Intersection algorithms exist for the majority of surfaces used in computer graphics today. These include polygons, quadrics ,superquadrics[Bar81], swept surfaces[Kaj83, Wij84], surfaces of revolution[Kaj83, Wij84], general swept surfaces[Wij85, BK85], implicit surfaces[Han83, KB89, Mic90], fractals[Kaj83], spline surfaces[RW80a, SA84, Ste84, Tot85, JB86, SB86, Yan87, LG90, SNK90], deformed surfaces[Bar87], fractals[Kaj83, Bou85, HSK89, HD91], etc[Hec87].

1.3.1 Current research

The existing research into ray tracing divides broadly into two areas. These areas are optical effects and geometric techniques. The optical effects are characterised by the initial assumption that a ray can be traced through the scene and the intersection points calculated. With this information, rays can be cast in any desired direction through the scene and the values they return combined to simulate the optical effects[Kaj86, Coo89]. Examples of these include depth of field, motion blur, lens, penumbra, and diffuse reflections. The geometric techniques, on the other hand, are concerned with efficiently intersecting the ray and the objects in the scene. This is performed by providing efficient routines to intersect a ray and an object, and by the use of special data structures[RW80b, WHG84, Gla84, FTI86, KK86, AK87, SB87, HT92] to reduce the number of ray-object intersections which must be performed.

In most cases there is a clear distinction between these techniques. This arises since the first group traces different types of rays and combines them to give their results and the second group trace the rays regardless of what the rays are used for.

The geometric techniques used are unrelated to the optical effects and vice-versa[3]. The work in this thesis is concerned with ray tracing displacement-mapped surfaces and thus falls into the geometric category. The techniques used are designed to be general and preserve the separation of optical and geometric techniques.

1.3.2 Geometric techniques in ray tracing

In recent years a large amount of research has been published on speeding up the ray tracing algorithm. This has concentrated on two main areas. The first is the ray-surface intersection calculation. The techniques used are specific to a given surface type, and fast algorithms exist to intersect rays with the most common surface types. This line of research has produced limited gains in speed. The speed of the intersection routine is inherently limited by the complexity of the surface and for complex surfaces (such as spline surfaces) this calculation is still comparatively slow.

The second area of geometric techniques in ray tracing has concentrated on reducing the number of ray-surface intersections which must be performed. In the earliest ray tracers the objects in the scene were usually stored in a linear list. This meant that, to find the closest intersection of the ray and the scene, the ray had to be intersected with every object. As the size of the scenes grew this problem became intolerable. The first attempts to solve this problem involved storing the scene in a hierarchical structure[RW80b, WHG84, KK86] with bounding boxes around the objects. If the ray missed the bounding box then the ray must miss all the objects contained therein providing a considerable saving. A related approach was to split the space in which the scene lies into cells[Gla84, FTI86, HT92] (uniformly or adaptively) and then to trace the path of the ray through the cells. If all the objects are classified to the cells they intersect then the ray need only be intersected with a small subset of the objects in the scene. The latest approaches[AK87] have split up not the space of the objects but the space of the rays. The objects are classified as to whether

[3]This is not strictly true as some geometric techniques are specific to given optical effects.

a given set of rays could hit them. This allows the objects on the path of the ray to be found very quickly. These three approaches, along with a myriad of hybrid combinations, have dramatically speeded up ray tracing. The state of the art techniques can process a ray in time almost independent of the number of objects in the scene.

1.4 Ray tracing displacement-mapped objects

Currently, little research has been conducted into ray tracing displacement-mapped objects. It is unclear exactly why this is so but a number of possible explanations exist.

- It is not considered worth the effort – Displacement mapping is not a common requirement of rendering systems. Since few people have access to it, it is not regarded as necessary and hence few people see a need for it. This is a poor reason for its absence, especially since bump mapping is commonly used and displacement mapping can greatly improve and extend the same effects.
- It is considered too expensive to be practical – A number of well known techniques can be thought a priori to be adaptable to handle displacement mapping. These can be based on polygonalisation or numerical techniques, both capable of handling general surfaces. Unfortunately, the complexity of displacement-mapped surfaces pushes both of these approaches beyond their practical limits.
- Optimisation problems in ray tracing are considered solved – Major journals, in particular SIGGRAPH, regard the optimisation problems in ray tracing as having been solved. The recent published work in the area has concentrated on hybrids of known techniques which although faster don't provide anything radically new. This climate is not conducive to new research in the area.

Together these reasons may explain the current inability of existing ray tracers to handle displacement-mapped surfaces.

Within the field of ray tracing complex surfaces, current software architectures for ray tracing suffer from a major space-time trade-off. Direct techniques for calculating the ray-surface intersection must use some form of numerical calculation if complex surfaces are used. This arises since no closed form solutions exist to equations of degree 5 or higher (a ray-bicubic patch intersection has degree 18). In these cases an iterative approach must be used and these, especially if solutions must be guaranteed, are very slow. This problem becomes worse as the complexity of the surface increases. Numerical techniques can be avoided by first reducing the complex surface to a number of simpler surfaces. These simpler surfaces are chosen so that they can be ray traced quickly. The primitive surface chosen is usually a planar polygon. This approach replaces the single complex numerical calculation with a series of simple ray-polygon intersections. Although this speeds up the ray tracer, a considerable amount of memory must be used to store the polygons (and the related hierarchy). This places an upper bound on the complexity of the scenes which can be handled.

Displacement-mapped surfaces, being very complex, multiply these problems. The mathematical complexity of the surfaces makes numerical techniques prohibitively expensive while the irregularity of the surface necessitates very large numbers of polygons. This makes current techniques impractical for displacement mapping.

1.5 Layout of thesis

This thesis presents three new algorithms for ray tracing displacement-mapped surfaces. The first of the algorithms is "inverse displacement mapping" presented in Chapter 2. This provides a direct solution to the intersection calculation and stems from an analysis of the geometry of the intersection calculation. The problem is inverted from a straight ray and a complex surface to a curved ray and a simple surface. This inverse geometry is then analysed to extract the information necessary to quickly calculate the intersection. The second algorithm presented in Chapter 3 calculates the solution to the ray displacement-mapped surface intersection directly by the use of numerical techniques. The problem is cast algebraically without inverting it or using specific geometric information. This facilitates a solution in cases where an analysis of the geometry is impractical. The final algorithm, presented in Chapter 4, provides an indirect algorithmic solution to the intersection problem. The intersections are calculated indirectly by polygonalising the surface and then intersecting the ray with the generated polygons. Algorithmic techniques are developed to efficiently handle the generation, storage and retrieval of these polygons. This approach provides a solution which, unlike the previous ones, is relatively independent of the algebraic or geometric complexity of the underlying surface. The final chapter of this thesis presents the results of the three algorithms and shows that, for the first time, displacement-mapped surfaces can be practically ray traced.

Chapter 2

Inverse Displacement Mapping

2.1 Introduction

This chapter presents the technique of "inverse displacement mapping"[PHL91, LP95]. This is a geometric approach which provides a direct solution to the ray displacement-mapped surface intersection problem. This approach is designed for the case of the unit normal as the displacement function. The name "inverse" displacement mapping arises from performing the intersection calculation in the inverse (i.e. parametric) space of the surface not directly in Euclidean space. The algorithm uses an iterative approach. The intersection points are first bound on segments of the ray between two surfaces. These surfaces are iteratively (recursively in the implementation) brought closer together until, at the limit, the solution is found. A procedure exists to determine if a segment of the ray contains no solutions, allowing large segments to be rejected quickly. The work given in this chapter differs from the original presentation of the work in[PHL91] in a number of ways. The algorithms here are presented for a variety of free-form surfaces (not just spheres) and a number of important optimisations are developed.

The algorithm itself is divided into two parts. The first is the base case. This is a special case of the ray displacement-mapped surface intersection problem which can be solved efficiently. The second is segment classification. This handles the general case by reducing it to a number of instances of the base case. This design allows an efficient implementation by removing much of the complexity to a (per ray) preprocessing stage. It also allows a separation of the surface specific features in the classification from the generic base case algorithm, thus allowing the algorithm to be applicable to a wide range of surfaces. Unfortunately, this theoretical range of surfaces cannot be realised in practice. The speed of the algorithm is critically dependent on the complexity of the underlying surface and becomes too slow for use with complex base surfaces. This said, inverse displacement mapping is still a very powerful technique for ray tracing displacement-mapped surfaces.

The details of inverse displacement mapping are given in the remainder of this chapter. Before the algorithm is presented the definitions and terminology used are given. This is followed by the base case and classification algorithms. Finally, a discussion of the theoretical and practical aspects of inverse displacement mapping are presented. The practical details necessary to implement it for a variety of surfaces are given in Appendix A.

2.2 Definitions and terminology

A number of definitions are required to describe inverse displacement mapping. Throughout this chapter the surface to which the displacements are applied, the base surface, is denoted by $f(u,v)$. The range for u and v is $(u,v) \in [u_{min}, u_{max}] \times [v_{min}, v_{max}]$. The texture function is denoted by $t(u,v)$ with the same range as f.

When the ray is projected into texture space all calculations are performed in terms of u, v and h. The third dimension, h, gives the height of a point above the base surface. All the heights are measured as the distance along the unit normal from the surface. The offset surface $O_{f,h}$ to f at a height h is defined as the surface a constant distance h above f. It is defined by

$$O_{f,h} = \left(f + \frac{\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}}{\left| \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} \right|}.h \right) (u,v)$$

Since the base surface is f and the parametric variables are (u,v) throughout, this can be abbreviated to O_h without confusion.

Two other functions are needed to describe inverse displacement mapping. If the ray under consideration is $\underline{r}$ then these are $D_f(\underline{r})$ and $P_f(\underline{r})$, the distance from the ray to the surface and the path of the ray over the surface respectively. In both cases, points are projected along the normal to the surface onto the surface itself. These can theoretically be described as functions of a single parameter (i.e. the ray parameter). In practice, since a point on the ray may correspond to more than one valid point on the surface, this can be difficult. This problem is alleviated since D_f and P_f are used only in the classification stage and then only indirectly. Further discussion of D_f and P_f will be postponed until their uses are shown.

With the terminology defined above the algorithm can be presented.

2.3 The base case

The base case algorithm is an algorithm to intersect a ray and a displacement-mapped surface. It works only for one special case of the problem but the general case can be reduced to a finite number of instances of the base case. Before the base case algorithm is given, the properties necessary for it will be presented and explained.

2.3.1 Defining properties

There are three defining properties for the base case algorithm.

1. O_h is well defined for $h_{min} \leq h \leq h_{max}$
2. D is an increasing or decreasing function
3. P is an increasing or decreasing function

These properties raise two questions, "What do they mean?" and "Why were they chosen?".

The first property ensures that the offset surface can be safely used for all heights between h_{min} and h_{max} (The limits denoting the minimum and maximum displacement heights). This property also ensures that D and P are well defined. This will be justified later (see 2.4).

The second property ensures that the ray segment is either approaching or leaving the surface as shown in figure 2.1. This implies that for any height h between h_{min} and h_{max} there is a unique point on the ray segment which corresponds to the height. Also, the maximum and minimum heights occur at the end points of the segment. These heights are used to control the base case algorithm and this property ensures they can be calculated efficiently.

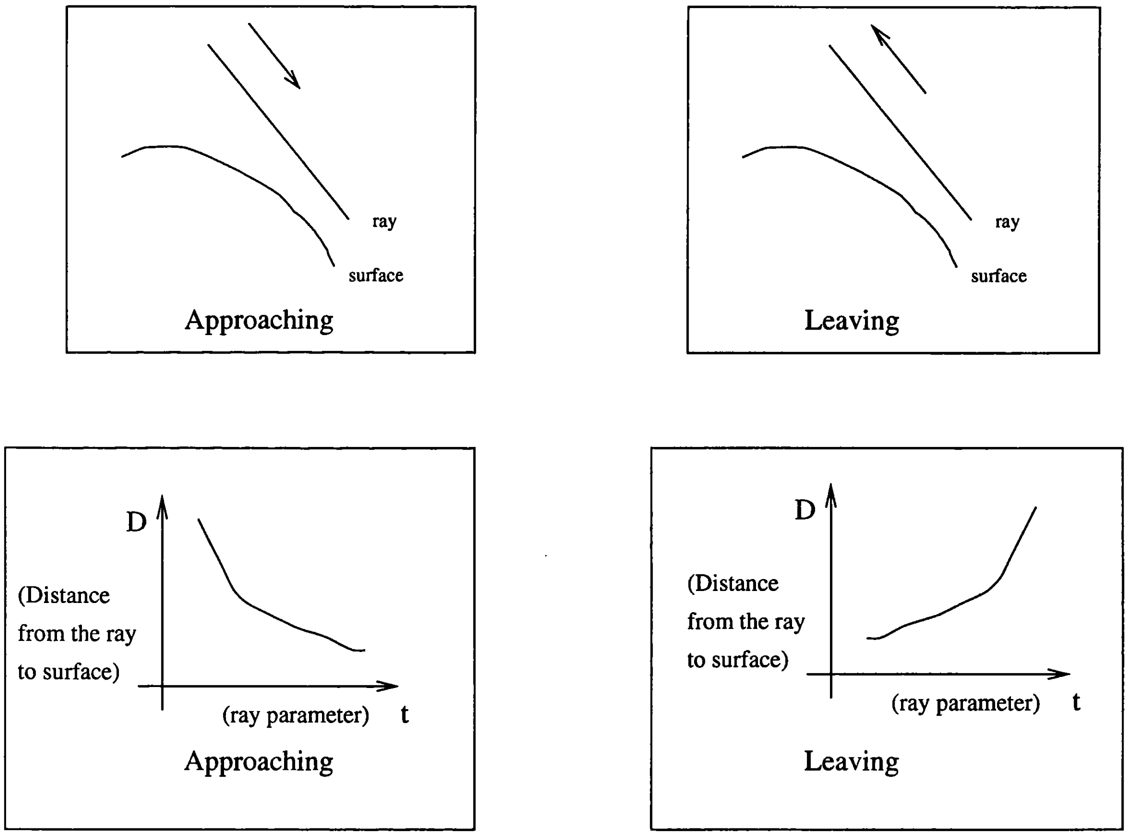


Figure 2.1: Geometric meaning of property 2

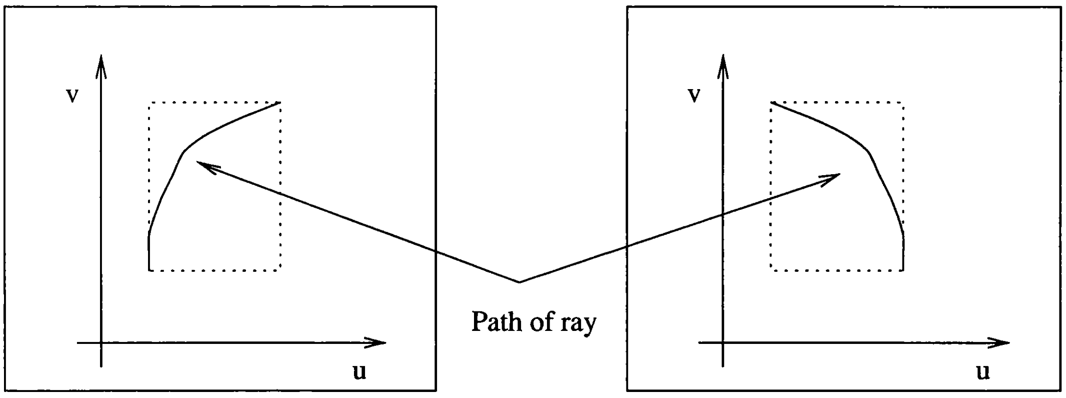


Figure 2.2: Geometric meaning of property 3

The third property ensures that the path of the ray over the surface has no local maxima or minima on the segment of the ray as shown in figure 2.2. Thus, as in the previous case, the end-points of the segment (when projected into UV-space) give the maximum and minimum UV-values. This allows an efficient bound on the area of the texture over which the ray passes to be calculated. This area will be searched for maximum and minimum height values to decide if the ray segment contains no solutions. This property ensures a tight and quickly calculated bound for the search.

These properties together are defined in such a way that, if true for a region of parameter space then, they are also true for any subregion. This allows the algorithm to proceed recursively without explicitly checking the properties are true for the subregions.

2.3.2 Base case algorithm

This section presents the algorithm to intersect a ray and a displacement-mapped surface subject to the conditions in the previous section.

The inputs to the algorithm are two height values h_{start} and h_{end}. These define the heights of the offset surfaces which bound the ray segment under consideration. The height h_{start} corresponds to the surface which the ray segment hits first. Initially h_{start} and h_{end} will correspond to the maximum and minimum displacement heights (i.e. h_{max} and h_{min}).

The first stage of the algorithm is to intersect the ray with the offset surfaces $O_{h_{start}}$ and $O_{h_{end}}$. This generates two intersection points (u_{start}, v_{start}) and (u_{end}, v_{end}) as shown in figure 2.3. These points are tested to see if they are close enough together to find if a solution can be determined. By joining the opposite corners of an individual texel, four

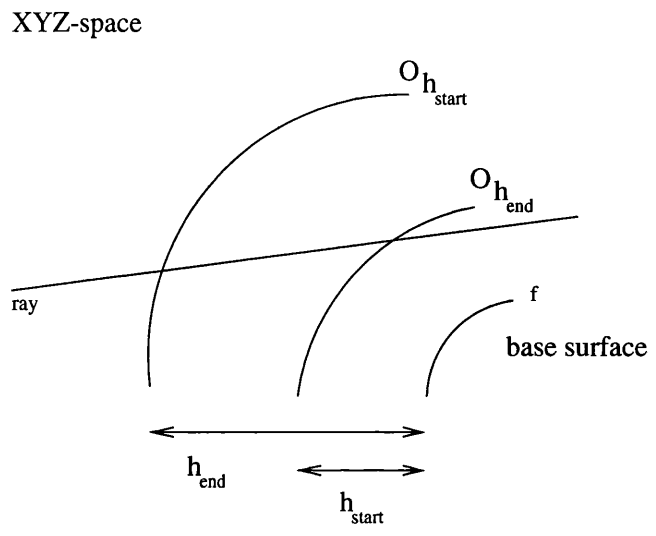


Figure 2.3: Initial geometry for the base case

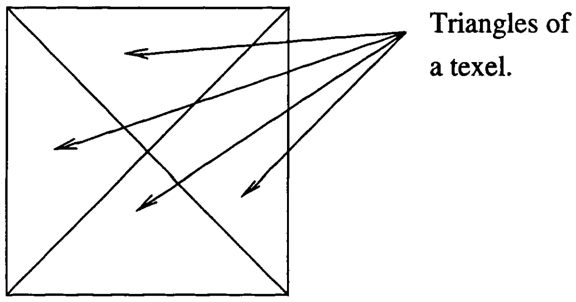


Figure 2.4: Texel geometry for termination

triangles are formed as shown in figure 2.4. If both solution points lie within the same triangle of a texel and the distance between the points is less than a preset tolerance then the intersection can be determined. When the points are close enough together their values are averaged and the height of this point calculated from the texture function. If this height is greater than the minimum of h_{start} and h_{end} then the ray misses the surface, otherwise the ray hits the surface. If required the XYZ-space intersection point can be calculated and returned.

If the intersections with the bounding surfaces are too far apart then the ray segment is split into two and each segment recursively processed. The segment is split at the height $h_{mid} = (h_{start} + h_{end})/2$ (by definition this uniquely defines a point on the ray segment). The ray is then intersected with the offset surface $O_{h_{mid}}$ to generate the intersection point (u_{mid}, v_{mid}). This defines two new segments from h_{start} to h_{mid} and h_{mid} to h_{end}, which will

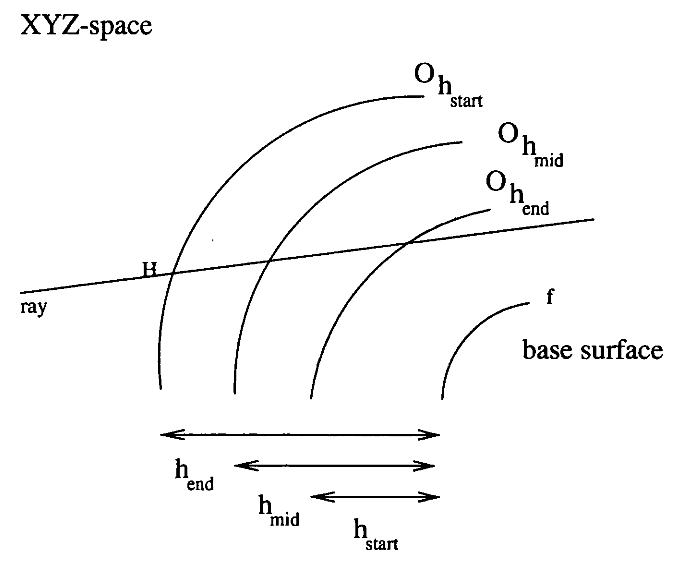


Figure 2.5: Mid-point geometry

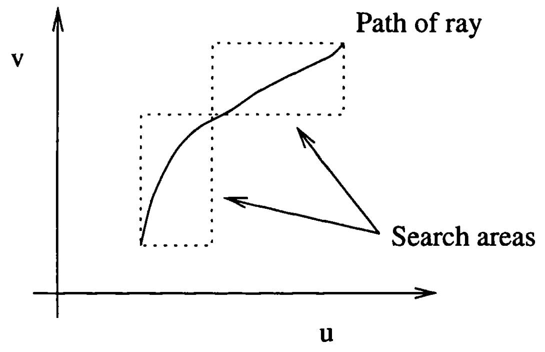


Figure 2.6: Path geometry

be recursively processed. This is shown in figure 2.5. If the segments are processed in this order then the intersection points will be calculated in order of increasing ray parameter. By calculating the solutions in this order the first solution determined will be the solution closest to ray origin. The algorithm may terminate here if, as is frequently the case, only the closest solution is required.

The points (u_{start}, v_{start}) and (u_{mid}, v_{mid}) define a rectangle in UV-space which bounds the path of the ray over the surface as shown in figure 2.6. This area of the texture is searched to find the maximum and minimum texture heights, h_{max} and h_{min} respectively. These bound the height of the displacement-mapped surface under the segment of the ray. If all points on the ray segment are higher than the maximum height of the texture under the ray then the ray must miss the surface. This condition occurs when

$$\min(h_{start}, h_{mid}) > h_{max}$$

If this is true the segment can be rejected. If the segment cannot be rejected it is recursively processed by calling the base case algorithm. For the recursive call the heights for the start and end surfaces are

$$h_{start} = \begin{cases} \max(h_{start}, h_{min}) & h_{start} < h_{mid} \\ \\ \min(h_{start}, h_{max}) & h_{start} > h_{mid} \end{cases}$$

$$h_{end} = \begin{cases} \min(h_{mid}, h_{max}) & h_{start} < h_{mid} \\ \\ \max(h_{mid}, h_{min}) & h_{start} > h_{mid} \end{cases}$$

If no intersections are found on the first ray segment then the segment h_{mid} to h_{end} must be considered. This is done in a similar way to the first segment except the segment is rejected if

$$\min(h_{mid}, h_{end}) > h_{max}$$

and the recursive call is made with heights

$$h_{start} = \begin{cases} \max(h_{end}, h_{min}) & h_{end} > h_{mid} \\ \\ \min(h_{end}, h_{max}) & h_{end} < h_{mid} \end{cases}$$

$$h_{end} = \begin{cases} \min(h_{mid}, h_{max}) & h_{end} > h_{mid} \\ \\ \max(h_{mid}, h_{min}) & h_{end} < h_{mid} \end{cases}$$

If no intersection is found on the second segment of the ray then no intersections exist.

The complete base case algorithm is summarised in pseudo code in figure 2.7.

Before the general case is considered a number of points must be made about the base case algorithm. The intersection of the ray with the offset surface can generate multiple

```
base_case_intersect($h_{start}, h_{end}$)
    begin
    Intersect ray with $O_{h_{start}}$ to get $(u_{start}, v_{start})$
    Intersect ray with $O_{h_{end}}$ to get $(u_{end}, v_{end})$
    if within_texel_triangle then
        if solution exists return true
    endif

    Intersect ray with $O_{h_{mid}}$ to get $(u_{mid}, v_{mid})$

    Search texture $(u_{start}, v_{start})$ to $(u_{mid}, v_{mid})$
    if min$(h_{start}, h_{mid}) > h_{max}$ then
        if $h_{start} < h_{mid}$ then
            if base_case_intersect(max$(h_{start}, h_{min})$,min$(h_{mid}, h_{max})$)then
                return true
            endif
        else
            if base_case_intersect(min$(h_{start}, h_{max})$,max$(h_{mid}, h_{min})$)then
                return true
            endif
        endif
    endif

    Search texture $(u_{end}, v_{end})$ to $(u_{mid}, v_{mid})$
    if min$(h_{end}, h_{mid}) > h_{max}$ then
        if $h_{mid} < h_{end}$ then
            if base_case_intersect(max$(h_{mid}, h_{min})$,min$(h_{end}, h_{max})$)then
                return true
            endif
        else
            if base_case_intersect(min$(h_{mid}, h_{max})$,max$(h_{end}, h_{min})$)then
                return true
            endif
        endif
    endif

    end
```

Figure 2.7: The base case algorithm

valid intersection points. The one to be used by the base case algorithm is the only one within the bounds of the ray segment (The uniqueness is ensured by the defining properties of the base case algorithm.). This may be easily accomplished if the u, v and ray parameter values are passed along with the offset height. Further, in many cases, the height of the offset surface is unchanged after comparison with the heights from the texture search. In this case, the intersection with the offset surface need not be recalculated. This provides an important optimisation.

2.4 Classifying the special cases

The algorithm, as presented so far, will work only for one special case of the problem. This section will show how inverse displacement mapping can be handled in the general case. Instead of extending the base case algorithm, the general case will be reduced to a number of instances of the base case. This is done by splitting the ray into segments such that each satisfies the conditions for the base case. The conditions for the base case are clearly defined and these will be examined in turn to decide where the ray must be split into segments.

The first condition necessary for the base case algorithm is that the offset surface, O_f, is well defined between the maximum and minimum displacement heights. For the offset surface to be well defined the base surface, f, must be well defined. This will be assumed as inverse displacement mapping is applicable only to well defined surfaces. The second condition for the offset surface is that the normal, $\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}$, is well defined. This occurs at all points except where one or both of the partial derivatives is zero or the partial derivatives are parallel. The partial derivatives are parallel at points where the parametric surface collapses to a single point, e.g. at the poles of a sphere. These points usually occur at the boundaries of the parametric domain and, since the ray is split at these points anyway, this condition poses no problems. Certainly, for the surfaces to which inverse displacement mapping can be practically applied this is the case.

If either partial derivative term is zero then the cross-product term will be zero and the normal is undefined. This occurs at turning points on the surface. These may occur at isolated points or along curves on the surface and can be local maxima, local minima or points of inflection. If the turning points occur at isolated points then the surface can be split into sub-surfaces at these points. This can be performed as a preprocessing stage and will thus not directly affect the algorithm. If the turning points occur along curves on the surface the ray can be split at the points where the path of the ray over the surface, P, intersects these curves. This solution glosses over the practical difficulties and computational expense of computing the curves and their intersections. As shall be seen later the normal to the surface is well-defined for those surfaces for which inverse displacement mapping is practical and this condition presents no problems.

The second condition for the base case algorithm is that the distance function, D, is an increasing or decreasing function. This means that ray must be either approaching or leaving the surface. A typical situation for the distance from the ray to the surface is as shown in figure 2.8. In this case the ray must clearly be split at four points. These points occur at the boundaries where the ray hits the largest and smallest offset surfaces, i.e where the ray hits $O_{h_{max}}$ and $O_{h_{min}}$, and at points between these where the distance function has a local maximum or minimum. Theoretically, the function D can be expressed as a function of one variable, α, the ray parameter. If this can be done then the local maxima and minima occur when

$$\frac{dD}{d\alpha} = 0$$

This condition is also true when D has a point of inflection. It is not necessary to classify the type of turning point or remove splits at the points of inflection. The turning points must only be culled to ensure that all points generated lie within the parametric bounds of the surface.

The local maxima and minima can be calculated even if no simple expression exists for D in terms of the ray parameter α. It can be shown that for $f = (f_x, f_y, f_z)$, $n = \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} = (n_x, n_y, n_z)$ and $r = (a_x, a_y, a_z) + \alpha.(b_x, b_y, b_z)$

$$D = \frac{(f_y - a_y)b_x - (f_x - a_x)b_y}{n_x b_y - n_y b_x}$$

defines the distance from a point on the ray to points on the surface. This equation, subject to the constraint that the points lie on the path of the ray over the surface (see later for derivation), allows the maximum and minimum distances to be calculated by the

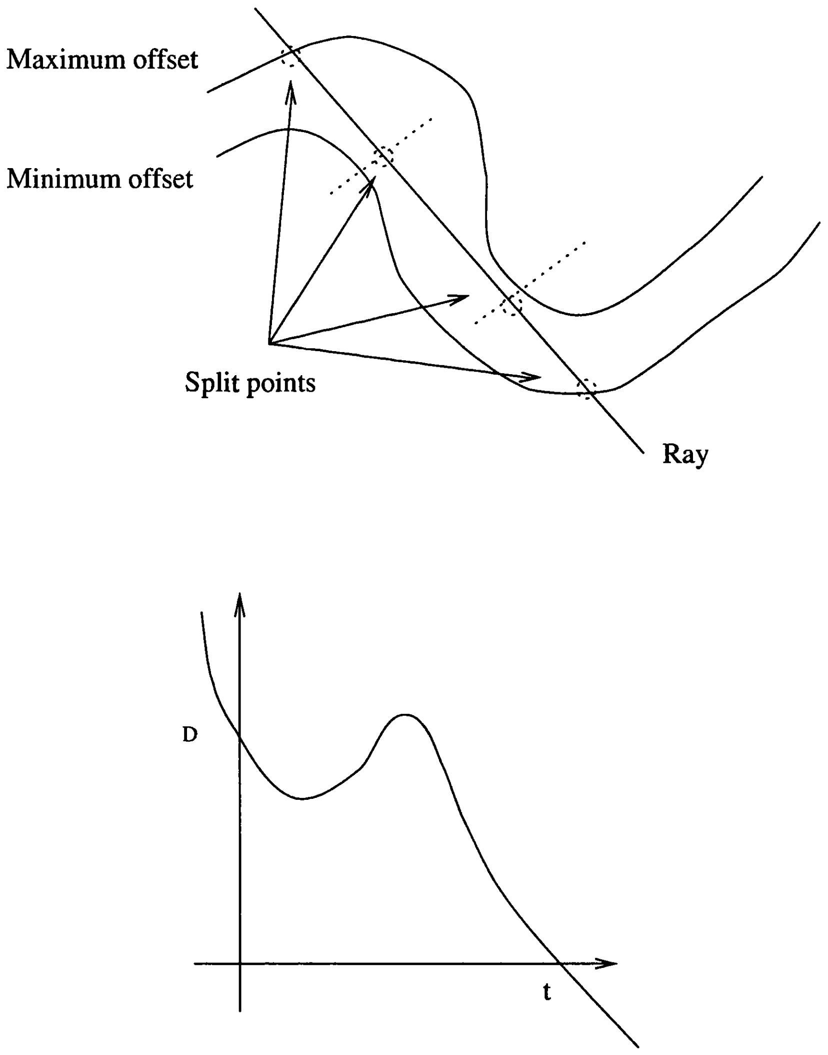


Figure 2.8: Ray-surface distance geometry

use of Lagrange multipliers. This reduces to solving the equations

$$\frac{\partial G}{\partial u} = 0, \quad \frac{\partial G}{\partial v} = 0, \quad \frac{\partial G}{\partial \lambda} = 0$$

where

$$G = \frac{(f_y - a_y)b_x - (f_x - a_x)b_y}{n_x b_y - n_y b_x} + \; nonumber \tag{2.1}$$

$$\lambda \begin{vmatrix} f_y(u,v) - a_y & f_x(u,v) - a_x \\ b_y & b_x \end{vmatrix} \begin{vmatrix} \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} \big|_x (u,v) & \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} \big|_z (u,v) \\ b_x & b_z \end{vmatrix}$$

$$- \lambda \begin{vmatrix} f_z(u,v) - a_z & f_x(u,v) - a_x \\ b_z & b_x \end{vmatrix} \begin{vmatrix} \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} \big|_x (u,v) & \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v} \big|_y (u,v) \\ b_x & b_y \end{vmatrix} \tag{2.2}$$

This produces a system of non-linear equations which must, in general, be solved by robust numerical methods. This analysis defines all the points necessary for the distance function if the offset surfaces enclose a volume of space (e.g a sphere). When the offset surfaces, over their valid parametric domain, do not enclose a volume then a volume must be created by considering "side" surfaces as shown in figure 2.9. This is necessary because a ray may pass over the base surface without intersecting the initial bounding volumes. This problem is easily solved by intersecting the ray with the four side surfaces. These surfaces are defined by

$$S(u,h) = f(u,\beta) + \frac{\frac{\partial f(u,\beta)}{\partial u} \times \frac{\partial f(u,\beta)}{\partial v}}{\left| \frac{\partial f(u,\beta)}{\partial u} \times \frac{\partial f(u,\beta)}{\partial v} \right|}.h$$

$$S(v,h) = f(\beta,v) + \frac{\frac{\partial f(\beta,v)}{\partial u} \times \frac{\partial f(\beta,v)}{\partial v}}{\left| \frac{\partial f(\beta,v)}{\partial u} \times \frac{\partial f(\beta,v)}{\partial v} \right|}.h$$

where $\beta \in \{0,1\}$ fixes u or v to lie on the boundary of the surface.

This analysis allows the ray to be split in such a way that the distance function is an increasing or decreasing function and thus satisfies the second property for the base case algorithm.

The third condition for the base case algorithm is that P, the path of the ray over the surface, is an increasing or decreasing function as shown in figure 2.10. As before P can theoretically be written as a function of the ray parameter α, so $P(\alpha) = (P_u(\alpha), P_v(\alpha))$.

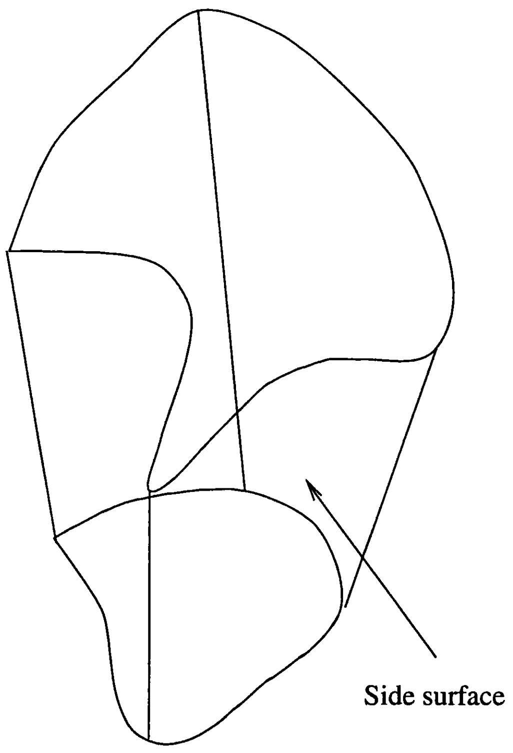


Figure 2.9: Side surfaces for the offset surface

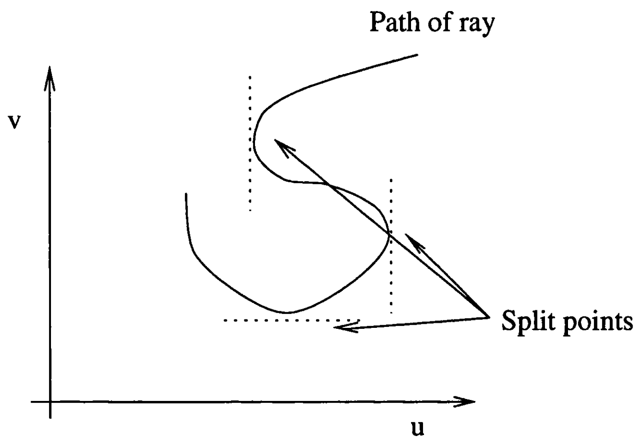


Figure 2.10: Path of ray over surface

This function can be split into appropriate segments by using the points where

$$\frac{dP_u}{d\alpha} = 0 \qquad \text{or} \qquad \frac{dP_v}{d\alpha} = 0$$

These are local maxima and minima in the UV-plane. In practice, the path of the ray can more easily be expressed as an implicit equation in u and v. This equation, using the previous nomenclature, is

$$P(u,v) = \begin{vmatrix} f_y(u,v) - a_y & f_x(u,v) - a_x \\ b_y & b_x \end{vmatrix} \begin{vmatrix} \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\big|_x (u,v) & \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\big|_z (u,v) \\ b_x & b_z \end{vmatrix}$$
$$- \begin{vmatrix} f_z(u,v) - a_z & f_x(u,v) - a_x \\ b_z & b_x \end{vmatrix} \begin{vmatrix} \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\big|_x (u,v) & \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\big|_y (u,v) \\ b_x & b_y \end{vmatrix} = 0$$

In this form the local maxima and minima occur when

$$\frac{du}{dv} = 0 \qquad \text{or} \qquad \frac{dv}{du} = 0$$

Geometrically, when the gradient is zero or infinite. The end points for the path are the same as the end points for the distance function and need not be considered again. This analysis will allow the ray to be split in such a way that the path of the ray over the surface is an increasing or decreasing function. Thus, the third property for the base case algorithm can be satisfied.

One more segment point must be considered when splitting the ray into segments. This point is related to the ray tracing itself, not the validity of the conditions for the base case. The point at which the ray parameter is zero (i.e. at the origin of the ray) is needed if the ray starts between the maximum and minimum offset surfaces. This will happen for shadow, reflected and refracted rays. In this case, if the ray starts at the point p, the equation

$$\underline{p} = \underline{f}(u,v) + \frac{\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}}{\left|\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\right|}.h$$

must be solved for u, v and h. All of the points behind this one can be removed allowing segments behind the ray origin to be culled.

Once all the segment split points have been calculated and those behind the origin removed, the remainder are sorted by increasing order of the ray parameter. Each successive pair of points define a segment and are used as input for the base case algorithm. By processing the points (which define segments) in this order, the algorithm can be terminated after the first intersection is found if desired. This intersection will be the closest one.

The discussion so far has been concerned with the presentation of the algorithm in a general setting. The next section will discuss the computational and practical aspects of inverse displacement mapping.

2.5 Computational requirements

Since inverse displacement mapping is concerned with the intersection calculations for rays and surfaces it will be carried out in the innermost loop of a ray tracer. This implies that the algorithm will be run a very large number of times (anything from tens of thousands to tens of millions of times). Thus it is vital that it runs fast. If it is assumed that the majority of the time for the algorithm is spent processing the base case, then two main tasks will dominate the computation.

1. Searching the texture.

2. Calculating the intersection of the ray and the offset surface.

By suitable preprocessing it is possible to search the texture very fast[1]. Hence the main task will be that of intersecting the ray with the offset surface. This can be very time-consuming. In the case of a plane, sphere or torus (in fact any shape made from straight lines and circular arcs) the offset surface will be of the same type as the base surface. This means, for example, that a sphere offsets to a sphere. For other surface types this is not the case and a far more complex surface type is formed. The ray must be intersected numerically with these surfaces, and as shall be seen in the next chapter, this is a very time consuming task.

One way to avoid the complexity of the numerical intersection with the offset surface is to approximate it with the same representation as the base surface. The approximation

[1]An adapted mip-mapping approach[Wil83] can be used to give very fast texture searches.

of the offset of a spline curve by a another spline curve of the same type has been addressed by a number of authors[Kla83, TH84, Coq87, JH88, Far89]. This will allow prisms and surfaces of revolution with cubic spline cross-section to be handled. The problem of approximating the offset of a spline surface has also been tackled[TH84, AU90]. Unfortunately the intersection calculation for a general spline surface is very time consuming and due to the large number of intersection calculations involved, inverse displacement mapping is impractical for general spline surfaces. For the limited number of surfaces left efficient algorithms can be found.

The above discussion relies on the assumption that the majority of the time for the algorithm is spent in the base case. If this is not so then, the majority of the time for the algorithm will be spent classifying the ray into segments. For these surfaces the base case, although faster than the classification, will still be too time consuming and inverse displacement mapping will be impractical in these cases. This causes no problems as the classification will be shown to be fast for the surfaces still under consideration.

The actual details needed to implement inverse displacement mapping for spheres, cylinders, tori and swept surfaces are given in Appendix A.

2.6 Results

The images in figures 2.11 to 2.14 show the results of inverse displacement mapping for a number of different base surfaces. The images can be seen to capture the geometry of the true displacement-mapped surface accurately. The false colour images beside figures 2.11 to 2.13 show where the time is spent in the algorithm[2], the colour brightness showing the number of recursive calls made to the base case algorithm. The majority of the time is spent around the edges of the spikes. This arises since the base case must recurse many times before the terminating condition is reached. This is what is expected as the geometry is far more complex in these regions. Further the false colour images show that the algorithm can handle the "easy" cases, where the texture heights are constant, efficiently. A more detailed discussion of the performance of inverse displacement mapping is given in Chapter 5.

[2]The data for the false colour images deals only with primary rays.

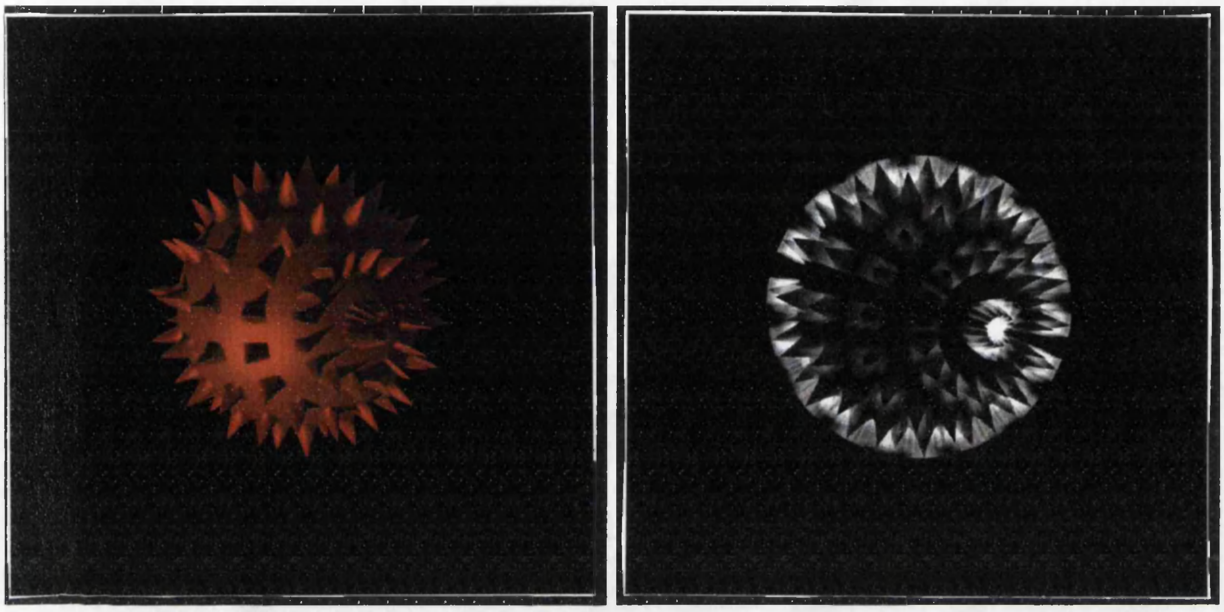


Figure 2.11: Inverse displacement-mapped sphere

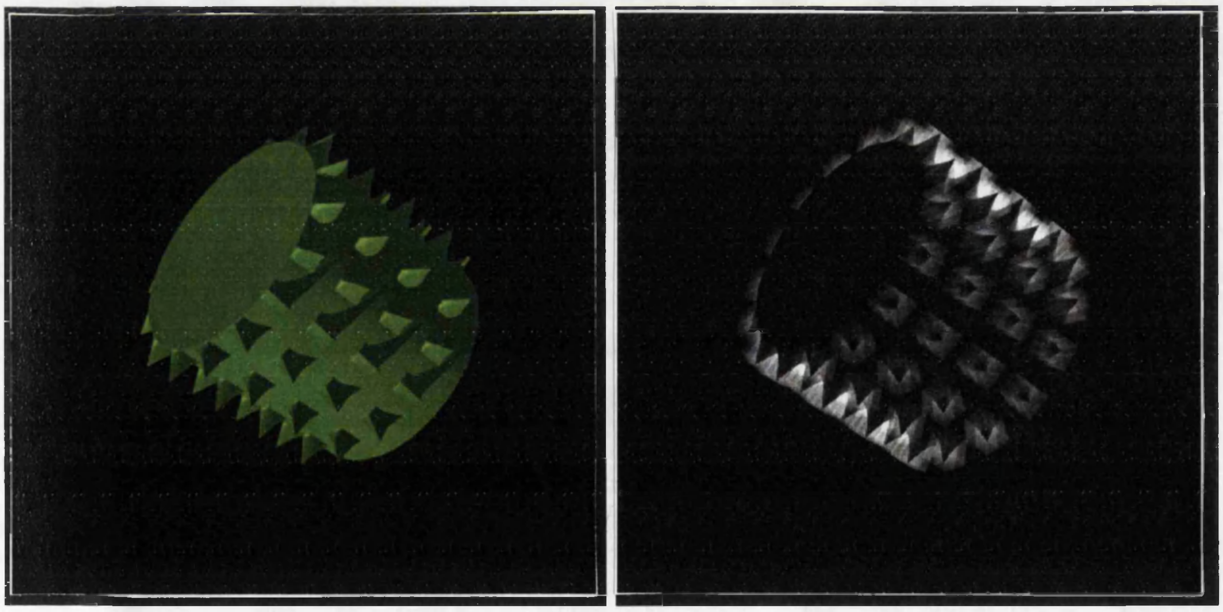


Figure 2.12: Inverse displacement-mapped cylinder

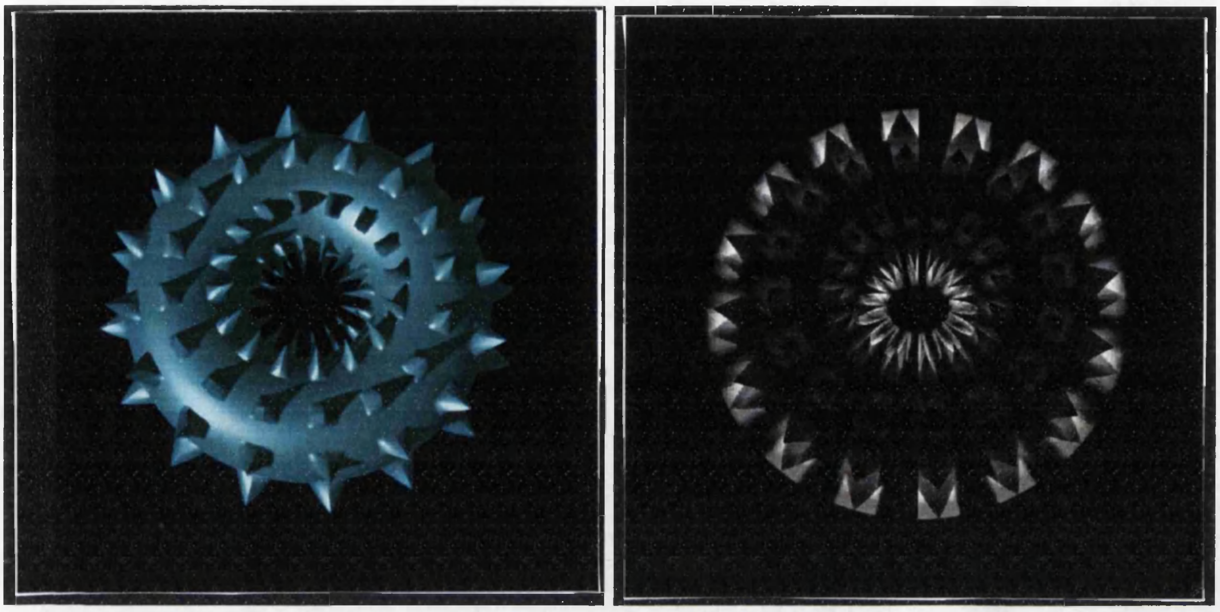


Figure 2.13: Inverse displacement-mapped torus

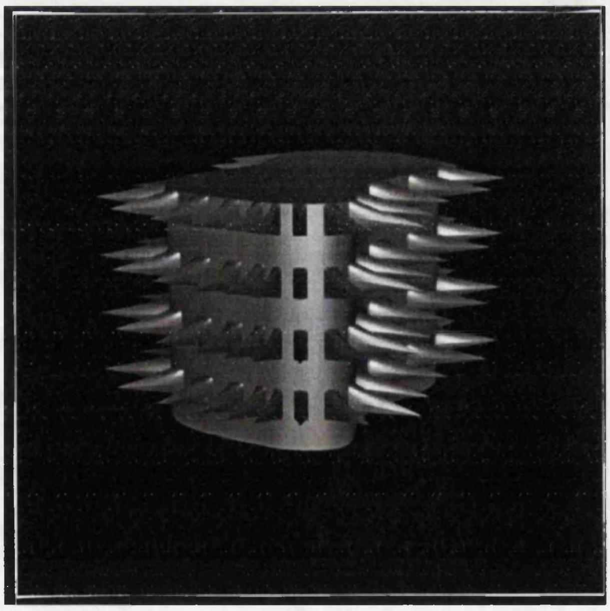


Figure 2.14: Inverse displacement-mapped swept surface

Chapter 3

A Direct Approach

3.1 Introduction

This chapter explores direct solutions to the ray displacement-mapped surface intersection. The algorithm developed is very general and flexible, although at the expense of a considerable computational burden. The algorithm in the previous chapter was constructed by using a large amount of a priori geometric knowledge of the intersection calculation. This allowed an efficient algorithm to be built but also limited its applicability to cases where the geometry of the intersection calculation could be easily and quickly analysed. The approach taken in this chapter is to assume as little information as possible about the surface. For the direct techniques presented here, the surface (and its derivatives) need only be calculated at individual points and bound over an area of the parametric domain. This can be done easily, although not necessarily efficiently, for almost all surface primitives used in computer graphics today.

The intersection of the ray and the displacement-mapped surface is solved by a two part algorithm. The first part proceeds by subdividing the surface until simple enough surfaces are reached so that, in the second part, existing numerical algorithms can be used to calculate the solution. The solution itself is calculated by formulating the problem as a set of non-linear equations. This system is extremely general; it can handle any continuously differentiable parametric surface, allowing the calculation of the solution of the intersection equations. Many numerical techniques exist to solve systems of non-linear equations. These, in general, use Newtonian iteration to calculate solutions with some form of preprocessing to guide the algorithm to likely solutions. Techniques of this form have been used by a number of researchers in the field of ray-tracing[Tot85, SB86, JB86, Bar87, Yan87, LG90, SNK90]. The algorithms used vary greatly in their generality and robustness. The direct numerical rendering of displacement-mapped surfaces has not been addressed but many of the algorithms can be extended to handle it.

This chapter starts with the formulation of the problem and an overview of how it can be solved. This is followed by a discussion of existing techniques, their performance, generality, robustness and whether they can be used to handle displacement-mapped surfaces. This is followed by an algorithm to allow direct rendering of displacement-mapped surfaces. Finally, a discussion of the scope and performance of this approach is given.

3.2 Formulating the problem

This section gives the general formulation of the ray surface intersection. Let the surface under consideration be $f(u,v) = (f_x(u,v), f_y(u,v), f_z(u,v))$. This may be any parametrically defined surface, although here consideration is specifically intended for displacement-mapped surfaces. Further, if the ray is $r(\alpha) = \underline{a} + \alpha\underline{b} = (a_x, a_y, a_z) + \alpha(b_x, b_y, b_z)$, then the ray intersects the surface when

$$f(u,v) = r(\alpha) \tag{3.1}$$

Hence,

$$\begin{aligned} f_x(u,v) - a_x - \alpha b_x &= 0 \\ f_y(u,v) - a_y - \alpha b_y &= 0 \\ f_z(u,v) - a_z - \alpha b_z &= 0 \end{aligned} \tag{3.2}$$

This defines a system of 3 non-linear equations in 3 variables. The problem can be recast by algebraically eliminating the variable α, as

$$\begin{aligned} b_i(f_j - a_j) - b_j(f_i - a_i) &= 0 \\ b_i(f_k - a_k) - b_k(f_i - a_i) &= 0 \end{aligned}$$

The subscripts $i, j, k \in \{x, y, z\}$ are chosen by

	$\lvert b_x\rvert > \lvert b_y\rvert, \lvert b_z\rvert$	$\lvert b_y\rvert > \lvert b_x\rvert, \lvert b_z\rvert$	$\lvert b_x\rvert > \lvert b_x\rvert, \lvert b_y\rvert$
i	x	y	z
j	y	z	x
k	z	x	y

This ensures that no division by zero occurs when recasting the problem. This formulation will return only the (u, v) values for the intersection points. The ray parameter, α, can easily be calculated from any of the equations in 3.2 once u and v are known.

The standard way to solve the system of non-linear equations in 3.2 or 3.3 is to use Newtonian iteration. Newtonian iteration works by taking an initial approximation to the solution and then iteratively refining it to the desired accuracy. If the system of equations is

$$g(\underline{v}) = 0$$

where

$$\underline{v} = (v_1, ..., v_n)$$

$$g(\underline{v}) = (g_1(\underline{v}), ..., g_m(\underline{v}))$$

then, for $\underline{v}_0$ the initial approximation, the Newtonian iteration is given by

$$\underline{v}_{k+1} = \underline{v}_k - J^{-1} g(\underline{v}_k)$$

The matrix J is the Jacobian of g, i.e.

$$J = \begin{bmatrix} \frac{\partial g_1}{\partial v_1} & \cdots & \frac{\partial g_1}{\partial v_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial v_1} & \cdots & \frac{\partial g_m}{\partial v_n} \end{bmatrix}$$

This method works when J is non-singular, and g is a continuously differentiable function. The major difficulty with Newtonian iteration is the choice of initial starting value. If $\underline{v}_0$ is poorly chosen then the system may converge to the wrong (unwanted) solution, converge very slowly, or fail to converge at all. Also, the system above can only find one solution, so if multiple solutions exist a starting value must be found for each one. Safe starting regions for Newtonian iteration can be found by utilising techniques from interval arithmetic[MJ77, Moo77, Moo78]. The core idea is that all calculations are performed using ranges of numbers (i.e. intervals) not single numbers. This approach makes it possible to deduce whether a real Newtonian system has (and will converge to) a unique solution. If the system cannot be solved directly, the region is split into smaller sub-regions and each sub-region is processed recursively. In this way all solutions, over a finite range, can be found. A description of interval arithmetic and an interval Newtonian scheme are given in Appendix B. This approach, although guaranteeing all solutions will be found, has an enormous computational penalty and is orders of magnitude slower than an "unsafe" real Newtonian scheme.

The next section describes existing applications of numerical techniques for intersecting rays and parametric surfaces. In most cases, the differences in the algorithms revolve around different methods of finding initial approximations.

3.3 Existing techniques

A number of authors[Tot85, SB86, JB86, Bar87, Yan87, LG90, SNK90] have addressed the problem of ray tracing complex parametric surfaces using numerical techniques. The majority of the techniques are based on Newtonian iteration although a number use variants of this method, or recast the problem in terms of differential equations. The major differences between the algorithms are in the way in which initial approximations are found. Some methods use an algorithmic approach with an auxiliary data structure, while other methods solve the problem directly using robust numerical techniques. A number of the algorithms are designed solely for ray tracing bicubic surface patches, due to their simple refinement properties. These algorithms are more general than usually presented as techniques from interval arithmetic[Moo66](see Appendix B) can be used to provide refinement properties for general parametric surfaces[MK84, Sny92, Duf92].

One approach to ray tracing parametric surfaces is characterised by the algorithms of Sweeney & Bartels[SB86] and Yang[Yan87]. Both algorithms use Newtonian iteration to calculate the solution with an initial subdivision to find starting values. The initial subdivision decomposes the surface into a large number of small elements which are stored in a tree structure. Sweeney & Bartels store the elements in a hierarchy of bounding volumes whereas Yang uses an octree. When the ray is intersected with the surface it is first tested against the tree structure to find a set of candidate parts of the surface which the ray could hit. Within each of these the ray is intersected with the surface numerically. The assumption is made that, if the surface is subdivided finely enough, the Newtonian iteration will converge to the unique solution within each surface element. This approach has a number of problems. The first is that to have a reasonable chance for convergence, the initial subdivision must be very fine. This generates very large data structures which impose severe memory requirements on the system. This limits the size of scenes which can be handled and removes one of the main advantages of parametric surfaces namely, that complex geometries can be built from compact representations. A more serious problem is that the algorithms are not robust. Both rely on the assumption that the initial subdivision is fine enough for the Newtonian iteration to converge correctly. In both algorithms the subdivision level is set by arbitrary parameters and checked visually. Any system using this approach will fail for certain classes of surface.

A different approach is presented by Toth[Tot85, LG90]. In this case no auxiliary data structures are used, instead techniques from interval mathematics[Moo66] are used to give a robust algorithm. Interval mathematics extends real arithmetic by performing computations over ranges of numbers, that is to say intervals. This allows bounds on all calculations. These bounds can be used to determine whether a real Newtonian scheme will converge to a unique solution, whether or not a solution exists or whether not enough information is present to determine the solution. Processing starts with the initial range of parametric values for the surface and subdivides these repeatedly until a solution is found. This process terminates with all solutions over a given interval. The only place where the algorithm fails is if the machine precision for numbers is reached before a solution is found. In this case some approximation is made to determine the solution. Since machine precision is frequently ignored in other algorithms this poses no problems. This approach has one serious drawback. It is very computationally expensive. The interval computations to determine safe starting regions are orders of magnitude slower then the iteration itself, thus making this approach infeasible in many situations.

The algorithm of Joy & Bhetanahotla[JB86] attempts to reconcile the speed gain from preprocessing the surface and the robustness of interval methods. In this case, the scene is preprocessed into cells with one, two or many possible solutions. By replacing Newtonian iteration with a quasi-Newtonian minimisation scheme the cases of one or two solutions can be dealt with quickly. All other cases are dealt with by a robust method. Since most cases are fairly simple this is claimed to be efficient. Unfortunately, no details of comparative timings, or more importantly how the classification is performed, are given in the paper. Since the crucial efficiency step in this algorithm is the preprocessing of the surface it is impossible to judge the effectiveness of this approach.

These techniques can be applied to displacement-mapped surfaces but the problems inherent in the techniques are amplified by the complexity of the displacement-mapped surfaces. The application to displacement-mapped surfaces is described in the remainder of the chapter.

3.4 Application to displacement mapping

The question now arises "Can these techniques be applied to displacement-mapped surfaces?".

In general the answer is "yes" as the equation of the displacement-mapped surface can be substituted in the equations in 3.3 and the intersection calculated. Unfortunately displacement mapping, due to its complexity amplifies many of the problems inherent in the previous algorithms. Displacement-mapped surfaces are, by their nature, very irregular. This leads frequently to multiple solutions within small areas of the surface and makes non-robust techniques prone to errors. Further, robust techniques depend heavily on the interval surface evaluations making displacement mapping excessively slow. One further complication with displacement-mapped surfaces is that they need not be continuously differentiable everywhere. At the boundaries of texels the texture function may not be differentiable, e.g. if a bilinear interpolation function is used, or even continuous. This implies that Newtonian iteration can only be applied inside individual texels not across the surface as a whole.

The inherent complexity of the displacement-mapped surfaces is their greatest asset (and the reason for studying them), so there is no way to avoid much of the complexity they entail. Although this complexity has a large impact on performance, numerical techniques can still be used to calculate the intersections. The only problem which preludes the use of numerical techniques is the last one. This problem can be overcome with an initial subdivision.

The algorithm works by subdividing the underlying parametric region of the surface until single texel-sized regions are reached. Once such a region has been found the ray is intersected numerically with the surface. To avoid an excessive number of numerical intersections, the ray is intersected with the bounding box of the surface as it is subdivided. In this way, numerical intersections are performed only when required and, as will be seen, the algorithm can be guided towards the closest solution. Caching can also be used to avoid repeatedly re-evaluating bounding boxes thus increasing the speed of the algorithm.

The algorithm makes few assumptions about the geometry of the underlying surface or the displacement function used. Thus it can be applied in a large variety of situations. This and its appealing simplicity lead to a general algorithm.

3.4.1 Subdividing the texture

The algorithm for intersecting rays and displacement-mapped surfaces works by subdividing the underlying parametric regions until regions of one texel in size are reached. This is necessary because only within such regions is the texture function $t(u, v)$ guaranteed to be a continuously differentiable function.

Thus within a single texel the displacement-mapped surface

$$\underline{S}(u, v) = \underline{f}(u, v) + \underline{d}(u, v).t(u, v)$$

is also continuously differentiable and the previous numerical algorithms can be used to calculate the intersection. In some cases it may be possible to stop the subdivision on a larger region of the texture. This can only be done if the texture function $t(u, v)$ is continuously differentiable over a number of texels. The use of this optimisation cannot be discussed in the general case as specific properties of the texture function and the texture map must be used. In the case where $t(u, v)$ is a bilinear interpolation of height values on the texture map (a very useful case) it can be easily calculated when texels can be combined to form larger areas which are continuously differentiable.

Now that the reasons for subdividing the texture have been given an efficient method of subdivision must be found. The obvious place to subdivide the region is at the the mid-point of its longest side. This will ensure that at each stage the regions are halved in size. Unfortunately one problem exists with this method. If the mid-point of the region falls within a texel (i.e. not on a texel boundary) then this texel will be placed into 2 subregions and may be intersected twice using the numerical intersection routine. This problem can be solved by forcing all subdivisions to take place on texel boundaries, thus ensuring each texel is in only one of the subregions. These observations lead to the following subdivision rule,

- subdivide at the texel boundary closest to the mid-point of the longest side of the region

If the mid-point lies exactly in the middle of a texel the boundary on either side may be chosen. It should be noted that to achieve maximum efficiency a texture of size $2^n \times 2^m$ for the initial region is ideal, as in this case the mid-point is the correct place to split the texture.

3.4.2 Guiding the subdivision

It is common in most ray tracing applications to generate only the closest solution to the ray starting point. Thus, the subregions formed should be processed in this order. This ordering can be found by using the ray parameter of the intersection of the ray and the bounding box for the surface. The subregions formed can be stored in a heap data structure ordered by the value of the ray parameter. In this way the closest solutions can be found first and once one solution has been found any regions further away can be discarded. Also, these bounding box tests provide an efficiency step which allows large regions of the texture to be rejected before they are subdivided to individual texels. If the ray misses the bounding box for a region then there can be no solution in that region and it can be rejected. As long as the box does bound the region of the surface the behaviour of the surface inside is irrelevant. There is no need for the surface even to be continuous (let alone continuously differentiable) as the numerical routines will never be called with this region.

```
set up the heap to contain the initial region
while the heap is non-empty do
    remove the top item from the heap
    if this region is of one texel in size then
        intersect the ray with the surface
        if a solution exists and it is the closest then
            store solution
            cull heap
        endif
        break
    else
        split the region into X1, X2 according to the rule mentioned earlier
        calculate the bounding boxes for each region
    endif
    for each region do
        if the ray hits the bounding box then
            add region to heap
        endif
    end for
end while
```

Figure 3.1: Pseudo code for direct displacement mapping

3.4.3 The algorithm

Now that the motivation and details of the algorithm have been discussed, the algorithm itself can be given. It is shown in figure 3.1.

This algorithm can intersect a ray with a displacement-mapped surface by using either a robust or a non-robust method to calculate the actual intersections.

In practise the performance of this algorithm can be very poor. This arises since the calculation of bounding boxes around regions of the texture is a time-consuming calculation and significantly degrades the performance. This can avoided by caching the bounding boxes after they have been calculated and using the pre-stored values for later iterations. Although this increases the memory requirements for the algorithm it is a very necessary optimisation.

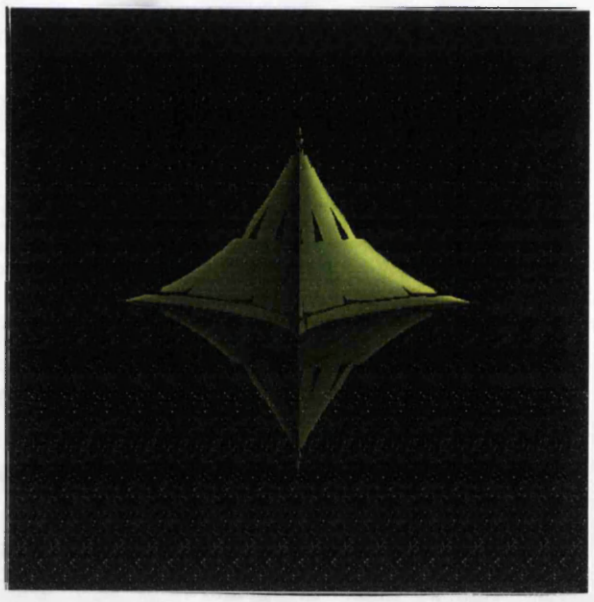


Figure 3.2: Displacement-mapped superellipsoid

3.5 Conclusions

The algorithm presented here to intersect a ray and a displacement-mapped surface has been implemented using both robust and non-robust methods to calculate the final solution. A detailed discussion of the results and performance is delayed until Chapter 5 although the algorithm behaves as expected. The time required to calculate the final solution using robust methods is very large whereas the time to calculate the final solution with non-robust methods is far less but visible artifacts result. These artifacts can be clearly seen in figure 3.2 on a superellipsoid. The splitting of the surface into texel sized regions also take a considerable time and the caching of the bounding boxes is essential for the speed of the algorithm. The image in figure 3.3 shows the results of this method for a teapot, defined by 32 bicubic patches and displacement-mapped with a spike texture.

This chapter has presented an algorithm to ray trace displacement-mapped surfaces in a very general way. Unfortunately the numerical techniques used were very slow. The speed is governed by the inherent complexity of the displacement mapped surface and, to provide a general and practical solution a different approach is needed. This is presented in the next chapter.

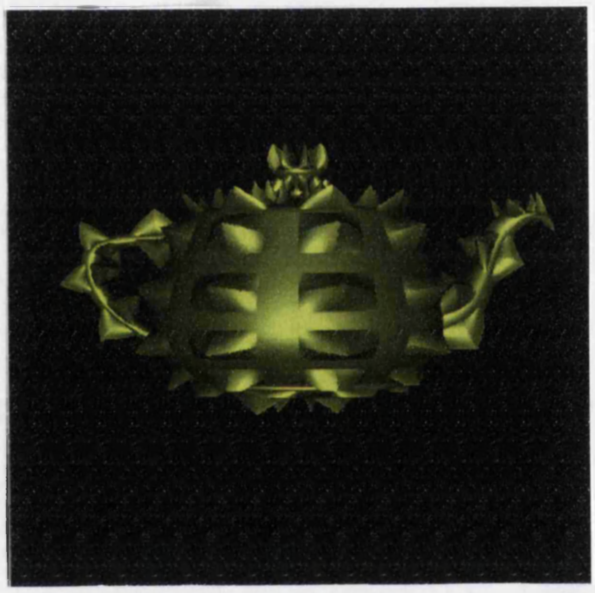


Figure 3.3: Displacement-mapped teapot

Chapter 4

LITUNI

4.1 Introduction

This chapter presents LITUNI, a powerful, general and practical system for ray tracing displacement-mapped surfaces. LITUNI is an acronym for "Leave It Till U Need It" – defining the philosophy behind the system. Any computation performed is done only when needed and any auxiliary results and structures are stored only while needed.

The system is designed to allow displacement-mapped surfaces of any base type to be ray-traced quickly. The solutions to the ray tracing of displacement-mapped surfaces given in Chapters 2 and 3 considered only the ray-object intersection calculations. This allowed the solutions to be used with existing ray tracing architectures but neither solution is practical and general. LITUNI is designed from one step back and considers the complete process of tracing a ray through the scene. By considering the whole process, the weaknesses of existing ray tracing architectures, particularly when handling displacement-mapped surfaces, can be identified and corrected. The LITUNI architecture, although designed for ray tracing displacement-mapped surfaces, is very flexible and general. It can handle scenes and object types which other systems cannot, without restricting itself to the one special case of displacement mapping.

Ray tracing is infamous amongst rendering techniques for its large computational expense and massive memory requirements. These two aspects are closely linked with a trade-off of one against the other. LITUNI tackles both of these problems. The first of these issues, speed, is dependent on the time for the intersection calculation. This computation is performed in the inner loop of the ray tracer and can account for up to 95% of the time. For displacement-mapped surfaces, as was seen in Chapters 2 and 3, the intersection calculation is complex and time consuming. LITUNI replaces the expensive ray-displacement-mapped surface intersection with a series of simple ray-polygon intersections. This is achieved by polygonalising the surface before intersections with the ray are calculated. Since a ray can be intersected with a polygon very quickly, this dramatically speeds up the system. Polygonalisation is in many ways a "RISC" approach to ray tracing, where one complex intersection is replaced with a number of simpler intersections. This approach has been tried in various guises before but has proved only partially successful. The problems which arise stem from the vast numbers of polygons needed to accurately represent a complex object. The handling of very large numbers of objects generate very large internal data structures to store the objects and process them efficiently. The critical area of LITUNI which solves the space problem is caching. Almost all data in the system is dynamic. It is generated on demand, stored only while needed and deleted thereafter. This leads to a space and time efficient system. The cache size can be kept manageable by exploiting coherence and if caching fails to reduce the memory usage its performance degenerates to a level more typical of conventional ray tracers.

Although LITUNI was designed as a tool for ray tracing displacement-mapped surfaces, its design incorporates much more. LITUNI provides a new basis for ray tracing all objects. It is flexible, general and extensible. The system is not specialised to displacement-mapped surfaces or any other surface, unlike many ray tracing techniques for complex surfaces. All ray tracing architectures can be divided into two parts – the optical effects which provide the lighting, shading, shadows, etc. and the geometric routines to efficiently intersect a ray with the objects in the scene. LITUNI retains the ability to handle all the optical techniques of existing systems but provides a reimplementation of the intersection calculation. The interface between these parts is well defined and LITUNI can be transparently integrated with existing algorithms for optical effects.

The rest of this chapter presents the design, motivation and structural details of LITUNI. First the design criteria are given. This is followed by a review of existing ray tracing systems and the design principles for LITUNI. This lead on to the internals of LITUNI itself, the handling of objects, polygonalisation and a caching hierarchy. Finally the performance and results of LITUNI are discussed.

4.2 Design criteria

This section describes the design criteria for LITUNI. The main criteria is to provide a system capable of rendering displacement-mapped surfaces, but for LITUNI to be truly useful, it must do considerably more. The criteria listed here are, in many ways, the goals of any ray tracer and LITUNI addresses only some of them. The issues addressed by LITUNI are those connected with practically and speed. The other issues can already be solved by existing techniques and incorporated into LITUNI.

The LITUNI system is designed to handle the following aspects of ray tracing:

Object Complexity LITUNI must be able to handle a wide range of object types from simple polygons to the most complex displacement-mapped surfaces. Many current ray tracers handle only simple objects such as polygons, quadrics, swept surfaces and surfaces of revolution. This is because a ray can be intersected quickly and easily with these surfaces. Other surface types, such as spline patches, generalised cylinders and fractals, are rarely found in ray tracers. For these, the intersection routines are difficult to write robustly and run very slowly. A truly useful ray tracer must be able to handle all surface types quickly and in a manner which is relatively independent of the surface complexity. It is only by ensuring this that displacement-mapped surfaces can ever be ray-traced practically.

Scene Complexity LITUNI must be able to handle very large scenes. It is not possible to define how big the scenes will be but the system must expect tens of thousands, if not hundreds of thousands of objects. One of the aims of rendering is to be able to generate images to photo-realistic quality and this requires very complex scenes. It should be noted that there is a dependence between the number of objects in a scene and the complexity of the objects. The effect of one displacement-mapped surface may need thousands of polygons to represent it. Thus, the number of objects needed (and hence the number which LITUNI must handle) depends on the type of the objects used.

Practicality LITUNI must be able to run in a reasonable amount of time without consuming excessive computing resources. Amongst the techniques in computer graphics, ray tracing (though famous for its quality) is infamous for its computational expense and massive memory usage. This is made worse by the trade-off between space and time when neither can be spared. LITUNI, using techniques which contradict existing thinking on how a ray tracer should work, removes the trade-off breaking the tie between space and time usage providing a fast and memory efficient system. This, in all but the worst cases, provides a great increase in performance.

Flexibility LITUNI must be able to provide a flexible basis for ray tracing. It must introduce no restrictions on object type or structure. This will allow existing intersection algorithms to be used where appropriate and new algorithms in other cases.

Image Quality The renderer must be able to generate faithfully the image of the scene. There must be no facets introduced to smooth surfaces nor "jaggies" along object boundaries. Also, there should be no aliasing effects from either incorrect sampling of textures or other parameters (e.g. time and strobe effects).

Shading Quality The renderer must be able to generate the shading and lighting for the image accurately. This is accomplished to a large degree by the choice of ray-tracing as the rendering model. Also, to model the properties of different surface types realistically, some type of programmable surface description/shading language[Coo84, HL90] is needed.

LITUNI is designed to handle the issues of object and scene complexity, as well as practicality and flexibility. Although it does not address image or shading quality directly, LITUNI is designed to allow the many existing techniques in these areas to be integrated seamlessly. This will allow the system to run efficiently without comprising on image quality.

4.3 Architectures of existing ray tracing systems

Before the design of LITUNI is described, existing ray tracing architectures (internal designs) will be reviewed. This will study their performance, basic principles, where they spend their time and any assumptions they depend on. This review will consider the principles behind the techniques, not the actual details. The result will be an assessment of the strengths and weaknesses of existing ray tracers.

The internal structure of all current ray tracers is as shown below.

```
Preprocessing
        Read in scene
        Build the hierarchy
Main ray tracing loop
        For each ray
        {
            Traverse the hierarchy to find objects on the path of the ray
Calculate the actual intersection
            For each object
                Intersect the ray and the object to calculate the closest intersection
Calculate the pixel colour
            Calculate the illumination at the intersection point
        }
```

Within the different techniques some of the stage boundaries may be blurred but all exist in some form.

Underlying the structure above are two major components – the hierarchy and the intersection routines. The hierarchy is designed to reduce the number of ray-object intersections. It does this by storing the whole scene in a data structure which allows those objects on the path of the ray to be found quickly and the rest trivially rejected. This data structure allows the ray to be intersected only with objects it is likely to hit. In doing so dramatically reducing the number of ray-object intersections and making the most recently designed hierarchies almost insensitive to the number of objects in the scene. The intersection routine calculates the ray-object intersection. There must exist an intersection routine for each object type and the speed of the intersection routine is highly dependent on the type.

The structure above shows a number of major points about current ray tracers. Firstly the hierarchy is built statically. This means the complete scene must be stored in memory at all times[1], and this imposes a ceiling on how large a scene may be ray-traced. Further,

[1] This is true even for ray classification(see later).

the hierarchy itself occupies memory, with the amount of memory used dependent on the number of objects in the scene. It is also clear that the inner loop is the intersection calculation. No matter how efficient the hierarchy is at reducing the number of intersection calculations performed, if the intersection routine is slow then the whole system will be slow. One way to avoid this is to polygonalise the scene before ray tracing but this amplifies the memory problems with the hierarchy.

This may seem to present an impossible situation. But, by rethinking the way objects are handled, the opposing constraints can be balanced. The design principles in the next section will show how these problems can be overcome.

4.4 Design principles

The design criteria for LITUNI and a survey of existing techniques lead to a set of features necessary for a successful implementation. The features necessary may seem contradictory but, as shall be shown, they can be reconciled.

Store only the data which is actually needed This principle is the way to avoid excessive memory demands on the system. The amount of data the system *must* store at any one time is the data needed to process only the current ray. In conventional ray tracing systems this is regarded as the complete scene but significantly less is really needed. Most objects in a scene are built hierarchically from a number of sub-objects e.g. a teapot is the list of the polygons/spline patches which describe it. To trace a ray through the scene only the bounding box for the object is needed (as well as a pointer to the file containing the data) for the majority of rays. If the object is read in only when a ray hits the bounding box for the object then the total amount of data needed for an individual ray will be far smaller that the size of the complete scene. The processing of objects in this way will alleviate many of the memory problems associated with ray tracing.

Generate data only on demand This principle, which is a form of lazy evaluation, gives efficient usage of memory and a fast system. If data is only created when needed then no space is wasted storing redundant data. In a conventional ray tracing system some parts of the system generate much data which is never accessed. The most common examples are polygons, generated via polygonalisation, which are never seen as they lie on the far side of the object and the parts of the hierarchy which must be built to accommodate them. This principle also provides an efficient system as the computation performed is used only for "useful" work. The LITUNI system derives its name from this principle as LITUNI is "Leave It Till yoU Need It".

Avoid recomputations (if possible) For a system to run efficiently, no unnecessary computations should be performed. In particular the same value should not be recomputed many times. To store all values computed in case they are needed in the future would use phenomenal amounts of memory and is impractical. Instead, a cache can be used to store all recently created data, and when the cache is full the data deemed not to be of further use should be deleted. Caching has been used to great effect in some areas of ray tracing before. In these cases one specific type of data is cached for future use. Within the LITUNI system all data[2] is cached including object descriptions and hierarchy related structures.

Use polygonalisation for complex surfaces This principle is the only way in which complex surface types can be handled in an efficient way. The alternative, to calculate the ray-object intersections numerically, is slow and difficult to implement robustly. The memory problems which have previously dogged polygonalisation, can be overcome by applying the first and second principles. Also, a system which is expected to handle dynamically created polygonal descriptions efficiently allows a great flexibility in the handling of different object types. For many object types it is comparatively easy to describe them as polygons but is far harder to write an intersection routine. This is clearly shown by the case of displacement-mapped objects.

Use simple routines for simple surfaces The system must not force any unnecessary constraints on the way objects are handled. Thus, it should be easy for simple objects such as spheres to use analytic intersection algorithms and not force them to be polygonalised.

[2]A small amount of data must always remain in the system and this must never be removed.

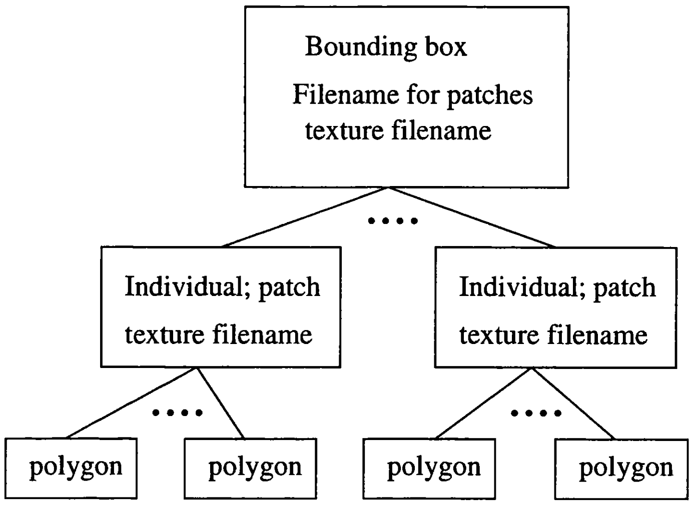


Figure 4.1: Levels of expansion

The principles above give the basis for the LITUNI system. Before an actual implementation can be given the way objects are handled must be discussed. Also, the details of dynamically storing the scene and running a cache will be given.

4.5 What LITUNI looks like

This section gives an overview of the way the LITUNI system behaves in operation. It is very different from existing ray tracers as it has a highly dynamic structure. The operation of LITUNI is data driven, the data being created, expanded and deleted as required by the ray tracing process. This is exemplified by the way objects are handled. At all times the minimum amount of data is stored to describe an object accurately. Initially, all that is needed is a bounding box for an object and a handle to a more complete object description. If no ray in the scene hits an object's bounding box then no rays hits the object. Thus the object, no matter how complex, is stored in a very compact form. Only when a ray hits the object's bounding box is more information required. If this happens the object is expanded to a more complete representation. This process continues for each object until objects which can be directly intersected with rays are reached. These need no further expansion. This can be seen by considering the various representations for a displacement-mapped teapot made from bicubic patches. The levels are shown in figure 4.1. At each level of expansion the amount of data increases and the object can be ray-traced more quickly. The object data must be stored in the hierarchy and the hierarchy expands to handle all the data it must store. The crucial area of LITUNI is that objects, and thus the hierarchy, also contract and data is removed when it is no longer needed. This is achieved by placing all data in a cache. As data is created the cache fills. When it is full the data which is no longer needed is removed and replaced with useful data. In this way the memory requirements for the system can be kept manageable.

4.6 Handling objects in LITUNI

This section presents the methods used to handle objects within LITUNI. To avoid storing the complete scene (i.e. all objects) in memory LITUNI uses a hierarchical method of specifying objects. The idea is to store as little information about the object in memory as is needed to ray-trace it efficiently. If a ray misses an object then all that is needed is the bounding box for the object. On the other hand if the ray hits the object then a more complete description is needed. This scheme allows an efficient usage of memory for handling objects. Also this method, which is really a conversion of representations, allows polygonalisation to be used to handle complex objects. This is achieved by expanding the object to a list of polygons at runtime when needed. The use of a hierarchical representation for the scene is not a major drawback as it is the method generally used in the modeller.

Within LITUNI objects are split into two distinct classes:

- **Primitives** These are objects which can be directly intersected with a ray.
- **Expandable objects** These are objects which are intersected with a ray indirectly by changing (expanding) their form until they are expressed as primitives.

The primitive objects will usually be polygons and quadrics, and the expandable objects will generally be bicubic patches and displacement-mapped surfaces. The two classes will be distinct within an implementation but need not be so across different implementations. This arises since the same object type may be directly intersected with a ray (e.g. using the algorithms in Chapter 2 or 3 for displacement-mapped surfaces) or may be expanded to polygons and then intersected with the ray (e.g. polygonalising displacement-mapped surfaces).

Primitive objects are easily handled within the system. They are specified by three routines **Bound**, **Intersect** and **Normal**. The function of these routines is the same as in a standard ray-tracer. In general, for an object type to be implemented as a primitive, it must have a very efficient intersection routine. The class of objects which are most efficiently handled as primitives will certainly include polygons and quadrics but may include some others. There are a number of object types which have "fairly" fast intersection routines and the best implementations for these must be found empirically. Examples of objects in this class are swept surfaces and surfaces of revolution.

Expandable objects are the critical feature of the LITUNI system. An expandable object is specified by giving three routines

- **Bound** Calculate a bounding box for the complete object.
- **Expand** Expand the object and return the list of sub-objects which the original object expands to.
- **Remove** Remove all parts of the object expansion.

The first of these routines **Bound** is necessary to allow the object to be inserted into the hierarchy. It should be possible to calculate a bounding box for the object without actually expanding the object. If necessary a bounding box for the object should be stored as part of the object description to avoid its calculation at run-time. The second routine **Expand** is the one which does all of the work. This routine alters the description of the object from its original form to a new form which is closer to one which can be ray-traced. The expansion process need not produce a list of primitives directly but must produce a list which, if all of its member were fully expanded, would produce a list of primitives. At each stage of the expansion the object will be described in a larger and more easily handled form. The expansion process itself does not produce the intersection of the ray and the object, instead it produces a description of the object which can later be used to calculate the intersection. The final routine **Remove** is the "unexpand" routine. This removes all the data allocated during the expansion and resets the object state to be unexpanded. It will be called when the object is removed from the cache.

The purpose of the **Expand** routine can also be seen by considering an example. Suppose that bicubic B-spline patch surfaces are to be handled within LITUNI (which they are). They can be specified by two levels of expansion and hence three levels of data. The first level specifies the minimum amount of data needed to describe the object. This is a bounding box for the object and a filename which provides a link to the rest of the surface data. When this object is expanded the filename is used to read in the file of control points. A new object is then created for each patch. Each of these objects contains a bounding box for the patch (easily calculated from the control points) and a pointer to the original object which specifies the control points for this patch. If one of the bicubic patch objects is expanded then it will produce a list of polygons which approximate the surface. These polygons are primitives, so no more expansions will occur.

4.6.1 Polygonalisation

Within LITUNI, polygonalisation is used as the primary expansion method for complex parametric surfaces. The expansion scheme allows polygonalisation to be used without incurring excessive memory usage and makes it practical, unlike most other ray tracing systems. A number of authors have addressed the problem of polygonalising complex parametric surfaces. There are two main approaches to the problem. The first is to use uniform subdivision. This method is simple and fast but generates very large numbers of polygons. The second approach is to use adaptive subdivision; this method is slower but can guarantee accuracy and uses polygons only where needed. These different approaches will now be discussed in greater detail.

Uniform subdivision was the earliest method of polygonalising surfaces. It is straightforward to implement and it runs efficiently. The underlying parametric domain is split into a regular grid and the surface points are evaluated. The points on this grid are then used to create triangles for rendering (see figure 4.2). Since the underlying surface is only used to evaluate the points, it is difficult to guarantee the accuracy of the polygonalisation. The usual solution is to evaluate the points on a very fine grid. This improves the accuracy[3] but generates very large numbers of polygons, most of which are very small. This introduces large numbers of unnecessary triangles which occupy memory and slow down the system. These problems can be solved by using adaptive subdivision.

Adaptive subdivision uses a different approach to polygonalising surfaces. The surface

[3]No point sampling scheme can ever guarantee accuracy as too little information is available.

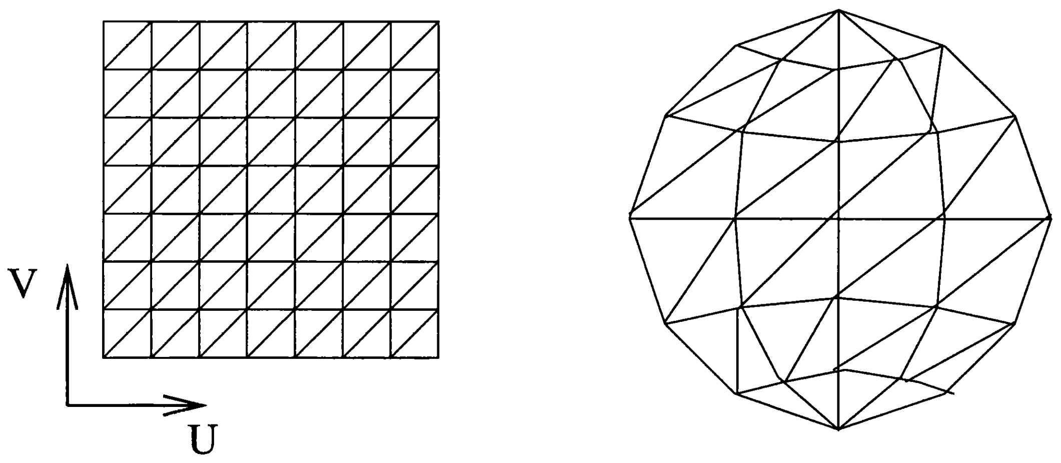


Figure 4.2: Uniform subdivision

is sampled to create polygons and the resulting polygons are tested against a number of criteria to ensure the approximation is an accurate one. This problem has been tackled by a number of authors[HB87, Vla90, FK90]. Barr and Von Herzen[HB87] create a polygon and if it fails their stopping criteria then the region of the surface is split into smaller regions and each is processed recursively. The opposite approach is taken by Vlassopoulos[Vla90]. His method starts by producing a very fine uniform subdivision of the surface and the polygons produced are then merged to form larger polygons which still provide a good approximation to the surface. The criteria used to test the quality of the approximation in both cases are based, in general, around the flatness of the surface. This can be tested by considering the difference in the normals and tangents to the surface at the corners of the polygons. If these lie within a given tolerance then the surface is taken to be flat and is approximated by a single polygon. The major problem with adaptive subdivision is that cracks can appear between the polygons of the surface as shown in figure 4.3. These cracks arise from the joining of different-sized polygons since common edges can be approximated in different ways. If the subdivision is stored in a quad-tree then cracks are easily formed. These cracks can be removed in a number of ways. Extra polygons may be added to the surface to fill in the cracks or the edges may be forced to coincide[TJ85, FK90]. This is the approach taken by Barr and Von Herzen. They store the polygonalisation in a restricted quad-tree[4] which guarantees that common edges always agree.

Although these methods may be applied to displacement-mapped surfaces, one compli-

[4]A restricted quad-tree is a quad-tree in which neighbouring nodes differ in depth by at most one.

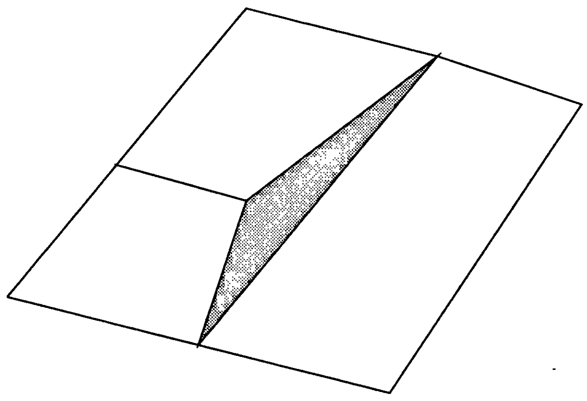


Figure 4.3: Cracks with adaptive subdivision

cation arises. Displacement-mapped surfaces may not be differentiable, or even continuous, at texel boundaries. This means that the tangents and the normal to the surface are not defined. This is shown in figure 4.4. At the corners of texels there are four distinct normals and at other boundaries two distinct normals (One normal for each texel bordering the point). This problem can be addressed by uniformly subdividing the surface to texel level and then polygonalising within the texel. In this case, the surface can be polygonalised easily, by defining which texel is used.

4.7 The hierarchy

Within any ray-tracer it is necessary to store the scene in a hierarchy which allows all objects on the path of the ray to be found efficiently. If an unordered list is used then each ray has linear time complexity on the total number of objects. This can be improved to logarithmic or better time complexity by use of a suitable data structure. There exist three main data structures for this purpose. These are based on spatial subdivision, a hierarchy of bounding boxes or on ray classification.

The first of these, spatial subdivision, divides the space in which the objects lie into cells and tracks the path of the ray through these cells. Every object in the scene is placed in each cell within which it lies. If these cells are all of the same size then it is called uniform spatial subdivision[FTI86]. This approach allows very fast traversal from one cell to another but does have a large memory overhead. It is particularly suited to scenes in which the objects are uniformly spread throughout space. The other form of

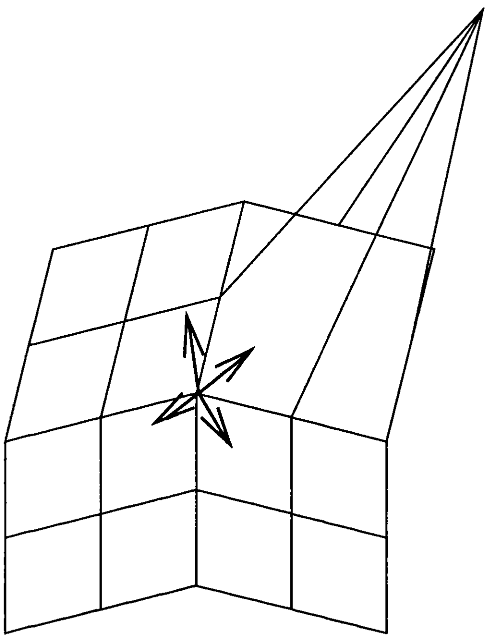


Figure 4.4: Normals to a displacement-mapped surface

spatial subdivision is adaptive spatial subdivision[Gla84, HT92]. This method uses cells of varying sizes depending on how many objects lie in the region of space. If there are a large number of objects then a correspondingly large number of cells are used. This method has the advantage of lower memory requirements than uniform subdivision as cells are only created where they are needed. The main disadvantage of this scheme is that it takes significantly longer to move from cell to cell. With both uniform and adaptive schemes, care must be taken to ensure that the ray intersection is performed only once with each object, even if the object lies in a number of cells on the path of the ray.

The second method is to create a hierarchy of bounding volumes[KK86, SB87]. Each object in the scene is surrounded by a volume which encloses it. These bounding volumes are used to build a tree structure in which each internal node has a bounding volume large enough to contain all of its children and each of the leaf nodes contains an object. When a ray is traced through the scene it is intersected with these bounding volumes. If the ray misses the bounding volume then it must also miss all objects enclosed within that volume, and that part of the tree can therefore be ignored. On the other hand, if the ray hits the bounding volume then it must be intersected with every child of that node. If a leaf node is reached the ray can be intersected with the object. If the list of tree nodes still to be processed is ordered by their distance along the ray then the objects in the scene can be processed in the order in which they occur. The one drawback of this method is that the initial tree of bounding volumes is difficult to build[Gol87] and a poor tree can significantly affect performance.

The third hierarchy type uses ray classification[AK87] to find a small set of objects which the ray is likely to intersect. The most important difference between this method and the previous ones is that it is based on rays, not objects. The idea is to partition the space of rays (a 5D space) and store with each element of the partition a list of objects which could be hit by any ray in that element. Thus the ray need only be intersected with the objects in this list. The correct list can be found very quickly. Since there is a very large number of possible rays, the data structure for the partition is built dynamically (the objects are inserted initially and the details of the data structure are filled in at runtime). This allows ray classification to be used without classifying for every possible ray. Although the classification procedure is time consuming, its benefit is spread over a large number of rays and the average cost per ray is small. Also with a suitable implementation it is possible to sort the list of candidate objects so that they are processed in the correct order along the ray. This method is reported to outperform spatial subdivision and hierarchies of bounding volumes, although at the cost of very large memory requirements.

The question now arises: can any of these hierarchies be used with LITUNI? The hierarchy used must be capable of handling insertions and deletions at any time. Of the existing hierarchies, none were designed to allow dynamic insertions or deletions. Also, since objects are only added to the hierarchy when they are needed in expanded form, the positioning of objects in space will be highly non-uniform. These criteria exclude the hierarchy of bounding volumes as the hierarchy is too difficult to maintain in an efficient form (recall that a good tree was necessary for efficiency and this is very tricky to build). Also uniform spatial subdivision can be excluded as there will be a highly non-uniform distribution of objects. The adaptive spatial subdivision scheme presents a number of problem for deletion algorithms and can also be excluded. This leaves only the ray classification scheme. This method has a number of useful properties which make it suitable for use in LITUNI.

1. The method is based on rays, not objects, thus it is easier to add and delete objects dynamically. This stems from having only to classify objects and add them to lists,

and not having to rebuild parts of the hierarchy.

2. It is fast. This is very important for making a general purpose, useful system.

3. It is built dynamically. This means that the parts of the hierarchy not used are never created and if parts of the hierarchy are deleted it will be possible to rebuild them.

The one disadvantage of ray classification is its large memory requirements but this can be avoided by the use of caching, inherent in LITUNI.

The ray classification scheme will now be described in more detail.

4.8 Ray classification

This section provides the basic details of the ray classification algorithm of Arvo and Kirk[AK87]. More details and some optimisations can be found in the original paper. The main idea behind this algorithm is to subdivide the space of rays into small sets and associate with each of these a list of candidate objects which the rays could intersect. A ray may be defined uniquely by giving five values. The first three of these define the x, y and z co-ordinates of the ray origin in XYZ-space. The other two values define the ray direction. Usually the direction of a ray is defined by a direction vector with three components but only two are needed. This can be seen by considering spherical polar co-ordinates. The two angles in this case define all possible directions. These five values are used to build five dimensional hypercubes. The hypercubes are split into smaller cells as needed until there is only a small set of objects associated with each leaf node. This proceeds in a manner analogous to the building of an octree. If a traversal of the hypercube tree needs to access a part of the structure which has not yet been built then it is built as part of the traversal procedure. In this way only the hypercubes which are needed are constructed, providing a considerable saving in time and space. Initially, the scene is bounded in XYZ-space and all rays are considered to start within this bound. This will be true for all rays except rays from the eye point. For these rays the origin of the ray is moved to lie on the boundary of the scene bounding box. In this way all rays can be made to start within a given volume in XYZ-space.

As was mentioned above any ray can be defined by five values – the origin and two spherical polar angles. Unfortunately if the hypercubes are built using this classification of

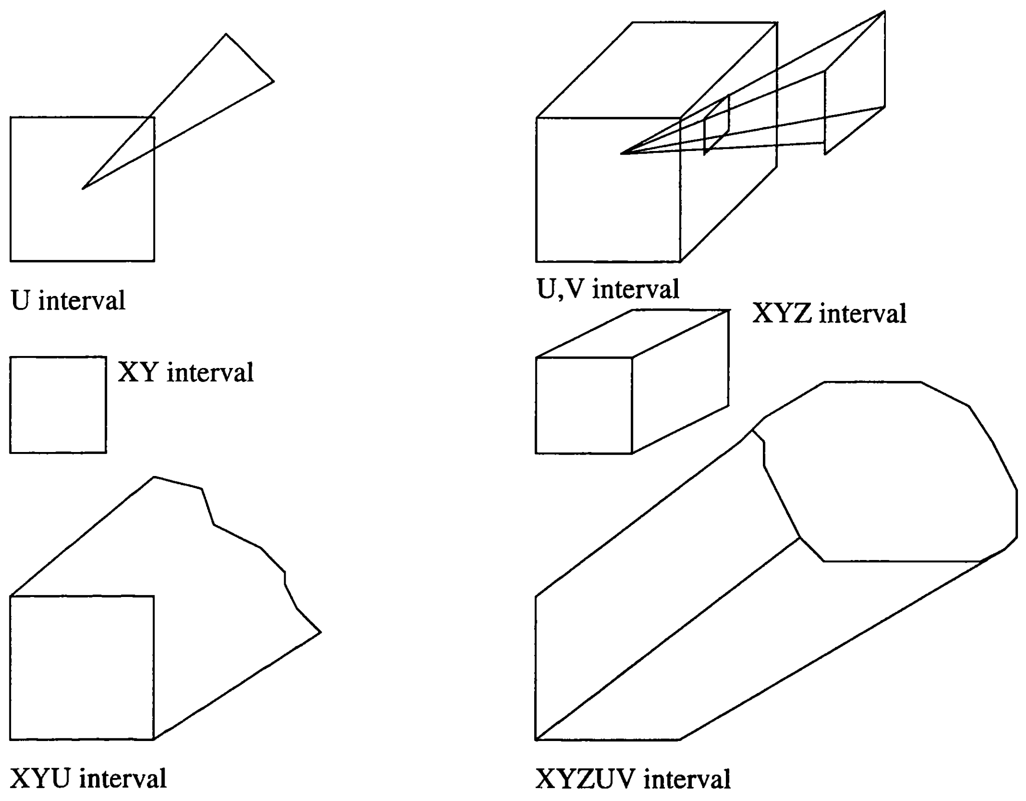


Figure 4.5: Beams in 2 and 3 space

rays the subdivision of hypercubes is highly non-uniform. This arises from the definition of spherical polar co-ordinates. In this coordinate system a similar change in angle has a far larger effect at the equator than it has at either of the poles. This problem can be overcome by defining the ray direction using a direction cube. The ray directions are defined by the points at which the ray intersects a unit cube centred at the origin. This gives a mapping from each ray direction to a 3-tuple defined by the face of the cube the ray hits (which can be calculated from the signed dominant axis of the ray) and the intersection point on the plane of the cube. This mapping gives a uniform subdivision of the 5-D space of the rays. Thus each hypercube defines a beam as shown in figure 4.5.

Initially, the hypercube tree contains one node. This corresponds to the set of ray origins for the complete scene (see above) and all ray directions. This node has a candidate list which corresponds to the complete list of objects in the scene. When a ray is traced, it is tested for intersection with the scene. If it hits the scene bounding box then the 5-D hypercube tree is traversed from the root until a leaf node is found. Since the hypercubes partition the space of rays, the ray will always lie in a distinct leaf node. There exist two types of leaf node in the hypercube tree (the reason for which will become clear later). If the candidate list for a leaf node has been calculated then the ray is intersected with each object in the list to find the actual intersection for the ray and the scene. If the candidate list for the ray has not been calculated then it must be found. This can be done by classifying the node and the candidate list for the parent node (see later for details). Once this list has been found the number of objects in the list is tested against a threshold value to decide if the list is small enough to become a candidate list. If the list of candidate objects is too large then the node is split into its 32 sub-nodes. This is done by subdividing the node at the mid-point of each axis of the parent node. Each of these nodes is marked as not having had its candidate list calculated. In this case the traversal continues down the tree. To avoid infinite recursion down the tree a maximum depth must be specified. Thus this traversal procedure, which as a side effect calculates parts of the hierarchy, will always return a candidate list for a given ray.

Now the traversal and construction procedures have been dealt with, the only remaining detail is the classification. The purpose of this is to find for a given beam (set of rays) the list of objects which intersect the beam. This can be done in a number of ways. If the scene contains only polygons then the list can be calculated exactly by formulating the classification as a linear program and solving with the simplex method. Although this gives exact candidate lists it is very slow. On the other hand if the objects are tested for intersections with the planes defining the beam then the candidate list can be calculated much more quickly. Unfortunately this scheme will classify some objects erroneously as intersecting the beam when they do not do so. Although a few extra objects are placed in the candidate lists, this method provides superior performance. If the scene is built from spherical objects (or objects with spherical bounding volumes) then the objects may be classified efficiently by approximating the beam with a cone and using a sphere-cone intersection routine[Ama84].

The basic algorithm above can be optimised in a number of ways. Firstly the objects of the candidate lists for each of the 6 root hypercubes can be sorted by their value along the dominant axis. If this is done then, once the ray is intersected with the candidate list, the objects will be found in their order along the ray. This sorting need only be done once as the new candidate lists created can retain the original order. Also since the set of directions for a ray is known, any back-facing parts of opaque surfaces can be culled from the candidate list. This is very useful as on average about half the polygons in a scene will be back facing. Finally, although traversal of the hierarchy is very efficient, it can be speeded up by caching the leaf node from the previous ray. If the next ray lies in that node then the hierarchy need not be traversed at all.

4.9 Caching in LITUNI

Now that the hierarchy to be used in LITUNI and the concepts of expandable objects have been presented, the remaining problem is how to combine them to form a practical system. The essence of this problem is to decide on strategies for the insertion and deletion of objects and the building of the related parts of the hierarchy. The strategies developed will be based on the underlying principles of LITUNI given in section 4.4. In particular, LITUNI should store only the data needed to run the system, generate data only when needed and avoid recomputations. These principles do not immediately provide a solution as there are many conflicts between storing only what is needed and avoiding recomputations. LITUNI provides a flexible basis for ray tracing and thus makes no assumptions about what type or number of rays will be generated. This means that, in theory, it is impossible to decide exactly when an object is needed in expanded form or when it can subsequently be deleted. In practice, it is well known that the rays generated are coherent and this has been exploited in many areas of ray tracing[WHG84, JB86, Han86, GD89, GP90]. Thus, similar rays are likely to hit a similar set of objects. This property gives a locality of access to the objects in scene and hence to the hierarchy and to the object expansions. By ensuring that the rays are processed in a coherent manner, the parts of the hierarchy they access, and the object expansions used, will also be similar. Further very different rays will access very different parts of the hierarchy and very different object expansions. This means that it should be possible to construct a strategy for insertions and deletions which ensures that at any time only a small number of expanded objects and a small part of the hierarchy is needed. This can be done without the need to frequently recompute expansions and hierarchy structures. The following sections discuss the general issues concerning insertions and deletions to the hierarchy and the expansion and removal of objects. This will culminate with a description of the expansion depth strategy currently used in LITUNI.

4.9.1 Dynamic insertions and object expansion

This section discusses the issues related to strategies for triggering object expansion and where in the hierarchy the new objects should be inserted. The strategy divides into two parts:

- When to trigger an object expansion.
- How to add the new objects to the hierarchy.

The first of these is the easiest to handle. An object expansion should take place only when necessary. This avoids any time or space overheads. The only time when the expansion of an object is needed is when a ray hits the bounding box for the object. At this point it is necessary to intersect the ray with the sub-objects in the expansion to calculate the actual intersection point.

The second part of the insertion strategy is more complex. The list of sub-objects generated by the object expansion must be inserted into the current node. But should they also be inserted elsewhere?

A naive answer to this question would be to say "no" – to insert the sub-objects elsewhere would generate data that no other part of the system needs. This would invalidate the principles of storing only the data needed and generating data only on demand. On closer inspection though, a problem arises. This is related to the actual insertion of objects in the hierarchy. To perform this task four operations must be performed:

- Identify where to insert the objects.
- Classify the sub-objects against the beam of the current node.
- Insert the classified objects into the current node maintaining sorted order.
- Split the current node if its candidate list is too large.

Of these tasks the classification is the most time consuming as the complete list of objects in the expansion must be considered. To understand why this is a problem, consider what would happen when a (precomputed) list of objects is used with the original ray classification hierarchy. Firstly, all the objects are inserted into the root of the hypercube tree. As the hypercubes are split a new classification takes place, but this new classification takes place using a subset of the objects in the scene (except at the first level), i.e.

just those in the parent node. This means that the vast majority of the classifications are performed using small lists (small compared to the total number of objects in the scene). Thus to insert the sub-objects dynamically only at the current node will involve classification against a far larger list than otherwise would have been the case.

The solution to this problem is to insert the sub-objects into a larger number of nodes. This can produce a faster average time per insertion since the complete list of sub-objects need only be classified against the node at the root of the insertion. For all descendants only the objects which intersect the parent beam need to be considered. There are many possible choices as to where the insertion can be performed:

1. The current node and parent only.
2. All ancestor nodes to certain depth.
3. All ancestor nodes to root.
4. All descendant nodes from a given ancestor.
5. All nodes.

Each of these strategies involves a different trade-off between the amount of work necessary, the memory used and the possible usefulness of the extra insertions.

One other point about the choice of insertion strategy must be made. The insertion strategy defines where in the hierarchy the sub-objects are found. This has many implications for the deletion of objects, since it is necessary to know where the elements of an object expansion are so that the deletion of an object expansion does not produce an invalid hierarchy.

The choice of insertion strategy for LITUNI is given in section 4.9.3. It is one which gives a fast classification, makes insertions into few unnecessary nodes and makes deletion easy.

4.9.2 Dynamic deletions and removing object expansions

This section discusses strategies for deleting object expansions and removing parts of the hierarchy. The first issue is when should deletions, of either type, take place? The answer to this question stems from the philosophy of LITUNI that only useful work should be done. Thus, deletions should take place only when necessary. It may seem that this is never, or at least not until ray tracing has finished. The current generation of workstations have large amounts of main memory with 32+ megabytes not uncommon, and operating systems which can handle hundreds of Megabytes of virtual memory. Unfortunately, as any programmer knows, no matter how much real and virtual memory is available it is never enough. In particular, ray tracing the polygonalisations of displacement-mapped objects typically involves tens if not hundreds of thousands of objects. These must be stored and a hierarchy built to accommodate them. This can easily swamp the resources of even the most powerful workstations. Deletions should therefore occur when the available memory for the program is exhausted. Within the current implementation of LITUNI all data is stored in a cache in main memory and the program generates deletion requests when this is full.

Now that the time for object deletions has been found it must be decided what to delete. Any data in the system which has been dynamically created is eligible for deletion as, if needed in the future, it can be recomputed. The data of this type falls into two categories:

- Object expansions
- Hierarchy nodes (except the root)

Between these types it is far easier to delete parts of the hierarchy than it is to delete the objects. This stems from the need to maintain the hierarchy in a consistent state. When an object is deleted any part of the hierarchy which contains references to the expansion will become invalid and must also be deleted (or at least parts of it rebuilt). On the other hand, since objects do not reference the hierarchy, parts of the hierarchy can be deleted without problems.

The choice of which objects in the system to delete is governed by the principle of LITUNI that recomputations should be avoided. Thus nothing which is likely to be used again should be deleted. As was discussed earlier, the rays generated in a typical ray tracer are coherent and an object which was intersected recently is likely to be intersected again. This observation provides the basis for the deletion strategies, for without some idea of the pattern of accesses each object would be equally useful at any time. The time[5]

[5]The usual measure of time, really progress, in ray tracing is the count of the number of rays fired.

an object was last accessed can be stored with it and all objects not accessed recently can be deleted. This same principle can be applied to hierarchy nodes and used as a basis for deleting them. This alone is not enough as it is possible for recently accessed parts of the hierarchy to contain seldom accessed expansions. Thus the deletion strategy must take great care to ensure that the hierarchy is maintained in a consistent state.

The final issue concerning deletions is to ensure that they are as quick as possible. The strategy should avoid having to rebuild any parts of the hierarchy and in particular should avoid performing any reclassifications. Also, if possible, an object deletion should not require a traversal of the complete hierarchy to ensure no dangling references exist. This is not possible if deletions are considered in isolation, but it is possible if the pattern of insertions is known.

4.9.3 Expansion depth insertions and deletions

This section describes a method of efficiently inserting into and deleting objects from the hierarchy. In many way it is a simplistic strategy but it does have many useful properties:

1. Insertions occur in many nodes at once for efficiency.
2. Insertions are localised to "useful" parts of the hierarchy.
3. Objects and hierarchy nodes can be easily chosen for deletion.
4. Deletions require little computation.
5. Hierarchy consistency can be easily maintained.

The critical idea behind this strategy is the expansion depth. This is the depth within the hierarchy at which expansions take place. For all nodes above this depth the objects are maintained in their unexpanded form. This allows all insertions to be localised to parts of the hierarchy which contain few objects. When an object is expanded the sub-objects are added to all nodes above the node at which the expansion began up to the expansion depth. This allows the insertion to affect a large portion of the hierarchy, but does not require reclassification of the complete hierarchy. It should be noted that this insertion

Thus ray 1 starts at time 1 and ray n starts at time n. This measure has the useful properties that it is directly related to the progress of the ray tracer and is independent of the absolute speed of the computer used.

strategy affects more nodes than just those on the path up to the expansion depth. Any leaf nodes off this path which have inherited candidate lists (i.e. not fully classified lists) will also inherit the new expansion.

The deletion strategy chosen for LITUNI was one based on minimum object lifetime. Each object expansion and each hierarchy node is marked with a last hit counter, to record the number of the last ray to intersect them. This counter is used to decide whether an object can be deleted from the cache. The idea is to ensure that only objects which have not been accessed recently are deleted and gives all cached entries a minimum lifespan. In this way, recently accessed objects will be forced to remain in the cache and there is no chance of useful data begin rejected. This strategy presents two practical difficulties. The first is to choose the minimum lifetime for the object. If too small a value is chosen then the system will "thrash" by creating and immediately deleting the object expansions and hierarchy nodes. This will cause the system to slow down dramatically since all of the time will be spent on the expansion process. Alternatively, if the lifetime of objects is made too large, the system still runs fast but the memory requirements rise substantially. This is caused by the cache holding many objects it could delete. In practise, it has been found that the correct level for the minimum lifetime is the level which ensures all data exists for the time taken to ray trace one scanline. The second difficulty with this strategy is maintaining cache coherence. It is possible for a situation to arise where an object has not been accessed recently but a recently accessed part of the hierarchy has references to it. In this case, it is necessary to first delete the objects and then delete the hierarchy nodes which have not been accessed recently and any others which access the deleted objects. These nodes can be easily found as an invalid reference will exist at the expansion depth and all nodes below must be deleted.

The one problem yet to be resolved is where to choose the expansion depth. Results have shown that the best choice is at half the maximum depth, although the performance of the system is not adversely effected by any other "reasonable" value for this parameter.

4.10 Improving the performance of LITUNI

There are a number of improvements which can be made to LITUNI. These are concerned with ensuring that the rays generated are coherent and that object expansions happen in a manner which suits the structure of LITUNI.

The standard order for generating the primary rays in a ray tracer is to use scanline order. With this scheme the rays are coherent along the horizontal axis of the image, but not so along the vertical axis. This arises since there is a jump from the far end of one scanline to the near end of the next one. It is possible to choose an ordering for the generation of rays which is coherent in both axes by using a space filling curve such as a Hilbert curve. With this, each ray is one pixel on the screen from the previous ray in all cases. This scheme can produce a modest improvement in performance but doesn't give a large gain. This is because some rays which are close in the image are widely separated with a Hilbert curve order. Such a situation almost guarantees an object expansion will need to be recomputed.

There are a number of properties of object expansions which improve the performance of LITUNI. The first is that an object expands to many objects which are much smaller than the original object. This ensures that parts of the expansion will be inserted into local areas. If a section of the expansion covers a large volume then it will need to be inserted into many areas. The second useful property for object expansions is that they happen smoothly. This means that the factor of increase in size (the ratio of the number of objects at one level divided by the number of objects at the parent level) is not too large. This can clearly be seen by considering the expansion of a displacement-mapped sphere. One method of doing this would be to expand directly from a sphere (1 object) to a complete polygonalisation (typically thousands of objects). This gives an expansion with a large list of objects to classify and a one which may have to remain in the system for a long time. An alternative is to add an intermediate level of expansion by dividing the displacement-mapped sphere into sub-regions. Each of these is again an expandable object which will produce a polygonalisation. This scheme may, at first sight, seem less efficient as extra work is involved in creating and inserting the intermediate objects. In general though, this scheme will be more efficient since:

- classification will involve smaller lists
- not all parts of the displacement-mapped sphere may be needed, e.g. back-facing regions, and only useful parts will need to be polygonalised
- it generates better cache usage as expansions are more localised

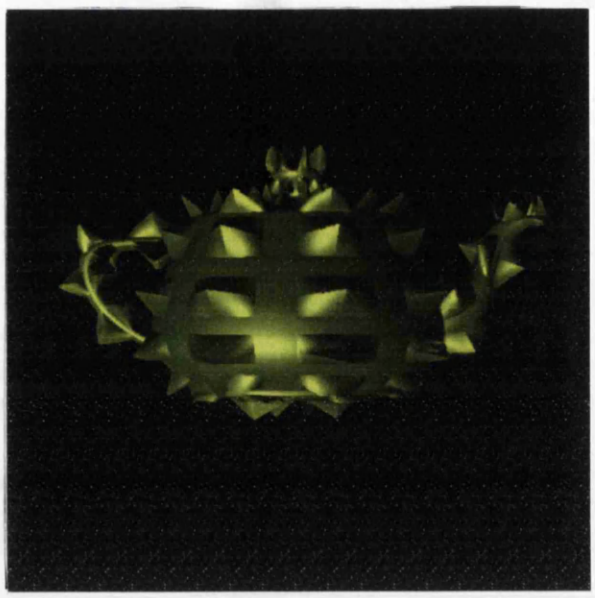


Figure 4.6: Displacement-mapped teapot

- it generates many shorter lived polygonalisations

Within the current implementation this scheme can be used with all expandable objects.

The final issue concerning improvements to the efficiency of LITUNI is cache performance. The size of the cache has a major affect on the performance of the system. A cache size which is too small will cause frequent deletions and subsequent re-expansions of objects and the hierarchy. In the worst case each ray deletes all the information from the previous rays and recomputes all expansions for itself. This gives rise to a situation analogous to thrashing in a virtual memory system. It is not desirable to solve this problem by just giving LITUNI as much memory as possible since this is very wasteful. (It is not wasteful for LITUNI, as LITUNI will run faster in more memory but is wasteful of system resources which other programs may need.) The solution to this problem is to start LITUNI with a small cache and allow LITUNI to increase the cache size if deletions become too frequent or no objects can be deleted. In this way the cache size can be adaptively set for the given scene.

4.11 Implementation and results

The LITUNI system as described here has been implemented in C++ under Unix. It has been used to ray trace a large number of surface types including spheres, tori and bicubic

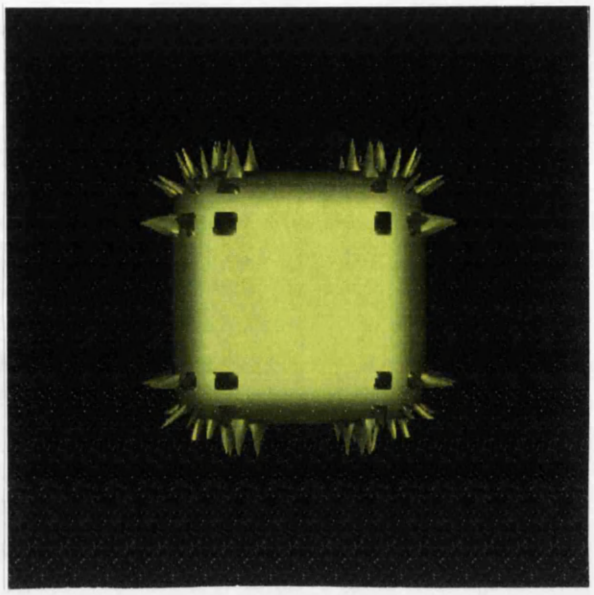


Figure 4.7: Displacement-mapped superellipse

patches. All of these have been implemented for both the base surface and displacement-mapped surfaces. Examples of these are shown in figures 4.6 and 4.7. The results show that for displacement-mapped surfaces the images can be generated at least an order of magnitude quicker than with any other technique, the increase in speed becoming more pronounced as the complexity of the surface increases. The memory requirements, although higher than for inverse or numerical techniques, are reasonable for ray tracing; maximum cache sizes of 4-8M are needed for the system to run efficiently, the cache size being an order of magnitude smaller than would be needed if the scene was completely polygonalised and stored in its entirety. -*-latex-*-

Chapter 5

Results and Conclusions

5.1 Introduction

This chapter presents the results of the three methods discussed earlier. The performance of each of the three algorithms will be given and comparisons between the algorithms drawn. Firstly, the performance of inverse displacement mapping and numerical displacement mapping will be given. This will be followed by a comparison of these two techniques, which will show that, although the two methods appear very different, they share a great similarity. Ultimately their performance is limited by the same factor, namely the complexity of the base surface. Next, the performance of LITUNI is given. This will show that LITUNI provides a fast and practical algorithm for ray tracing displacement-mapped surfaces. The performance of LITUNI and the two direct methods is then compared and LITUNI is shown to outperform the other techniques in all areas. Once this (the main result of the thesis) has been presented a discussion of future extensions, improvements and further research is given.

5.2 Performance of inverse displacement mapping

This section gives a performance analysis of the technique of inverse displacement mapping presented in Chapter 2. The results are presented for four base surfaces namely; cylinders, spheres, tori and cubic swept surfaces. In all cases the results were generated at 512 by 512 resolution on a 90MHz Pentium processor running the NeXTStep operating system.

The first set of results show the performance of inverse displacement mapping for a

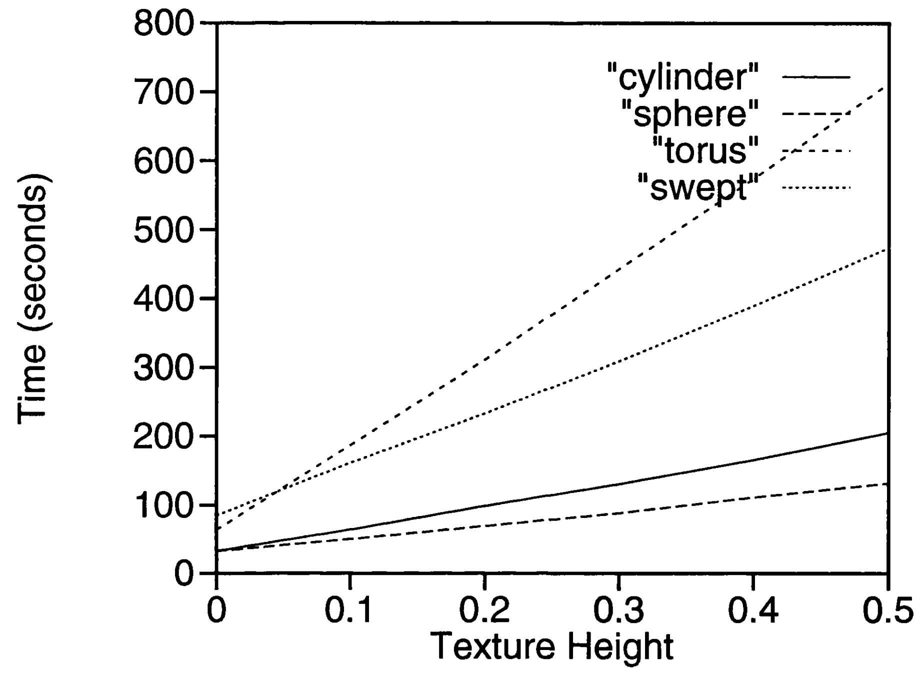


Figure 5.1: Performance of inverse displacement mapping at varying texture heights

texture defining a single spike tiled around the surface, the maximum height of the spike being varied from 0.0 to 0.5 in increments of 0.1. The results are presented in figure 5.1.

From these results it is clear that the performance of inverse displacement mapping varies linearly with the magnitude of the texture. This can be explained by two factors. The first is that the objects are simply bigger, so more rays will hit the displacement-mapped surface as the texture gets bigger. The second, and main, reason is that the algorithm takes longer to recurse to the solution. This arises from the initial bounding volumes' heights being greater and more iterations being needed to converge to the solution. This is especially marked for rays near the silhouette of the object as in this case the initial bounds cover a large area of the texture.

The second set of results for inverse displacement mapping show the performance for a fixed surface and number of tiles but with textures of increasing complexity. In this case a texture with from 0 to 4 spikes on it (the texture size is constant – just the contexts change). The measure of complexity is the number of spikes over a fixed area. The results are presented in figure 5.2.

From these results it is clear that the performance of inverse displacement mapping varies linearly with the complexity of the texture used. This can be explained by an

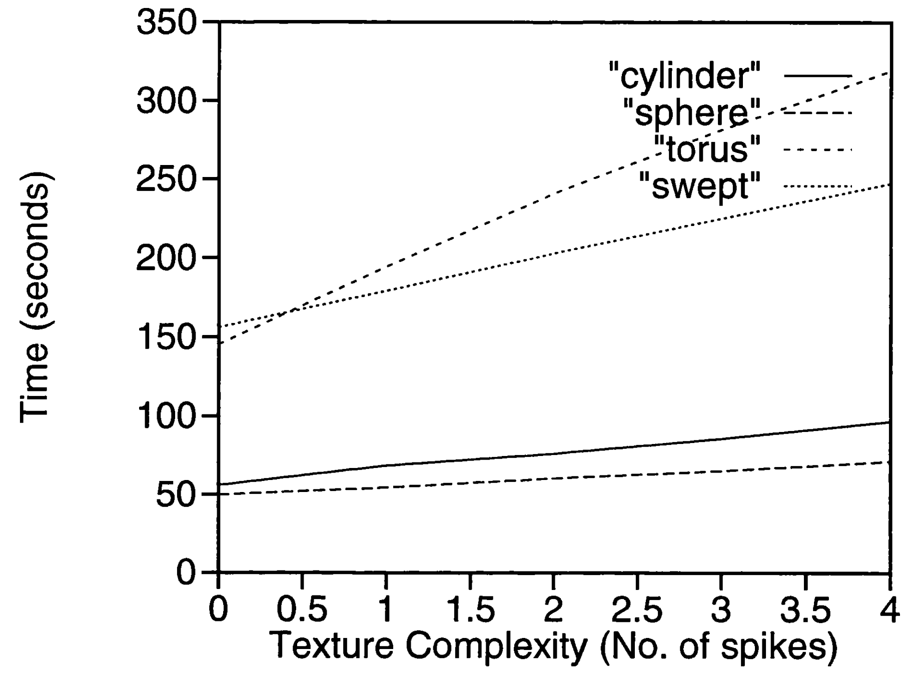


Figure 5.2: Performance of inverse displacement mapping at varying texture complexities

increased number of "difficult" intersections to be found. In areas of the texture of constant magnitude (i.e. in the spaces between spikes), the solution can be found very quickly as the two bounding volumes will quickly converge to the solution point. In areas on or near a spike, the "difficult" intersections, the convergence will be much slower.

The final set of results for inverse displacement mapping show the performance for a fixed surface and number of tiles but with tiles of varying size. The tiles range in size from 3 by 3 to 12 by 12 texels. The results are presented in figure 5.3.

These results show that the performance of inverse displacement mapping varies linearly with the size of the tile defining the displacement map. This can be explained by the increased time needed to search the larger texture and the increased likelihood of largely differing texture heights.

5.3 Performance of numerical displacement mapping

This section gives a performance analysis of the techniques for numerically ray tracing displacement-mapped surfaces presented in Chapter 3. The results are presented for four base surfaces namely; spheres, tori, superellipsoids and a Bézier patch surface (a teapot[Cro87]). In all cases the results were generated at 128 by 128 resolution on a

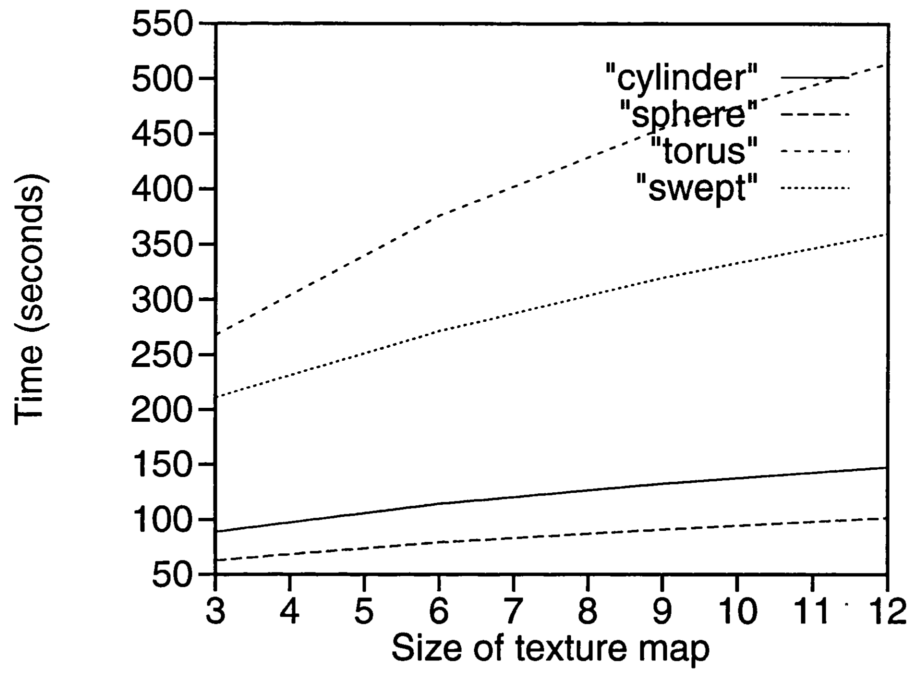


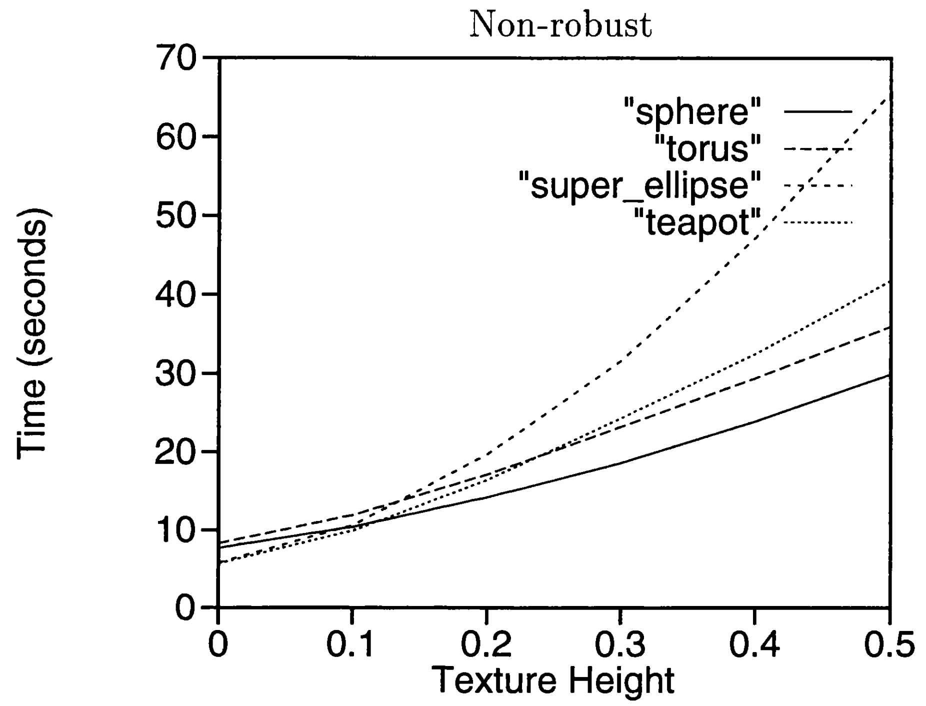
Figure 5.3: Performance of inverse displacement mapping at varying texture sizes

90MHz Pentium processor running the NeXTStep operating system. The reason for generating the images at low resolution (compared to the inverse and LITUNI tests) is the enormous time taken for the numerical techniques, which made larger images impractical.

The results shown use the same texture data and heights as in the previous section. The results are shown in figures 5.4 to 5.6. These results are given in pairs with the top graph using Newtonian iteration starting from the centre of the texel and the bottom graph using robust methods to calculate the final solution.

The first set of results shows the performance of the numerical techniques as the texture height is increased. The results are shown in figure 5.4.

From these these results it is clear that the performance of the numerical techniques is worst than linear as the texture height increases, this being especially pronounced in the case of the superellipsoid. This occurs partly, as before, from the increase in the size of the object but mainly from the increased time to numerically calculate the final solution. The speed of convergence of the numerical methods depends on the amount of variation of the partial derivatives to the surface. If there is a large variation in these then the algorithm will converge slowly. The increase in the texture height has the effect of increasing the values and the range of these derivatives and hence the time to calculate the solution



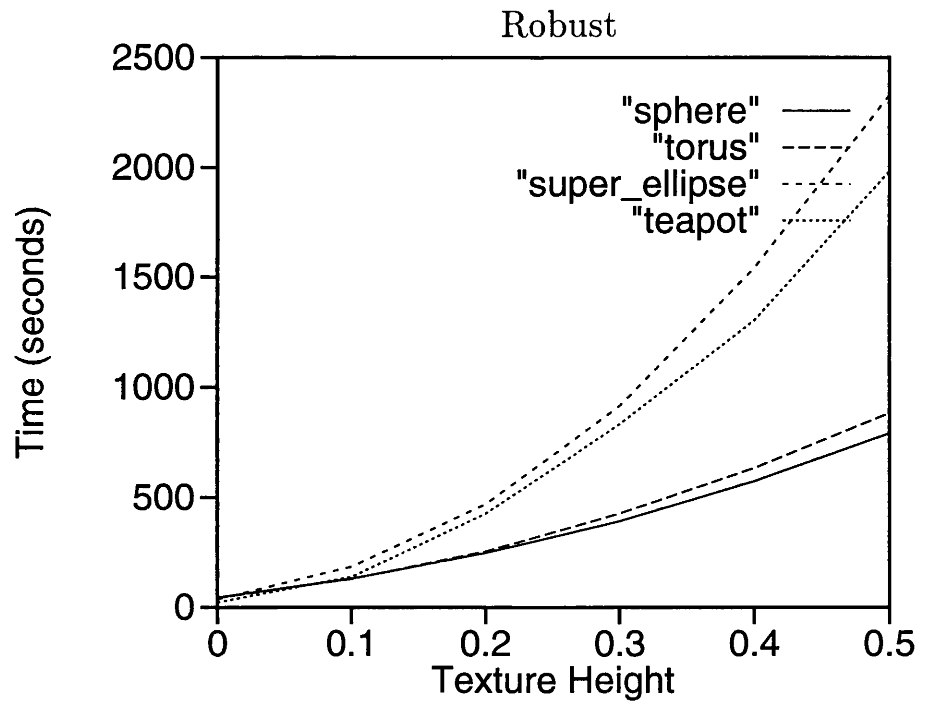


Figure 5.4: Performance of numerical displacement mapping at varying texture heights

points.

The second set of results for numerical displacement mapping show the performance as the texture complexity is increased. These results are shown in figure 5.5.

From these results it is clear that the performance of numerical displacement mapping varies linearly with the complexity of the texture used. The sudden raise in the simple Newtonian scheme arises from the jump from zero texture to non-zero texture. The explanation for these results is similar to the case of inverse displacement mapping. In simple areas of the texture the solution can be found quickly since there is little variation in the surface and its derivatives. At the areas near to spikes this is not the case and the intersection takes considerably longer.

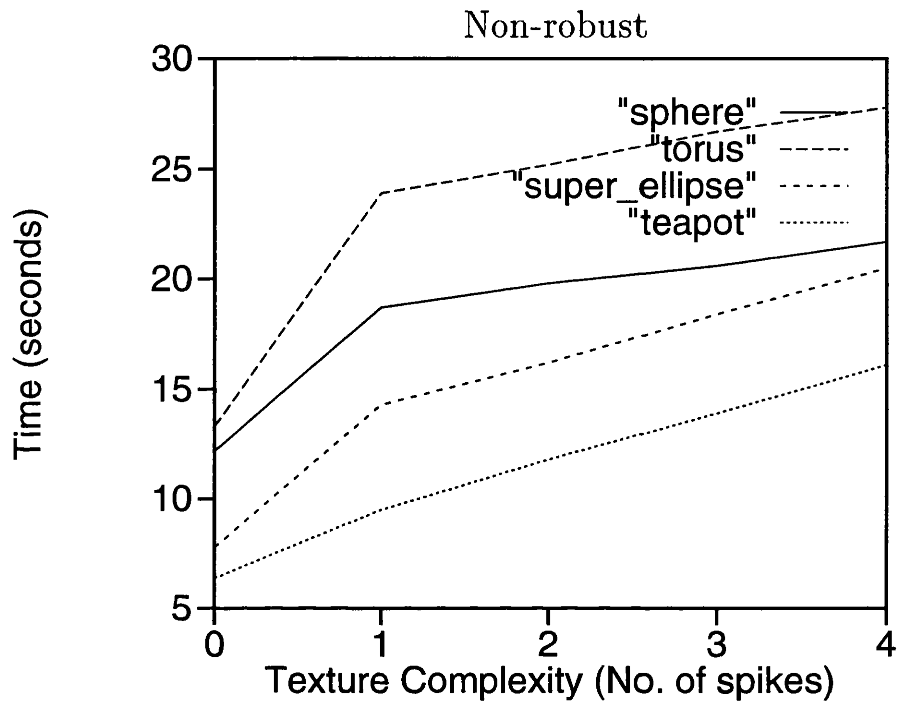
The final set of results for numerical displacement mapping shows the performance for a varying tile size. The results are presented in figure 5.6.

These results show that the performance of numerical displacement mapping varies linearly with the size of the texture. This arises since many more texels exist and will need to be used as starting points for numerical iterations. Although the starting regions will be small and the bounding box tests will exclude many possible numerical intersections, there will still be an increased number as the texture complexity increases.

An important result is the comparative speed of the two numerical methods. On average a simple Newtonian scheme is two orders of magnitude slower than the robust interval method. This factor arises from the significantly greater overhead for interval computations and the irregularity of the displacement-mapped surfaces. This large irregularity makes it very time-consuming to find safe starting regions for the Newtonian iteration. That said, a simple Newtonian scheme is not sufficient by itself as the irregularity of the surface causes many missed pixels.

5.4 Comparison of inverse and numerical techniques

The preceding two sections have given the details of inverse and numerical techniques for intersecting a ray and a displacement-mapped surface. From these results it is clear that in the cases where either may be used, inverse displacement mapping is the faster technique. It should always be remembered that the images for inverse displacement mapping are 16 times larger, and for fair comparison any numerical timing should be multiplied by this



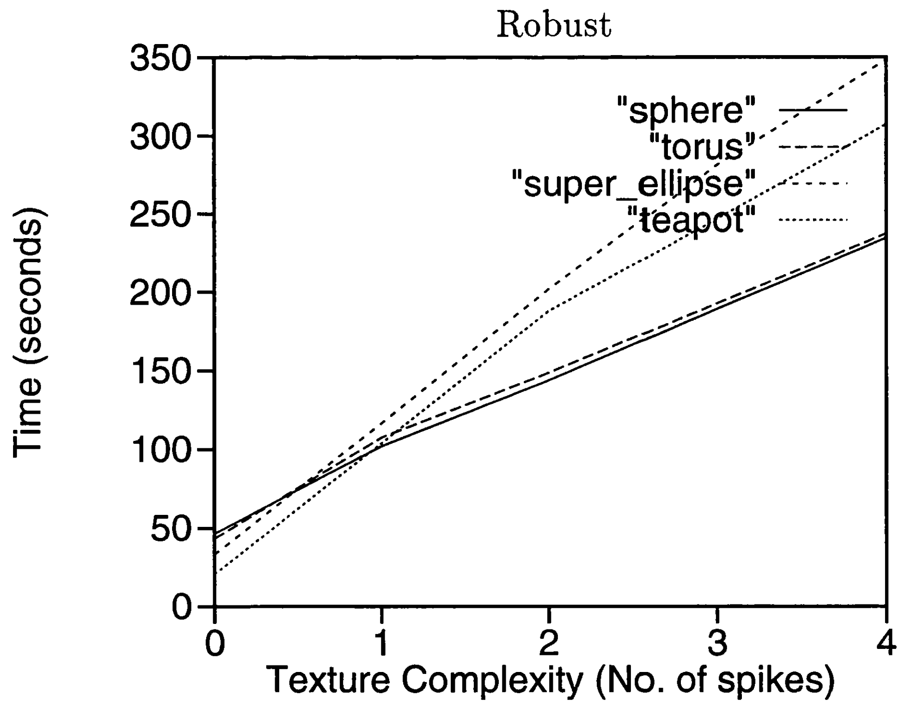
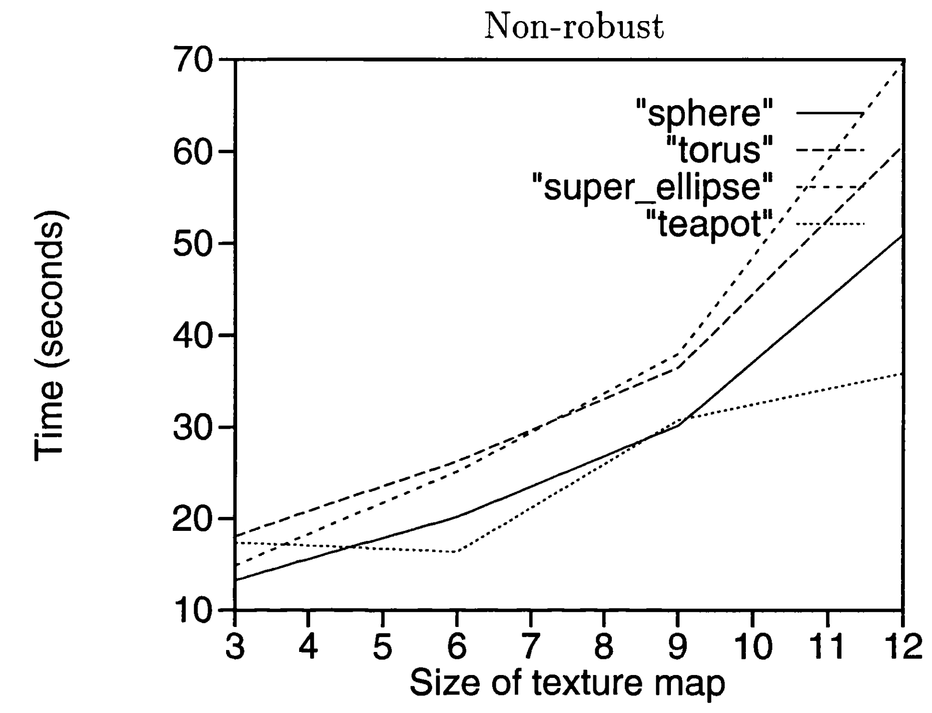


Figure 5.5: Performance of numerical displacement mapping at varying texture complexities



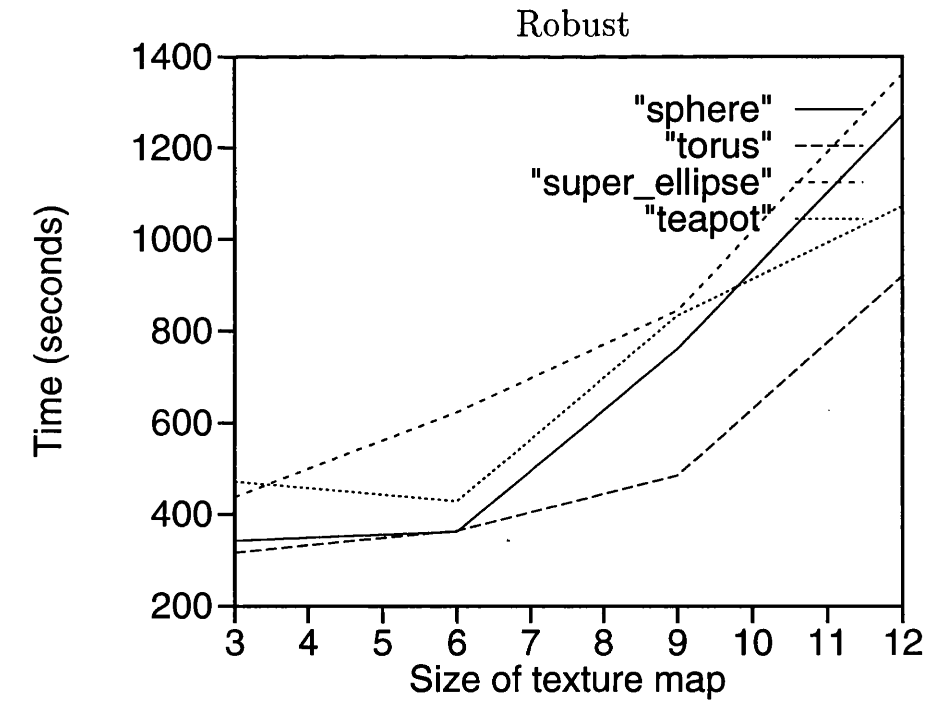


Figure 5.6: Performance of numerical displacement at varying texture sizes

factor. At first sight, it may seem difficult to draw more detailed conclusions about the comparative performance of the two algorithms as they are, on the surface, very different. However this is not actually the case; as there are in fact many parallels between the ways both algorithms proceed.

The similarities between the two algorithms arise in three main areas. These are (using the terminology of inverse displacement mapping), classification of the problem to a specific case, a core algorithm which calculates the solution by iterative means and the use of surface-specific properties. The first area of similarity is classification. This is obvious for inverse displacement mapping which classifies the problem directly to instances of its base case. The numerical techniques also use a form of classification. This arises in two forms. The first is the subdivision of the surface to texel sized regions. This leads to a number of subproblems which can be solved (remember the numerical techniques require a continuously differentiable surface which only exists within a single texel). If non-robust methods are used this concludes the classification, but for robust methods the solutions must be further classified until safe starting regions are found. The second area of similarity between the two algorithms is that both use an iterative base case. With inverse displacement mapping, the iteration (really recursion) is within the base case where the two bounding surfaces are brought closer together until a solution is found. With numerical techniques, the iteration is within the context of Newtonian iteration, as the starting value is refined until a solution of the desired accuracy is determined. The final similarity between the algorithms is that both use surface-specific properties. This is clear for inverse displacement mapping as the classification stage extracts much geometric information. For numerical techniques, surface-specific properties are used to allow efficiently computable and tight bounds within the interval arithmetic. This step is not necessary to calculate the result but it does allow the results to be produced faster.

With these parallels between the algorithms it becomes clear why inverse displacement mapping outperforms numerical techniques. The classification stage for inverse displacement mapping is tightly focussed using detailed geometric information to generate a few instances of the base case. The classification for numerical techniques classifies to a large number of Newtonian starting points and uses comparatively little geometric information.

Thus, the generality of the numerical techniques, the big advantage allowing them to be applied to a wide range of surfaces, is also their major weakness. Inverse displacement mapping, although limited in the range of surfaces it can handle, is significantly faster.

5.5 Performance of LITUNI

This section gives a performance analysis for LITUNI, the caching ray tracer for ray tracing displacement-mapped surfaces discussed in Chapter 4. The results are presented for four base surfaces namely; spheres, tori, superellipsoids and a Bézier patch surface (a teapot[Cro87]). In all cases the results were generated at 512 by 512 resolution on a 90MHz Pentium processor running the NeXTStep operating system. In all cases the cache block size was 1Mb and the minimum lifetime for both object expansions and hierarchy node in the cache was 512. The figure of 512 corresponds to 1 scanline ($\approx$ 0.2%) of the final image.

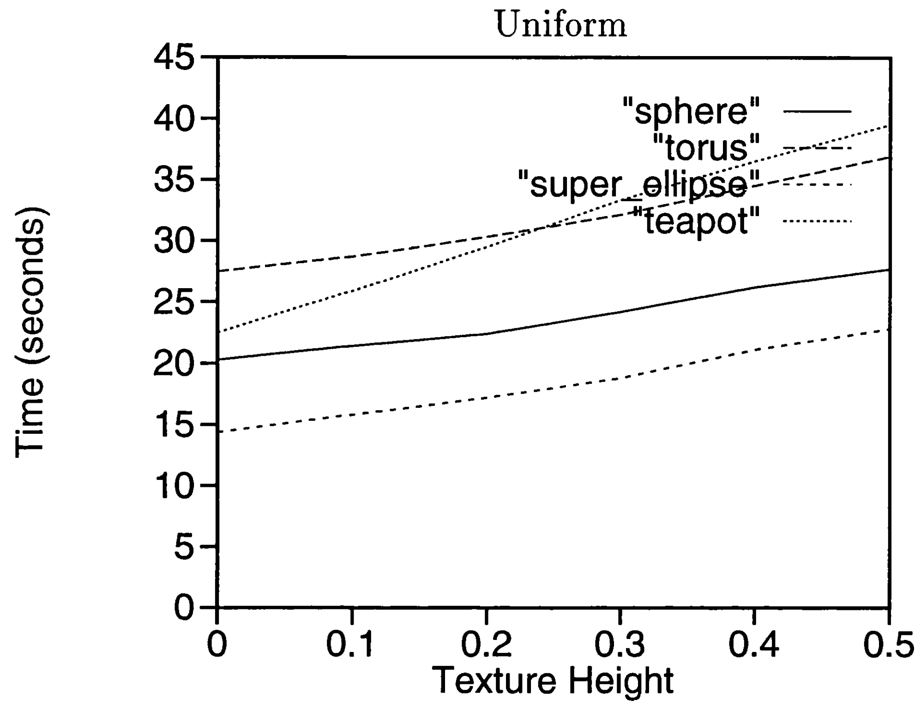
The results are based on the same texture data and heights as in the previous section, and are shown in figures 5.7 to 5.9. These results are given in pairs with the top graph using uniform subdivision to perform the polygonalisation and the bottom graph using adaptive subdivision to perform the polygonalisation.

The first set of results shows the performance of the LITUNI as the texture height is increased. The results are shown in figure 5.7.

These results show that, as in the previous cases, the performance of LITUNI varies linearly with the height of the texture. Although the performance is linear, the gradient of the lines show that the increase in time is minimal, this being most obvious when uniform subdivision is used to calculate the polygonalisations. The reason for the increase in time is that more polygons must be processed. With adaptive subdivision, this arises from the larger number of polygons needed to capture the geometry of the surface. Also, for both adaptive and uniform subdivision, the polygons will cover a larger area and this will increase both the time to classify the expanded lists and the number of primitives in the final candidate lists.

The second set of results show the performance as the texture complexity is increased. The results are shown in figure 5.8.

These results show that LITUNI is relatively insensitive to the complexity of the texture used. In many cases the graphs are close to constant and in the others there is only a modest linear increase in the rendering time. This is caused by the polygonalisation



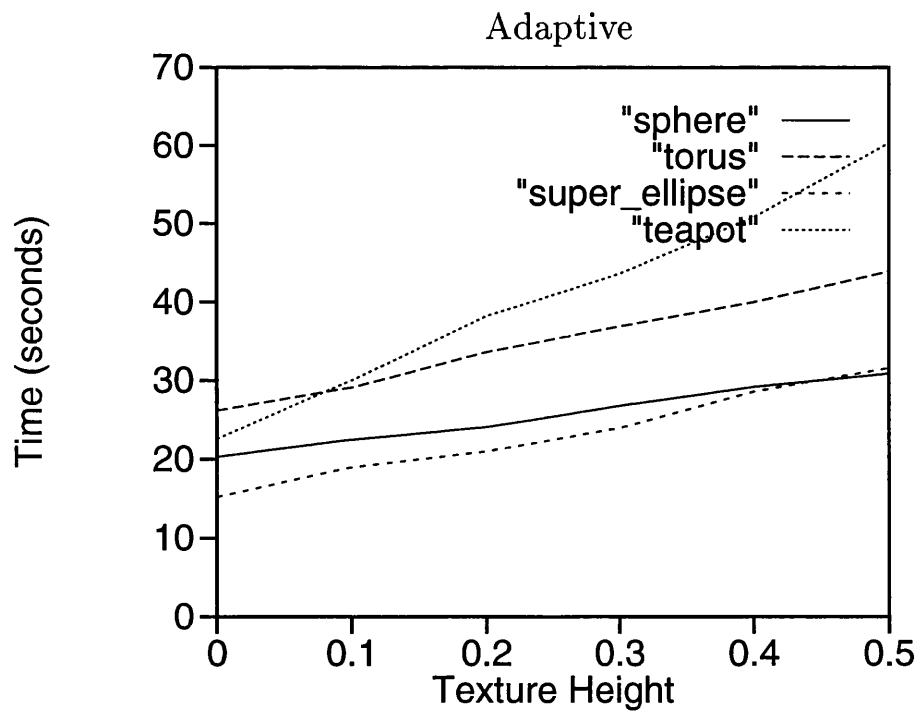
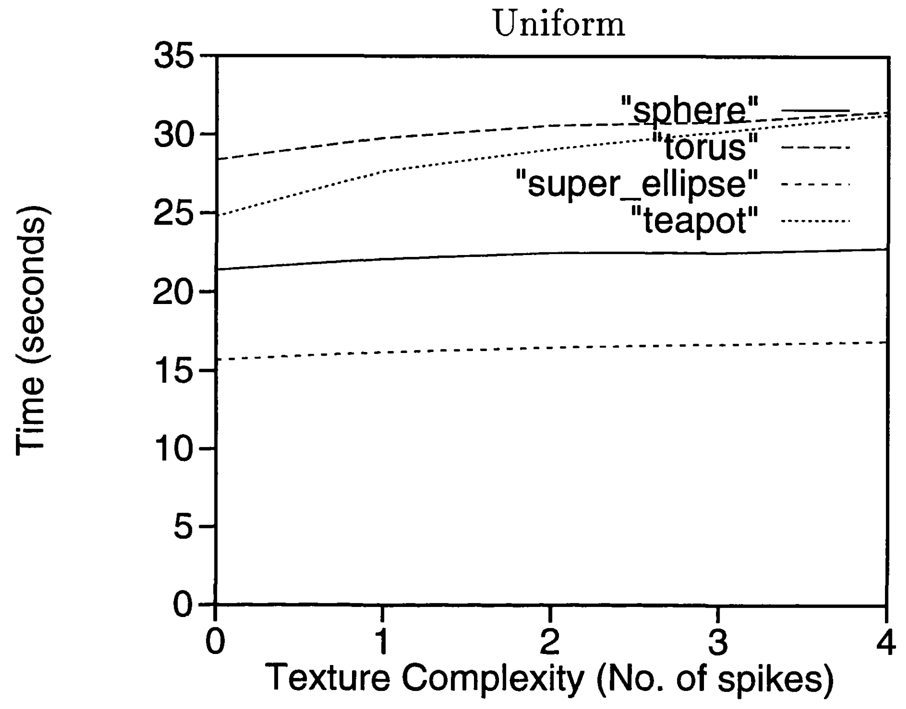


Figure 5.7: Performance of LITUNI at varying texture heights



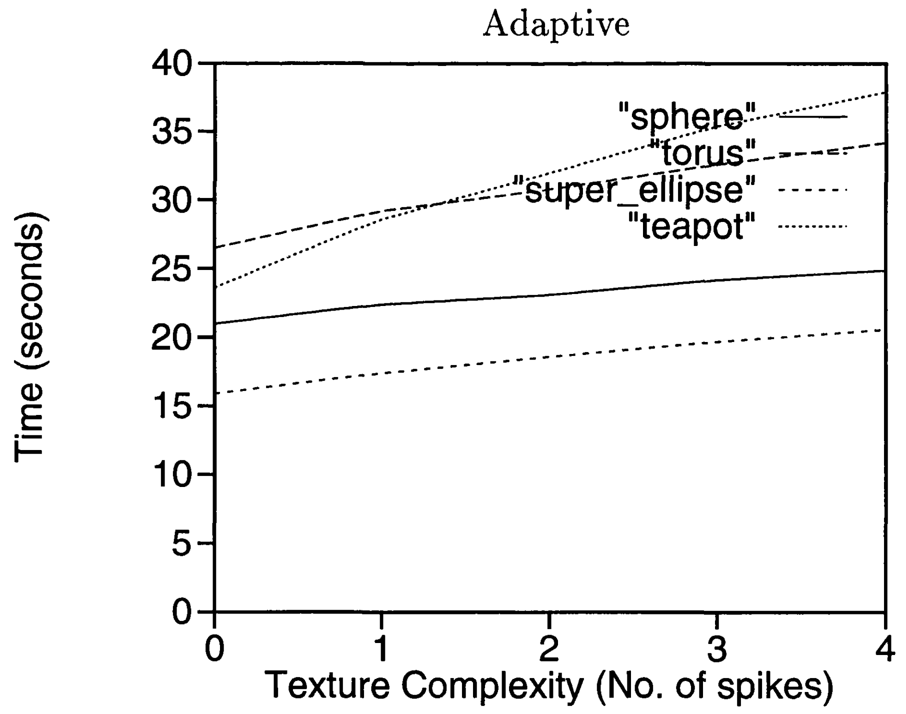


Figure 5.8: Performance of LITUNI at varying texture complexities

Method	Core Intersection	Preprocessing	Memory requirements
LITUNI	ray-polygon	Per object expansion	Medium
Inverse	ray-offset surface	Per ray	Low
Numerical	system of non-linear eqns.	Per ray	Low

Table 5.1: Comparison of Rendering Methods

used in the object expansions. The cost of the expansion and subsequent classification is amortised over many rays and adds only a small amount to the rendering time. The small increases in rendering time which are found are caused by the increased number and area of the generated polygons.

The final set of results for LITUNI show the performance for a varying tile size. The results are presented in figure 5.9.

These results show that the performance of LITUNI varies linearly with the texture size with a small growth factor. The increase in rendering time is caused by the increased number of polygons the system must deal with. This result show that LITUNI will scale well, since the greatly increased number of polygons generated does not translate to a significant rise in rendering time (recall that polygonalisation can only take place inside a texel).

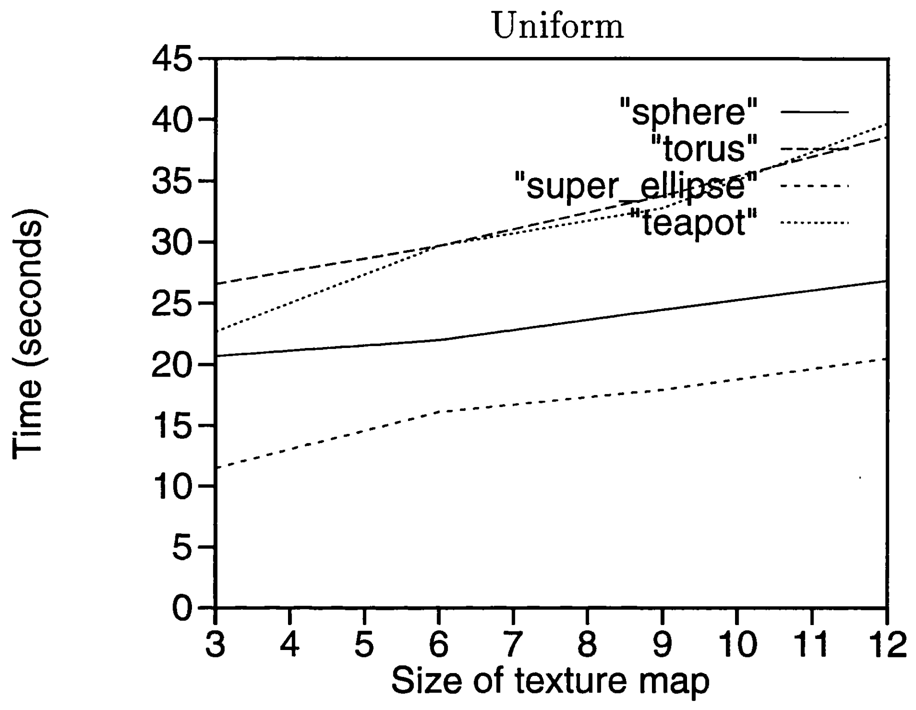
5.6 Comparison of LITUNI and the direct approaches

The results in the previous three sections show the performance for the techniques developed to ray trace displacement-mapped surfaces. From these results it is immediately clear that LITUNI provides the fastest rendering method. The results for LITUNI are an order of magnitude faster than for inverse displacement mapping, these in turn being two orders of magnitude faster then the robust[1] numerical techniques[2].

The explanation for these results can be seen by considering the table in figure 5.1. The core intersection for each method is the calculation carried out in the innermost loop

[1]It is reasonable to use the robust numerical methods for comparison as only these can guarantee a correct solution.

[2]The results for the numerical methods must be scaled up by a factor of 16 for fair comparison due to the smaller image size used.



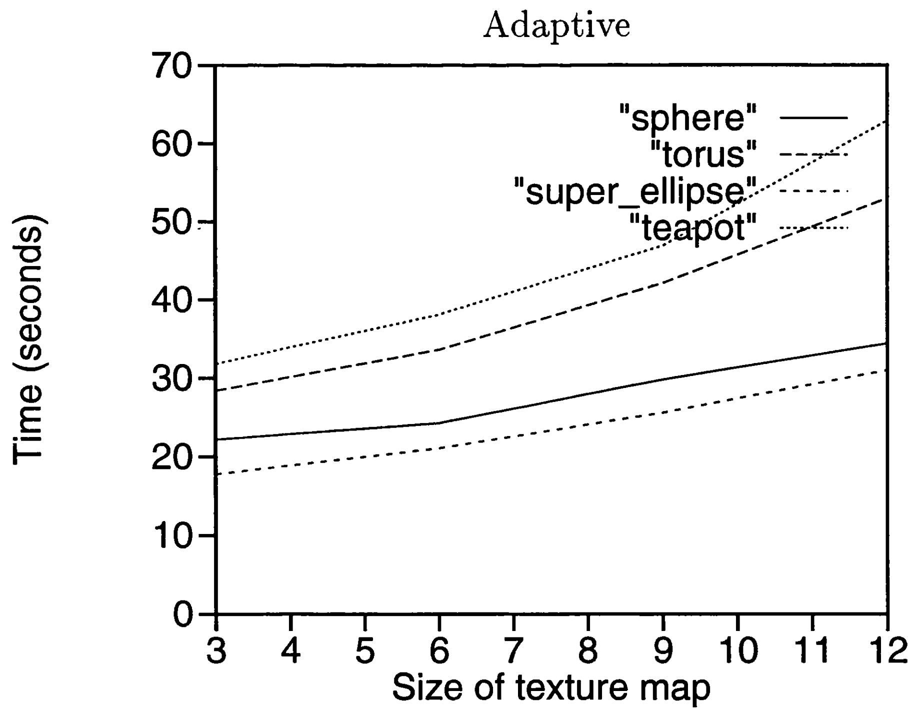


Figure 5.9: Performance of LITUNI at varying texture sizes

and is the dominant factor in the speed of the algorithms. For LITUNI this is a ray-polygon intersection which can be highly optimised and computed very efficiently. This explains the far superior performance of LITUNI compared to the other two cases where the inner loop involves much more computationally demanding tasks. LITUNI is also faster than the other techniques because less preprocessing is needed to prepare for the core intersection. With LITUNI the preprocessing is the object expansion which occurs infrequently and the benefit is spread over many rays. Further for the other techniques, the preprocessing is performed on a per-ray basis and any information calculated is lost thereafter. The one area in which the LITUNI performs worst is memory usage. Since polygonalisation is used, LITUNI needs to store a large amount of auxiliary data. This data can be kept to a reasonable level by the inherent caching design of LITUNI.

Of the rendering methods considered LITUNI, though having higher memory requirement, provides the fastest method to ray trace displacement-mapped surfaces.

5.7 Conclusions

The algorithms developed in this thesis show that, for the first time, it is practical to use ray tracing to render displacement-mapped surfaces. The algorithms developed tackle the problem from three different points of view.

The first algorithm, inverse displacement mapping, tackles the problem geometrically. The geometry of the intersection calculation is analysed for a range of base surfaces and the insight gained is used to provide a reasonably fast algorithm. The limiting factor for inverse displacement mapping is the complexity of the geometry. In cases where the base surface is too complex this technique become unusable as the analysis needed is prohibitively time consuming

The second algorithm, numerical displacement mapping, tackles the problem numerically. The problem is formulated as a system of non-linear equations and the ray displacement-mapped surface intersection reduced to a number of instances of these. This technique was largely unsuccessful as a solution to the problem of ray tracing displacement-mapped surfaces. Non-robust numerical techniques give many visual errors and robust techniques impose a computational burden so great as to make the algorithm unusable.

The final algorithm, LITUNI, tackles the problem algorithmically. The solution was to step back from solely the ray-surface intersection calculation and consider the whole ray tracing process. This leads to a novel architecture for ray tracing in which all data is dynamically created and stored in a cache for only as long as it is needed. The system allows polygonalisation to be used and the intersections calculated using a simple ray-polygon intersection routine. The LITUNI system provides a very fast system for ray tracing displacement-mapped surfaces while at the same time remaining within manageable memory constraints.

Ray tracing is inherently about images, they are what is really important. The actual technique used is unimportant as long as it generates the correct image, does so fast and in limited memory. This thesis extends ray tracing by making displacement mapping not only possible, but practical.

5.8 Future work

The techniques in this thesis show that, for the first time displacement-mapped surfaces can be practically ray-traced and, in doing so, opens up many new areas of research. The technique of displacement mapping has been used very little to date and there are many unanswered questions about where it is useful and how generally it can be applied.

This thesis has been concerned solely with the issue of ray tracing displacement-mapped surfaces and has not addressed the modelling of the surfaces. The surfaces shown in this thesis were created from base surfaces which were either trivial to model, e.g. spheres, or from pre-existing models, e.g. the teapot[Cro87], and textures defined by editing textual descriptions. What is needed is an integrated modelling system which allows the creation and previewing of displacement-mapped surfaces. Such a modeller would need three components to handle displacement-mapped surfaces:

- A modeller for the base surface.
- A texture builder.
- A composition module to apply displacements to a base surface.

The first of these already exists in current modelling packages. The second, a texture builder, could be built from an editor for height fields (a displacement map is in effect a height field). This would allow the texture to be constructed, separately from the base surface, on a plane in a manner independent of the base surface to which it will be applied. The textures built here could be stored and reused later. The third component of the modelling system for displacement-mapped surfaces is a composition module. This should allow the existing textures to be applied to existing surfaces and the results viewed. One important component of this would be a set of routines to render the displacement-mapped surfaces in real-time. Without the capability to view the displacement-mapped surface in real-time it would be difficult to know exactly what the model looked like. Another complex issue with modelling displacement-mapped surfaces is how to interact with them. Should the user be constrained to editing only the base surface and the texture or should direct interaction with the displacement-mapped surface be allowed? A system which allowed direct interaction with the desired model, i.e. the displacement-mapped surface, is certainly more desirable but is problematic. It is unclear how changes to the model would be mapped back to the base surface and the texture. This must be done in a way that the surface generated is in fact the result of applying the texture to the base surface. Further, if changes are made to the texture via the displacement-mapped model the effect, if the texture is tiled, may be very unintuitive. For displacement mapping to be widely used, a method of modelling displacement-mapped surfaces is critical, but it remains an open question as to how this can be done.

The next major area of displacement mapping research is to extend the range of displacement-mapped surfaces which can be generated. There are three independent ways in which this can be achieved. These are by extending each of the functions which define the surface; as described in section 1.2 these are,

- the base surface.
- the texture function.
- the displacement function.

The first two of these are the easiest to extend. For the base surface, any surface onto which a parametric co-ordinate system can be placed is suitable. Thus, displacement mapping can be extended readily to generalised cylinders, generalised swept surfaces and more complex spline representations such as NURBS. There is also no reason why a displacement-mapped surface cannot be displacement-mapped! As shall be discussed shortly this is more easily accomplished for some of the algorithms presented in this thesis than for others. The texture function can also be generalised easily. Throughout this thesis a bi-linear interpolation of height values has been used as the texture function. This choice was made because of its simplicity, the striking geometry it generates and since it generates nasty discontinuities of the derivative which the techniques here have been shown to handle. This texture function can be replaced easily by a more complex interpolation scheme, the obvious choice being bi-cubic interpolations, or by some other more exotic function. The only requirement is that the function is continuously differentiable within the individual texels.

The essence of displacement mapping is the displacement function and it is by extending this that the most striking extensions to displacement mapping become possible. The displacement function need not just be the unit normal to the surface, it can be any function whether derived from the geometry of the surface or not. As described here the displacement function uses only the UV-co-ordinates of the point on the surface to calculate the displacement direction. This information can be augmented with other information to produce a myriad of different displacements for the same point depending on where it is, or what it is doing at a given instant of time. The choice here is limitless and only a series of possibilities for what the displacement function could be are given.

One choice for the displacement function is to use the normal to the surface at a point instead of the unit normal. The magnitude of the normal is proportional to the magnitude of the derivatives so this function produces a large displacement when the surface is changing rapidly and a small displacement elsewhere. The use of the texture function can then define where it is to be applied. A possible use for this displacement function could be to exaggerate areas of the surface with high curvature. Another possibility for the displacement function is to displace a point in a given direction depending on where the point is in space. For example if the points were in the first octant then they could be displaced along a tangent and could be left unchanged elsewhere. A final example is to relate the displacement function to the acceleration of an object. The points on the surface could be displaced in the direction opposite to the direction of motion. This could simulate the affect of inertia.

With such a wide choice for the base surface, texture function and displacement function, the obvious question is to ask “Can these surfaces be handled by the algorithms in this thesis?". The technique of inverse displacement mapping uses a large amount of geometric information about both the base surface and the displacement function. This makes it difficult to extend to other surfaces and displacement functions. Also, in this case, the generalisations would be very specific and each surface and displacement function would have to be coded separately. The numerical methods and LITUNI make few assumptions about the geometry or behaviour of the functions they use and can be extended more easily. All that is required is that the displacement-mapped surface is continuously differentiable over texel-sized regions. This condition can be enforced easily for the base surface and the texture function but may present problems for some displacement functions. This is likely to be a particular problem if the displacement function is bound to areas of space and not to the model.

It should be clear from the conclusions of this thesis that any investigations in this area should concentrate on LITUNI, as it is the only system fast enough to handle the complexity involved.

[Before moving on, the issue of modelling must be addressed again. The methods of modelling discussed earlier can easily handle a range of base surfaces and texture functions. It is unclear how an arbitrary displacement function can be modelled.]

Another area of future work into displacement mapping is to improve and extend the algorithms in this thesis. Since the main conclusion of this thesis was that LITUNI was the only practical solution to ray tracing displacement-mapped surfaces, the work will concentrate on LITUNI. The future work in this area divides into two parts. The first part is on improvements to the existing structure of LITUNI; this will concentrate on improving the key aspects of cache and hierarchy performance, and on improving the speed and quality of the expansions. The second area of future work is extensions to LITUNI; this will concentrate on extending LITUNI to handle parallel ray tracing and ray tracing animations. Each of these will now be discussed in turn.

The first area of future research is to try different structures for the hierarchy which holds the scene. The choice of ray classification, and its ray-based hypercubes, was made because of its speed and ease of updates. Unfortunately it suffers from larger memory requirements than other ray tracing hierarchies. The extra memory requirement is not a major problem because of the caching in LITUNI but a more compact hierarchy would be advantageous. Research in this area should concentrate on octree based structures in preference to hierarchies of bounding boxes as it appears just too difficult, and therefore computationally expensive, to maintain a well-balanced bounding box hierarchy. The use of caching with an octree structure will require efficient algorithms to build and delete the structure. This may or may not turn out to be a fruitful area of research as the gains from decreased memory usage (and hence the ability to store more in the cache) may be outweighed by the extra computation performed. This can only be decided empirically.

The next major area of improvement to LITUNI is the design of better caching strategies. The expansion depth scheme could be replaced by a method which makes a more intelligent choice about where to insert objects and when to delete them. This could be done by using information about the size of the objects, i.e. the volume they enclose, to decide where to insert them. The idea here is that a large object is likely to be hit by more rays and should be inserted into a larger number of nodes in the hierarchy. Further, large objects should remain in the cache for longer. The cache strategy could also use information about what types of rays hit an object to decide how long it remains in the cache. An object hit by a primary ray is clearly more useful than one hit by a tenth-generation refracted ray. There are clearly many different strategies which can be employed and the usefulness of any one can only be measured by implementing it and testing it on a variety of scenes.

The final area of improvement to LITUNI is concerned with improving the efficiency and quality of the object expansions. The aim is to have a system which automatically expands any object into sub-objects such that the expansion:

- is generated as fast as possible
- introduces no visible artifacts
- can be inserted and deleted easily
- produces coherent groups of sub-objects to improve cache performance

To achieve this, the algorithms to expand objects must decide when to expand to polygons and when to split to new expandable objects automatically. It must also produce polygonalisations which accurately capture the geometry of the displacement-mapped surface but add no extraneous polygons and must have knowledge of the caching strategy to generate expansions favourable to that strategy. Currently many parameters which drive the expansion process are specified manually within the model. The automation of this process is a complex task but could provide significant performance gains.

The current implementation of LITUNI has proved that the concept of a caching hierarchy combined with object expansions produces a very powerful and efficient system. It was designed using techniques with low overheads and a simple caching strategy. The essence of any research into improving on this is to make more complex and informed decisions in such a way that the improved speed of the hierarchy, object expansions and caching strategy is not outweighed by the extra overheads.

The future work described so far has been concerned with improving the existing structure of LITUNI. The next area of future work is to extend LITUNI to calculations the current system does not address. The first of these is a parallel implementation. The LITUNI system can be mapped in a straightforward manner onto a MIMD (Multiple Instruction Multiple Data) parallel architecture. This can be achieved by splitting the image into a number of sub-images and having each processor compute part of the result. This method of parallel ray tracing requires that each processor must have access to the complete scene which will in general be too big to store in the local memory of the individual processors. It is here that the advantages of LITUNI for parallel processing become clear as the caching inherent to LITUNI can be applied to the local memory of each processor. This provides a way to ensure that only a small amount of data must be stored at each node. With this system there is also very little communication overhead. Once a processor has been initialised the only data that needs to be sent is the data to create the object expansions and the returned image. This data, such as spline control meshes and displacement maps, will, in general, be very compact. The very small amount of communication means LITUNI is especially suited to implementations on a network of workstations where communications can be very time consuming (e.g. communication over a busy Ethernet).

Another way in which LITUNI can be extended is to add the capability to handle animated sequences. To do this all the objects and rays, as well as the structures to hold them, must be augmented with an extra time parameter. This means that the rays, objects and the hierarchy must all be extended. Further the caching policy must also be extended so that it can exploit temporal as well as spatial coherence. The easiest part of LITUNI to extend to the temporal dimension is the ray classification hierarchy. This can be done by making the 5-D hypercubes into 6-D hypercubes with the extra dimension partitioning the rays over time. The extension of objects to time is straightforward as the surfaces they define just need to be augmented with information about how the object moves and changes shape. Unfortunately, the expansion of these objects presents a number of problems. In a simple implementation, the objects can be expanded at each discrete time interval where the animation frames lie and the polygonalisations at these times used. If the effect of motion blur is to be simulated then the rays generated can occur at any time interval and polygonalisations at all of these are needed. This problem can be solved by polygonalising at the discrete time intervals where the frames lie and the actual polygons at times between these calculated by interpolation. This is more computationally expensive as each polygon must be calculated by interpolation for each ray. It should be noted that if the shape of an object remains constant and only its position changes then the object need only be polygonalised once and the remaining objects calculated by transforming the polygon vertices. A further problem with extending LITUNI to handle animations is to design a caching strategy which takes advantage of temporal coherence. All of the objects in one frame are likely to be needed in the next frame, but the cache may not be big enough to accommodate this much information. This problem can be solved by calculating the animation in an order other than a frame at a time. One approach to this could be to preprocess the scene and calculate roughly where each of the major objects will lie in each frame and then ray-trace the parts of the animation for these objects in each frame. Once this has been done the remaining parts of each image could then be calculated. In this way a strong element of temporal coherence can be introduced. Further, once an expansion has been calculated the information related to it could be saved to disk and reused later if needed.

There are many other area of further work which arise from this thesis related to extending the ray tracing model itself. These include improvements to the lighting model used and in particular the handling of diffuse reflections. This problem continues to receive much attention by computer graphics practitioners. It is hoped that the computational advantages offered by LITUNI may allow techniques in many areas to now become tractable.

Appendix A

Specific Surfaces for Inverse Displacement Mapping

This appendix provides the surface specific details needed to allow inverse displacement mapping to be implemented for spheres, cylinders, tori and surfaces defined by sweeping a planar cubic curve. To implement inverse displacement mapping for a specific surface type three equations must be considered. These are the equations of:

1. The offset surface
2. The distance from the ray to the surface
3. The path of the ray over the surface

These will be given for the surfaces to which inverse displacement mapping has been applied.

The details of inverse displacement mapping will now be given for a number of surfaces namely – spheres, cylinders, tori and cubic swept surfaces. In all cases the details are presented for some "standard" surface without reference to scaling, rotation or translation. These are accounted for, before calling the inverse displacement mapping routines, by first transforming the ray.

Suppose the standard form of the surface is denoted by $f(u, v)$. This surface can be scaled, rotated and translated by applying a matrix transformation. If all points are specified in homogeneous co-ordinated then the transformation matrix for the surface, M, is

$$M = G.R.V$$

where

G	=	global scaling matrix
R	=	composition of rotation matrices about the origin
V	=	translation matrix

The ray $\underline{r} = \underline{a} + \alpha\underline{b} = (a_x, a_y, a_z) + \alpha(b_x, b_y, b_z)$ can be transformed to a ray relative to the standard surface by post-multiplication by M^{-1}, hence the new ray will be $\underline{r}' = \underline{a}M^{-1} + \alpha\underline{b}M^{-1}$. Solution points calculated for the standard surface can be transformed back to the given instance by multiplication with M. In the discussion that follows all surfaces will be in standard form and all rays are assumed to have been transformed.

A.1 Spheres

For spheres the standard surface is a unit sphere at the origin. This is defined by the parametric equations, $-\pi \leq u \leq \pi$ and , $-\frac{\pi}{2} \leq v \leq \frac{\pi}{2}$ and

$$f(u, v) = [\sin(u)\cos(v), \sin(v), \cos(u)\cos(v)]$$

For a unit sphere, the unit normal is the same as the position vector for the point on the surface, hence the offset surface for a sphere is again a sphere with equation,

$$O_{f,h}(u, v) = (1 + h)\,[\sin(u)\cos(v), \sin(v), \cos(u)\cos(v)]$$

The intersection of a ray and an offset sphere can most easily be calculated in XYZ-space. In this space the intersection is described by the implicit equation

$$(a_x + \alpha b_x)^2 + (a_y + \alpha b_y)^2 + (a_z + \alpha b_z)^2 - (1 + h)^2 = 0$$

This will generate the ray parameter and hence the XYZ-space solution points of the ray-offset surface intersection. The solution points in XYZ-space can be transformed back to UVH-space by the equations,

$$\begin{aligned} u =& \quad atan2(x,z) \\ v =& atan2(y,\sqrt{x^2+z^2}) \\ h =& \sqrt{x^2+y^2+z^2}-1 \end{aligned} \tag{A.1}$$

where $atan2(u,v)$ is the inverse tangent function giving the correct results over the range $[-\pi,\pi]$.

It can be shown geometrically that the ray need only be split due to the ray-surface distance when the ray hits the outer bounding surface and not the inner bounding surface[PHL91]. Further, the closest point to the surface lies midway between the two intersections with outer bounding surface. The UVH co-ordinates can be calculated from equation A.1.

It can be shown[PHL91] that the path of the ray over the surface is

$$v = \frac{2}{\pi}\tan^{-1}\left(k\cos(\pi u+\beta)\right)$$

where

$$\begin{aligned} k &= \tfrac{P_1^2+P_2^2}{P_3} \\ \beta &= \tan^{-1}\tfrac{P_1}{P_3} \end{aligned}$$

$$P_0 = \begin{vmatrix} a_y & a_z \\ b_y & b_z \end{vmatrix}, P_1 = \begin{vmatrix} a_x & a_z \\ b_x & b_z \end{vmatrix}, P_2 = \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

and hence the path has a local maximum or minimum when

$$-2k\cos^2\left(\frac{\pi v}{2}\right)\sin\left(\pi u+\beta\right) = 0$$

Finally, if the ray origin lies within the two bounding surfaces, the UVH values for this point can be calculated from equation A.1.

A.2 Cylinders

For cylinders the standard surface is a cylinder centred at the origin, height 1, bound by the planes $z = 0$ and $z = 1$. This defines a cut cylinder which is really three surfaces –

the side, the base plane and the cap plane. The ray must be intersected with all three surfaces, although the displacement map is only applied to the side of the cylinder.

The side surface for the cylinder is defined by the equation, for $-\pi \leq u \leq \pi$ and $0 \leq v \leq 1$,

$$f(u,v) = [\cos(u), \sin(u), v]$$

The unit normal to this surface is $n(u,v) = [\cos(u), \sin(u), 0]$, hence the offset surface for a cylinder is

$$O_{f,h}(u,v) = [(1+h)\cos(u), (1+h)\sin(u), v]$$

This surface is also a cylinder. A ray can most easily be intersected with this offset surface in XYZ-space. Here the intersection is described by the implicit equation

$$(a_x + \alpha b_x)^2 + (a_y + \alpha b_y)^2 - (1+h)^2 = 0$$

This will generate solution points in XYZ-space which can be converted to UVH-space by the equations, for a point (x,y,z),

$$\begin{aligned} u &= atan2(y,x) \\ v &= z \\ h &= \sqrt{x^2+y^2} - 1 \end{aligned} \tag{A.2}$$

It can be shown, as in the case of a sphere, that the ray need only be split due to the ray-surface distance when the ray hits the outer bounding surface and not the inner bounding surface. Further, the closest point to the surface lies midway between the two intersections with the outer bounding surface. The UVH co-ordinates can be calculated from equation A.2.

The path of a ray over a cylinder can be shown to be a monotonic function and the derivate is never zero. Thus, since the path of the ray over the surface can have no local maxima or minima, the ray need not be split.

Since the offset surface does not enclose a volume of space, side surfaces must be considered. These lie in the base and cap planes. To handle the side surface at $z = 0$ the ray is initially intersected with this plane in XYZ-space and the result converted to

UVH-space. If the height is less than h_{min} then the ray intersects the base of the cylinder. If the height is greater than h_{max} then the ray misses the surface on the base plane, Finally, if $h_{min} \leq h \leq h_{max}$ then h is compared with the texture height at $t(u,0)$. In this case if h is less than the texture height the point is a hit with the base otherwise the point is a segment start value. The plane $z = 1$ is handled in a similar manner.

A.3 Tori

For tori the standard surface is a torus centred at the origin, axis parallel to the Z-axis, with inner radius α and outer radius β. This is defined parametrically by the equations, for $-\pi \leq u, v \leq \pi$,

$$f(u,v) = [\cos(u)(\alpha + \beta\cos(v)), \sin(u)(\alpha + \beta\cos(v)), \beta\sin(v)]$$

The unit normal to a torus is $n(u,v) = (\cos(u)\cos(v), \sin(u)\cos(v), \sin(v))$ and hence the offset surface is

$$O_{f,h}(u,v) = [\cos(u)(\alpha + (\beta + h)\cos(v)), \sin(u)(\alpha + (\beta + h)\cos(v)), (\beta + h)\sin(v)]$$

This surface is again a torus. As in the case of a sphere, a ray can most easily be intersected with a torus by considering it in implicit form. The implicit equation for the intersection is

$$((a_x + tb_x)^2 + (a_y + tb_y)^2 + (a_z + tb_z)^2 - \alpha^2 - (\beta + h)^2)^2 - 4\alpha^2((\beta + h)^2 - (a_z + tb_z)^2) = 0$$

The solutions to this equation will generate points in XYZ-space, these can be transformed back to UVH-space by the equations

$$\begin{aligned} u =& \quad atan2(y, x) \\ v =& \quad atan2(z, \sqrt{x^2 + y^2} - \alpha) \\ h =& \sqrt{x^2 + y^2 + z^2 + \alpha^2 - 2\alpha(x\cos(u) + y\sin(u))} - \beta \end{aligned} \tag{A.3}$$

The path of the ray over the torus can be described by the implicit equation

$$((a_x + tb_x)^2 + (a_y + tb_y)^2 + (a_z + tb_z)^2 - \alpha^2 - (h)^2)^2 - 4(h)((a_x + tb_x)^2 + (a_y + tb_y)^2) = 0$$

If this is differentiated with respect to h then the distance from the ray to the surface has local maxima or minima when,

$$((a_x+tb_x)b_x+(a_y+tb_y)b_y+(a_z+tb_z)b_z)^2((a_x+tb_x)^2+(a_y+tb_y)^2)-\alpha^2((a_x+tb_x)b_x+(a_y+tb_y)b_y)^2 = 0$$

The solution generated can be converted to UVH-space via equation A.3 and if the heights are between h_{min} and h_{max} then the ray split at these points.

Finally, the equation of the path of a ray over a torus is,

$$\tan(v) = \frac{b_z\,(P_0 \sin(u) - P_1 \cos(u))}{b_y P_0 - b_x P_1 + \alpha b_z\,(b_y \cos(u) - b_x \sin(u))}$$

and the ray must be split when,

$$\frac{dv}{du} = 0$$

A.4 Cubic swept surfaces

For cubic swept surfaces the standard surface is built by sweeping a closed cubic B-spline curve defined in the XY-plane and bound by the planes $z = 0$ and $z = 1$. This defines three surfaces – the side, the base plane and the cap plane. The ray must be intersected with all three surfaces, although the displacement map is only applied to the side of the swept surface.

The side surface is defined, for a curve segment $C(u) = (C_x(u), C_y(u))$ with $0 \leq u, v \leq 1$, by

$$f(u,v) = [C_x(u), C_y(u), v]$$

Since the *unit* normal to this surface has a complicated form the definition of the offset surface is such that a ray cannot be easily (or quickly) intersected with it. This problem can be worked around by approximating the offset surface with a surface of the same form. This problem reduces to finding the offset of the cubic curve $C(u)$. Let the offset of C by

1 be denoted by C'. The offset surface is

$$O_{f,h}(u,v) = [C_x(u) + h(C'_x(u) - C_x(u)), C_x(u) + h(C'_x(u) - C_x(u)), v]$$

This is again a swept cubic surface. The approximation involved gives the unit normal as $N(u) = C(u)' - C(u) = (N_x(u), N_y(u))$.

The distance from the ray to the surface is then,

$$h = \frac{(C_y(u) - a_y)b_x - (C_x(u) - a_x)b_y}{N_y(u)b_x - N_x(u)b_y}$$

This equation can be differentiated w.r.t. u and the maximum and minimum values of $\frac{dh}{du} = 0$ found. It should be noted that the resulting equation is degree 4 and can be solved algebraically[1].

The path of a ray over the surface can be described by the equation,

$$v = \frac{b_z(N_y(u)(C_x(u) - a_x) - N_x(u)(C_y(u) - a_y)) + a_z(b_x N_y(u) - b_y N_x(u))}{b_y - b_x}$$

This can be differentiated to find the local maxima and minima.

Since the offset surface does not enclose a volume of space, side surfaces must be considered. The intersection points with these can be found by intersecting the ray with the cut planes to get intersection points (x,y) and solving

$$C_x(u) + (C_x(u)' - C_x(u)).h = x \quad , \quad C_y(u) + (C_y(u)' - C_y(u)).h = y$$

for h. The processing from here is as for a cylinder.

[1]The equation appears to be degree 5 but the co-efficient of u^5 is zero.

Appendix B

Interval Arithmetic

B.1 Basic definitions

Interval arithmetic[Moo66] is an extension to real arithmetic which works with ranges of numbers (intervals) instead of individual numbers. In this context an interval is defined as a closed and bounded set on the real line, i.e. $\{x \in \mathcal{R} : a \leq x \leq b\}$ and is denoted by $[a, b]$[1]. The basic arithmetical operations are defined on intervals in the obvious way. Thus for intervals $X = [a, b]$ and $Y = [c, d]$,

$$
\begin{aligned}
X + Y &= \{x + y : x \in [a, b] \ \ and \ y \in [c, d]\} \\
&= [a + c, b + d] \\
X - Y &= \{x - y : x \in [a, b] \ \ and \ y \in [c, d]\} \\
&= [a - d, b - c] \\
X * Y &= \{x * y : x \in [a, b] \ \ and \ y \in [c, d]\} \\
&= [\ \min(a * c, a * d, b * c, c * d), \\
&\quad \max(a * c, a * d, b * c, c * d)\ ] \\
X/Y &= \{x/y : x \in [a, b] \ \ and \ y \in [c, d]\} \\
&= [a, b] * [1/d, 1/c]
\end{aligned}
$$

[1]It is assumed $a \leq b$.

Clearly division is only defined if $0 \notin Y$. Also, any real number x can be considered to be the interval $[x, x]$. Thus real arithmetic subsumes interval arithmetic. From the above definitions it is easily shown that $+$ and $*$ are associative and commutative, i.e. for intervals X, Y and Z

$$\begin{aligned} X + Y &= Y + X \\ X * Y &= Y * X \\ X + (Y + Z) &= (X + Y) + Z \\ X * (Y * Z) &= (X * Y) * Z \end{aligned}$$

At this point the first difference between real and interval arithmetic arises in that interval arithmetic is not distributive so, in general, for intervals X, Y and Z,

$$X * (Y + Z) \neq X * Y + X * Z$$

This can be seen by taking $X = [1, 2]; Y = [1, 2], Z = [-2, -1]$

$$\begin{aligned} X * (Y + Z) &= [1, 2] * ([1, 2] + [-2, -1]) \\ &= [1, 2] * ([-1, 1]) \\ &= [-2, 2] \end{aligned}$$

whereas

$$\begin{aligned} X * Y + X * Z &= [1, 2] * [1, 2] + [1, 2] * [-2, -1] \\ &= [1, 4] + [-4, -1] \\ &= [-3, 3] \end{aligned}$$

Although not distributive interval arithmetic is sub-distributive so

$$X * (Y + Z) \subset X * Y + X * Z$$

Interval arithmetic is also inclusion monotonic. This means that if $X_1 \subset X_2$ and

$Y_1 \subset Y_2$ then $X_1 + Y_1 \subset X_2 + Y_2$ and similarly for $-, *$ and $/$. This may be easily verified from the definition of the operations.

B.2 Interval extensions to functions

One important concept in interval mathematics is the interval extension of a real function f to the interval function F. The interval extension is such that

$$f(x_1, x_2, ..., x_n) = F(x_1, x_2, ..., x_n)$$

for all real numbers $x_1, x_2, ..., x_n$. These extensions, for rational polynomials, can be calculated by replacing all real operations by interval ones. Any function built in this way is also inclusion monotonic.

When an interval function is evaluated the result is an interval which contains all the values the function could take. It is not necessarily the exact range of the function, e.g. let $F = X^2 - X$ then $F([0,1]) = ([0,1])^2 - [0,1] = [-1,1]$ whereas the exact range is $[-0.25, 1]$. In general, the more interval operations used the larger the resulting interval will be. Thus, if a polynomial is the be evaluated, a tighter interval will be obtained if Horner's rule is used. It is also possible to obtain tighter intervals by using knowledge of the function.

This may be easily done for a number of common functions. The squaring function $(X \longmapsto X^2)$ is an obvious candidate. If it is evaluated by multiplication then a wide interval is obtained whereas by considering a number of cases a much tighter interval can be found. First if a number is squared the result is positive, hence the lowest value there can be in the resulting interval is zero. By considering the signs of the end points, it can be shown that

$$sqr([a,b]) = \begin{cases} [a^2, b^2] & if a \geq 0 \\ [b^2, a^2] & if b \leq 0 \\ [0, \max(a^2, b^2)] & otherwise \end{cases}$$

This form of calculation may seem more complicated but will in general be more efficient[2]

[2]It may be more computationally expensive to calculate but the tighter interval will cause any use of

as the resulting interval will be smaller. Two questions now arise:

- Can interval versions of other functions e.g. cos, sin, exp, etc be found?
- Can tight or even exact intervals be found for all functions?

The answers to these questions are the subject of the next section.

B.3 Interval functions and narrow ranges

It can be shown that the error (i.e. the distance from the exact range) of an interval function evaluation is related to the size of the interval. Thus, if smaller intervals are used[Ala85] then a smaller error occurs. If an interval $[a, b]$ is split into smaller intervals $[a_0, a_1], [a_1, a_2], ..., [a_{n-1}, a_n]$ where $a_0 = a, a_n = b$ and $a_i \leq a_{i+1}$ then $F([a_0, a_1]) \cup F([a_1, a_2]) \cup ... \cup F([a_{n-1}, a_n]) \subset F([a, b])$. Since the error in each term is smaller than the total error a tighter bound is formed.

There is another approach to this problem[MK84]. Consider a real-valued function $f(x_1, ...x_n)$. This function has extreme values (i.e. maxima and minima) at points where the derivatives vanish, i.e. when

$$\frac{\partial f}{\partial x_1}(x_1, ..., x_n) = 0$$

$$\vdots$$

$$\frac{\partial f}{\partial x_n}(x_1, ..., x_n) = 0$$

The only other points where maxima or minima could occur are at infinity. Thus the function, evaluated over a finite interval, can have maximum or minimum values (giving the exact range) at a point where the derivatives vanish or along the boundaries of the interval.

Consider first the case where $n = 1$ and f represents a curve. In this case the boundaries are points and the maxima and minima must occur at isolated[3] points. These are called

the result to proceed faster.

[3]If the curve is constant then the maximum and minimum values are curves but this case can be ignored as the exact range for any interval is just this constant.

the characteristic points of the curve. Now, if f is split into a number of segments at these points, the range of f over a given interval can be calculated by considering only its end-point values. These must be the maximum and minimum values for the curve segment (by definition) as the interval has no internal extreme values. This is very useful as it gives the exact range for the function and can be calculated by evaluating only the end-points (Clearly real arithmetic is faster than interval arithmetic). If the given interval spans a number of curve segments then the result is the union of the results from the individual segments.

In the case where $n \geq 2$ the problem is more complex as the maxima and minima can lie along a curve and the surface can't be split. Although this is true in general, a number of particular surfaces commonly used in computer graphics can be split. These include spheres, planes, cones, cylinders, tori, prisms, surfaces of revolution and bicubic surfaces. If the surface is known to have other bounding properties these can also be used to calculate interval extensions. This is the case for spline surfaces where the surface is known to lie within the convex hull of the control points. If a spline surface is refined to a given interval (i.e. part of the surface) then the minimum bounding box of the control points provides an accurate (although not exact) bound for the surface.

The above discussion also shows the way to handle interval extensions to irrational functions such as log, exp and the trigonometric functions. For these the extreme values are easily found and the exact range over a given interval can be calculated. For example $\frac{d}{dx}e^x \neq 0$ for all real x and since e^x is an increasing function

$$e^{[a,b]} = \left[e^a, e^b\right]$$

B.4 Systems of non-linear equations

This section discusses the use of Newtonian iteration to solve a system of non-linear equations. The method presented is general and robust. Interval arithmetic is used extensively to find safe starting regions for real iteration and the method is guaranteed to find all solutions over a finite region. Initially, ordinary Newtonian iteration is presented and its limitations discussed. Next an interval version of Newtonian iteration is given. This finds all solutions over a finite region (an interval) and guarantees safe starting regions. An algorithm to perform these computations is given.

B.5 Newtonian iteration

The standard method of solving a system of non-linear equations is to use Newtonian iteration. This method takes an initial approximation to the solution and refines it to the required accuracy. If the system is given by

$$f(\underline{v}) = 0$$

where $\underline{v} = (v_1, ...v_n)$, $f(\underline{v}) = (f_1(\underline{v}), ..., f_n(\underline{v}))$ and the initial approximation is $\underline{v_0}$ then Newtonian iteration is given by

$$\underline{v}_{k+1} = \underline{v}_k - J^{-1} f(\underline{v}_k)$$

The matrix J is the Jacobian of f evaluated at $\underline{v}_k$. This method is very powerful but has a number of drawbacks. The first problem is that an initial approximation to the solution is required. If this approximation is poor then the system may converge very slowly, converge to the wrong solution or even fail to converge at all. The system if it converges will only find one solution, if a number of valid solutions exist a starting value must be found for each. Also if the Jacobian matrix is singular[4] then the system will breakdown.

All of the above problems can be solved by using an interval version of Newtonian iteration. This is the subject of the next section.

B.6 Interval mathematics and Newtonian iteration

This section presents a robust algorithm for solving a system of non-linear equations. The algorithm uses an interval version of Newtonian iteration to find regions in which a real Newtonian scheme is guaranteed to converge to a unique solution or to find regions in which no solution can exist.

The algorithm is based on Krawczyk's operator[MJ77, Moo77, Moo78] which is defined

[4] In this case the inverse of J is undefined.

as

$$K(X, y, Y) = y - Yf(y) + (I - YF'(X))(X - y) \tag{B.1}$$

where X is an interval vector, y is any real vector in X and Y is any interval matrix. It can be shown that this operator provides, for $x \in X$, an interval extension to

$$v(x) = x - J^{-1}.f(x)$$

which is the real Newtonian scheme. The operator K can be shown to contain all solutions to $f(x) = 0$ for $x \in X$. Thus if $K(X, y, Y) \cap X = \emptyset$ then there can be no solutions to the system of equations in X.

Moreover $K(X, y, Y)$ can be used to find safe starting regions for Newtonian iteration. To this end define

$$r = ||I - Y.F'(X)||$$

where $||A|| = \max_i \sum_j |A_{ij}|$. This definition arises as a distance function for a metric space on interval matrices which is part of a rigorous mathematical treatment of the subject. If $r < 1$ then on successive iterations of $K(X, y, Y)$ the values become closer together and the system converges.

From this it can be proved that,

- If $K(X, y, Y) \subset X$ and $r < 1$ then the real Newtonian scheme will converge to a unique solution for any $x_0 \in X$.
- If $a = min(d_i)$, where $(d_0, ..., d_n) = w(X)/2$ then if $r < 1$ and $||m(X) - (x - Yf(m(X)))|| < a(1 - r)$ then the interval Newtonian scheme converges to a unique solution for any $x_0 \in \{x \in X : ||x - m(X)|| \leq a\}$
- If $r < 1$ then the system

  $$X_{n+1} = K(X_n, m(X_n), Y_n) \cap X_n$$

  where $Y_0 = Y$ and $X_0 = X$ will either converge to a unique solution or will break down due to empty intersection (showing that no solution exists).

The conditions defined above allow regions which contain no solutions or a unique solution to be identified. Before an algorithm to find all solutions in a region can be given a method to handle regions which meet none of the previous conditions must be found. This can be done by recursively splitting the region into a number of subregions and then testing these. Since machine computation can only handle numbers to a finite precision the recursion must be stopped at some predefined level to avoid floating point errors. If this is done it is possible to form a region which cannot be split any further and meets none of the criteria to safely start iteration. These regions must be handled but some other method[5]. A general algorithm to find all solutions to $f(x) = 0$ for $x \in X$ is given in figure B.1.

[5]Since the purpose of this work is to calculate the intersection of a ray and a surface, specialised methods can be found to handle this case.

```
Put $X$ into the list of regions still to be processed
**while** there are still regions to be processed **do**
    Let $A$ equal the first region in the list
    Compute K=$K(A, m(A), m(F'(A)))$
    **if** $K \cap A = \emptyset$ **then**
        **break**
    **endif**

    Compute $r = ||I - m(F'(A))||$

    **if** $r < 1$ **then**
        **if** $K \subset A$ **then**
            Calculate the unique solution by simple Newtonian iteration
            **break**
        **else**
            Calculate the solution (or determine that no solution exists)
                by interval iteration
            **break**
        **endif**
    **else**
        **if** $A$ is too small to be split **then**
            Add $A$ to a list of regions to be processed by some other technique
        **else**
            Split the region $A$ along its longest side
            Add the two subregions to the list of regions still to be processed
        **endif**
    **endif**
    **end while**
```

Figure B.1: Solving non-linear equations

Bibliography

[AK87] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):55–64, 87.

[Ala85] J. Alander. On interval arithmetic range approximation methods of polynomials and rational functions. *Computers and Graphics*, 9(4):365–372, 85.

[Ama84] J. Amanatides. Ray tracing with cones. *Computer Graphics*, 18(3):129–135, 84.

[AU90] S. Aomura and T. Uehara. Self-intersections of the offset surface. *Computer Aided Design*, 22(7):417–422, 90.

[AWW85] G. Abram, L. Westover, and T. Whitted. Efficient alias-free rendering using bit-masks and look-up tables. *Computer Graphics*, 19(3):53–59, 85.

[Bar81] A. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

[Bar87] A. Barr. Ray tracing deformed surfaces. *Computer Graphics*, 20(4):287–196, 87.

[BK85] W. F. Bronsvoort and F. Klok. Ray tracing generalized cylinders. *ACM Trans. Grap.*, 4(4):291–303, 85.

[Bli78a] J. F. Blinn. Models of light reflection for computer synthesised pictures. *Computer Graphics*, 11(2):192–198, 78.

[Bli78b] J. F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics*, pages 286–292, 78.

[BN76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[Bou85] Christian Bouville. Bounding ellipsoids for ray-fractal intersection. *Computer Graphics*, 19:45–52, July 1985.

[Cat74] E. Catmull. *A subdivision Algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, Salt Lake City, 1974.

[CCC87] R. L. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. *Computer Graphics*, 21(4):95–102, 87.

[CG85] Richard J. Carey and Donald P. Greenberg. Textures for realistic image synthesis. *Computers and Graphics*, 9(2):125–138, 85.

[Coo84] R. L. Cook. Shade trees. *Computer Graphics*, 18(3):223–231, 84.

[Coo89] R. L. Cook. Stochastic sampling and distributed ray tracing. *An Introduction to Ray Tracing*, 89.

[Coq87] S. Coquillart. Computing offsets of b-spline curves. *Computer Aided design*, 19(6):305–309, 87.

[Cro87] Franklin C. Crow. The origins of the teapot. *IEEE Computer Graphics and Applications*, 7(1):8–19, January 1987.

[Cro77] F. C. Crow. The aliasing problem in computer generated shaded images. *Graphics and Image Processing*, 20(11):799–805, 77.

[CT81] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3):307–316, August 1981.

[Duf92] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics*, 26(2):131–138, 92.

[Far89] G. Farin. Curvature continuity and offsets for piecewise conics. *ACM Trans. Grap.*, 8(2):89–99, 89.

[FK90] D. Forsey and R. V. Klassen. An apadtive subdivision algorithm for crack prevention in the display of parametric surfaces. *Graphics Interface*, pages 1–6, 90.

[FLC80] E. A. Feibush, M. Levoy, and R. L. Cook. Synthetic texturing using digital filters. *Computer Graphics*, 14(3):294–301, July 1980.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwa¡ta. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, pages 16–26, 86.

[GD89] S. A. Green and Paddon D, J. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, pages 12–25, 89.

[Gla86] A. S. Glassner. Adaptive precision in texture mapping. *Computer Graphics*, 20:297–306, August 1986.

[Gla84] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 15–22, October 84.

[Gla89] A. S. Glassner. An overview of ray tracing. *An Introduction to Ray Tracing*, 89.

[Gol87] J. Goldsmith. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 87.

[GP90] S. A. Green and D. J. Paddon. A highly flexible multiprocessor solution for ray-tracing. *Visual Computer*, 6:62–73, 90.

[Gre86] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11), November 1986.

[Han83] P. Hanrahan. Ray tracing algebraic surfaces. *Computer Graphics*, 17(3):83–90, 83.

[Han86] P. Hanrahan. Using caching and breadth first search to speed up ray-tracing. *Graphics Interface*, pages 56–61, 86.

[HB87] B. Von Herzen and A. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics*, 21(4):103–110, 87.

[HD91] John C. Hart and Thomas A. DeFanti. Efficient anti-aliased rendering of 3D linear fractals. *Computer Graphics*, 25:91–100, July 1991.

[Hec87] Paul S. Heckbert. Ray tracing Jell-O brand gelatin. *Computer Graphics*, 21:73–74, July 1987.

[Hec86] P. S. Heckbert. A survey of texture mapping. *IEEE Computer Graphics and Applications*, pages 56–67, November 86.

[HL90] P. Hanrahan and J. Lawson. A language for shading and lighting calulations. *Computer Graphics*, 24(4):289–298, 90.

[HSK89] John C. Hart, Daniel J. Sandin, and Louis H. Kauffman. Ray tracing deterministic 3-D fractals. *Computer Graphics*, 23:289–296, July 1989.

[HT92] P. Hsiung and R. Thibadeau. Accelerating arts. *Visual Computer*, 8:181–190, 92.

[JB86] K. L. Joy and M. N. Bhetanabhotla. Ray tracing parametric surface patches utilising numerical techniques and ray coherence. *Computer Graphics*, 20(4):279–285, 86.

[JH88] N Wissel J. Hoschnek. Optimal approximate conversion of spline curves and spline approximation of offset surfaces. *Computer Aided Design*, 20(8):475, 88.

[Kaj86] J. T. Kajiya. The rendering equation. *Computer Graphics*, 20:143–150, August 1986.

[Kaj83] J. T. Kajiya. New techniques for ray tracing procedurally defined objects. *Computer Graphics*, 17(3):91–102, 83.

[Kaj85] J. T. Kajiya. Anisostropic reflection models. *Computer Graphics*, 19(3):15–21, 85.

[KB89] D. Kalra and A. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics*, 23(3):297–306, 89.

[KK86] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–276, 86.

[Kla83] R. Klass. An offset spline approximation for plane cubic splines. *Computer Aided Design*, 15(5):297–299, 83.

[LG90] D. Lischinski and J. Gonczarowski. Improved techniques for ray-tracing parametric surfaces. *Visual Computer*, 6:134–152, 90.

[LP95] J. R. Logie and J. W. Patterson. Inverse displacement mapping in the general case. *Computer Graphics Forum*, Accepted for Publication, 95.

[Max88] N. L. Max. Horizon mapping shadows for bump-mapped surfaces. *Visual Computer*, pages 109–117, 88.

[Mic90] D.P. Michtell. Robust ray intersection with interval arithmetic. *Graphics Interface*, 90.

[MJ77] R. E. Moore and S.T. Jones. Safe starting regions for iterative methods. *SIAM J. Numerical Analysis*, 14(6):1051–1065, December 77.

[MK84] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, pages 7–17, February 84.

[Moo66] R. E. Moore. *Interval Analysis.* Prentice-Hall, 66.

[Moo77] R. E. Moore. A test for existance of solutions to non-linear systems. *SIAM J. Numerical Analysis*, 14(4):611–615, September 77.

[Moo78] R. E. Moore. A computational test for convergence of iterative methods for non-linear systems. *SIAM J. Numerical Analysis*, 15(6):1194–1196, December 78.

[NS94] T. Noma and K. Sumi. Shadows on bump-mapped surfaces in ray tracing. *Visual Computer*, 10(6):300–336, 1994.

[PF90] P. Poulin and A. Fournier. A model for anisotropic reflection. *Computer Graphics*, 24(4):273–282, 90.

[PHL91] J. W. Patterson, S. G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10:129–139, 91.

[Pho75] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[RSC87] W. Reeves, D. H. Salessin, and R. L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, 21(4):283–291, 87.

[RW80a] S Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Comms of ACM*, 14:110–116, 80.

[RW80b] S. M. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 80.

[SA84] T. W. Sederberg and D. A. Anderson. Ray tracing steiner patches. *computer Graphics*, 18(3):159–164, 84.

[SB86] M. A. J. Sweeney and R. H. Bartels. Ray tracing free-form b-spline surfaces. *IEEE Computer Graphics and Applications*, pages 41–49, February 86.

[SB87] J. M. Snyder and A. H. Barr. Ray tracing complex models containing surface tesselations. *Computer Graphics*, 21(4):119–127, 87.

[SNK90] T. W. Sederberg, T. Nishita, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics*, 24(4):337–345, 90.

[Sny92] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, 92.

[Ste84] H. A. Steinberg. A smooth surface based on biquadratic patches. *IEEE Computer Graphics and Applications*, 84.

[TH84] W. Tiller and E. G. Hanson. Offsets of two-dimensional profiles. *IEEE Computer Graphics and Applications*, pages 36–46, September 84.

[TJ85] M. Tamminen and F. W. Jansen. An integrity filter for recursive subdivision meshes. *Computers and Graphics*, 9(4):351–363, 85.

[Tot85] D. L. Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19(3):171–179, 85.

[Ups90] S. Upstill. *The Renderman Companion.* Addison Wesley, 90.

[Vla90] V. Vlassopoulos. Adaptive polygonization of parametric surfaces. *Visual Computer*, 6:291–298, 90.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics*, 26:265–272, July 1992.

[WAT92] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions for complex surfaces. *Computer Graphics*, 26:255–264, July 92.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[Wij84] J. J. Van Wijk. Ray tracing objects defined by sweeping planar cubic splines. *ACM Trans..on Grap.*, 3(3):223–237, 84.

[Wij85] J. J. Van Wijk. Ray tracing objects defined by sweeping a sphere. *Computers and Graphics*, 9(3):283–290, 85.

[Wil83] L. Williams. Pyramidal parametrics. *Computer Graphics*, 17(3):1–11, 83.

[Yan87] C. Yang. On speeding up ray tracing of b-spline surfaces. *Computer Aided Design*, 19(3):122–130, April 87.

[YYiT92] Takami Yasuda, Shigeki Yokoi, and Jun ichiro Toriwaki. A shading model for cloth objects. *IEEE Computer Graphics and Applications*, 12(6):15–24, November 1992.