Advanced Graphics Lecture Notes

Neil Dodgson[*]

University of Cambridge Computer Laboratory

Overview

Syllabus

- **Introduction.** Revision of the ray tracing and polygon scan conversion methods of making images from 3D models; the pros and cons of each approach. Current uses of computer graphics in animation, special effects, Computer-Aided Design and marketing. [0.4 lecture]

- **The polygon.** Drawing polygons. Graphics cards. Polygon mesh management: data structures. [0.6 lecture]

- **Ray tracing.** The primitive geometric shapes used in ray tracing: plane, polygon, sphere, cylinder, cone, box, disc, torus. Ray intersection calculations and normal calculations for these. Converting the primitives into polygons for use in polygon scan conversion. [1.5 lectures]

- **Splines for modelling arbitrary 3D geometry** (splines are the standard 3D modelling mechanism for Computer-Aided Design). Features required of surface models in a Computer-Aided Design package. Bezier curves and surfaces. B-splines, from uniform, non-rational B-splines through to non-uniform, rational B-splines (NURBS). [2.5 lectures]

- **Subdivision surfaces** (an alternative mechanism for representing arbitrary 3D geometry, now widely used in the animation industry). Introduction to subdivision. Pros and cons when compared to NURBS. [1 lecture]

- **Other ways to create complex geometry.** Generative models: extrusion, revolution, sweeping, generalised cylinders. Constructive solid geometry (CSG): set theory applied to solid objects; different implementations of this using ray tracing and polygons. Implicit surfaces and voxels: 3D pixels and the marching cubes algorithm; medical applications of this. [2 lectures]

[*]Written 10/99, modifications made 09/00, 10/02, and 09/04. ©1999, 2000, 2002, 2004 Neil A. Dodgson

Why *Advanced* Graphics? The title "Advanced Graphics" dates from the year in which the course was first proposed. At this time a 16 lecture course on various advanced topics in graphics was envisaged. The course is now only 8 lectures long (the other 8 have been allocated to HCI). It is almost exclusively about 3D modelling techniques, so the course title is, perhaps, a little misleading. Computer Laboratory policy is, however, to minimize changes to course titles. 3D modelling is important because it underpins all of the practical uses of 3D computer graphics.

What's examinable? Everything except where explicitly noted otherwise. This means that anything that is covered in the lectures is examinable, even if it is not in the notes, and that anything that is in the notes is examinable, unless noted otherwise.

Lecture handouts and supervision material Some of the lecture course material is available on the web. This material is also printed out to provide these lecture notes. Other material is bound in with these notes (this material cannot be put on the web for copyright reasons). There are exercises scattered throughout the notes. These can usually be found at the end of sections. My thanks to Jonathan Pfautz and Andy Penrose (now both graduated with their PhDs) for some of the exercises.

Book list and their abbreviations The following books are referred to in the course. Each is preceded by the abbreviation used in these notes to refer to that book.

- **FvDFH** Foley, J.D., van Dam, A., Feiner, S.K. & Hughes, J.F. (1990). *Computer Graphics: Principles and Practice*. Addison-Wesley (2nd ed.). The traditional "bible" of Computer Graphics. It tends to be fairly terse on many topics but it has wide coverage of all of the basics.

- **F&vD** Foley J.D. & van Dam, A. (1984). *Fundamentals of Interactive Computer Graphics*. Addison-Wesley (1st ed.). The earlier version of **FvDFH**. It contains only about half of the material of the second edition, but is still fairly comprehensive about the basics of computer graphics.

- **SSC** Slater, M., Steed, A. & Chrysanthou, Y. (2002). *Computer Graphics & Virtual Environments*. Addison Wesley. A more recent book which covers all the basics. Also has sections on Constructive Solid Geometry (Ch. 18), Quadrics (also Ch. 18), Radiosity (Ch. 15), and an introduction to Bezier and B-Spline curves and surfaces (Ch. 19).

- **R&A** Rogers, D.F. & Adams, J.A. (1990). *Mathematical Elements for Computer Graphics*. McGraw-Hill (2nd ed.). A fairly thorough coverage of the mathematics of the 2D and 3D representation of shape as it was understood in the year of publication. Explains Bezier, B-spline, and NURBS curves and surfaces in great detail. Also covers conics and quadrics.

- **Farin** Farin, G. (2002, 5th ed.; 1997, 4th ed.). *Curves and Surfaces for CAGD*. Morgan Kaufmann (5th ed.). Academic Press (4th ed.). A good alternative source for information on Bezier, B-Spline, NURBS, and conics. Regularly updated since its original publication in 1988.

- **W&W** Warren, J. & Weimer, H. (2002). *Subdivision Methods for Geometric Design*. Morgan Kaufmann. The only book on the market devoted entirely to subdivision methods.
- **GG I-V** *Graphics Gems I* (1990) to *Graphics Gems V* (1995). Acadmic Press. A collection of five books containing a wide variety of algorithms for use in computer graphics. A wide range of tips, tricks and techniques is included.

Other material in the handout In addition to these notes I have included copies of the following:

1. Lorenson and Cline's 1987 SIGGRAPH paper "Marching cubes: a high resolution 3D surface construction algorithm", *Proc SIGGRAPH '87*, pages 163–169. This is relevant to the section on voxels and marching cubes (section 8.3.2).
2. Extracts from Rogers and Adams (**R&A**): table 4-8 (conics), figure 6-18 (quadrics), and *parts* of sections 6-2 (surfaces of revolution), 6-3 (sweeps), 5-8 (Bezier curves), 5-9 (B-splines), and 5-13 (NURBS).

Note on copyright material Both items in the above list are copyrighted material provided under the University of Cambridge's license from the Copyright Licensing Agency. This allows us to make one copy for each student and supervisor ("tutor") on the course *within certain limits*. These are: no more than three works and no more than 5% or one whole article or chapter from each work. This material is provided solely for the student's own study. Further copying of this handout is a breach of copyright.
Be warned: to fit inside these limits I have heavily edited the extracts from **R&A**. In particular, I have included none of the worked examples. To thoroughly understand the material I suggest that you read this extract and then borrow (or buy) a copy of **R&A** in order to go through the examples.

1 Basic 3D modelling

1.1 Ray tracing *vs* polygon scan conversion

These are the two standard methods of producing images of three-dimensional solid objects. They were covered in some detail in the Part IB course. If you want to revise them then check out **FvDFH** sections 14.4, 15.10 and 15.4 *or* **F&vD** sections 16.6 and 15.5. Line drawing is also used for representing three-dimensional objects in some applications. It is briefly covered later on.

1.1.1 Ray tracing

Ray tracing has the tremendous advantage that it can produce realistic looking images. The technique allows a wide variety of lighting effects to be implemented. It also permits a range of primitive shapes which is limited only by the ability of the programmer to write an algorithm to intersect a ray with the shape. It is considered by many to be the natural or obvious way to render 3D objects.

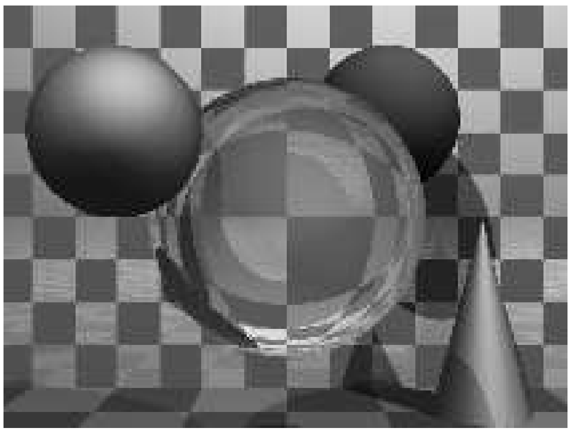


Figure 1: A basic ray traced model showing reflection and refraction.

Ray tracing works by firing one or more rays from the eye point through each pixel. The colour assigned to a ray is the colour of the first object that it hits, determined by the object's surface properties at the ray-object intersection point and the illumination at that point. The colour of a pixel is some average of the colours of all the rays fired through it. The power of ray tracing lies in the fact that secondary rays are fired from the ray-object intersection point to determine its exact illumination (and hence colour). This spawning of secondary rays allows reflection, refraction, and shadowing to be handled with ease. A simple raytraced image can be seen in Figure 1 and the basic algorithm in Figure 12.

Ray tracing's big disadvantage is that it is slow. It takes minutes, or hours, to render a reasonably detailed scene. Until recently, ray tracing had never been implemented in hardware. A Cambridge company, Advanced Rendering Technologies, has now done this. The quality of the images that they can produce is extraordinarily high compared with polygon scan conversion. This is their main selling point. However, ray tracing is so computationally intensive that it is not possible to produce images at the same speed as hardware assisted polygon scan conversion. Other researchers are trying to do this by using multiple processors (dozens to hundreds), but ray tracing will always be slower than polygon scan conversion.

Ray tracing therefore is only used where the visual effects cannot be obtained using polygon scan conversion. This means that it is, in practice, used by a minority of movie and television special effects companies, advertising companies, and enthusiastic amateurs.

1.1.2 Example

The kitchen in Figure 2 was rendered using the ray tracing program *Rayshade*. *Rayshade* is now very old, it has not been updated for at least a decade. An alternative ray tracer, which is kept up to date, is *POVray*, with which you may like to experiment. It is worth visiting the *POVray* website to see the stunning imagery which has been produced using the ray tracer .

The close-ups of the kitchen scene in Figures 3 and 4 show some of the power of ray

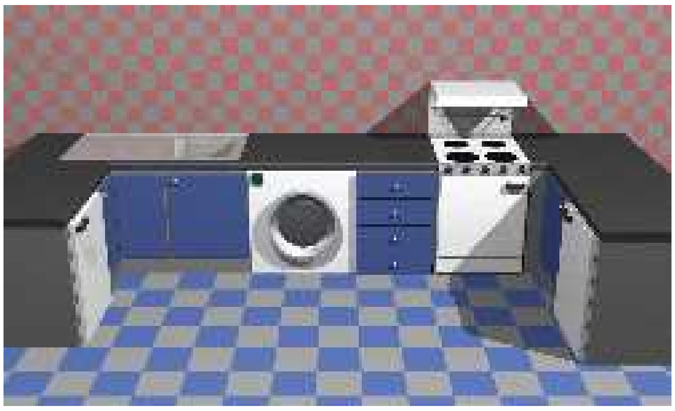


Figure 2: A ray traced model of a kitchen design.

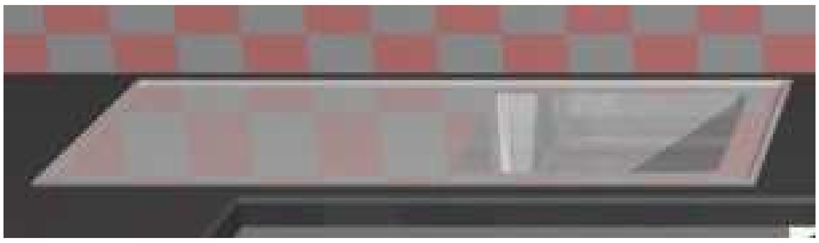


Figure 3: A close up of the kitchen sink.

tracing. The kitchen sink reflects the wall tiles. The bench top in front of the kitchen sink has a specular highlight on its curved front edge. The washing machine door is a perfectly curved object (impossible to achieve with polygons). The inner curve is part of a cone, the outer curve is a cylinder. You can see the floor tiles reflected in the door. Both the washing machine door and the sink basin were made using CSG techniques (see section 8.2). The grill on the stove casts interesting shadows (there are two lights in the scene). This sort of thing is much easier to do with ray tracing than with polygon scan conversion.

1.1.3 Polygon scan conversion

This term encompasses a range of algorithms where polygons are rendered, normally one at a time, into a frame buffer. The term *scan* comes from the fact that an image on a CRT is made up of *scan lines*. Examples of polygon scan conversion algorithms are the painter's algorithm, the z-buffer, and the A-buffer (see your Part IB lecture notes, **FvDFH** chapter 15 *or* **F&vD** chapter 15). In this course we will generally assume that polygon scan conversion refers to the z-buffer algorithm or one of its derivatives.

The advantage of polygon scan conversion is that it is fast. Polygon scan conversion algorithms are used in computer games, flight simulators, and other applications where interactivity is important. To give a human the illusion that they are interacting with a 3D model in real time, you need to present the human with animation running at

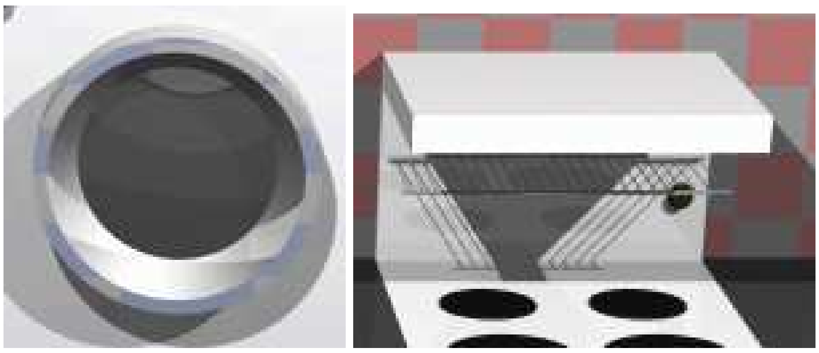


Figure 4: Close up views of the washing machine door and the grill on the stove.

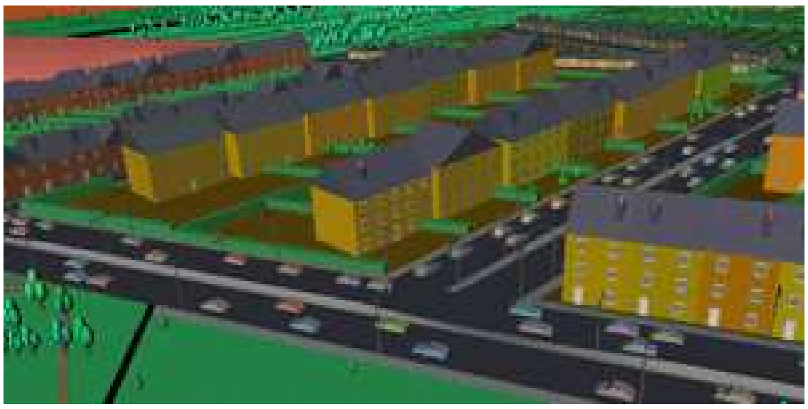


Figure 5: A scan converted model of a city (courtesy of Jon Sewell).

10 frames per second or faster for passive viewing on a monitor, TV, or movie screen. Research at the University of North Carolina has experimentally shown that for immersive virtual reality applications this is not high enough and at least 15 frames per second is a minimum. Polygon scan conversion is capable of providing this sort of speed. The NVIDIA GeForce4 graphics processing unit (GPU), for example, can process 136 million vertices per second in its geometry processor and 4.8 billion antialiased samples per second in its pixel processor making it feasible to render scenes containing several million triangles. The GPU is capable of over 1.2 trillion operations per second. While we may hope that scientific or medical applications were considered important applications of computer graphics, it is in fact the game industry which is currently driving the development of graphics card technology.

One problem with polygon scan conversion is that it can only support simplistic lighting models, so images do not necessarily look realistic. For example: transparency can be supported, but refraction requires the use of an advanced and time-consuming technique called “refraction mapping”; reflections can be supported – at the expense of rendering a “reflection map” before rendering the scene; shadows can be produced using “shadow maps”, a more complicated method than ray tracing. Where ray tracing is a clean and simple algorithm, polygon scan conversion uses a variety of tricks of the trade to get

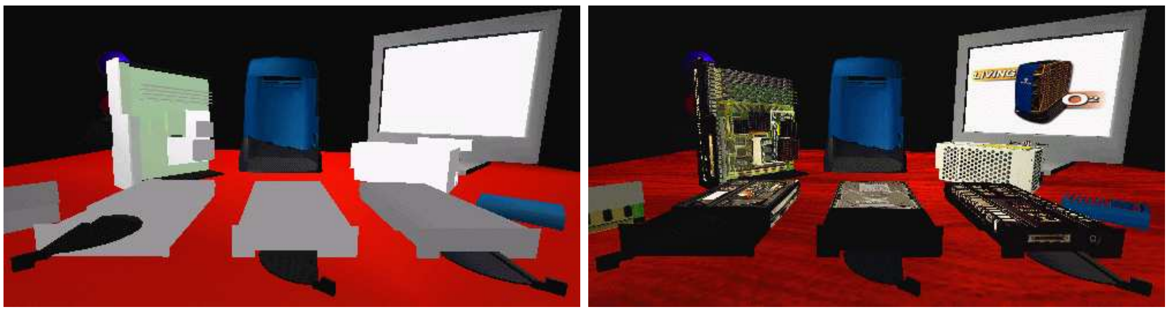


Figure 6: An SGI O2 computer and components drawn with and without texture maps.

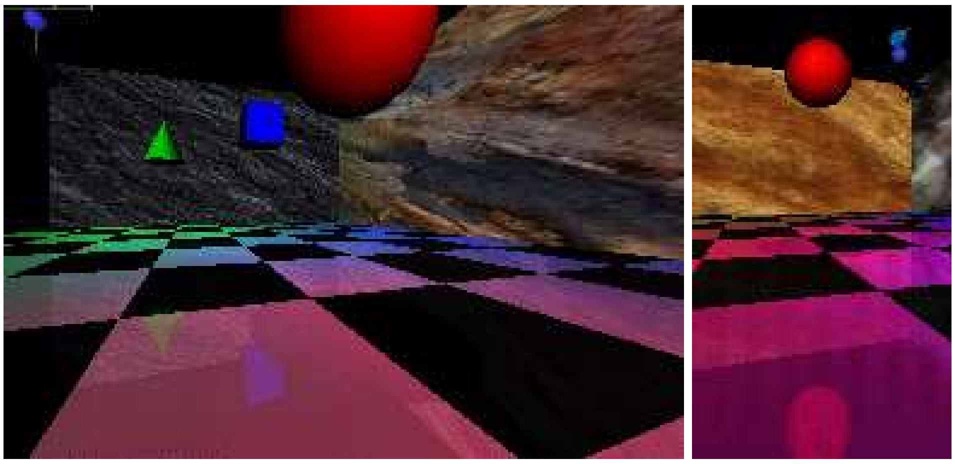


Figure 7: Left: some floating objects in a simulated environment. Right: a close up of the red ball showing the reflection of the ball in the shiny floor.

the desired results. The other limitation of polygon scan conversion is that it only has a single primitive: the polygon, which means that everything is made up of flat surfaces. This is especially unrealistic when modelling natural objects such as humans or animals, unless you use polygons that are no bigger than a pixel, which is indeed what happens these days. An image generated using a polygon scan conversion algorithm, even one which makes heavy use of texture mapping, will still tend to look computer generated.

1.1.4 Examples

Texture mapping is a simple way of making a polygon scan conversion (or a ray tracing) scene look better without introducing lots of polygons. The images in Figure 6 show a scene both with and without any texture maps. Obviously this scene was designed to be viewed with the texture maps turned on. This example shows that texture mapping can make very simple geometry look interesting to a human observer.

The images in Figure 7 were generated using polygon scan conversion. Texture mapping has been used to make the back and side walls more interesting. All the objects are reflected in the floor. This reflection is achieved by duplicating all of the geometry,

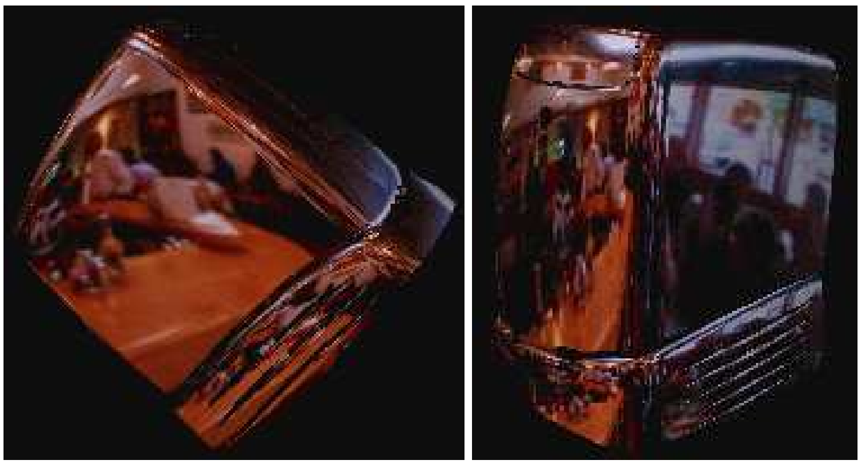


Figure 8: An example of environment mapping: a silvered SGI O2 computer reflecting an environment map of the interior of a cafe.

upside-down, under the floor, and making the floor partially transparent. The close-up shows the reflection of the red ball, along with a circular "shadow" of the ball. This shadow is, in fact, a polygonal approximation to a circle drawn on the floor polygon and bears no relationship to the lights whatsoever. You may need to look at the version of these images on the Lab's website to see the images more clearly and in colour.

Environment mapping (Figure 8) is another clever idea which makes polygon scan conversion images look more realistic. In environment mapping we have a texture map of the environment which can be thought of as wrapping completely around the entire scene (you could think of it as six textures on the six inside faces of a big box). The environment map itself is not drawn, but if any polygon is reflective then the normal to the polygon is found at each pixel (this normal is needed for Gouraud shading anyway) and from this, and a vector pointing to the eye, the appropriate point (and therefore colour) on the environment map can be located. You may note that finding the correct point on the environment map is actually a very simple (and easily optimised) piece of ray tracing.

1.1.5 Line drawing

An alternative to the above methods is to draw the 3D model as a wire frame outline. This is obviously unrealistic, but is useful in particular applications. The wire frame outline can be either see through or hidden lines can be removed (**FvDFH** section 15.3 *or* **F&vD** section 14.2.6). In general, the lines that are drawn will be the edges of the polygons which would be drawn by a polygon scan conversion algorithm.

Line drawing has historically been faster than polygon scan conversion. However, modern graphics cards can handle both lines and polygons at about the same speed. Line drawing of 3D models is used in Computer Aided Design (CAD) and in 3D model design. The software which people use to design 3D models tends to use line drawing in its user interface with polygon scan conversion providing preview images of the model. I find it interesting that, when **R&A** was first written in 1976, the authors had only

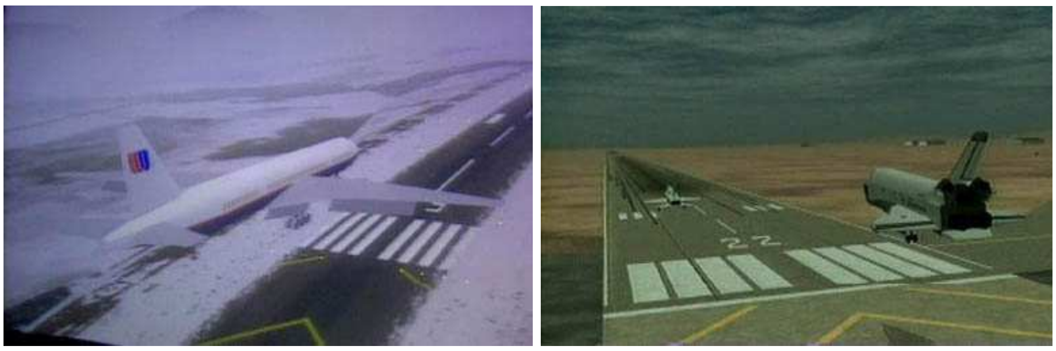


Figure 9: Screen shots from commercial flight simulators.

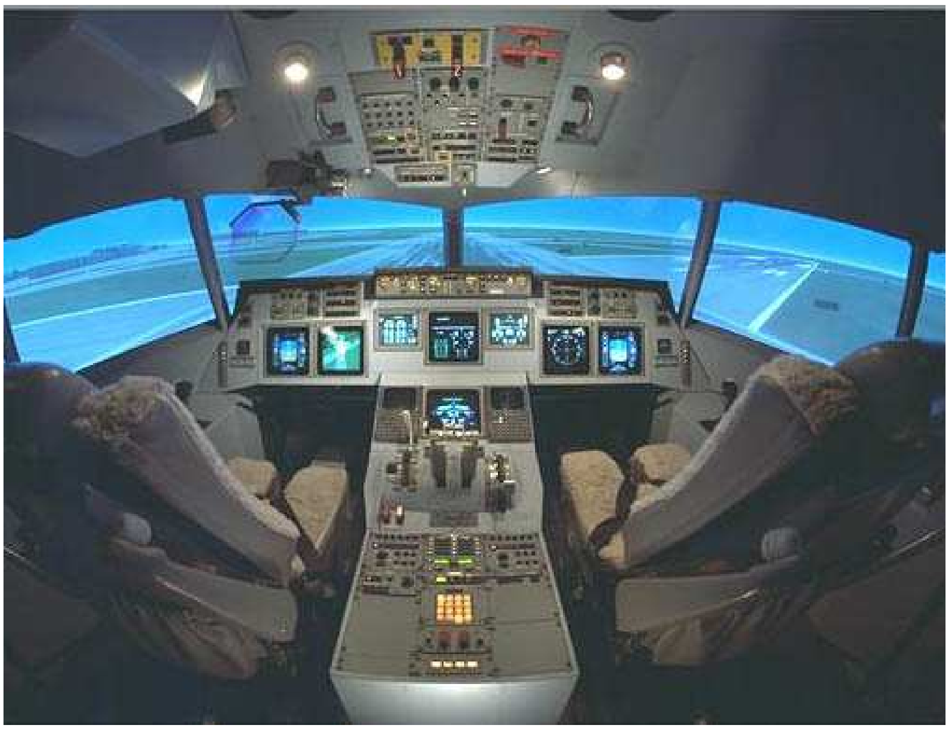


Figure 10: The real cockpit of a commercial flight simulator: an exact replica of the equivalent airplane's cockpit.

line drawing algorithms with which to illustrate their 3D models. Only one figure in the entire book did not use exclusively line drawing: Fig. 6-52, which had screen shots of a prototype polygon scan conversion system. Technology has moved on enormously since then.

1.1.6 Applications of computer graphics

Visualisation generally does not require realistic looking images. In science we are usually visualising complex three dimensional structures, such as protein molecules, which have no “realistic” visual analogue. In medicine we generally prefer an image that helps in diagnosis over one which looks beautiful. Polygon scan conversion is therefore normally used in visualisation (although some data require voxel rendering – see section 8.3.2).

Simulation uses polygon scan conversion because it can generate images at interactive speeds. At the high (and very expensive) end a great deal of computer power is used. In 1990, the most expensive flight simulators (those with full hydraulic suspension and other fancy stuff) cost about £10M, of which £1M went on the graphics kit. Similar rendering power is available today on a graphics card which costs a hundred pounds and fits in a PC. Figure 9 shows screen shots from two commercial flight simulators in the mid-1990s; Figure 10 shows the simulator’s cockpit, which is an exact physical replica of the cockpit on a real aircraft. Although the cost of the graphics has dropped dramatically, the cost of the physical kit has not.

3D games (for example Quake, Unreal, Descent) use polygon scan conversion because it gives interactive speeds. A lot of other “3D” games (for example SimCity, Civilisation, Diablo) use pre-drawn sprites (small images) which they simply copy to the appropriate position on the screen. This essentially reduces the problem to an image compositing operation, requiring much less processor time. The sprites can be hand drawn by an artist or created in a 3D modelling package and rendered to sprites in the company’s design office. Donkey Kong Country (mid-1990s), for example, was the first game to use sprites which were ray traced from 3D models.

You may have noticed that the previous sentence is the first mention of ray tracing in this section. It transpires that the principal uses of ray tracing, in the commercial world, are in producing a small quantity of super-realistic images for advertising and in producing a small proportion of the special effects for film and television. Despite what you may have expected, most special effects are done using sophisticated polygon scan conversion algorithms.

The first movie to use 3D computer graphics was *Star Wars* [1977]. Graphics were not used for the space ships , animals or sets, however. You may recall that there were some line drawn computer graphics toward the end of the movie in the targeting interfere on the X-wing fighter. All of the spaceship shots, and all of the other fancy effects, were done using models, mattes (hand-painted backdrops), and hand-painting on the actual film. Computer graphics technology has progressed incredibly since then. The 25th anniversary re-release of the Star Wars trilogy included a number of computer graphic enhancements, all of which were composited into the original movie.

A more recent example of computer graphics in a movie is the (rather bloodythirsty) *Starship Troopers* [1997]. Most of the giant insects in the movie are completely computer generated. The spaceships are a combination of computer graphic models and real models. The largest of these real models was 18’ (6m) long: so it is obviously still worthwhile spending a lot of time and energy on the real thing.

Special effects are not necessarily computer generated. Compare *King Kong* [1933] with *Godzilla* [1998]. The plots have not changed that much, but the special effects have improved enormously: changing from hand animation (and a man in a monkey suit) to swish computer generated imagery. Not every special effect you see in a modern movie is computer generated. In *Starship Troopers*, for example, the explosions are real. They were set off by a pyrotechnics expert against a dark background (probably the night sky), filmed, and later composited into the movie. In *Titanic* [1997] the scenes with actors in the water were shot in the warm Gulf of Mexico. In order that they look as if they were shot in the freezing North Atlantic, cold breaths had to be composited in later. These were filmed in a cold room over the course of one day by a special effects studio. Film makers obviously need to balance quality, ease of production, and cost. They will use whatever technology gives them the best trade off. This is increasingly computer graphics, but computer graphics is still not useful for everything by quite a long way.

In the three Lord of the Rings movies, almost anything which could be shot in live action was shot this way. Computer graphics were used only where they were easier or cheaper or the only feasible way to do something. For example, in Return of the King, the lava was originally to be produced by computer graphics simulation. When the results were found to be not realistic enough, some of the shots were re-done using real gunk flowing down a real model of a mountainside. Helms Deep, in The Two Towers, consisted of some computer graphics, a small-scale model of the whole thing, a quarter-scale model of the wall and citadel and a full-scale model of parts of the citadel for real actors to perform on. Compositing all the component of any given shot is an interesting image processing task. In a typical movie, each frame (at 24 frames per second) will have anything from twenty to over a hundred separate elements which need to be composted to make the final image.

A recent development is the completely computer generated movie. *Toy Story* [1995] was the world's first feature length computer generated movie. Two more were released in 1998 (*A Bug's Life* [1998] and *Antz* [1998]). *Toy Story 2* [1999], *Dinosaur* [2000], *Shrek* [2001], *Monsters Inc* [2001], and the graphically less sophisticated *Ice Age* [2002] have followed. *Finding Nemo* [2003] and *Shrek 2* [2004] are the most recent. More are in the pipeline. Note the subject matter of these movies (toys, bugs, dinosaurs, monsters, sea life, fairytale character). It is still very difficult to model humans realistically and much research is being undertaken in the field of realistic human modelling. *Shrek 2* represent the commercial state-of-the-art in representing humans.

1.1.7 Polygon scan conversion or ray tracing for special effect?

While ray tracing gives a better range of lighting effects than polygon scan conversion, we usually get acceptable results with polygon scan conversion through the use of techniques such as environment mapping and the use of enormous numbers of tiny polygons. The special effects industry still dithers over whether to jump in and use ray tracing. Many special effects are done using polygon scan conversion, with maybe a bit of ray tracing for special things (giving a hybrid ray tracing/polygon scan conversion algorithm).

Toy Story [1995], for example, used Pixar's proprietary polygon scan conversion algorithm. It still took between 1 and 3 hours to render each frame (these frames have a resolution of 1526 by 922 pixels) and over 800,000 CPU hours were absorbed in the making of the movie (roughly a CPU century). More expensive algorithms can be used in less time if you are rendering for television (I estimate that about one sixth of the pixels are needed compared to a movie) or if you are only rendering a small part of a big image for compositing into live action.

At SIGGRAPH 98 I had the chance to hear about the software that some real special effects companies were using. Two of these companies use ray tracing and two are pretty happy using polygon scan conversion.

BlueSky—ViFX Everything is ray traced using CGI-Studio.

Digital Domain Use ray tracing in commercial software, except when the commercial software cannot do what they want. Used MentalRay on *Fifth Element* [1997]; used Alias models (NURBS) passed to Lightwave (polygons) for one advertisement; used MentalRay plus Renderman for another advertisement.

Rhythm + Hues Use a proprietry renderer, which was about 10 years old at the time. It has been rewritten many times. They make only limited use of ray tracing.

Station X Use Lightwave plus an internally developed renderer which is a hybrid between ray tracing and z-buffer.

At Eurographics 2002 and SIGGRAPH 2002, it was apparent that little had changed over the intervening four years: the computers had got faster and artists were producing more detailed work but polygon scan conversion is still the technology of choice for almost all commercial applications of computer graphics.

1.2 Exercises

1. Compare and contrast the capabilities and uses of ray tracing and polygon scan conversion.
2. In what circumstances is line drawing more useful than either ray tracing or polygon scan conversion.
3. (a) When is realism critical? (b) Give 5 examples of applications where different levels of visual realism are necessary and explain what sort of rendering is needed for each and why.
4. "The quality of the special effects cannot compensate for a bad script." Discuss with reference to movies that you have seen.

2 The polyyon

2.1 Polygon mesh management

In order to do polygon scan conversion or line drawing we need to know how to handle polygon meshes.

2.1.1 Drawing polygons

In order to draw a polygon, you obviously need to know its vertices. To get the shading correct you also need to know its normal. The direction of the normal tells you which side is the front of the polygon and which is the back. Many systems assume one-sided polygons: the front side is shaded and the back side either is coloured matt grey or black or is not even considered. This is sensible if the polygon is part of a closed polyhedron. In many applications, all objects consist of closed polyhedra; but you cannot guarantee that this will always be the case, which means that you will get unexpected results when the back sides of polygons are actually visible on screen.

The normal vector does not need to be specified independently of the polygon's vertices because it can be calculated from the vertices. As an example: assume a polygon has three vertices, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. The normal vector can be calculated as: $\mathbf{N} = (\mathbf{C} - \mathbf{B}) \times (\mathbf{A} - \mathbf{B})$.

Any three adjacent vertices in a polygon can be used to calculate the normal vector but the *order* in which the vertices are specified is important: it changes whether the vector points up or down relative to the polygon. In a right-handed co-ordinate system the three vertices must be specified anti-clockwise round the polygon as you look down the desired normal vector (i.e. as you look at the front side of the polygon). If there are more than three vertices in the polygon, they must all lie in the same plane, otherwise the shape will not be a polygon.

Thus, for drawing purposes, we need to know only the vertices and surface properties of the polygon. The vertices naturally give us both edge and orientation information. The surface properties are such things as the specular and diffuse colours, and details of any texture mapping which may be applied to the polygon. These things are generally specified at the vertices (diffuse colour, specular colour, texture co-ordinates) for use in Gouraud or Phong shading.

2.1.2 Interaction with polygon mesh data

The above is fine for drawing but, if you wish to manipulate the polygon mesh (for example, in a 3D modelling package), then it is useful to know quite a lot more about the connectivity of the mesh. For example: if you want to move a vertex, which is shared by four polygons, you do not want to have to search through all the polygons in your data structure trying to find the ones which contain a vertex which matches your vertex data, you want some data structure which allows easy access to the relevant information.

The various versions of the *winged-edge data structure* are particularly useful for handling polygon mesh data. The version shown in Figure 11 contains explicit links for all of the relationships between vertices, edges and polygons, thus making it easy to find, for example, which polygons are attached to a given vertex, or which polygons are adjacent to a given polygon (by traversing the edge list for the given polygon, and finding which polygon lies on the other side of each edge).

The vertex object contains the vertex's co-ordinates, a pointer to a list of all edges of which this vertex is an end-point, and a pointer to a list of all polygons of which the vertex is a vertex. It also has a pointer to the vertex's surface properties (such as colour and texture coordinates).

The polygon object contains (a pointer to) the polygon's surface property information (such as its texture map), a pointer to a list of all edges which bound this polygon, and a pointer to an ordered list of all vertices of the polygon.

The edge object contains pointers to its start and end vertices, and pointers to the polygons which lie to the left and right of it.

Figure 11 shows just one possible implementation of a polygon mesh data structure. **FvDFH** section 12.5.2 describes another winged-edge data structure which contains slightly less information, and therefore requires more accesses than the one shown here to find certain pieces of information. The implementation that would be chosen depends on the needs of the particular application which is using the data structure. The trade-off is between ease of extracting information and ease of updating the data structure.

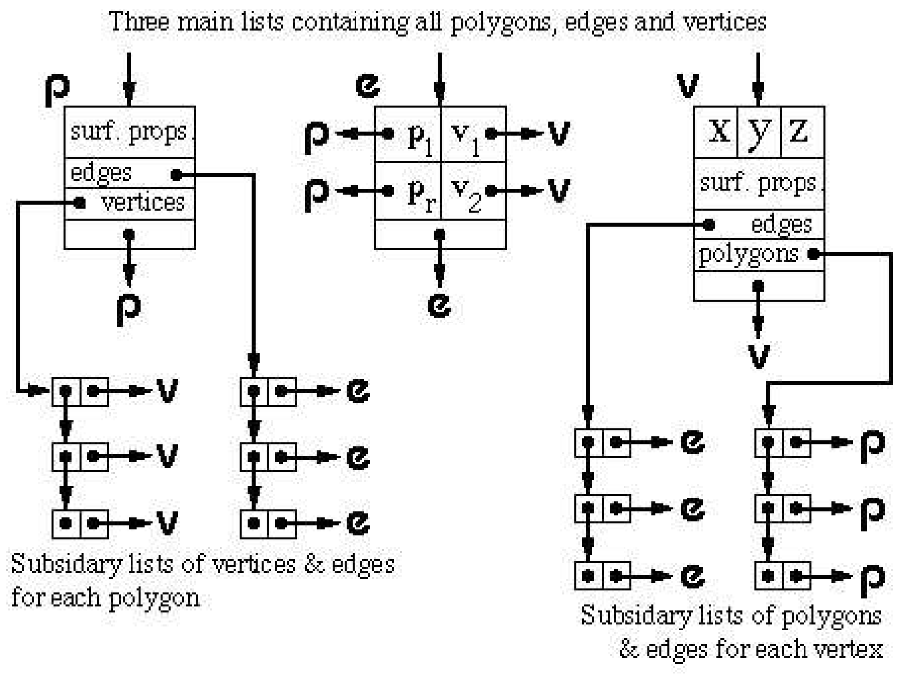

Three main lists containing all polygons, edges and vertices
p
e
v
surf. props.
edges
vertices
p
p
v
x y z
surf. props.
edges
polygons
Subsidary lists of vertices & edges for each polygon
Subsidary lists of polygons & edges for each vertex

Figure 11: One version of the winged edge data structure.

F&vD section 13.2 also contains a bit of information on polygon meshes.

In general, we will want a polygon mesh to form a manifold surface. This is where the surface is what a human would naturally think of as a surface, without any three-way joins or other peculiar features; a surface which you could flatten onto a plane given sufficiently many cuts and a bit of stretching here and there. Mathematically, a manifold surface is where the neighbourhood of every point is topologically equivalent to a disc (except at the edges of the manifold, where it is topologically equivalent to a half disc). The principal upshot of this is that each edge in the polygon mesh can be the edge of either one or two polygons, no more and no less.

2.2 Hardware polygon scan conversion quirks

A piece of polygon scan conversion hardware, such as the Silicon Graphics Reality Engine or the nVIDIA GeForce family of graphics cards, will generally consist of a *geometry* engine and a *rendering* engine. The geometry engine will handle the transformations of all vertices and normals, and some of the shading calculations. The rendering engine will implement the polygon scan conversion algorithm on the transformed data. Modern graphics cards allow for user programming in both the geometry and rendering engine. Machine instructions are provided for the usual operations (addition, multiplication), and also for such necessary things as taking the dot product of two vectors. The geometry and rendering engines both have multiple copies of the same hardware to allow for multiple vertices and polygons to be processed in parallel. These are generally built with a SIMD (single instruction, multiple data) parallel processor architecture. The architecture is optimised for processing graphics, so the user is somewhat limited is what he or she can program. However, the most recent graphics cards allow for a good deal of flexibility. Whereas the previous generation of cards allowed a limited number of instructions (a maximum of 128–256 instructions in the whole program, no more than twelve working registers, no jumps or loops, no access to general memory), the latest cards allow up to 64k instructions, with jumps and loops. The introduction of jumps and loops causes interesting issues with the SIMD architecture, requiring different pipes to be able to chose whether or not to execute any given instruction.

To give you an idea of the complexity which is possible, on a recent generation of nVIDIA cards, the information which is passed to the geometry engine, for a single vertex, is position, weight, normal, primary and secondary colour, fog coordinate, and eight texture coordinates; all sixteen of these are floating-point four-component vectors. The output from the geometry engine is homogeneous clip space position, primary and secondary colours for front and back faces of the polygon, fog coordinate, point size, and texture coordinate set; where they are all again floating-point four-component vectors except for the output fog coordinate and the point size[1]. Both geometry and rendering engines have read access to the texture buffers. Graphics cards are now so powerful that they are being used as general purpose co-processors for a variety of mathematics-intense computation tasks, using texture buffers for storing intermediate results.

2.2.1 Triangles only

When making a piece of hardware to render a polygon, it is much easier to make the hardware handle a fixed number of vertices per polygon, than a variable number. Most hardware implementations thus implement only triangle drawing. This is not a serious drawback. Polygons with more vertices are simply split into triangles.

2.2.2 The triangle strip set and triangle fan set

In addition to simple triangle drawing, rendering hardware may also implement either or both of the triangle strip set and triangle fan set to speed up processing through the geometry engine. Each triangle in the set has two vertices in common with the previous triangle. Each vertex is transformed only once by the geometry engine, giving a factor of three speed up in geometry processing.

For example, assume we have triangles **ABC**, **BCD**, **CDE** and **DEF**. In naïve triangle rendering, the vertices would be sent to the geometry engine in the order **ABC BCD CDE DEF**; each triangle's vertices being sent separately. With a triangle strip set the vertices are sent as **ABCDEF**; the adjacent triangles' vertices overlapping.

A triangle fan set is similar. In the four triangle case we would have triangles **ABC**, **ACD**, **ADE** and **AEF**. The vertices would again be sent just as **ABCDEF**. It is obviously important that the rastering engine be told whether it is drawing standard triangles or a triangle strip set or a triangle fan set.

[1]You are not expected to remember all of these input and output registers, but they give you an idea of the complexity of the processing which can go on inside a graphics card.

2.2.3 The vertex cache

The triangle strip and fan sets work because there is a vertex cache which can hold all the relevant data about two vertices. More recent graphics cards have a vertex cache which can hold the twenty most recently used vertices, hence obviating the need to be explicitly specify fan sets and strip sets, although you still need to send the triangles to the card in some reasonably coherent order and you do need to let the graphics card know that the triangles form a set with the same surface properties. You also need to index the vertices so that you refer to each by its index rather than by sending the (x, y, z) coordinates again.

2.3 Exercises

1. Calculate both surface normal vectors (left-handed and right-handed) for a triangle with points (1, 1, 0), (2, 0, 1), (-1, -2, -1).
2. Confirm that the following statements provide a definition of a polygon mesh which represents a manifold surface:
   (a) A vertex belongs to at least two edges.
   (b) A vertex is a vertex of at least one polygon.
   (c) An edge has exactly two end points.
   (d) An edge is an edge of either one or two polygons.
   (e) A polygon has at least three vertices.
   (f) A polygon has at least three edges.
3. Work out the algorithm that is required to modify a winged-edge data structure when an edge is split. You may ignore surface property information for the polygons and you may assume that the edge that is split is split exactly in half. The algorithm could by called by the function call:

   ```
   split_edge( vertex_list v, edge_list e,
               polygon_list p, edge edge_to_split )
   ```

   where the winged-edge data structure is made up of the three linked lists of objects (vertices, edges, and polygons).
4. [2002/7/9] Describe the situations in which it is sensible to use a winged-edged data structure to represent a polygon mesh and, conversely, the situations in which a winged-edged data structure is not a sensible option for representing a polygon mesh. What is the minimum information which is required to successfully draw a polygon mesh using Gouraud shading? [4 marks]

3 Ray tracing primitives

A *primitive* is a shape for which a ray-shape intersection routine has been written. More complex objects can be built out of the primitives. Most ray tracers will have a variety of primitives. They are limited only by the ability of the programmer to write a function to analytically intersect a ray with the surface of the shape. The basic raytracing algorithm is given in Figure 12. Figure 14 shows some of the common primitives.

For practical experience, download and play with either *POVray* or *Rayshade*.

3.1 Mathematical preliminaries

3.1.1 Vector arithmetic

It is helpful to remember your vector arithmetic. A 3D vector is represented thus:

$$\mathbf{V} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1}$$

For ease of writing such definitions in text we may say $\mathbf{V} = (x, y, z)$, where we understand that this ordered triple is equivalent to the vector.
The magnitude of vector $\mathbf{V}$ is:

$$|\mathbf{V}| = \sqrt{x^2 + y^2 + z^2} \tag{2}$$

The dot product of two vectors ($\mathbf{A} = (x_A, y_A, z_A)$ and $\mathbf{B} = (x_B, y_B, z_B)$) is:

$$\mathbf{A} \cdot \mathbf{B} = x_A x_B + y_A y_B + z_A z_B \tag{3}$$

which could also be written as a matrix multiplication:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{A}^T\mathbf{B} = [x_A y_A z_A] \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} \tag{4}$$

The dot product is a scalar value equal to:

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| \cos\theta \tag{5}$$

where θ is the angle between the two vectors. Note that this means that:

$$\mathbf{V} \cdot \mathbf{V} = |\mathbf{V}|^2 \tag{6}$$

and hence:

$$|\mathbf{V}| = \sqrt{\mathbf{V} \cdot \mathbf{V}} \tag{7}$$

Also note that the dot product between two mutually perpendicular vectors is always zero.
The cross product (vector product) of these two vectors is:

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} y_A z_B - z_A y_B \\ z_A x_B - x_A z_B \\ x_A y_B - y_A x_B \end{bmatrix} \tag{8}$$

The cross product is a vector which is perpendicular to both $\mathbf{A}$ and $\mathbf{B}$. This is a very handy way to make a new vector which is guaranteed perpendicular to a given vector. It also means that:

$$\mathbf{A} \cdot (\mathbf{A} \times \mathbf{B}) = 0 \tag{9}$$

$$\mathbf{B} \cdot (\mathbf{A} \times \mathbf{B}) = 0 \tag{10}$$

though this may be getting a bit esoteric and so let's move on.

```
Select eye point ($E$), look point ($L$), and up vector
Set screen plane to be centred on $L$, perpendicular to vector $EL$
Set screen size and number of pixel
For each pixel {
    Define $D$ to be the unit vector pointing from $E$ to the centre of the pixel
    Raytrace($E$, $D$)
}

function Raytrace($E$, $D$) returns Colour {
    nearest_t = $\infty$
    nearest_object = NULL
    for each object {
        find $t$, the smallest, non-negative real solution of
            the ray/object intersection equation
        if $t$ exists {
            if $t <$ nearest_t { nearest_t = $t$ ; nearest_object = current object }
        }
    }
    colour = black
    if nearest_object exists {
        find normal vector, $N$, at intersection point
        if object is reflective {
            reflected_colour=Raytrace(intersection point, reflection vector)
            colour += reflection_coeff * reflected_colour
        }
        if object is refractive {
            refracted_colour = Raytrace(intersection point, refracted vector)
            colour += refraction_coeff * refracted_colour
        }
        for each light {
            if shadow_ray(intersection point, light position) returns No_Shadow {
                calculate light's colour contribution by doing the illumination calculations
                    using $D$, $N$, the current light, and the object properties
                colour += lights colour contribution
            }
        }
    }
    return colour
}
```

Figure 12: A simplistic pseudocode version of the basic ray tracing algorithm. The function *shadow_ray* can be found in Figure 13.

```
function shadow_ray(point1, point2) returns Shadow or No_Shadow {
    ray defined with $E$=point1, $D$=point2-point1
    nearest_t = $\infty$
    nearest_object = NULL
    for each object {
        find $t$, the smallest, non-negative real solution of
            the ray/object intersection equation
        if $t$ exists {
            if $t <$ nearest_t { nearest_t = $t$ }
        }
    }
    if nearest_t $< 1$ return Shadow
    else return No_Shadow
}
```

Figure 13: A simplistic pseudocode version of the function *shadow_ray* used by the ray tracing algorithm in Figure 12.

3.1.2 Points and displacements

Both points and displacements are three-tuples of real numbers. However, they are different beasts and must be treated differently. They point is an absolute position in space relative to some fixed origin, $\mathbf{O} = (0, 0, 0)$. A displacement is an offset, specifying a distance and direction but *not* an absolute position.

Only certain arithmetic operations are permitted on points and displacements. Let $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \dots$ represent points and $\mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3, \dots$ represent displacements. Displacements can be added,

$$\mathbf{D}_3 = \mathbf{D}_1 + \mathbf{D}_2, \tag{11}$$

but points cannot be added. The difference of two displacements is a displacement,

$$\mathbf{D}_3 = \mathbf{D}_1 - \mathbf{D}_2, \tag{12}$$

but the difference of two points is also a displacement:

$$\mathbf{D}_1 = \mathbf{P}_1 - \mathbf{P}_2. \tag{13}$$

You can add a displacement to a displacement,

$$\mathbf{D}_3 = \mathbf{D}_1 + \mathbf{D}_2, \tag{14}$$

or a displacement to a point,

$$\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{D}_1, \tag{15}$$

but *not* a point to a point. It is possible to take a weighted average of points,

$$\mathbf{P}_{\text{new}} = \alpha_1 \mathbf{P}_1 + \alpha_2 \mathbf{P}_2 + \cdots + \alpha_k \mathbf{P}_k, \ \sum_{i=1}^{k} \alpha_i = 1 \tag{16}$$

provided that the weights sum to one, otherwise the operation makes no sense.

If you transform your coordinate system, then all points scale, rotate, and translate as you would expect. Displacements also scale and rotate but they do not translate. To see why this is the case, consider $\mathbf{D}_1 = \mathbf{P}_1 - \mathbf{P}_2$. Let the two points be translated: $\mathbf{P}'_1 = \mathbf{P}_1 + \mathbf{D}_o$ and $\mathbf{P}'_2 = \mathbf{P}_2 + \mathbf{D}_o$. Then:

$$\begin{aligned}
\mathbf{D}'_1 &= \mathbf{P}'_1 - \mathbf{P}'_2 && (17)\\
&= (\mathbf{P}_1 + \mathbf{D}_o) - (\mathbf{P}_2 + \mathbf{D}_o) && (18)\\
&= (\mathbf{P}_1 - \mathbf{P}_2) + (\mathbf{D}_o - \mathbf{D}_o) && (19)\\
&= (\mathbf{P}_1 - \mathbf{P}_2) && (20)\\
&= \mathbf{D}_1 && (21)
\end{aligned}$$

3.2 Equation of a ray

A ray is defined by an origin or eye point, $\mathbf{E} = (x_E, y_E, z_E)$, and an offset displacement, $\mathbf{D} = (x_D, y_D, z_D)$. The equation for the ray is:

$$\mathbf{P}(t) = \mathbf{E} + t\mathbf{D}, t \geq 0 \quad (22)$$

This is obviously equivalent to the three equations:

$$\left.\begin{array}{l} x(t) = x_E + tx_D \\ y(t) = y_E + ty_D \\ z(t) = z_E + tz_D \end{array}\right\} t \geq 0 \quad (23)$$

When finding a ray-object intersection point, we are looking for the intersection point with the *lowest non-negative* value of t.

3.3 Equations for the primitives

3.3.1 Sphere

The sphere is the simplest finite object with which to intersect a ray. Practically any ray tracing program will include the sphere as a primitive. Scaling a sphere by different amounts along the different axes will produce an ellipsoid: a squashed or stretched sphere. There is thus no need to include the ellipsoid as a primitive provided that your ray tracer contains the usual complement of transformations (it would be a poor ray tracer if it did not). There is, however, some subtlety in how normals are transformed in such an anisotropically scaled object, which we discuss below in section 3.4.2.

The unit sphere, centred at the origin, has the implicit equation:

$$x^2 + y^2 + z^2 = 1 \quad (24)$$

In vector arithmetic, this becomes:

$$\mathbf{P} \cdot \mathbf{P} = 1 \quad (25)$$

To find the intersection between this sphere and an arbitrary ray, substitute the ray equation (Equation 23) in the sphere equation (Equation 24):

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 + (z_E + tz_D)^2 = 1 \quad (26)$$

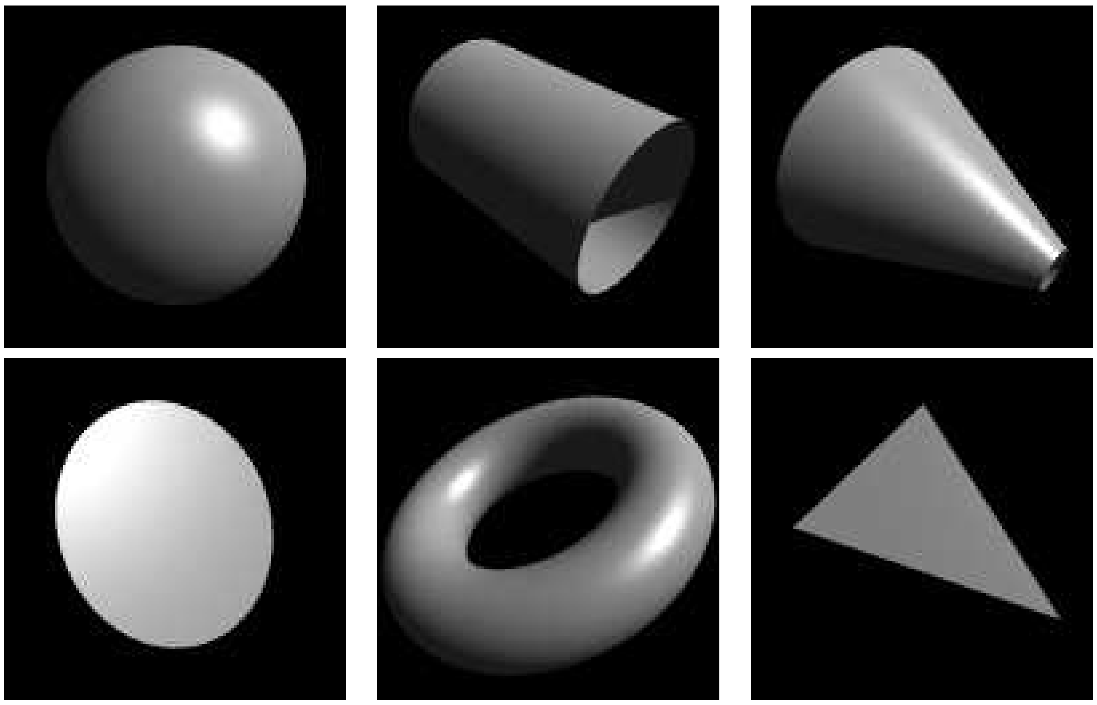


Figure 14: Some ray traced primitives: sphere, cylinder, cone, disc, torus, polygon

$$
\begin{aligned}
\Rightarrow\quad & t^2(x_D^2 + y_D^2 + z_D^2) + t(2x_Ex_D + 2y_Ey_D + 2z_Ez_D) \\
& +(x_E^2 + y_E^2 + z_E^2 - 1) = 0 && (27) \\
\Rightarrow\quad & at^2 + bt + c = 0 && (28) \\
\Rightarrow\quad & t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} && (29)
\end{aligned}
$$

where $a = x_D^2 + y_D^2 + z_D^2$, $b = 2x_Ex_D + 2y_Ey_D + 2z_Ez_D$, and $c = x_E^2 + y_E^2 + z_E^2 - 1$. This gives zero, one, or two real values for t. If there are zero real values then there is no intersection between the ray and the sphere. If there are either one or two real values then chose the *smallest, non-negative* value of t. This gives the intersection point (at $P(t)$). If there are only negative values of t, then the line (of which the ray is a part) *does* intersect the sphere, but the intersection point is not on the part of the line which constitutes the ray. In this case there is again no intersection point between the ray and the sphere.

An alternative formulation is to use the vector versions of the equations (Equations 22 and 25):

$$
\begin{aligned}
& (\mathbf{E} + t\mathbf{D}) \cdot (\mathbf{E} + t\mathbf{D}) = 1 && (30) \\
\Rightarrow\quad & t^2(\mathbf{D} \cdot \mathbf{D}) + t(2\mathbf{E} \cdot \mathbf{D}) + (\mathbf{E} \cdot \mathbf{E} - 1) = 0 && (31) \\
\Rightarrow\quad & at^2 + bt + c = 0 && (32) \\
\Rightarrow\quad & t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} && (33)
\end{aligned}
$$

Where $a = \mathbf{D} \cdot \mathbf{D}$, $b = 2\mathbf{E} \cdot \mathbf{D}$, and $c = \mathbf{E} \cdot \mathbf{E} - 1$. In other words, exactly the same result, expressed in a more compact way. *Graphics Gems I* (p. 388) describes yet another way of arriving at the same result.

You will remember, from Part IB, that we need to know the normal vector (which is, of course, a displacement) at the intersection point in order to calculate the illumination and/or find the reflection ray. For a sphere centred at the origin, the normal is a vector passing through the origin and the point of intersection.

$$\mathbf{N} = (x, y, z) - (0, 0, 0) \tag{34}$$

$$= (x, y, z) \tag{35}$$

Remember that points and displacements are different beasts. Equation 34 is the subtraction of one point from another while Equation 35 is a displacement. So (x, y, z) in Equation 34 is a point while (x, y, z) in Equation 35 is a displacement.

To simplify the ray tracing algorithm, we may choose to assume that the normal vector points out of the surface on the same side of the surface as the eye point. In this case, we need to be careful about whether the eye point lies inside or outside of the sphere. If the origin of the ray lies inside the sphere then the normal at the intersection point is the negative of that in Equation 35, that is, for the case of **E** inside the sphere,

$$\mathbf{N} = (-x, -y, -z). \tag{36}$$

To check whether **E** is inside the unit sphere we simply need to check whether $|\mathbf{E}| < 1$.

3.3.2 Cylinder

Intersecting a ray with an infinitely long cylinder is practically as easy as intersecting one with a sphere. The tricky bit, if it can be called that, is to intersect a ray with a more useful finite length cylinder. This is achieved by intersecting the ray with the appropriate infinitely long cylinder and then ascertaining where along the cylinder the intersection lies. If it lies in the finite length in which you are interested then keep the intersection. If it does not then ignore the intersection. Note that the ray tracer used to render the cylinder in Figure 14 has cylinders without end caps. This is the correct result if you follow the procedure outlined above. Adding end caps to your cylinders requires extra calculations. We consider both cases in more detail below.

The *infinite* unit cylinder aligned along the z-axis is defined as:

$$x^2 + y^2 = 1 \tag{37}$$

To intersect a ray with this, substitute Equation 23 in Equation 37.

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 = 1 \tag{38}$$

$$\Rightarrow \quad t^2(x_D^2 + y_D^2) + t(2x_E x_D + 2y_E y_D) + (x_E^2 + y_E^2 - 1) = 0 \tag{39}$$

$$\Rightarrow \quad at^2 + bt + c = 0 \tag{40}$$

$$\Rightarrow \quad t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{41}$$

where $a = x_D^2 + y_D^2$, $b = 2x_E x_D + 2y_E y_D$, and $c = x_E^2 + y_E^2 - 1$.

The *finite* open-ended unit cylinder aligned along the z-axis is defined as:

$$x^2 + y^2 = 1, z_{\min} < z < z_{\max} \tag{42}$$

The only difference between this and Equation 37 being the restriction on z. To handle this finite length cylinder, solve Equation 41 above. This gives, at most, two values of t. Call these t_1 and t_2. Calculate z_1 and z_2 using Equation 23 ($z_1 = z_E + t_1 z_D$ and $z_2 = z_E + t_2 z_D$) and then check $z_{\min} < z_1 < z_{\max}$ and $z_{\min} < z_2 < z_{\max}$. Whichever intersection point passes this test and, if both pass the test, has the smallest non-negative value of t, is the closest intersection point of the ray with the open-ended finite cylinder.

If we wish the finite length cylinder to be closed we must formulate an intersection calculation between the ray and the cylinder's end caps. The end caps have the formulae:

$$z = z_{\min}, \qquad x^2 + y^2 \leq 1 \tag{43}$$
$$z = z_{\max}, \qquad x^2 + y^2 \leq 1 \tag{44}$$

and we could find explicit intersections between the ray and these two discs. However, for the cylinder, there is a more efficient trick: once you have calculated the solutions to Equation 41 you will either know that there are no intersections with the infinite cylinder or you will know that there are one or two real intersection points represented by t_1 and t_2, which may be negative at this point in the algorithm. The previous paragraph explained how to ascertain whether these correspond to points on the finite length open-ended cylinder. Now, if z_1 and z_2 lie either side of $z_{\min}$ we know that the ray intersects the $z_{\min}$ end cap, and can calculate the intersection point as:

$$t_3 = \frac{z_{\min} - z_E}{z_D} \tag{45}$$

A similar equation holds for the $z_{\max}$ end cap. Note that the ray may intersect both end caps, for example when $z_1 < z_{\min}$ and $z_2 > z_{\max}$. Also, note that a ray parallel to the z-axis is a special case and needs to be handled separately. It is left as an exercise how to check for and how to handle this case (see exercises below).

The normal vector of this cylinder, at intersection point (x, y, z) on the surface of the infinite cylinder, will be $(x, y, 0)$ or $(-x, -y, 0)$ depending on whether the ray hits the inside or outside surface first. For a finite cylinder, if the intersection point is on an end cap, then the normal vector will be $(0, 0, -1)$ or $(0, 0, 1)$ depending on which end cap is hit and whether the origin of the ray is inside or outside the finite cylinder. The details of how to ascertain whether the ray's origin, E, is inside or outside of the cylinder in both the finite and infinite cases are left as an exercise.

3.3.3 Cone

Cones are very like cylinders, mathematically. Like the infinite cylinder, there is a simple mathematical definition of an infinite cone which makes it easy to write a ray-cone intersection algorithm. Note that a cone does not need to have a point – it can be truncated short of its 'top', as illustrated in Figure 14. The particular ray tracer used does not add end caps to cones.

The *infinite* double cone[2] aligned along the z-axis and having unit slope is defined as:

$$x^2 + y^2 = z^2 \tag{46}$$

To intersect a ray with this, substitute Equation 23 in Equation 46.

$$\begin{aligned}
& (x_E + tx_D)^2 + (y_E + ty_D)^2 = (z_E + tz_D)^2 && (47)\\
\Rightarrow\quad & t^2(x_D^2 + y_D^2 - z_D^2) + t(2x_Ex_D + 2y_Ey_D - 2z_Ez_D) && \\
& +(x_E^2 + y_E^2 - z_E^2) = 0 && (48)\\
\Rightarrow\quad & at^2 + bt + c = 0 && (49)\\
\Rightarrow\quad & t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} && (50)
\end{aligned}$$

where $a = x_D^2 + y_D^2 - z_D^2$, $b = 2x_Ex_D + 2y_Ey_D - 2z_Ez_D$, and $c = x_E^2 + y_E^2 - z_E^2$.

The *finite* open-ended cone aligned along the z-axis is defined as:

$$x^2 + y^2 = z^2, z_{\text{min}} < z < z_{\text{max}} \tag{51}$$

The only difference between this and Equation 46 being the restriction on z. Note that if z_{min} and z_{max} are both positive or both negative then you get a single cone with its top truncated. If either z_{min} or z_{max} is zero you get a single cone with its apex at the origin.

To handle this finite length cone you proceed as for the finite length cylinder, with several subtle modifications. In particular, it transpires that it is simpler to calculate the intersection of the ray with the two end-cap discs rather than use the trick we used for cylinders. This is because of the increased number of special cases which would need to be considered for cones. It is left as an exercise to work out how to handle this. It would be instructive to try to do this both ways: first by using explicit ray-disc intersection calculation and second by modifying the trick we used for cylinders, taking all special cases into account.

3.3.4 Torus

Toroids are reasonably rare in real life (doughnuts and tyre inner tubes notwithstanding). They are somehow alluring to the kinds of people who implement ray tracers and, having a reasonably straightforward mathematical definition, are reasonably simple to implement. They thus appear as primitives in many ray tracers. They become more useful when combined with Constructive Solid Geometry (see section 8.2).

A torus is defined by two parameters: the radius of the torus (that is the radius of the torus's defining circle, measured from the origin) and the radius of the tube (the perpendicular distance from the defining circle to the surface of the torus). These are R and r respectively. Normally $R > r$.

An implicit definition of the torus is:

$$\left(\sqrt{x^2 + y^2} - R\right)^2 + z^2 = r^2 \tag{52}$$

[2]*double cone* means that it is two "standard" cones joined at their apices.

The torus can also be defined parametrically in terms of two angles, θ and ϕ, where θ can be thought of as the angle around the defining circle and ϕ the angle around the inside of the tube:

$$\begin{aligned} x &= (R + r\cos\phi)\cos\theta & (53)\\ y &= (R + r\cos\phi)\sin\theta & (54)\\ z &= r\sin\phi & (55) \end{aligned}$$

Substituting these three equations into Equation 52 will show that they are correct and is a useful exercise in algebraic manipulation.

To find the intersection points of a ray with a torus you need to substitute Equation 23 in Equation 52. Equation 56 is that substitution with the $(\sqrt{x^2+y^2} - R)^2$ term expanded, the resulting square root term placed on one side of the equals sign, and all other terms placed on the other side:

$$\begin{aligned} & 2R\sqrt{x_E^2 + 2tx_Ex_D + t^2x_D^2 + y_E^2 + 2ty_Ey_D + t^2y_D^2} \\ = \; & R^2 + x_E^2 + 2tx_Ex_D + t^2x_D^2 + y_E^2 + 2ty_Ey_D + t^2y_D^2 + z_E^2 + 2tz_Ez_D + t^2z_D^2 - r^2 \quad (56) \end{aligned}$$

If we now square both sides we will get a quartic equation in t. This can be solved using a standard quartic root finder to find the four roots of the equation[3] (there are up to four intersection points between a torus and an arbitrary ray). A quartic root finder is described in *Graphics Gems V* (p. 3).

Finding the normal to this torus requires finding the intersection point on the surface, $(x, y, z) = ((R + r\cos\phi)\cos\theta), (R + r\cos\phi)\sin\theta, r\sin\phi)$, and the nearest point on the ring through the middle of the torus, $(R\cos\theta, R\sin\theta, 0)$. Subtracting one from the other gives the normal as $\mathbf{N} = (r\cos\phi\cos\theta, r\cos\phi\sin\theta, r\sin\phi)$.

3.3.5 Plane

The infinite plane is a simple object with which to intersect a ray. On its own it can represent boundary objects such as the ground or the sky or perhaps an infinite wall. Intersection with the infinite plane is a useful building block in a ray tracing system as it forms the initial step in polygon and disc calculations. It is also useful in Computational Solid Geometry (section 8.2).

A plane can be defined by a normal vector, $\mathbf{N}$ and a point on the plane, $\mathbf{Q}$. A point, $\mathbf{P}$, is on the plane if:

$$\mathbf{N} \cdot (\mathbf{P} - \mathbf{Q}) = 0 \qquad (57)$$

To find the ray/plane intersection substitute Equation 22 in Equation 57:

$$\begin{aligned} & \mathbf{N} \cdot (\mathbf{E} + t\mathbf{D} - \mathbf{Q}) = 0 & (58)\\ \Rightarrow \quad & t = \frac{\mathbf{N} \cdot (\mathbf{Q} - \mathbf{E})}{\mathbf{N} \cdot \mathbf{D}} & (59) \end{aligned}$$

[3]The equation itself is rather fierce and you would not be expected to do the full expansion in an exam. For those who are interested, it looks like this:
$t^4(x_D^4 + y_D^4 + z_D^4 + 2x_D^2y_D^2 + 2x_D^2z_D^2 + 2y_D^2z_D^2) + t^3(4x_D^3x_E + 4y_D^3y_E + 4z_D^3z_E + 4x_D^2y_Dy_E + 4x_D^2z_Dz_E + 4x_Dx_Ey_D^2 + 4y_D^2z_Dz_E + 4x_Dx_Ez_D^2 + 4y_Dy_Ez_D^2) + t^2(-2R^2x_D^2 - 2R^2y_D^2 + 2R^2z_D^2 - 2r^2x_D^2 - 2r^2y_D^2 - 2r^2z_D^2 + 6x_D^2x_E^2 + 2x_E^2y_D^2 + 8x_Dx_Ey_Dy_E + 2x_D^2y_E^2 + 6y_D^2y_E^2 + 2x_E^2z_D^2 + 2y_E^2z_D^2 + 8x_Dx_Ez_Dz_E + 8y_Dy_Ez_Dz_E + 2x_D^2z_E^2 + 2y_D^2z_E^2 + 6z_D^2z_E^2) + t(-4R^2x_Dx_E - 4R^2y_Dy_E + 4R^2z_Dz_E - 4r^2x_Dx_E - 4r^2y_Dy_E - 4r^2z_Dz_E + 4x_Dx_E^3 + 4x_E^2y_Dy_E + 4x_Dx_Ey_E^2 + 4y_Dy_E^3 + 4x_E^2z_Dz_E + 4y_E^2z_Dz_E + 4x_Dx_Ez_E^2 + 4y_Dy_Ez_E^2 + 4z_Dz_E^3) + (R^4 - 2R^2x_E^2 - 2R^2y_E^2 + 2R^2z_E^2 + r^4 - 2r^2R^2 - 2r^2x_E^2 - 2r^2y_E^2 - 2r^2z_E^2 + x_E^4 + y_E^4 + z_E^4 + 2x_E^2y_E^2 + 2x_E^2z_E^2 + 2y_E^2z_E^2) = 0$

If $\mathbf{N} \cdot \mathbf{D} = 0$ then the ray is parallel to the plane, and there is no intersection point. If $t < 0$ then the plane is behind the eye point and there is no intersection. If $t \geq 0$ then the intersection point is $\mathbf{E} + t\mathbf{D}$. The normal vector is obviously either $\mathbf{N}$ or $-\mathbf{N}$ depending on whether $\mathbf{E}$ is on the side of the plane pointed at by $\mathbf{N}$ or not.

3.3.6 Polygon

Having the polygon as a ray tracing primitive allows a ray tracer to render anything that a polygon scan conversion algorithm could. To find the intersection of a ray with a polygon, first find the intersection of the ray with the infinite plane in which the polygon lies. Then ascertain whether the intersection lies inside or outside the polygon: this is a reasonably straightforward two dimensional graphics operation.

A polygon can be defined by an ordered set of vertices: $(\mathbf{P_1}, \mathbf{P_2}, \mathbf{P_3}, \ldots)$. To find the intersection point between a polygon and a ray, we first find the intersection point between the polygon's plane and the ray, and then ascertain whether this intersection point lies inside the polygon or not.

The normal of the polygon's plane can be found by the simple cross product: $\mathbf{N} = (\mathbf{P_3} - \mathbf{P_2}) \times (\mathbf{P_1} - \mathbf{P_2})$. A point on the plane's surface is even easier to find: $\mathbf{Q} = \mathbf{P_1}$. The intersection calculation then proceeds as above, for the polygon.

If an intersection point between the ray and the plane is found then we can check whether or not the point lies inside the polygon in the following manner. First we project the intersection point and the polygon to two dimensions by simply throwing away one coordinate. The most obvious thing to do is throw away the coordinate in which the polygon has the smallest extent. We then test to see if the intersection point lies inside the two dimensional polygon using the odd/even test.

The odd/even test checks to see whether or not an arbitrary point lies inside an arbitrary polygon in two dimensions. This is done by drawing an arbitrary ray from the point to infinity. If the ray crosses an even number of polygon edges then the point lies outside the polygon. Contrariwise, if the ray crosses an odd number of polygon edges then the point lies inside the polygon. The easiest implementation of this simply runs the arbitrary ray parallel to one of the axes. A full discussion of the implementation details of this, and other point-in-polygon algorithms, can be found in *Graphics Gems IV* pp. 24–46.

3.3.7 Disc

The disc is essential in ray tracers which implement cones and cylinders without end caps (such as *Rayshade* which was used to render the images in Figure 14). In Figure 14, the disc has been positioned so that it just catches the light – this illustrates how specular reflection varies across a completely flat surface. For comparison, the sphere, cone and torus images illustrate how specular reflection varies across three curved surfaces.

The disc is similar to the polygon. Both are planar objects. A disc can be defined by its centre, $\mathbf{Q}$, its radius, r, and a normal vector, $\mathbf{N}$. Finding the intersection point, $\mathbf{P}$, of the ray with the disc's plane proceeds as for a plane/ray or polygon/ray intersection. The routine to check if the intersection with the plane lies inside or outside of the disc is simpler than the equivalent routine for the polygon. Discovering whether $\mathbf{P}$ lies inside the disc requires you to check only that:

$$(\mathbf{P}-\mathbf{Q})\cdot(\mathbf{P}-\mathbf{Q}) \leq r^2 \tag{60}$$

3.3.8 Box

A box is essentially six polygons and could be ray traced as such. However, intersection with an axis-aligned box can be optimised. Any box can be axis-aligned by appropriate transformations. We can thus write a routine to intersect an arbitrary ray with an axis-aligned box and then transform the ray under consideration in exactly the same way as we transform the box which we are trying to intersect with it. This sort of idea generalises neatly to the concept of specifying any object in a convenient *object coordinate system* and then applying transforms to the whole object to place it at the appropriate point in the *world coordinate system*.

3.4 Intersections with arbitrarily positioned primitives

Of the above primitives, only the plane and polygon are arbitrarily defined[4]. The sphere, cylinder, cone, and torus are all defined as being centred at the origin, and all have other restrictions on their definition[5]. In order to ray trace one of these primitives in an arbitrary location we have two alternatives: (1) find general intersection algorithms between a ray and the arbitrarily located versions of the primitives; or (2) use geometric transforms to scale, rotate, and translate these primitives into the desired locations. This second option will be followed here.

3.4.1 Transforming the primitives

The basic idea is simple. We specify a scaling, a rotation, and a translation which, between them, transform the primitive from its standard position to the desired location. You will remember that this was covered in the IB *Computer Graphics & Image Processing* course. To perform the intersection we take the inverse transform of the ray, intersect this with the primitive in its standard position, and then transform the resulting intersection point to its correct location.

Remember that there is a difference between points and displacements. When transforming objects, displacements are scaled and rotated but *not* translated. Think of it this way: if you translate two points, the displacement between them stays exactly the same. But if you scale or rotate the two points, the displacement between them scales or rotates accordingly.

All this is by way of explaining how we transform a ray in order to intersect the appropriate ray with the primitive in its standard position. Let us assume that the primitive object in its standard position, $\hat{\mathbf{B}}$, undergoes transformation[6] $\mathbf{TRS}$ to get into

[4]The disc is also arbitrarily defined if you want a circular disc, but it could be transformed to provide an ellipse.

[5]The sphere has unit radius; the cylinder has unit radius and is aligned with the z-axis; the cone is similarly aligned and has unit slope; the torus is aligned with the z-axis.

[6]Where $\mathbf{T}$ is a translation, $\mathbf{R}$ a rotation, and $\mathbf{S}$ a scaling.

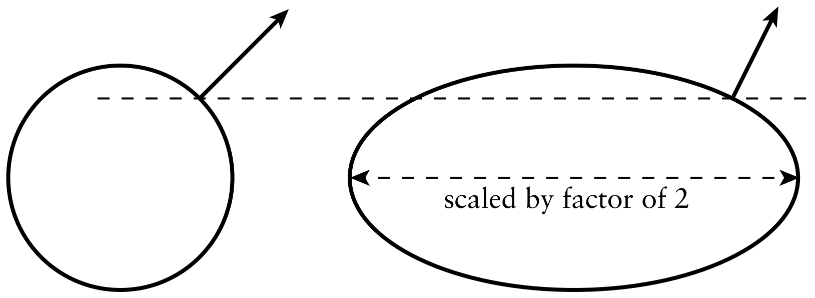


Figure 15: At left is a circle and a normal vector to the circle at a particular point. If we scale the circle by a factor of two horizontally, the normal vector scales by a factor of a half in the same dimension.

the desired position $\mathbf{B} = \mathbf{TRS}\hat{\mathbf{B}}$. To intersect ray, $\mathbf{E} + t\mathbf{D}$, with $\mathbf{B}$ we transform the point, $\mathbf{E}$, and the displacement $\mathbf{D}$ as follows:

$$\hat{\mathbf{E}} = \mathbf{S}^{-1}\mathbf{R}^{-1}\mathbf{T}^{-1}\mathbf{E} \tag{61}$$
$$\hat{\mathbf{D}} = \mathbf{S}^{-1}\mathbf{R}^{-1}\mathbf{D} \tag{62}$$

The point is translated, rotated and scaled but the displacement is only rotated and scaled.

Now intersect ray, $\hat{\mathbf{E}} + t\hat{\mathbf{D}}$, with the object in its standard position, $\hat{\mathbf{B}}$, as described in the previous sections. This gives the value of t and consequently allows you to directly calculate $\mathbf{P} = \mathbf{E} + t\mathbf{D}$. You do not even have to transform back because t has the same value in both standard and transformed coordinate systems.

In addition to scaling, rotating and translating the standard primitives, this mechanism allows us to stretch and squash them by scaling them by different factors in the different dimensions. For example, an ellipsoid is simply a stretched sphere. However, such anisotropic scaling causes interesting problems with the normal vectors, as discussed in the following section.

3.4.2 Transforming normal vectors

In ray tracing we need to know not just the intersection point but also the normal vector to the surface at the intersection point. This normal vector is used in the illumination calculations. However, if we scale an object anisotropically (different amounts in each coordinate), the normal vector scales as the *inverse* of the object scaling, although it rotates in the *same* way as the object. Figure 15 graphically illustrates this phenomenon.

One way to understand this behaviour is through the following mathematics. Remember that the normal vector is related to the derivative of the surface. The normal vector is perpendicular to the surface, which means that it is perpendicular to any derivative vector. As an example, consider the ellipsoid created by scaling the unit sphere by the matrix:

$$\mathbf{S} = \begin{bmatrix} l & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{63}$$

The intersection point between the unit sphere and the inverse transformed ray will be at some point $\hat{\mathbf{P}} = (\hat{x}, \hat{y}, \hat{z})$. This equates to the spherical polar coordinate $(1, \theta, \phi)$ where[7] $(\hat{x}, \hat{y}, \hat{z}) = (\cos\phi\cos\theta, \cos\phi\sin\theta, \sin\phi)$. By virtue of the fact that the normal to every point on the sphere passes through the centre of the sphere, it is easy to see that the normal vector, $\hat{\mathbf{N}}$, is also $\hat{\mathbf{N}} = (\cos\phi\cos\theta, \cos\phi\sin\theta, \sin\phi)$.

Transforming $\hat{\mathbf{P}}$ to the true intersection point $\mathbf{P}$ is simply a matter of applying $\mathbf{P} = \mathbf{S}\hat{\mathbf{P}}$. Thus the intersection point of the ray with the ellipsoid is at rectangular coordinate $(x, y, z) = (l\cos\phi\cos\theta, m\cos\phi\sin\theta, n\sin\phi)$.

To find the normal vector to the ellipsoid at this intersection point, we can find two non-parallel derivative vectors to the surface at the intersection point, and take their cross product to give the normal vector, $\mathbf{N}$. Two derivative vectors are:

$$\frac{\partial(\mathbf{P})}{\partial\phi} = \frac{\partial(x,y,z)}{\partial\phi} = (-l\sin\phi\cos\theta, -m\sin\phi\sin\theta, n\cos\phi) \tag{64}$$

$$\frac{\partial(\mathbf{P})}{\partial\theta} = \frac{\partial(x,y,z)}{\partial\theta} = (-l\cos\phi\sin\theta, m\cos\phi\cos\theta, 0) \tag{65}$$

This leads to the normal vector being[8]:

$$\mathbf{N} = \frac{\partial(\mathbf{P})}{\partial\phi} \times \frac{\partial(\mathbf{P})}{\partial\theta} \tag{66}$$

$$= (\frac{1}{l}\cos\phi\cos\theta, \frac{1}{m}\cos\phi\sin\theta, \frac{1}{n}\sin\phi) \tag{67}$$

Thus, we conclude that while:

$$\mathbf{P} = \mathbf{S}\hat{\mathbf{P}} \tag{68}$$

$$\mathbf{N} = \mathbf{S}^{-1}\hat{\mathbf{N}} \tag{69}$$

This leaves open the question of what to do about the rotation and translation components of the transformation. It is intuitively obvious that rotating an object causes the normal vector to rotate by the same amount and, because normal vectors are displacements rather than points, they should not be translated. So the final analysis is that:

$$\mathbf{P} = \mathbf{TRS}\hat{\mathbf{P}} \tag{70}$$

$$\mathbf{N} = \mathbf{RS}^{-1}\hat{\mathbf{N}} \tag{71}$$

This piece of mathematics is specific to the unit sphere, but the same result holds for any object.

[7] If you are wondering where r has disappeared to, remember that, in a unit sphere, $r = 1$.

[8] If you try this at home you will find that you will need to divide through by the factor $-1/(lmn\cos\phi)$.

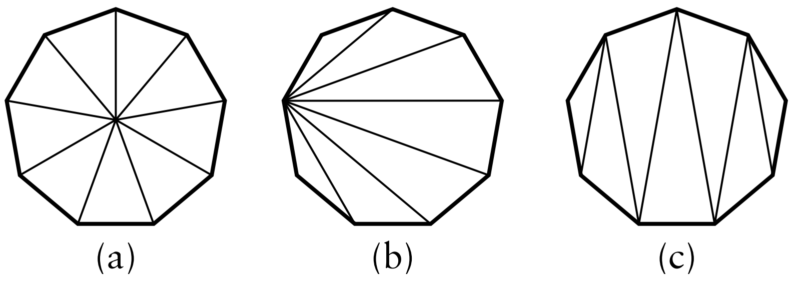


Figure 16: Three ways to split a polygon into triangles: (a) define a central vertex and connect every edge to this vertex to make n isosceles triangles; (b) select one vertex on the n-gon, and make a triangle fan emanating from this vertex; or (c) start at one edge of the n-gon and make a triangle strip set which proceeds from this edge of the polygon to the opposite edge.

3.5 Converting the primitives to polygons

We may be in a situation where we define objects in terms of the various primitives, but where we wish to draw the objects using polygon scan conversion. In this case we need to convert primitives into polygons. Some polygon scan conversion algorithms (notably their implementation in hardware in graphics cards) can only deal with triangles; in these cases we may need to do a little bit of extra work to ensure that all of the generated polygons are triangles.

Converting the curved primitives (sphere, cylinder, cone, torus, disc) involves approximating a curved profile by a series of straight line segments. The simplest example is the disc. This can be approximated by a regular n-gon, where n is chosen to give an adequate approximation to the disc. "Adequate" in this case will depend on the desired rendering resolution, the desired speed of rendering, and the desired quality of the final image. If necessary, this n-gon can be converted to triangles in one of a number of ways, as shown in Figure 16.

A cylinder or cone can be converted to polygons by polygonising the discs at either end into n-gons, and then connecting corresponding vertices on the two n-gons. In the special case of a cone with a point, one of the n-gons obviously degenerates to that point.

Spheres and tori can be most easily converted to polygons by considering their parameterisation in terms of θ and ϕ. For a sphere these are Equations 72–74 (below); for a torus they are Equations 53–55.

$$x = r\cos\phi\cos\theta \tag{72}$$
$$y = r\cos\phi\sin\theta \tag{73}$$
$$z = r\sin\phi \tag{74}$$

By selecting appropriate steps in the two parameters we can generate a set of quadrilaterals which approximates the curved primitive.

It should be noted that spheres can be polygonised with more uniform polygons by starting with one of the five Platonic solids, and subdividing its faces accordingly. The details of this are left as an exercise to the reader.

3.6 Conics, quadrics, and superquadrics

The ray tracing primitives, described above, have relatively simple mathematical definitions. This is what makes them attractive: the simple mathematical definition allows for simple ray-object intersection code. Following from this, it would seem logical to investigate other shapes with simple mathematical definitions. Spheres, cones and cylinders are part of a more general family of parametric surfaces called *quadrics* (N.B. tori are *not* quadrics). Quadrics are the 3D analogue of 2D conics. We describe these general families below, but it turns out that they are of little practical use. It would seem that the general quadrics are a "dead end" in graphics research.

3.6.1 Conics

A conic is a two dimensional curve described by the general equation:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

This general form can be rotated, scaled, and translated so that it is aligned along the axes of the coordinate system. It will then have the simpler equation:

$$ax^2 + by^2 = k \text{ or } ax^2 + by = k$$

The useful conics are the ellipse (of which the circle is a special case), the hyperbola, and the parabola. For more details see **R&A** section 4-10, especially Table 4-8 on page 242. Table 4-8 is included in the handout.

3.6.2 Quadrics

The quadrics are the three dimensional analogue of the conics. The general equation is:

$$Ax^2 + By^2 + Cz^2 + Dxy + Eyz + Fzy + Gx + Hy + Jz + K = 0$$

This general form can be rotated, scaled, and translated so that it is aligned along the axes of the coordinate system. It will then have the simpler equation:

$$ax^2 + by^2 + cz^2 = k \text{ or } ax^2 + by^2 + cz = k$$

The useful conics are the ellipsoid (of which the sphere is a special case), the infinite cylinder, and the infinite cone. Various hyperboloids, and paraboloids are also defined by these equations, but these have little real use unless one is designing satellite dishes (paraboloid), headlamp reflectors (also paraboloid), or power station cooling towers (hyperboloid). For more details see **R&A** section 6-4, especially Figure 6-18 on page 403. Figure 6-18 is included in the handout.

3.6.3 Superquadrics

These are an extension of quadrics, where the power on the coordinate does not have to be 2. The general form of a superquadric centred at the origin and aligned along the coordinate axes is:

$$(ax)^n + (by)^n + (cz)^n = k$$

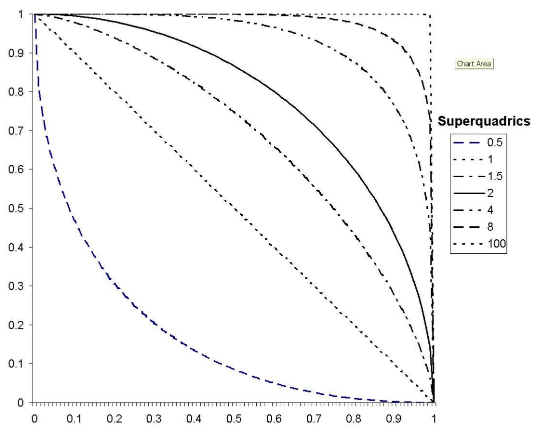


Figure 17: Superquadric functions for $n = 0.5, 1, 1.5, 2, 4, 8, 100$.

Super-ellipsoids tend to be the only members of this family that are actually used, and even they are only used in very limited areas. The effect of n on a super-ellipsoid is roughly as follows: $n = 2$ is a standard ellipsoid; $n > 2$ becomes closer to a box as n increases; in the limit, as $n \to \infty$, the shape does become a box; $n < 2$ is a more pointy version, the "points" being along the main axes; $n = 1$ is an octahedral shape; and $n < 1$ has sharp points on the main axes. Figure 17 shows examples of the shapes of these curves.

The interested student may like to have a quick look at Alan Barr's two papers on superquadrics. The papers can be found in the Computer Laboratory library in *IEEE Transactions on Computer Graphics and Applications* volume 1, number 1 (January 1981), pages 11-23, and volume 1, number 3 (July 1981), pages 41-47.

3.7 Exercises

1. Give mathematical equations which define a plane, a sphere, an infinitely long cylinder, an infinitely long cone, and a torus. You will find it helpful to centre each primitive at the origin and to align it in a sensible way with respect to the coordinate axes.

2. Show how to intersect a ray with each of the five primitives from Question 1. You may assume that you are provided with functions to find the roots of linear, quadratic, cubic and quartic equations. Also show how to compute the normal vector at the intersection point.

3. Show how to intersect a ray with a finite length closed cylinder. Ensure that you handle all special cases, including that of a ray which is parallel to the axis of a finite length cylinder. Give both intersection point and normal vector for all cases in which an intersection occurs.

4. Give a complete algorithm for intersecting a ray with a finite length closed cone, including calculation of both intersection point and normal vector.

5. Work out if there exists a faster intersection algorithm for an axis aligned $2 \times 2 \times 2$ unit box than just six separate polygon intersection calculations.

6. Show how to convert a cylinder into a polygon mesh. What changes do you have to make if the mesh may contain only triangles?

7. Show how to convert a torus into a polygon mesh.

8. Show how to convert a sphere into a triangle mesh. How can you get the most even distribution of triangle vertices across the sphere?

9. [1999/7/11] (a) Give a parametric definition of a torus centred at the origin and aligned with the coordinate axes. (b) Outline how you would find the first intersection point, if any, of a ray with the torus from the previous part.

10. [2000/9/4] (a) Show how you would calculate the first intersection point between an arbitrary ray and a finite length open cylinder of unit radius aligned along the x-axis. [**Note:** an open cylinder is one which has no end caps.] Having found the intersection point, how would you find the normal vector?

11. [2001/7/9] (b) (i) Show how to find the first intersection between a ray and a finite-length, open-ended cone, centred at the origin, aligned along the x-axis, for which both ends of the finite-length are on the positive x-axis (i.e. $0 < x_{\min} < x_{\max}$). [6 marks]
    (ii) Extend this to cope with a closed cone (i.e. the same cone, but with end caps). Take care to consider any special cases. [5 marks]
    (iii) Extend this further to give the normal vector at the intersection point. [3 marks]

12. [2002/7/9] (a) A disc is a finite, planar, circular object. Describe an algorithm to find the point of intersection of an arbitrary ray with an arbitrary disc in three dimensions. Ensure that you describe the parameters used to define both the ray and the disc. [6 marks]
    (b) Given the above algorithm and an algorithm to find the intersection of an arbitrary ray with a finite-length open cylinder, a programmer has two choices for implementing an algorithm to find the intersection with a finite-length closed cylinder. She could simply use the finite-length open cylinder primitive and two disc primitives. Alternatively she could implement the finite-length closed cylinder as a primitive in its own right by adding extra code to the open cylinder algorithm. Compare the two alternatives in terms of efficiency and accuracy. [4 marks]

4 Introduction to splines

While the above primitives allow us to specify particular types of curved surface, we find ourselves in need of some more general way of specifying arbitrary curved surfaces. We want some mechanism which allows us to specify any smooth curved surface which we desire. This problem was first faced in the 1960s for the design of aeroplanes and cars. We will look at three solutions: Bezier surfaces, B-spline surfaces (including NURBS) and subdivision surfaces. It transpires that one of the most important problems is getting different patches of surface to connect together smoothly, that is: with continuity of position ($C0$), slope ($C1$) and curvature ($C2$). These are continuity of the function, its first and its second derivatives, respectively. Much of the ensuing discussions consider how to achieve such continuity.

5 Bezier curves

Bezier curves were covered in the Part IB *Computer Graphics and Image Processing* course. This section gives some of the mathematical details, as does **R&A** Section 5-8. Parts of this Section of **R&A** are included in the handout.

If you want to experiment with Bezier curves then there are a number of on-line tutorials. One such is available from the Technion in Israel.

A Bezier curve is a weighted sum of $n+1$ control points, $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$, where the weights are the Bernstein polynomials:

$$\mathbf{P}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i, 0 \le t \le 1 \tag{75}$$

The Bezier curve of order $n+1$ (degree n) has $n+1$ control points. Below are the first three orders of Bezier curve definitions.

$$\begin{aligned}
\text{linear} \quad & \mathbf{P}(t) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 && (76)\\
\text{quadratic} \quad & \mathbf{P}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 && (77)\\
\text{cubic} \quad & \mathbf{P}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 && (78)
\end{aligned}$$

5.1 Ways of thinking about Bezier curves

There are several useful ways in which you can think about Bezier curves. Here are the ones that I use.

Linear interpolation. Equation 76 is obviously a linear interpolation between two points. Equation 77 can be rewritten as a linear interpolation between linear interpolations between points:

$$\mathbf{P}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \tag{79}$$

Equation 78 can be rewritten as a linear interpolation between linear interpolations between linear interpolations between points. This is left as an exercise for the reader.

Weighted average. A Bezier curve can be seen as a weighted average of all of its control points. Because all of the weights are positive, and because the weights sum to one, the Bezier curve is guaranteed to lie within the convex hull of its control points.

Refinement of the control polygon. A Bezier curve can be seen as some sort of refinement of the polygon made by connecting its control points in order. The Bezier curve starts and ends at the two end points and its shape is determined by the relative positions of the $n-1$ other control points, although it will generally not pass through any of these other control points. The tangent vectors at the start and end of the curve pass through the end point and the immediately adjacent point.

Rogers and Adams list the properties of the Bezier curve on page 291.

5.2 Continuity

You should note that each Bezier curve is independent of any other Bezier curve. If we wish two Bezier curves to join with any type of continuity, then we must explicitly position the control points of the second curve so that they bear the appropriate relationship with the control points in the first curve.

Any Bezier curve is infinitely differentiable within itself, and is therefore continuous to any degree (C^n-continuous, $\forall n$). We therefore only need concern ourselves with continuity across the joins between curves. Assume that we have two Bezier curves of the same order: $\mathbf{P}(t)$, defined by $(\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n)$, and $\mathbf{Q}(t)$, defined by $(\mathbf{Q}_0, \mathbf{Q}_1, \ldots, \mathbf{Q}_n)$. C^0-continuity (continuity of position) can be achieved by setting $\mathbf{P}(1) = \mathbf{Q}(0)$. This gives a formula for $\mathbf{Q}_0$ in terms of the $\mathbf{P}_i$s:

$$\mathbf{Q}_0 = \mathbf{P}_n. \tag{80}$$

Similarly for C^1-continuity, we need C^0-continuity and $\mathbf{P}'(1) = \mathbf{Q}'(0)$, giving:

$$\mathbf{Q}_1 - \mathbf{Q}_0 = \mathbf{P}_n - \mathbf{P}_{n-1} \tag{81}$$

Combining Equations 81 and 80 gives a formula for $\mathbf{Q}_1$ in terms of the $\mathbf{P}_i$s:

$$\begin{aligned} \mathbf{Q}_1 &= 2\mathbf{P}_n - \mathbf{P}_{n-1} && (82) \\ &= \mathbf{P}_n + (\mathbf{P}_n - \mathbf{P}_{n-1}) && (83) \end{aligned}$$

Continuing in this vein, we find that the requirements for C^2-continuity (i.e. C^1-continuity and $\mathbf{P}''(1) = \mathbf{Q}''(0)$) give:

$$\mathbf{Q}_2 - 2\mathbf{Q}_1 + \mathbf{Q}_0 = \mathbf{P}_n - 2\mathbf{P}_{n-1} + \mathbf{P}_{n-2} \tag{84}$$

Combining Equations 84, 81, and 80 gives a formula for $\mathbf{Q}_2$ in terms of the $\mathbf{P}_i$s:

$$\begin{aligned} \mathbf{Q}_2 &= 4\mathbf{P}_n - 4\mathbf{P}_{n-1} + \mathbf{P}_{n-2} && (85) \\ &= \mathbf{P}_{n-2} + 4(\mathbf{P}_n - \mathbf{P}_{n-1}) && (86) \end{aligned}$$

5.3 Bezier surfaces

We learnt in the IB course that the simplest way to construct a Bezier surface is as the tensor product of Bezier curves. A tensor product Bezier surface of order $n+1$ is defined by $(n+1)^2$ control points. It is called a Bezier patch.

$$\mathbf{P}(s,t) = \sum_{i=0}^{n} \binom{n}{i} (1-s)^{n-i} s^i \sum_{j=0}^{n} \binom{n}{j} (1-t)^{n-j} t^j \mathbf{P}_{i,j} \tag{87}$$

You can think about this as moving the control points of one Bezier curve along a set of Bezier curves to sweep out a surface. Continuity across a boundary between two Bezier patches is only guaranteed if each of the Bezier curves across the join obey the curve continuity conditions. Again, this was covered in the IB course.

5.4 Exercises

1. Explain what $C0$-, $C1$-, $C2$-, Cn-continuity mean.

2. Derive the constraints on control point positions which ensure that two quartic Bezier curves join with (a) $C0$-continuity, (b) $C1$-continuity, and (c) $C2$-continuity.

6 B-splines

B-splines are covered in some detail below and also in **R&A** Section 5-9. Parts of this Section of **R&A** are included in the handout. Beware that none of the worked examples are in the handout. These may come in useful, and you will need to get hold of a real copy of **R&A** if you wish to work your way through them.

B-splines are a more general type of curve than Bezier curves. In a B-spline each control point is associated with a *basis function*,$N_{i,k}$.

$$\mathbf{P}(t) = \sum_{i=1}^{n+1} N_{i,k}(t)\mathbf{P}_i, t_{\min} \le t < t_{\max} \tag{88}$$

There are $n+1$ control points, $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_{n+1}$. The $N_{i,k}$ basis functions are of order k (degree $k-1$). k must be at least 2 (linear), and can be no more than $n+1$ (the number of control points). The important point here is that the order of the curve (2 [linear], 3 [quadratic], 4 [cubic],...) is therefore not dependent on the number of control points (which it is for Bezier curves, where k must always equal $n+1$).

Equation 88 defines a piecewise continuous function. The $N_{i,k}$ are defined by a *knot vector*, $(t_1, t_2, \ldots, t_{k+(n+1)})$, must be specified. This determines the values of t at which the pieces of curve join, like knots joining bits of string. It is necessary that:

$$t_i \le t_{i+1}, \forall i \tag{89}$$

The $N_{i,k}$ depend *only* on the value of k and the values, t_i, in the knot vector. $N_{i,k}$ is defined recursively as:

$$\begin{aligned} N_{i,1}(t) &= \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases} \\ N_{i,k}(t) &= \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) \end{aligned} \tag{90}$$

This is essentially a modified version of the idea of taking linear interpolations of linear interpolations of linear interpolations...

At this point it would be instructive for you to work out $N_{1,1}$, $N_{2,1}$, $N_{3,1}$, $N_{1,2}$, $N_{2,2}$, $N_{1,3}$ for the knot vector $[0, 2, 3, 6]$. It helps if you draw the graphs for these functions.

There are several things that you should note about these equations. Each $N_{i,k}(t)$ depends only on the $k+1$ knot values from t_i to t_{i+k}. $N_{i,k}(t) = 0$ for $t < t_i$ or $t \ge t_{i+k}$ so $\mathbf{P}_i$ only influences the curve for $t_i \le t < t_{i+k}$. Formally, $\mathbf{P}(t)$ is a polynomial of order k (degree $k-1$) on each interval $t_i \le t < t_{i+1}$. Across the knots $\mathbf{P}(t)$ is C^{k-2}-continuous. $\mathbf{P}(t)$ is, of course, continuous in all its derivatives between the knots. Remember from

Equation 16 that a weighted sum of points only makes sense if the weights sum to one. $\mathbf{P}(t)$ is therefore validly defined only where

$$\sum_{i=1}^{n+1} N_{i,k}(t) = 1. \tag{91}$$

This is the range $t_{\min} \leq t < t_{\max}$ where $t_{\min} = t_k$ and $t_{\max} = t_{n+2}$. Even more properties of B-splines are described in Rogers and Adams pp. 306–7.

6.1 The knot vector

The above explanation shows that the knot vector is very important. The knot vector can, by its definition, be any sequence of numbers provided that each one is greater than or equal to the preceding one. Some types of knot vector are more useful than others. Knot vectors are generally placed into one of three categories: uniform, open uniform, and non-uniform.

Uniform. These are knot vectors for which

$$t_{i+1} - t_i = \text{constant}, \forall i \tag{92}$$

For example:

$$\begin{array}{l} [1, 2, 3, 4, 5, 6, 7, 8] \\ [0, 1, 2, 3, 4, 5] \\ [0, 0.25, 0.5, 0.75, 1.0] \\ [-2.5, -1.4, -0.3, 0.8, 1.9, 3.0] \end{array}$$

All of the basis functions are just shifted versions of one another and so the implementation is very easy.

Open Uniform. These are uniform knot vectors which have k equal knot values at each end:

$$\begin{array}{rl} t_i = t_1, & i \leq k \\ t_{i+1} - t_i = \text{constant}, & k \leq i < n+2 \\ t_i = t_{k+(n+1)}, & i \geq n+2 \end{array} \tag{93}$$

For example:

$$\begin{array}{ll} [0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4] & (k = 4) \\ [1, 1, 1, 2, 3, 4, 5, 6, 6, 6] & (k = 3) \\ [0.1, 0.1, 0.1, 0.1, 0.1, 0.3, 0.5, 0.7, 0.7, 0.7, 0.7, 0.7] & (k = 5) \end{array}$$

This is essentially just a simple modification to the uniform case which allows the curve to go through its two end points.

Non-uniform. This is the general case, the only constraint being the standard $t_i \leq t_{i+1}, \forall i$ (Equations 89). For example:

$$\begin{array}{l} [1, 3, 7, 22, 23, 23, 49, 50, 50] \\ [1, 1, 1, 2, 2, 3, 4, 5, 6, 6, 6, 7, 7, 7] \\ [0.2, 0.7, 0.7, 0.7, 1.2, 1.2, 2.9, 3.6] \end{array}$$

The shapes of the $N_{i,k}$ basis functions are determined entirely by the *relative* spacing between the knots. Scaling ($t'_i = \alpha t_i, \forall i$) or translating ($t'_i = t_i + \Delta t, \forall i$) the knot vector has no effect on the shapes of the $N_{i,k}$ nor on the shape of the actual curve $\mathbf{P}(t)$.

The above gives a description of the various types of knot vector but it doesn't really give you any insight into how the knot vector determines the shape of the curve. The following subsections look at the different types of knot vector in more detail. However, the best way to get to feel for these is to derive and draw the basis functions yourself.

6.1.1 Uniform knot vector

For simplicity, let $t_i = i$ (this is allowable given that the scaling or translating the knot vector has no effect on the shapes of the $N_{i,k}$). The knot vector thus becomes $[1, 2, 3, \ldots, k + (n + 1)]$ and Equation 90 simplifies to:

$$\begin{aligned} N_{i,1}(t) &= \begin{cases} 1, & i \le t < i+1 \\ 0, & \text{otherwise} \end{cases} \\ N_{i,k}(t) &= \frac{t-i}{k-1} N_{i,k-1}(t) + \frac{i+k-t}{k-1} N_{i+1,k-1}(t) \end{aligned} \qquad (94)$$

You should be easily able to graph the first few of these for yourself. The principle thing to note about the uniform basis functions is that, for a given order k, the basis functions are all simply shifted versions of one another. See Rogers and Adams Figure 5-36.

6.1.2 Things you can change about a uniform B-spline

With a uniform B-spline, you obviously cannot change the basis functions (they are fixed because all the knots are equispaced). However you can alter the shape of the curve by modifying a number of things:

Moving control points. Moving the control points obviously changes the shape of the curve.

Multiple control points. Sticking two adjacent control points on top of one another causes the curve to pass closer to that point. Stick enough adjacent control points on top of one another and you can make the curve pass through that point (Rogers and Adams, Figure 5-45).

Order. Increasing the order k increases the continuity of the curve at the knots, increases the smoothness of the curve, and tends to move the curve farther from its defining polygon. (Rogers and Adams, Figure 5-44).

Joining the ends. You can join the ends of the curve to make a closed loop. Say you have M points, $\mathbf{P}_1, \ldots, \mathbf{P}_M$. You want a closed B-spline defined by these points. For a given order, k, you will need $M + (k - 1)$ control points (repeating the first $k - 1$ points): $\mathbf{P}_1, \ldots, \mathbf{P}_M, \mathbf{P}_1, \ldots, \mathbf{P}_{k-1}$. Your knot vector will thus have $M + 2k - 1$ uniformly spaced knots.

6.1.3 Open uniform knot vector

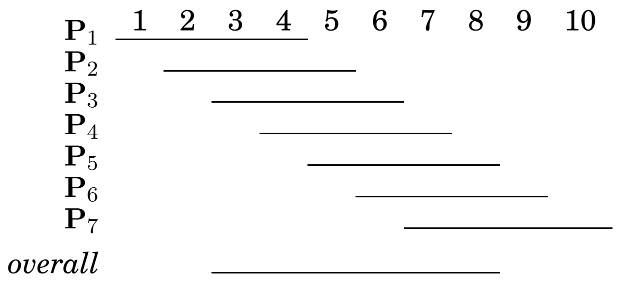
The previous section intimated that uniform B-splines can be used to describe closed curves: all you have to do is join the ends as described above. If you do not want a closed curve, and you use a uniform knot vector, you find that you need to specify control points at each end of the curve which the curve doesn't go near (e.g. Rogers and Adams, Figure 5-44, the order 4 curve).

If you wish your B-spline to start and end at your first and last control points then you need an open uniform knot vector (e.g. Rogers and Adams, Figure 5-41). The only difference between this and the uniform knot vector being that the open uniform version has k equal knots at each end.

An order k open uniform B-spline with $n+1 = k$ points is the Bezier curve of order k. It would be a useful exercise for you to prove this for $k = 3$. For ease of calculation take the knot vector to be $[0, 0, 0, 1, 1, 1]$.

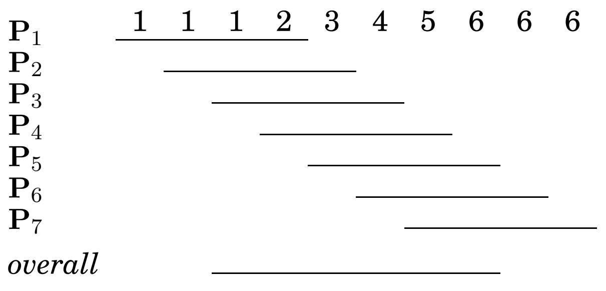
6.1.4 The difference between uniform and open uniform

It may help, at this stage, to compare a particular uniform and an equivalent open uniform knot vector. This is a uniform knot vector for $n + 1 = 7$, $k = 3$:



The lines show the range of t over which each $\mathbf{P}_i$ is non-zero. The B-spline itself (the *overall* line in the diagram) is defined over the range $t_3 \le t < t_8$, i.e. over the range $3 \le t < 8$.

By comparison an open uniform knot vector for $n + 1 = 7$, $k = 3$ is:



The B-spline itself is defined over the range $t_3 \le t < t_8$, i.e. over the range $1 \le t < 6$. By the definition of a open uniform knot vector $t_3 = t_1$ and $t_8 = t_{10}$ and so an open uniform B-spline is defined over the *full* range of t from t_1 to $t_k + n + 1$.

6.1.5 Non-uniform knot vector

Any B-spline whose knot vector is neither uniform nor open uniform is non-uniform. Non-uniform knot vectors allow any spacing of the knots, including multiple knots (adjacent knots with the same value). We need to know how this non-uniform spacing affects the basis functions in order to understand where non-uniform knot vectors could be useful. It transpires that there are only three cases of any interest: (1) multiple knots (adjacent knots equal); (2) adjacent knots more closely spaced than the next knot in the vector; and (3) adjacent knots less closely spaced than the next knot in the vector. Obviously, case (3) is simply case (2) turned the other way round.

Multiple knots. A multiple knot reduces the degree of continuity at that knot value. Across a normal knot the continuity is C^{k-2}. Each extra knot with the same value reduces continuity at that value by one. This is the only way to reduce the continuity of the curve at the knot values. If there are $k-1$ (or more) equal knots then you get a discontinuity in the curve.

Close knots. As two knots' values get closer together, relative to the spacing of the other knots, the curve moves closer to the related control point.

Distant knots. As two knots' values get further apart, relative to the spacing of the other knots, the curve moves further away from the related control point.

6.1.6 Use of non-uniform knot vectors

Standard procedure is to use uniform or open uniform B-splines unless there is a very good reason not to do so. Moving two knots closer together tends to move the curve only slightly and so there is usually little point in doing it. This leads to the conclusion that the main use of non-uniform B-splines is to allow for multiple knots, which adjust the continuity of the curve at the knot values.

However, non-uniform B-splines are the general form of the B-spline because they incorporate open uniform and uniform B-splines as special cases. Thus we will talk about *non-uniform B-splines* when we mean the general case, incorporating both uniform and open uniform.

6.1.7 What can you do to control the shape of a B-spline?

- Move the control points.
- Add or remove control points.
- Use multiple control points.
- Change the order, k.
- Change the type of knot vector.
- Change the relative spacing of the knots.
- Use multiple knot values in the knot vector.

6.1.8 What should the defaults be?

If there are no pressing reasons for doing otherwise, your B-spline should be defined as follows:

- $k = 4$ (cubic);
- no multiple control points;
- uniform (for a closed curve) or open uniform (for an open curve) knot vector.

6.2 B-spline patches

We generalise from B-spline curves to B-spline surfaces in the same way as we did for Bezier patches. Take a tensor product of two versions of Equation 88.

$$\mathbf{P}(s,t) = \sum_{i=1}^{m+1} \sum_{j=1}^{n+1} \mathbf{P}_{i,j} N_{i,k}(s) N_{j,l}(t),\, s_{\min} \le s < s_{\max}, t_{\min} \le t < t_{\max} \tag{95}$$

where it is usual for the patch to have the same order (i.e. $k = l$) in both directions. Patches are thus defined by a quadrilateral grid of control points of size $(m+1) \times (n+1)$.

6.3 Why B-splines?

B-splines have many nice properties when compared to other families of curves which could be used. They:

- minimise the order of the polynomial pieces (order k)
- maximise the continuity between pieces (continuity $C(k-2)$)
- minimise the number of control points controlling a piece (k points)
- have positive basis functions
- have basis functions which partition unity, implying that each piece lies inside its control points' convex hull
- are invarient with respect to affine transforms

6.4 Exercises

1. How many control points are required for a quartic Bezier and how many for a quartic B-spline?
2. Why are cubics the default for B-spline use?
3. Explain the difference between Uniform, Open Uniform, and Non-Uniform knot vectors. What are the advantages of each type?

4. [2000/9/4] (b) A non-rational B-spline has knot vector $[1, 2, 4, 7, 8, 10, 12]$. Derive the first of the third order (second degree) basis functions, $N_{1,3}(t)$, and graph it.
   If this knot vector were used to draw a third order B-spline, how many control points would be required? [7 marks]

5. [2001/8/4] (a) For a given order, k, there is only one basis function for uniform B-splines. Every control point uses a shifted version of that one basis function. How many different basis functions are there for open-uniform B-splines of order k with $n + 1$ control points, where $n >= 2k - 3$? [6 marks]
   (b) Explain what is different in the cases where $n < 2k - 3$ compared with the cases where $n >= 2k - 3$. [3 marks]
   (c) Sketch the different basis functions for $k = 2$ and $k = 3$ (when $n >= 2k - 3$). [4 marks]
   (d) Show that the open-uniform B-spline with $k = 3$ and knot vector $[0, 0, 0, 1, 1, 1]$ is equivalent to the quadratic Bezier curve. [7 marks]

6. [2002/7/9] (d) Derive the formula of and sketch a graph of $N_{3,3}(t)$, the third of the quadratic B-spline basis functions, for the knot vector $[0, 0, 0, 1, 3, 3, 4, 5, 5, 5]$. [6 marks]

6.5 NURBS

NURBS are covered below and in some detail in **R&A** Section 5-13. Parts of this Section of **R&A** are included in the handout.

Non-uniform rational B-splines are the curves that are currently used in any graphics application that requires curves and surfaces with more functionality than Bezier curves can offer. In most cases, you would actually use the special case of non-rational B-splines (those described in the previous section) but it is useful to have the more general rational versions available for certain types of curve and surface. In addition to the features listed above for B-splines, NURBS are invariant with respect to perspective transforms.

NURBS are generally rendered by converting them to lots of small polygons and then using polygon scan conversion. They can also by ray traced, but a general analytic ray-NURBS intersection algorithm is a nightmare, so numerical techniques are used to find the intersection point.

NURBS curves incorporate – as special cases – uniform B-splines, non-rational B-splines, Bezier curves, lines, and conics. NURBS surfaces incorporate planes, quadrics, and tori. Note that this does not quite mean what it says. It is tricky to get NURBS to represent *infinite* surfaces, but they can certainly represent finite sections of infinite surfaces such as planes, paraboloids, and hyperboloids.

If you want to experiment with NURBS curves then there are a number of on-line tutorials. One such is available from the Technion in Israel.

Rational B-splines have all of the properties of non-rational B-splines plus the following two useful features:

- They produce the correct results under projective transformations (while non-rational B-splines only produce the correct results under affine transformations).

- They can be used to represent lines, conics, non-rational B-splines; and, when generalised to patches, can represent planes, quadrics, and tori.

In this case *rational* means "one polynomial divided by another" (see Equation 96). The antonym of *rational* is *non-rational* (i.e. a non-rational B-spline is just a polynomial (see Equation 88). Non-rational B-splines are a special case of rational B-splines, just as uniform B-splines are a special case of non-uniform B-splines. Thus, *non-uniform rational B-splines* encompass almost every other possible 3D shape definition. *Non-uniform rational B-spline* is a bit of a mouthful and so it is generally abbreviated to *NURBS*.

We have already learnt all about the the *B-spline* bit of *NURBS* and about the *non-uniform* bit. So now all we need to know is the meaning of the *rational* bit and we will fully(?) understand NURBS.

Rational B-splines are defined simply by applying the B-spline equation (Equation 88) to homogeneous coordinates, rather than normal 3D coordinates. We discussed homogeneous coordinates in the IB course. You will remember that these are 4D coordinates where the transformation from 4D to 3D is:

$$(x', y', z', w) \to \left(\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w}\right) \tag{96}$$

Last year we said that the inverse transform was:

$$(x, y, z) \to (x, y, z, 1) \tag{97}$$

This year we are going to be more cunning and say that:

$$(x, y, z) \to (xh, yh, zh, h) \tag{98}$$

Thus our 3D control point, $\mathbf{P}_i = (x_i, y_i, z_i)$, becomes the homogeneous control point, $\mathbf{C}_i = (x_i h_i, y_i h_i, z_i h_i, h_i)$.

A NURBS curve is thus defined as:

$$\mathbf{P}_H(t) = \sum_{i=1}^{n+1} N_{i,k}(t)\mathbf{C}_i, t_{\text{min}} \le t < t_{\text{max}} \tag{99}$$

Compare Equation 99 with Equation 88 to see just how easy this is!

We now want to see what a NURBS curve looks like in normal 3D coordinates, so we need to apply Equation 96 to Equation 99. In order to better explain what is going on, we first write Equation 99 in terms of its individual components. Equation 99 is equivalent to:

$$x'(t) = \sum_{i=1}^{n+1} x_i h_i N_{i,k}(t) \tag{100}$$

$$y'(t) = \sum_{i=1}^{n+1} y_i h_i N_{i,k}(t) \tag{101}$$

$$z'(t) = \sum_{i=1}^{n+1} z_i h_i N_{i,k}(t) \tag{102}$$

$$h(t) = \sum_{i=1}^{n+1} h_i N_{i,k}(t) \tag{103}$$

Equation 96 tells us that, in 3D:

$$x(t) = x'(t)/h(t) \tag{104}$$
$$y(t) = y'(t)/h(t) \tag{105}$$
$$z(t) = z'(t)/h(t) \tag{106}$$

Thus the 4D to 3D conversion gives us the curve in 3D:

$$\mathbf{P}(t) = \frac{\sum_{i=1}^{n+1} N_{i,k}(t)\mathbf{P}_i h_i}{\sum_{i=1}^{n+1} N_{i,k}(t)h_i}, t_{\text{min}} \le t < t_{\text{max}} \tag{107}$$

This looks a lot more fierce than Equation 99, but is simply the same thing written a different way.

So now, we need to define an additional parameter, h_i, for each control point, $\mathbf{P}_i$. The default is to set $h_i = 1, \forall i$. This results in the denominator of Equation 107 becoming one, and the NURBS equation (Equation 107) therefore reducing to the non-rational B-spline equation (Equation 88).

Increasing h_i pulls the curve closer to point $\mathbf{P}_i$. Decreasing h_i pushes the curve farther from point $\mathbf{P}_i$. Setting $h_i = 0$ means that $\mathbf{P}_i$ has no effect on the curve at all. See Rogers and Adams Figure 5-58 for an example, and play with an on-line NURBS tutorials such as the one mentioned above.

6.6 An example: a circle defined by NURBS

This subsection provides an example of a shape which cannot be represented by non-rational B-splines: a circle. A non-rational B-spline or a Bezier curve cannot exactly represent a circle. An interesting exercise is to place a cubic Bezier curve's end points at $(0, 1)$ and $(1, 0)$, with the other control points at $(\alpha, 1)$ and $(1, \alpha)$. Now see how close this "quarter circle" comes to the real quarter circle defined by $x^2 + y^2 = 1$, i.e. what is the value of α for which the Bezier curve most closely matches the quarter circle. You will find that you can get a match which is almost, but not quite, circular.

NURBS *can* be used to represent circles, and all of the other conics. NURBS surfaces can be used to represent quadric surfaces. As an example, let us consider one way in which NURBS can be used to describe a true circle. Rogers and Adams cover this on pages 371–375. The ways in which this is done require the designer to specify several things correctly at the same time, as we shall see. The details are so complicated that I would not expect you to remember it in an exam but I would expect you to remember that it can be done and have some idea of where to look it up if you needed it.

The method is as follows. Construct eight control points in a square. Let $\mathbf{P}_1$, $\mathbf{P}_3$, $\mathbf{P}_5$, and $\mathbf{P}_7$ be the vertices of the square. Let $\mathbf{P}_0$, $\mathbf{P}_2$, $\mathbf{P}_4$, and $\mathbf{P}_6$ be the midpoints of the respective sides, so that the vertices are numbered sequentially as you proceed around the square. Finally, you need a ninth point to join up the curve, so let $\mathbf{P}_8 = \mathbf{P}_0$.

Use a quadratic B-spline basis function with the knot vector
$[0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4]$. This means that the curve will pass through $\mathbf{P}_0$, $\mathbf{P}_2$, $\mathbf{P}_4$, $\mathbf{P}_6$ and $\mathbf{P}_8$, and allows us to essentially treat each quarter of the circle independently. That is, we can just examine $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$, along with the knot vector $[0, 0, 0, 1, 1, 1]$. If this makes a quarter circle then the other three quarters will also be correct.

We finally need to specify the homogeneous co-ordinates. As a circle is symmetrical it should be obvious that that $h_1 = h_3 = h_5 = h_7 = \alpha$ and $h_0 = h_2 = h_4 = h_6 = h_8 = \beta$. As we would like the curve to pass through the even numbered points we know that $\beta = 1$. All we therefore need to determine is α, the value of the odd numbered homogeneous co-ordinates.

If $\alpha = 1$ then the NURBS curve will bulge out more than a circle. If $\alpha = 0$, it will bow in. This gives us limits on the value of α. To find the exact value we take the NURBS curve definition for the quarter circle:

$$\mathbf{P}(t) = \frac{(1-t)^2\mathbf{P}_0 + 2\alpha t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2}{(1-t)^2 + 2\alpha t(1-t) + t^2}, 0 \le t < 1 \tag{108}$$

Assume now that $\mathbf{P}_0 = (0, 1)$, $\mathbf{P}_1 = (1, 1)$, and $\mathbf{P}_2 = (1, 0)$. Insert Equation 108 into the equation for the unit circle ($x(t)^2 + y(t)^2 = 1$). The resulting equation is:

$$\frac{((1-t)^2 + 2\alpha t(1-t))^2 + (2\alpha t(1-t) + t^2)^2}{((1-t)^2 + 2\alpha t(1-t) + t^2)^2} = 1, 0 \le t < 1 \tag{109}$$

Now solve this for α. Equation 109 is essentially:

$$\frac{a_N t^4 + b_N t^3 + c_N t^2 + d_N t + e_N}{a_D t^4 + b_D t^3 + c_D t^2 + d_D t + e_D} = 1, 0 \le t < 1 \tag{110}$$

From this we can conclude that we require $a_N = a_D$, $b_N = b_D$, $c_N = c_D$, $d_N = d_D$, and $e_N = e_D$. The first three all solve to give the result that $\alpha = 1/\sqrt{2}$, while the last two cancel out totally to give the tautology $0 = 0$. Thus $\alpha = 1/\sqrt{2}$.

This derivation is not at all intuitive and similar cleverness is required to handle representations of other conics. The beauty of NURBS is that they allow us to do this sort of thing and unify all shapes into a single representation. The difficulty is that, in order to achieve this unification, we need to have this rather complicated but general mathematical mechanism.

6.7 Exercises

1. Review from IB: What are homogeneous coordinates and what are they used for in computer graphics?

2. Explain how to use homogeneous coordinates to get rational B-splines given that you know how to produce non-rational B-splines.

3. What are the advantages of NURBS over Bezier curves? (i.e. why have NURBS, in general, replaced Bezier curves in CAD?)

4. Show that you understand why NURBS includes Uniform B-splines, Non-Rational B-splines, Beziers, lines, conics, quadrics, and tori.

5. [1998/7/12] Consider the design of a user interface for a NURBS drawing system. Users should have access to the full expressive power of the NURBS representation. What things should users be able to modify to give them such access and what effect does each have on the resulting shape? [6 marks]

6. For each of the items (in the previous question) that the user can edit: (i) Give sensible default values; (ii) Explain how they would be constrained if a 'demo' version of the software was to be limited to cubic Uniform Non-rational B-Splines.

7. [1999/7/11] (c) Show how to construct a circle using non-uniform rational B-splines (NURBS). [8 marks]
   Note: this question is ludicrously hard unless you remember the worked example in these notes or **R&A** pages 371-375.
   (d) Show how the circle definition from the previous part can be used to define a NURBS torus. [4marks]
   Note: you need explain only the general principle and the location of the torus' control points.

7 Subdivision surfaces

Subdivision schemes work by taking a coarse polygon mesh and introducing new vertices to create a finer mesh. Iterating this process several times creates a very fine mesh of polygons. Given that we are interested in drawing things only to a certain level of accuracy (there is no point in having polygons that are much smaller than pixels), the easily understood subdivision idea has definite benefits over the mathematically complicated B-spline methods. In fact, two of the standard subdivision schemes (Doo-Sabin and Catmull-Clark) produce, in the limit, B-spline surfaces (uniform quadratic and uniform cubic respectively) except at their extraordinary points. Some of the mathematical detail of subdivision surfaces is given below. **W&W** survey the field and the related mathematical tools.

Subdivision schemes have been around for a long time. Subdivision methods for curves were first mathematically analysed in 1947. Their use in computer graphics dates from 1974 when Chaikin used them to derive a simple algorithm for generating curves quickly. In 1978 Doo & Sabin and Catmull & Clark generalised Chaikin's work from curves to surfaces. Much work has been done since then, but it seems that it is only since about 2000 that subdivision schemes have had widespread use owing to Pixar's adoption of them.

Subdivision schemes are increasingly being used as an alternative to NURBS. They combine mathematical elegance with an exceptionally simple implementation. For curves, given an arbitrary control polygon, we use the positions of the current vertices to determine the location of the new vertices in a new, refined, more detailed, control polygon. Generally, each old vertex gives rise to two new vertices. For example, you could place new vertices one-quarter and three-quarters of the way between each adjacent pair of old vertices. Connecting all the new vertices together, in the appropriate order, produces a more refined control polygon. Repeat this process several times and you produce a very good approximation to the uniform quadratic B-spline curve defined by the original set of vertices. In the limit, the refined control polygon becomes this uniform quadratic B-spline curve. The Doo-Sabin subdivision method is the extension of this idea to surfaces, where the refined control polygon has four times as many vertices as the source control polygon. Given the simplicity of the implementation and the fact that you can stop whenever you like, you can see how attractive this method is for computer graphics.

7.1 Mathematical details: curves

Take an arbitrary polygon defined by the sequence of control points:

$$\mathbf{P}^i = (\ldots, \mathbf{p}^i_{-1}, \mathbf{p}^i_0, \mathbf{p}^i_1, \mathbf{p}^i_2, \ldots)$$

Subdivision maps this sequence of control points to a new sequence, $\mathbf{P}^{i+1}$ by applying subdivision rules. This process doubles[9] the number of points, and there is one rule for the odd numbered points and one for the even. For example, the subdivision rules on which the Doo-Sabin method is based are:

$$\mathbf{p}^{i+1}_{2j} = \frac{3}{4}\mathbf{p}^i_j + \frac{1}{4}\mathbf{p}^i_{j+1} \tag{111}$$

$$\mathbf{p}^{i+1}_{2j+1} = \frac{1}{4}\mathbf{p}^i_j + \frac{3}{4}\mathbf{p}^i_{j+1} \tag{112}$$

while the subdivision rules on which the Catmull-Clark method is based are:

$$\mathbf{p}^{i+1}_{2j} = \frac{1}{8}\mathbf{p}^i_{j-1} + \frac{6}{8}\mathbf{p}^i_j + \frac{1}{8}\mathbf{p}^i_{j+1} \tag{113}$$

$$\mathbf{p}^{i+1}_{2j+1} = \frac{4}{8}\mathbf{p}^i_j + \frac{4}{8}\mathbf{p}^i_{j+1} \tag{114}$$

As is the way with much mathematics, we can write it in a more compact, more general, but less obvious, form as:

$$\mathbf{p}^{i+1}_j = \sum_{k=-\infty}^{\infty} \alpha_{2k-j}\mathbf{p}^i_k \tag{115}$$

where the α_j are coefficients depending on the subdivision rules. Note that the index $2k - j$ alternately selects the even indexed α_j and the odd indexed α_j. So, the two schemes given above, can be compactly described as:

$$\alpha = \frac{1}{4}(\ldots, 0, 0, 1, 3, 3, 1, 0, 0, \ldots) \tag{116}$$

and

$$\alpha = \frac{1}{8}(\ldots, 0, 0, 1, 4, 6, 4, 1, 0, 0, \ldots) \tag{117}$$

respectively. You will recognise the sequences in parentheses as being two rows from Pascal's triangle.

It would now be constructive for you to draw an arbitrary control polygon and perform a couple of subdivision steps using the first of the two subdivision schemes. Once you feel happy that you understand what is going on, you may like to try the second scheme. For those for whom these two tasks seem simple, you may like to consider what happens if you try to use the previous row from Pascal's triangle (1,2,1) and, for even more excitement, what happens if you try to use the next row (1,5,10,10,5,1). Both produce valid subdivision methods, but you will find that (1,2,1) has a minimal effect on the *shape* of the control polygon.

[9] It doesn't quite double the number of points when the sequence is open and of finite length, but we will gloss over that at the moment.

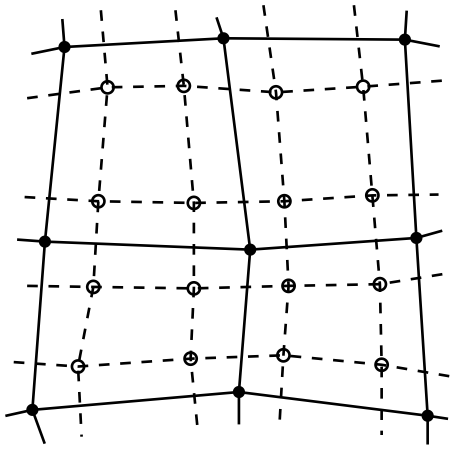
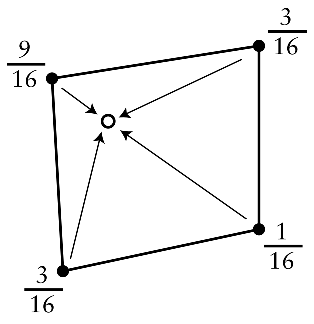



Figure 18: Doo-Sabin subdivision. On left a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). At right the weights used to generated one of the refined vertices.

7.2 Mathematical details: surfaces

The above subdivision methods can be easily extended from a control polygon to a quadrilateral mesh. This is a mesh where every polygon is a quadrilateral and every vertex is connected to four other vertices.

The Doo-Sabin subdivision method introduces four new vertices in each quadrilateral, and connects up vertices accordingly. The new vertices are blended mixtures of the old vertices in the proportions $9 : 3 : 3 : 1$ (derived from the tensor product of the univariate case: $3 \times 3 : 3 \times 1 : 1 \times 3 : 1 \times 1$). This is illustrated in Figure 18.

Catmull-Clark subdivision is not much more difficult to understand. The only difference here is that is not all of the new vertices are created using the same weights. A vertex is introduced in the centre of each quadrilateral, in the centre of each edge, and near to each old vertex. Each of these three types of vertex has a different set of weights as illustrated in Figure 19.

This all works beautifully for quadrilateral meshes. Now, suppose we have a quadrilateral mesh that contains extraordinary vertices, in other words a mesh that consists of quadrilaterals but has occasional vertices with other than four immediate neighbours. The Doo-Sabin scheme will still worked quite happily, because every polygon in the mesh is still quadrilateral. However, the Catmull-Clark subdivision scheme depends on every vertex having exactly four neighbours for the generation of the new vertex that is near to the old vertex position (the rightmost case in Figure 19). Catmull and Clark got around this problem by creating a new set of weights, one set of weights for each vertex valence (the valence of vertex is a number of other vertices to which it is connected). Instead of weights of $1/64$, $6/64$, and $36/64$ you can use weights of $1/4n^2$, $3/2n^2$, and $1 - 7/4n$, where n is the valence of the vertex. This particular set of weights was derived by Denis Zorin, other values can also be used.

However, this is not the only type of mesh with which we can deal. The Doo-Sabin scheme can be easily modified to cope with meshes in which some of the polygons are not quadrilateral, while still maintaining C^1-continuity everywhere. For a k-sided polygon,

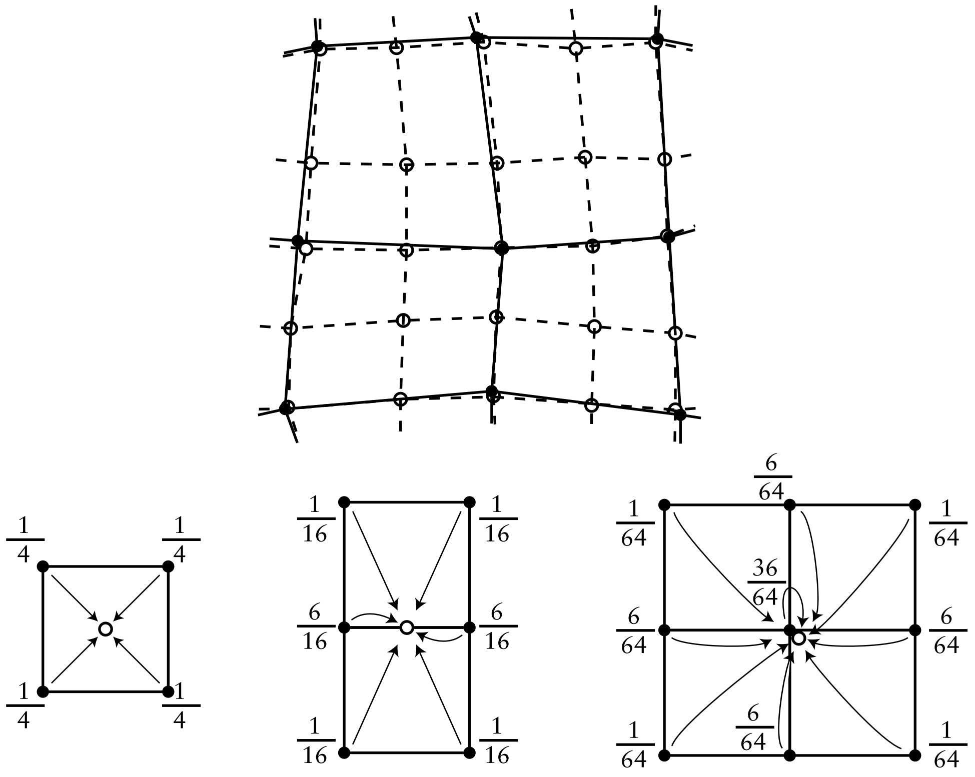


Figure 19: Catmull-Clark subdivision. Above: a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). Below: the weights used to generated each type of refined vertex: centre, edge, and modified old vertex.

the weights, α_k on the k vertices can be shown to be:

$$\alpha_0 = \frac{1}{4} + \frac{5}{4k} \tag{118}$$

$$\alpha_i = \frac{1}{4k}\left(3 + 2\cos\frac{2i\pi}{k}\right) \tag{119}$$

There are other schemes, notably the Loop scheme (named after Dr Loop) which works on triangular meshes. My reread group at the Computer Laboratory has been working on the theory of subdivision since 2000 and has produced some interesting results including some rather whacky alternative subdivision schemes.

7.2.1 Exercises

1. Do the “constructive” exercises at the end of section 7.1.

2. Explain how Doo-Sabin subdivision works for an arbitrary polygon mesh.

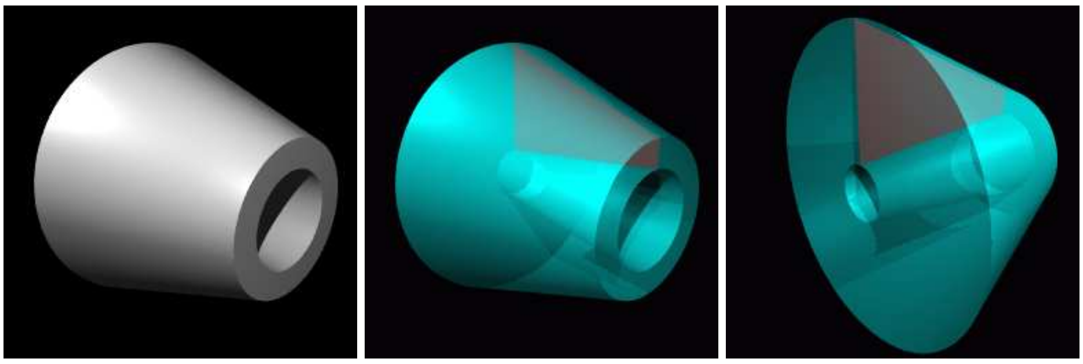


Figure 20: Left: an example surface of revolution. Middle and right: two views of its generating polygon (the red quadrilateral), with the generated 3D object shown in semi-transparent cyan.

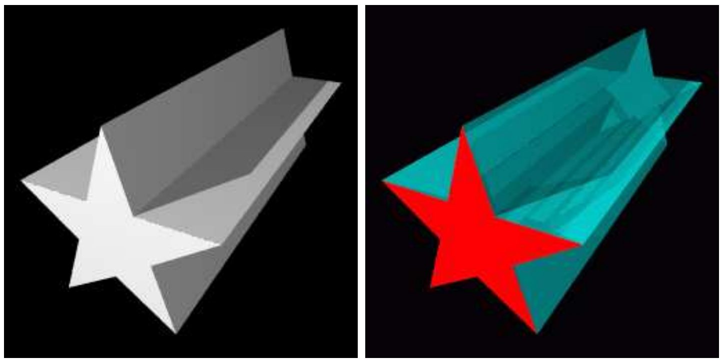


Figure 21: Left: an example extrusion. Right: its generating polygon (the red star), with the generated 3D object shown in semi-transparent cyan.

8 Other 3D modelling mechanisms

8.1 Sweeps

These are three dimensional objects generated by *sweeping* a two dimensional shape along a path in 3D. Two special cases of the *sweep* are *surfaces of revolution*, where the path is a circle (see Figure 20); and *extrusions*, where the path is a straight line (see Figure 21). Some surfaces can be generated in more than one way, as illustrated in Figure 22. Surfaces of revolution are covered in **R&A** section 6-2. More general *sweeps* are covered in **R&A** section 6-3 and **FvDFH** section 12.4. Parts of Sections 6-2 and 6-3 of **R&A** are included in the handout.

If we push the idea of a sweep to its limit we can think of many things which could be modified to produce a three dimensional swept shape:

- **Cross section** This is the two dimensional shape that is to be swept along the sweep path. It does not have to be circular. Figure 23 shows two swept objects, one with a circular cross-section, one with a polygonal cross-section.

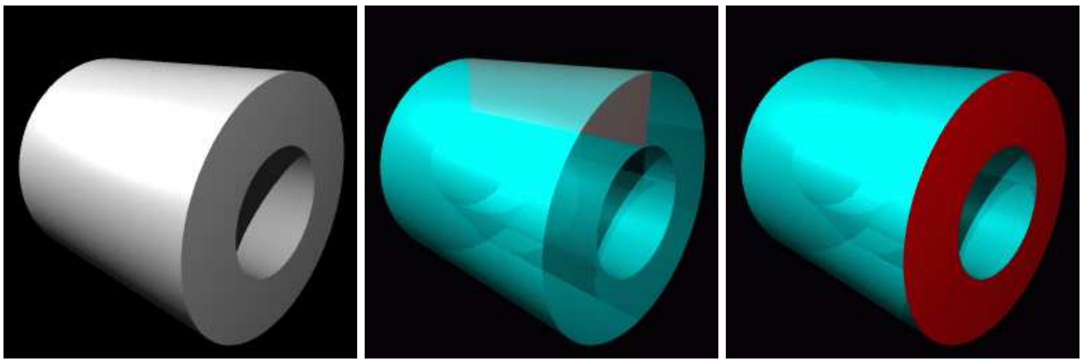


Figure 22: Left: an example object which could be an extrusion or a surface of revolution. Middle: generated as a surface of revolution from a rectangle. Right: generated as an extrusion from an annulus.

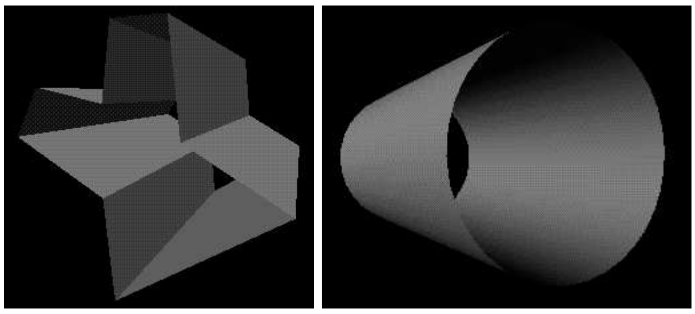


Figure 23: Two simple swept objects, one with a polygonal cross-section, one with a circular cross-section.

- **Sweep path** This is the path along which the two dimensional cross section is swept to produce the three dimensional shape. It may be any curve. Figure 24 shows two views of the same swept object: a polygonal cross-section is swept along a convoluted path.
- **Twist** This is how the cross section twists (rotates) as it moves along the sweep path. The default would be to have no twist at all. Figure 25 shows a swept object with and without some twist.
- **Scale** This is how the cross section scales (changes size) as it moves along the sweep path. The default would be to have it stay the same size along the whole path. Figure 26 shows a cylinder, and the same cylinder with different scales along its length.
- **Normal vector direction** The normal vector of the 2D cross section will usually point along the sweep path at each point. Changing this will change the nature of the swept object. See **R&A** Figure 6-17 (in the handout) for an example.

You may be able to think of parameters, other than those in the list above, which could be modify. The examples given here were all generated using a generalised cylin-

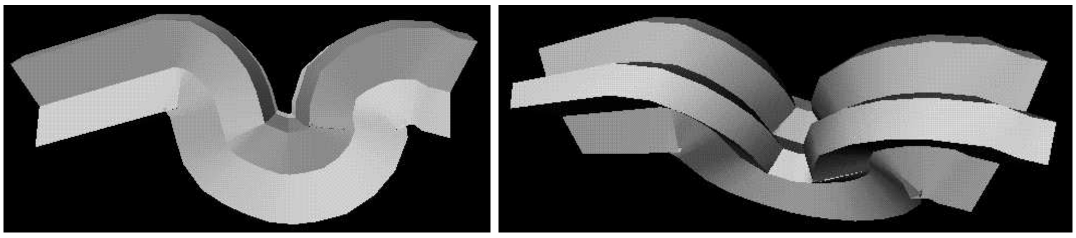


Figure 24: Two views of a polygonal cross-section swept along a path.

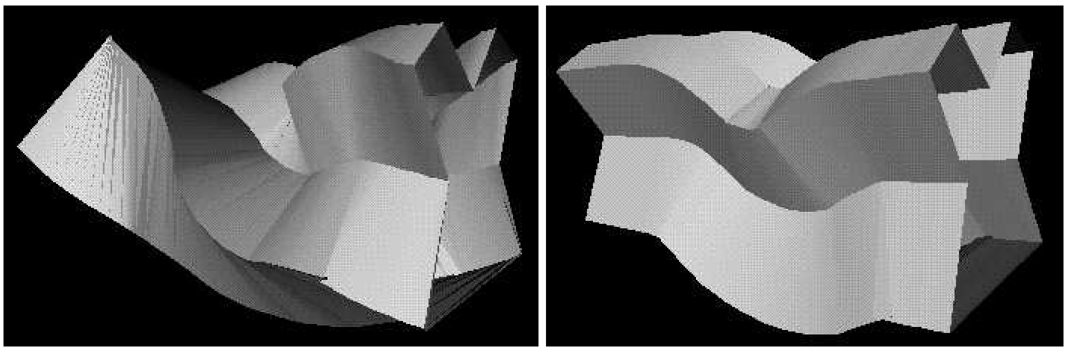


Figure 25: A polygonal cross-section swept along a path with (left) and without (right) twist.

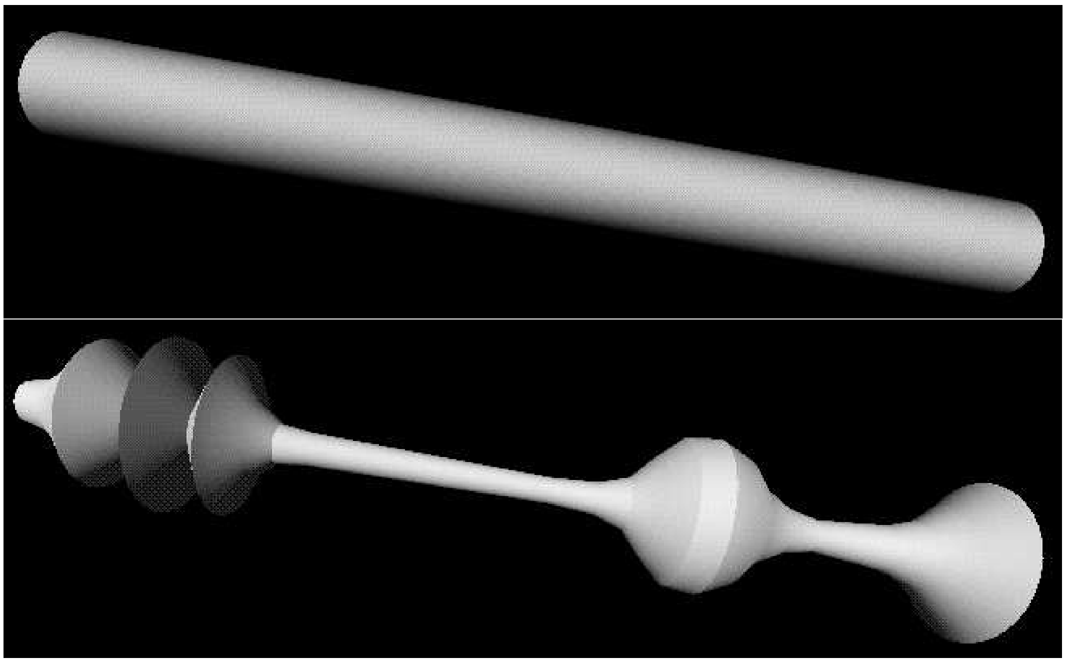


Figure 26: A cylinder (top) swept along a path with scale changes (bottom).

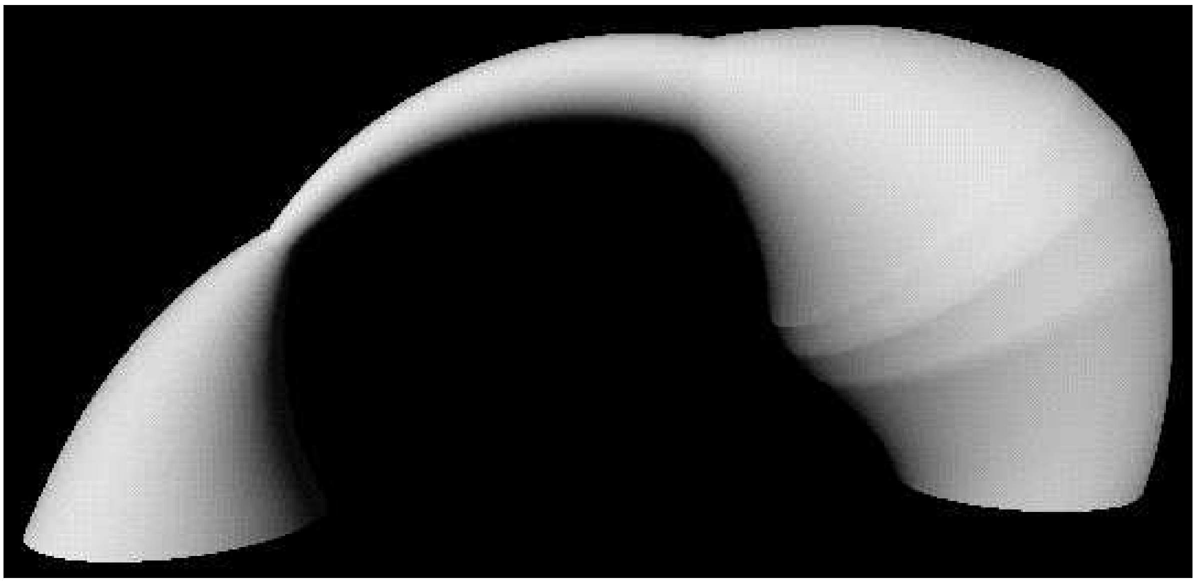


Figure 27: A swept object with a circular cross-section, semi-circular path, and varying scale.

der tool from Silicon Graphics would could vary cross-section, path, twist and scale. This is sufficient for most purposes. The input to such a system will either be NURBS curves or polylines. The output will be a NURBS surface or a polygon mesh.

8.1.1 Sweeping with NURBS

A NURBS surface is defined as the same two-dimensional extension to NURBS curves described in Equation 95, though obviously carried out in homogeneous co-ordinates. You can define sweeps using NURBS curves by using one NURBS curve as the sweep path, and another NURBS curve as the cross-section. You take the tensor product of the two curves. This is covered in **R&A**, pages 445–456, 465–477.

8.1.2 Converting swept objects to polygons

Swept objects are hard to ray trace. Imagine trying to write a ray/object intersection algorithm for a general swept object. This means that we generally need to polygonise swept objects in order to render them. For **polygon scan conversion** we obviously must convert them to polygons.

A swept surface may be easily converted to polygons by converting the outline of the 2D cross section to a polygon, and converting the sweep path to connected set of line segments. Moving the polygon to each vertex of the set of line segments, and connecting vertices accordingly, will produced a polygon mesh which approximates the swept surface.

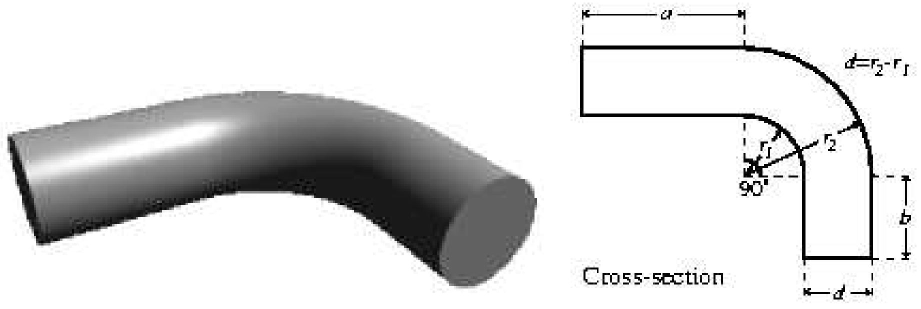


Figure 28: The object from the 1998 exam question.

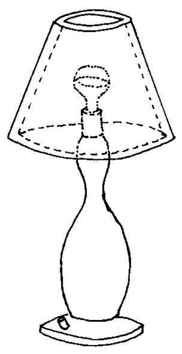


Figure 29: A lamp.

8.1.3 Exercises

1. [1998/7/12] Show how the object in Figure 28 can be represented as a swept object. Show also how to convert the swept object into polygons. Explain what extra work would you need to do if you had to convert it into triangles.

2. Use the following different methods of specifying a geometrical model for the object in Figure 29 (assuming it is a three dimensional model and not a line drawing). Come as close as you can to the original for each of the methods, and describe the difficulties in using a particular method for this model.

   (a) Extrusions

   (b) Surfaces of revolution

   (c) Sweeps along straight lines with scale changes.

3. For each of the following categories list five real-world objects which could be represented by the primitives in the category.

   (a) The ray-tracing primitives in section 3.3

   (b) Extrusions

   (c) Surfaces of revolution

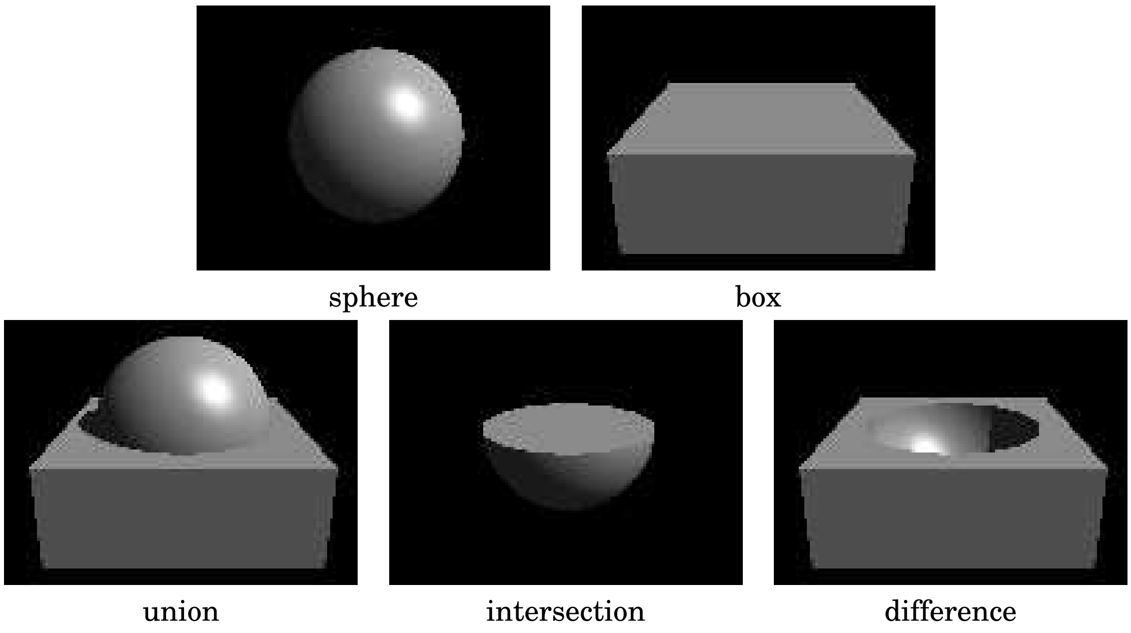


Figure 30: CSG example: two primitives (a sphere and a box) under the three CSG operations: union, intersection, and difference.

(d) General sweeps

4. A flume (water tunnel) at a swimming complex is modelled as a circle swept along a particular path. The designers also want to model the volume swept out by a person travelling down the flume. We can approximate the cross-section of a person with something roughly elliptical and we'll assume the 'virtual' person doesn't move legs or arms while hurtling along. Explain which parameters in the list would be need to be modified to specify the shape of the flume and which would need to be modified to model the volume swept out by a person travelling down the flume (alternatively, specify which parameters would be held constant, in each case, for the entire length of the sweep).

8.2 Constructive Solid Geometry

Constructive solid geometry (CSG) essentially consists of Boolean set operations on closed primitives in 3D space. CSG is covered in **FvDFH** sections 12.7 and 15.10.3.

The three CSG operations are *union*, *intersection* and *difference*. Figure 30 illustrates the three CSG operations in use on simple three dimensional primitives. Figure 31, based on **FvDFH** Plate III.2, shows an object for which CSG is (probably) the only sensible modelling technique. The object rendered in the right-hand image is constructed from the primitives shown in the left-hand image. It is mostly made out of cylinders, but you will recognise the extruded star from Figure 21. The cover of Hofstadters "Gödel, Escher, Bach" has a carved wood shape which casts shadows of the letters G, E, and B onto three orthogonal planes. In Figure 32, I try this treatment on my initials. Unfortunately the letters N, A, and D are not as amenable to this as the letters G, E, and

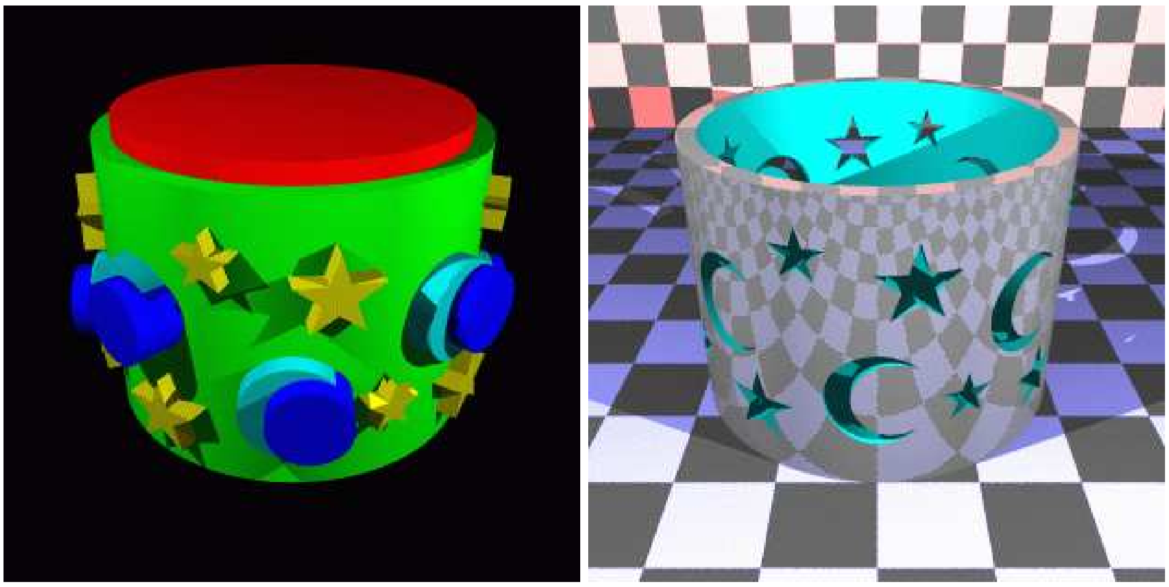


Figure 31: CSG example. Left: the primitives. Right: the finished object. It is difficult to see how this object could be produced easily in any other way.

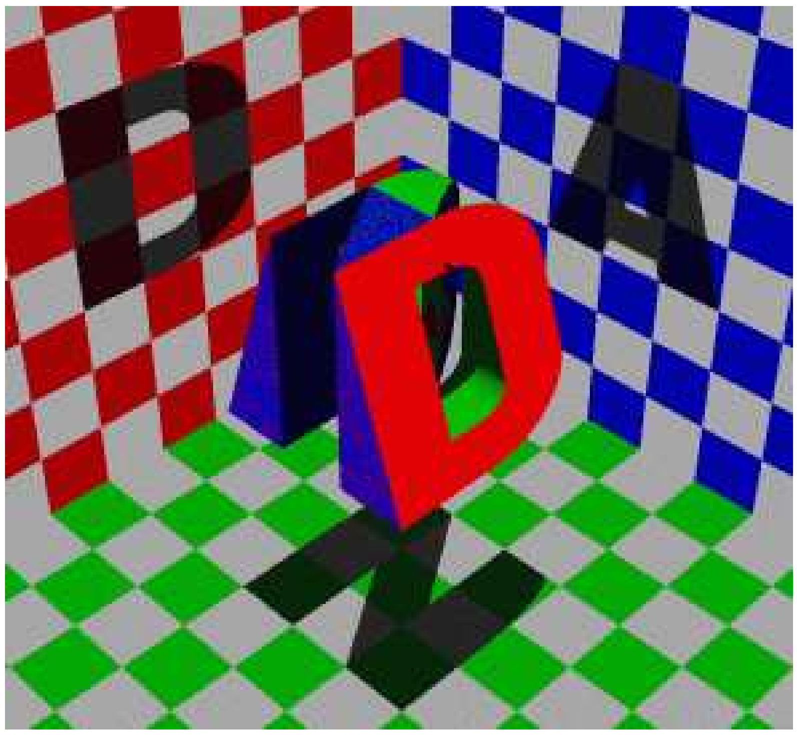


Figure 32: A CSG shape casting three shadows of three different letters.

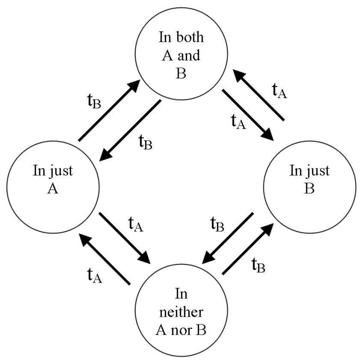


Figure 33: The simple state machine used in the CSG algorithm.

B: notice that the shadow of the N has a slight curve at its top right, owing to the N's intersection with the curve on the D and the slope on the A. Various other arrangements of the three letters were tried, all of which gave more noticeable artefacts than this. Each of the letters is a CSG object (the D, for example, is constructed from cylinders and boxes). The final effect is produced simply by intersecting the three letters.

More information is required to fully understand how CSG can be raytraced. Some of this is given in **SSC** Chapter 18, some below, and more will be said in lectures.

8.2.1 Some algorithmic details

This section deals with how to raytrace a CSG object. A CSG object can be represented as a binary tree where each node is one of the three CSG operators: union ($\cup$), intersection ($\cap$), and difference ($\backslash$). The leaves of the tree are primitive objects for which a ray-object intersection test exists. The basic information which is passed up the tree is a list of *all* intersection point, t, between the ray and the object represented by the sub-tree. Combining two sub-trees requires you to sort through these lists of intersection points, keeping those from both sub-trees which are intersection points with the combined object and discarding the others.

CSG of two objects, A and B, can be driven by a simple state machine, where there are four states:

- In both A and B.
- In just A.
- In just B.
- In neither A nor B.

See Figure 33. As one progresses along the ray, each intersection point changes the state. Each of the four possible operations ($\cup$, $\cap$, $A\backslash B$, $B\backslash A$) requires you to keep the intersection points which transfer you into or out of just one of the four states. Thus $A\backslash B$ is defined by the transitions into and out of the "In just A? state, while intersection is defined by the transitions into and out of the "In both A and B? state.

```
fun inter( (inA, tA::tAs), (inB, tB::tBs) )
    =   if( tA < tB ) then
            if( inB ) then
                tA :: inter( (not inA, tAs), (inB, tB::tBs) )
            else
                inter( (not inA, tAs), (inB, tB::tBs) )
        else
            if( inA ) then
                tB :: inter( (inA, tA::tAs), (not inB, tBs) )
            else
                inter( (inA, tA::tAs), (not inB, tBs) )
|   inter( (inA, []), (inB, tB::tBs) )
    =       if( inA ) then
                tB :: tBs
            else
                []
|   inter( (inA, tA::tAs), (inB, []) )
    =       if( inB ) then
                tA :: tAs
            else
                []
|   inter( (inA, []), (inB, []) )
    =           [] ;

fun intersection( (inA, tAs), (inB, tBs) )
    =   ( inA andalso inB, inter( (inA, tAs), (inB, tBs) ) ;
```

Figure 34: An ML function for calculating the CSG intersection of two objects

A ray's interaction with an object, A, can be stored as an ordered pair: `( inA : Boolean, tAs : float list )` where `inA` is `True` if the eye is inside object A and `False` otherwise. `tAs` is a list of t-values for which the ray intersects the surface of the object, sorted in increasing order. Of course, all t values are non-negative.

Figure 34 gives an ML function for finding the Constructive Solid Geometry intersection of two objects: $A \cap B$. You could try writing an equivalent function to find the difference of two objects, $A \backslash B$.

8.2.2 Exercises

1. Work out how to represent a Lego technic brick as a CSG Object (see Figure 35). You may assume that you have box and cylinder primitives.

2. [1998/7/12] Work out how to represent the object in Figure 28 using CSG. You may assume the following primitives: sphere, cylinder, cone, torus, box.

3. [1999/9/4] (c) List the three ways of combining objects using constructive solid geometry (CSG). Describe how an object built using CSG can be represented using a

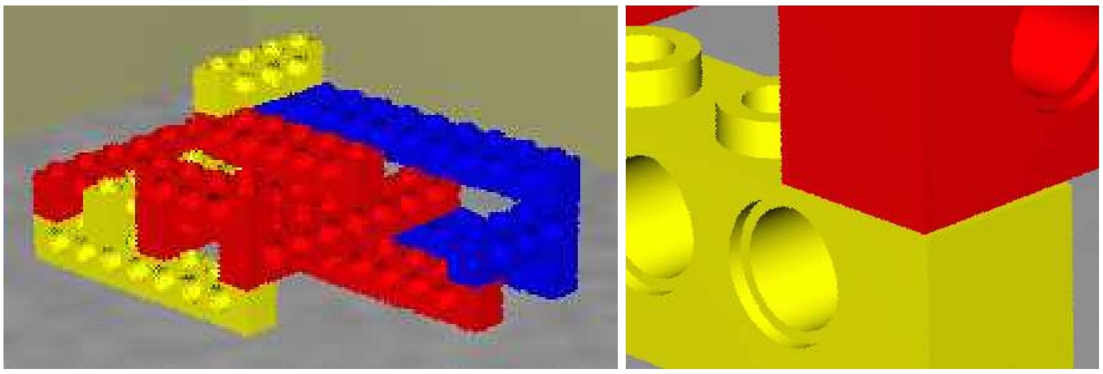


Figure 35: Some CSG Lego technic bricks.

binary tree. Given the intersection points of a ray with each primitive in the tree, explain how these points are passed up the tree by each type of combination node to produce a list of intersection points for the whole CSG object. [8 marks]

4. [2002/8/4] **(c)** Describe how an object built using constructive solid geometry (CSG) can be represented using a binary tree. Given the intersection points of a ray with each primitive in the tree, show how to calculate the first intersection point of the ray with the entire CSG object. [6 marks]

5. Write a function, similar to that in section 8.2.1 for $A \cap B$, to find the difference of two objects, $A \backslash B$.

8.3 Implicit surfaces, voxels and the marching cubes algorithm

8.3.1 Implicit surfaces

These will be described in lectures. Brian Wyvill has a good introduction to implicit surfaces on his website at the University of Calgary. Click on "Implicit Tutorial" for the introduction to implicit surfaces.

8.3.2 Voxels and the marching cubes algorithm

Voxels are the three dimensional analogue of pixels. Rather than storing a colour in a voxel, you will generally store a density value. Voxels and the marching cubes algorithm are both covered in Lorenson and Cline's 1987 SIGGRAPH paper "Marching cubes: a high resolution 3D surface construction algorithm", *Proc SIGGRAPH 87*, pages 163-169. This paper is included in the handout. One of the reasons for including the paper is to give you a taster of what a good graphics research paper looks like. Two of the exercises relate to evaluating this paper in terms of (i) its research content and (ii) its writing style.

8.3.3 Exercises

1. Give a definition of an implicit surface and give three examples of where such things might be useful.

2. Explain how voxel data can be thought of a defining an implicit surface (or surfaces). Explain, conversely, an implicit surface can be converted into voxel data.

3. [2001/7/9] (a) The marching squares algorithm is a two-dimensional version of marching cubes, where you generate line segments in 2D rather than triangles in 3D. It could be used, for example, where you have a regular grid of height values and want to draw contours of constant height. Sketch an implementation of this two-dimensional marching squares algorithm. [6 marks]

4. Medical data is captured in slices. Each slice is a 2D image of density data. The distance between slices may be different to the distance between the pixels within a slice (for example, see Lorenson and Cline, Section 7.1, p. 167). What effect, if any, does this difference have on the voxel data? What effect, if any, does it have on the marching cubes algorithm?

5. Consider Lorenson and Cline, Section 6. This research was done about fifteen years ago. Given your knowledge of processor performance, what differences in performance would you expect to see between then and now?

6. Lorenson and Cline is an example of a graphics research paper. Critically evaluate Lorenson and Cline. How good is this piece of research?

7. Research papers at the SIGGRAPH conference are limited in their length. Evaluate Lorenson and Cline in terms of the following questions. What has been left out that would have been useful? What has been included that could have been left out? Where could the explanation have been better? Are any of the figures extraneous? Where would an extra figure have been helpful? For light relief, list any grammatical or spelling errors that you find (there is at least one of each).

8. [2002/8/4] (b) Implicit surfaces are normally combined by adding the field functions together to create a "blobby" blended surface. Describe an alternative mechanism (or mechanisms) for combining implicit surfaces which would produce results more akin to CSG union and intersection. Explain why it produces these results. Given this mechanism, suggest a way of combining implicit surfaces to produce a result similar to CSG difference. [4 marks]