

Advanced Graphics Lecture Notes

Neil Dodgson*

University of Cambridge Computer Laboratory

Overview

The course is in two halves. The first taught by Prof. Neil Dodgson, the second by Dr Alex Benton. This first half of the course covers the three industry-standard ways of representing three-dimensional surfaces: polygon meshes, NURBS, and subdivision surfaces.

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 12 lectures long. This year, it is mainly concerned with 3D modelling techniques, so the course title is a little misleading.

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, unless I say otherwise, 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 but cannot be put on the web for copyright reasons. There are exercises at the end of each section (subsections [1.6](#), [2.3](#), [4.4](#), [5.4](#), [6.2](#), [7.3](#)). My thanks to Dr Jonathan Pfautz and Dr Andy Penrose 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. The NURBS part of the course is mainly based on material from **R&A**. You should be able to find some of these books in your College library.

- **R&A** Rogers, D.F. & Adams, J.A. (1990). *Mathematical Elements for Computer Graphics*. McGraw-Hill (2nd ed.). A good coverage of the mathematics of the 2D and 3D representation of shape as it was understood in the year of publication.

*Written 10/99, modifications made 09/00, 10/02, 09/04, 04/06, 03/07, 01/10, 03/12.©1999, 2000, 2002, 2004, 2006, 2007, 2010, 2012 Neil A. Dodgson

Explains Bézier, B-spline, and NURBS curves and surfaces in great detail. Also covers conics and quadrics.

- **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 terse 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 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 than **FvDFH**. It covers all the basics and also has sections on Constructive Solid Geometry (Ch. 18), Quadrics (also Ch. 18), Radiosity (Ch. 15), and an introduction to Bézier and B-Spline curves and surfaces (Ch. 19).
- **Buss** Buss, S.R. (2003). *3-D Computer Graphics*. Cambridge University Press. A book that has the best description of radiosity (Ch. XI) that I have ever read. It also contains chapters on Bézier curves (VII), B-Splines (VIII), ray tracing (IX and X) and animation (XII).
- **Farin** Farin, G. (2002, 5th ed.; 1997, 4th ed.). *Curves and Surfaces for CAGD*. Morgan Kaufmann (5th ed.). Academic Press (4th ed.). An alternative source for information on Bézier, B-Spline, NURBS, and conics. Requires more fluency with mathematics than **R&A**. Regularly updated since its original publication in 1988.
- **W&W** Warren, J. & Weimer, H. (2002). *Subdivision Methods for Geometric Design*. Morgan Kaufmann. The first book devoted entirely to subdivision methods.
- **P&R** Peters, J. & Reif, U. (2008). *Subdivision Surfaces*. Springer. The second book on subdivision. Aimed more at mathematical proof than **W&W**.
- **GG I-V** *Graphics Gems* volumes I (1990) to V (1995). Academic 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.

Note on copyright material I have included, in this handout, an extract from Rogers and Adams (**R&A**), which consists of *parts* of sections 5-8 (Bézier curves), 5-9 (B-splines), and 5-13 (NURBS). This is 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

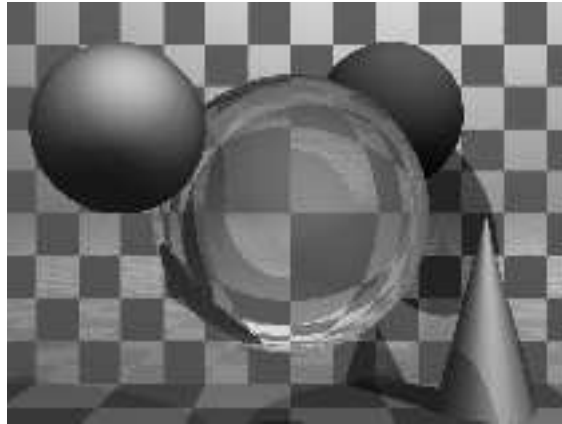


Figure 1: A basic ray traced model showing refraction and shadowing.

material I suggest that you read this extract and then borrow a copy of **R&A** in order to go through the examples.

1 Basic 3D modelling

1.1 Ray tracing vs polygon scan conversion

Ray tracing and polygon scan conversion are the two standard methods of producing images of three-dimensional solid objects. They were covered in the Part IB course. If you want to revise them then read through **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 in Section 1.4.

1.2 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 that 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.

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, the illumination at that point, and contributions from any reflection or refraction that occurs 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.

Ray tracing's big disadvantage is that it is slow. It takes minutes, or hours, to render a detailed scene. The quality of the images that can be produced is high, compared



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



Figure 3: A close up of the kitchen sink.

with polygon scan conversion. 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. A Cambridge company (Advanced Rendering Technology) investigated hardware-assisted ray tracing in the late 1990s and other researchers have used multiple processors (dozens to hundreds) to do ray tracing, 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. Polygon scan conversion uses a variety of tricks to immitate these special effects, including using parts of the ray tracing algorithm where it proves more effective than any other solution. This means that, in practice, pure ray tracing is used by a minority of movie and television special effects companies, advertising companies, and enthusiastic amateurs.

The kitchen in Figure 2 was rendered in 1990 using the ray tracing program *Rayshade*. At the time, *Rayshade* was able to produce images that could not be created efficiently by polygon scan conversion. However, *Rayshade* has not been updated since 1992. A more recent ray tracer 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 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

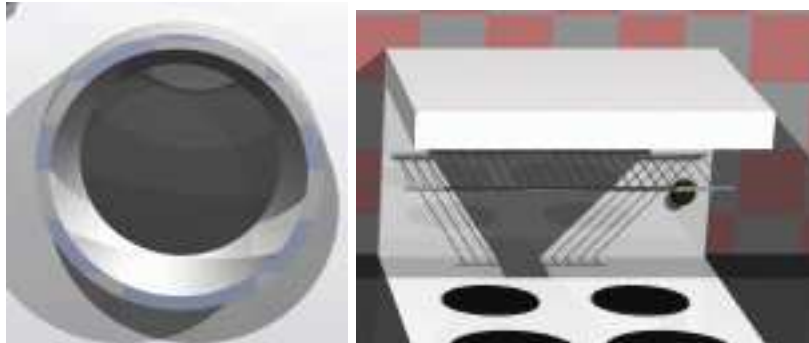


Figure 4: Close up views of the washing machine door and the grill on the stove.

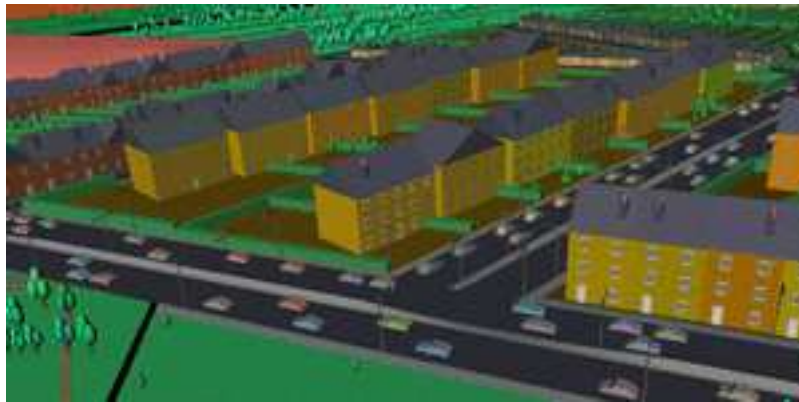


Figure 5: A scan converted model of a city (courtesy of Jon Sewell).

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 computational solid geometry. 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.

1.3 Polygon scan conversion

This term encompasses a range of algorithms where polygons are rendered, one at a time, into a frame buffer. The term *scan* is historic: when the original algorithm was developed, displays were cathode ray tubes, which display images as a sequence of horizontal *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 assume that polygon scan conversion refers to the *z*-buffer algorithm or one of its derivatives, such as the A-buffer. We will also assume that the algorithm is implemented in hardware on a graphics card.

The advantage of polygon scan conversion is that it is fast. Polygon scan conversion

algorithms are essential for 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 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 and is capable of being implemented as a parallel algorithm. For example, the [NVIDIA GeForce GTX 680](#) graphics processing unit (GPU), released in March 2012, has 1536 CUDA cores running at just over 1GHz. It can render up to 128.8 billion textured pixels per second, and can render animated scenes containing many millions of triangles in real time. While we might hope that scientific or medical applications were considered important applications of computer graphics, it is the game industry that is driving the development of graphics card technology.

One challenge with polygon scan conversion is that the basic algorithms support only simplistic lighting models, so images do not necessarily look realistic. For example: transparency can be supported, but refraction requires the use of a texture-mapping 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”. All of these are more complicated methods than those used in ray tracing. Where ray tracing is a clean and simple algorithm, polygon scan conversion uses a variety of tricks of the trade to get 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 unrealistic if you use largish polygons to model natural objects such as humans or animals. The solution, feasible with today’s technology, is to use polygons that are no bigger than a pixel. An image generated using a polygon scan conversion algorithm is usually distinguishable from a photograph, but it is becoming increasingly possible to produce images indistinguishable from reality, at least at a first glance.

1.4 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, in particular in the computer software used to design 3D models.

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 control mesh of the B-spline or subdivision surface (see Sections [5](#) and [7](#)).

Line drawing was historically faster than polygon scan conversion. However, modern graphics cards handle both lines and polygons at about the same speed.

To illustrate the speed with which computer graphics has developed, Compare a modern computer game with what was possible in the late 1970s. A modern game can produce near-photo-realistic animated graphics in real time. The first edition of [R&A](#), published in 1976, used only line drawing algorithms to illustrate its 3D models. Only one figure in the entire book did not use line drawing: Fig. 6-52, which had screen shots of a *prototype* polygon scan conversion system.



Figure 6: Screen shots from commercial flight simulators, circa 1995. The graphics used today are better than this, but not as good as you see in video games: most aircraft cockpits are a long way away from the scenery for most of the time, and pilots spend most of their time looking at the instruments, rather than looking out the window.

A second illustration of the development of computer graphics is in the movie industry. *Star Wars* [1977] and *Alien* [1979] were the first movies to use computer graphics. In both movies, the “computer graphics” were short sequences of line drawn graphics, where each frame took several minutes to render. Thirty years later, *Avatar* [2009] used real-time 3-D computer graphics during shooting, and non-real-time rendering of photo-realistic shaded computer graphics for most of the movie.

1.5 Applications of computer graphics

Almost all applications of computer graphics use polygon scan conversion.

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. Such visualisation therefore uses polygon scan conversion, although some data require *voxel* rendering.

Simulation uses polygon scan conversion because it can generate images at interactive speeds. Early advances in computer graphics (in the 1970s and 1980s) were driven by the commercial flight simulator market. At this expensive end of the market, a great deal of computer power used to be needed to do the graphics. The most expensive flight simulators are those with full hydraulic suspension and a full field-of-view out the cockpit windows. In 1990 such a simulator would 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. A commercial flight simulator today will still cost you about £10M, but less than £10k will be spent on the graphics kit. Figure 6 shows screen shots from two commercial flight simulators in the mid-1990s; Figure 7 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.

Computer games are a particular form of interactive simulation. They use polygon scan conversion because it gives interactive speeds.



Figure 7: The real cockpit of a commercial flight simulator: an exact replica of the equivalent airplane's cockpit.

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 visual effects for film and television. Most visual 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. 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 special 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 twenty-fifth anniversary re-release of the original Star Wars trilogy included a number of computer graphic enhancements, all of which were composited into the original movie.

Twenty years after *Star Wars*, we saw computer graphics effects of the kind found in movies such as *Starship Troopers* [1997]. Most of the giant insects in the movie were completely computer generated. The spaceships were a combination of computer graphic models and real models. The largest of these real models was 18' (6m) long: so it was obviously still worthwhile spending a lot of time and energy on the real thing.

Effects are not necessarily computer generated. The movie industry distinguishes between *special effects*, which are done on set, and *visual effects*, which are done in postproduction on a computer. Compare *King Kong* [1933] with *King Kong* [2005]. The plot has barely changed, but the effects have improved enormously: changing from hand

animation (and a man in a monkey suit) to computer generated imagery.

Not every 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 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* [2001–3], almost anything which could be shot in live action was shot that 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 flowing down Mount Doom 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 composited to make the final image.

Completely computer-generated movies have been with us since 1995. *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]). These were followed by *Toy Story 2* [1999], *Dinosaur* [2000], *Shrek* [2001], *Monsters Inc* [2001], *Ice Age* [2002], *Finding Nemo* [2003], and *Shrek 2* [2004]. The genre is now well established and there are many more recent examples. Note the subject matter of the early movies: toys, bugs, dinosaurs, monsters, sea life, and fairytale characters. It is very difficult to model humans realistically and research is still being undertaken in the field of realistic human modelling. *Final Fantasy* [2001] was the first serious attempt to represent fully human characters in a fully computer-generated movie. When we look at it today we see that the quality is not quite up to the standard that we see today in the top video games. *Avatar* [2009] successfully produced computer-generated (pink-skinned) humans that looked realistic, though you can still see something “not quite right” in still shots of the computer-generated human characters, and the majority of the shots of the (pink-skinned) humans were taken of live humans on a set, rather than depending on the computer.

In terms of computer time, both *Toy Story* [1995] and *Avatar* [2009] used about a century of CPU time for the final rendering. Compare the quality and the complexity of what is rendered to judge how far the industry moved in the 14 years between those two movies.

1.6 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. “The quality of the visual 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 of polygon edges) 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 if the back sides of polygons are actually visible on screen.

The normal vector of a polygon 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, **A**, **B** and **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.

An alternative is to have these things specified at the vertices (normal vector, diffuse colour, specular colour, texture co-ordinates). This is particularly useful for Gouraud or Phong shading, and particularly when the polygons have been generated from some curved surface for which we can calculate the true normal vector at each vertex.

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 more about the connectivity of

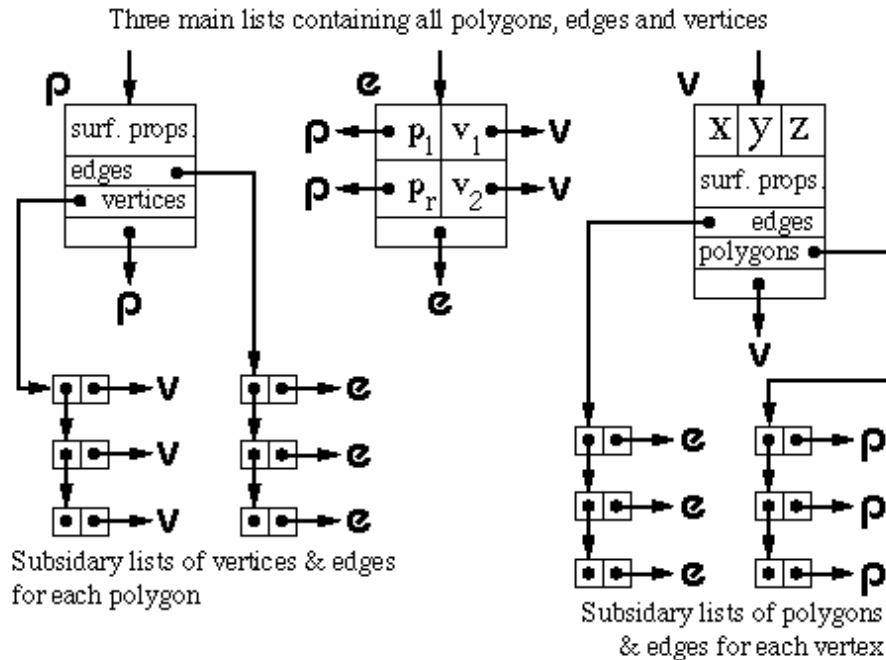


Figure 8: One version of the winged edge data structure.

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 useful for handling polygon mesh data. The version shown in Figure 8 is a highly-detailed structure, in that it 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 8 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. **F&vD** section 13.2 and **SSC** pp. 170–172 also contain some 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. It can be defined by these rules:

1. A vertex belongs to at least two edges.
2. A vertex is a vertex of at least one polygon.
3. An edge has exactly two end points.
4. An edge is an edge of either one or two polygons.
5. A polygon has at least three vertices.
6. A polygon has at least three edges.

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 **AMD** Radeon and the **NVIDIA** GeForce families of graphics cards, can be thought of as consisting of a *geometry* engine and a *rendering* engine. The geometry engine handles the transformations of all vertices and normals, and some of the shading calculations: doing calculations for each vertex. The rendering engine will implement the polygon scan conversion algorithm on the transformed data: doing calculations for each pixel. Early graphics cards (1980s and 1990s) were purely for hardware acceleration, with a fixed pipeline and fixed algorithm. Modern graphics cards allow for user programming. 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 in what he or she can program. However, the most recent graphics cards allow for a good deal of flexibility. Early generations of programmable cards allowed a limited number of instructions. For example, the NVIDIA GeForce 3 card (2001) had a maximum of 256 instructions in the whole program, no more than twelve working registers, no jumps or loops, no access to general memory. The NVIDIA GeForce 8 cards (2007) had thousands of registers and allow up thousands of 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

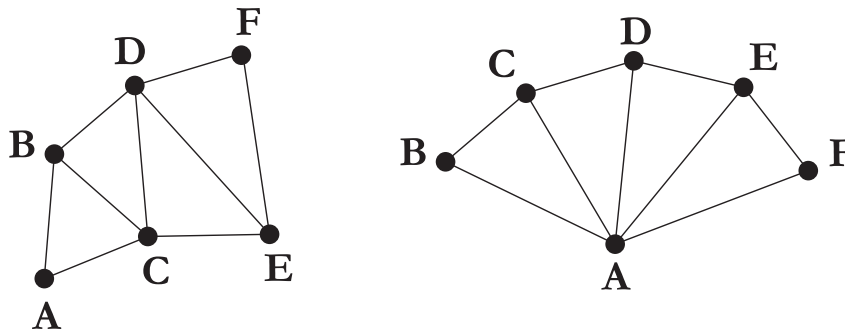


Figure 9: Left: a triangle strip set. Right: a triangle fan set.

whether or not to execute any given instruction. The latest generations of NVIDIA and AMD graphics engines have progressed (or reverted?) to a *unified shader* model of processing, where any processor can handle either the geometry processing or the pixel processing. This allows more efficient distribution of the processing load as appropriate to the objects being rendered on the screen.

Graphics cards are now sufficiently powerful that they are also used as general purpose co-processors for a variety of mathematics-intense computation tasks, using texture buffers for storing intermediate results.

To give you an idea of the complexity which is possible, on the GeForce4 generation of NVIDIA cards (2002), the information that was passed to the geometry engine, for a single vertex, was 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 size¹.

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 Triangle strip sets, triangle fan sets, and the vertex cache

In addition to simple triangle drawing, rendering hardware may also implement some way of caching vertices that will be used in multiple polygons in order to speed up processing through the geometry engine by preventing the engine from processing the same vertex more than once.

Early versions of this were the triangle strip set and triangle fan (see Figure 9). Each triangle in the set has two vertices in common with the previous triangle. Each vertex

¹You 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.

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.

The vertex cache, introduced around 2000, generalises this idea. Rather than caching just two vertices, the initial versions of the vertex cache held the twenty most recently used vertices, hence obviating the need to be explicitly specify fan sets and strip sets. It still requires you 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. 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 be 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).

3. [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 Introduction to splines

While polygons are good for rendering, we need some better way of generating curved surfaces. A designer cannot be expected to manipulate, directly, the millions of polygons that comprise the rendered model. We need a general way to specifying arbitrary curved surfaces, which can then be converted to polygons for rendering. Ideally, we want a 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: Bézier surfaces, B-spline surfaces (including NURBS) and subdivision surfaces. The Computer-Aided Design (CAD) industry uses NURBS surfaces as its standard definition mechanism. The visual effects industry uses both NURBS surfaces and subdivision surfaces.

The course handout contains a slide presentation introducing the concepts in this part of the course. Please look at this before continuing.

4 Bézier curves

Bézier 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. An extract from this Section of **R&A** is included in the handout. Please read that extract before continuing.

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

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

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

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

$$\text{linear} \quad \mathbf{P}(t) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \quad (2)$$

$$\text{quadratic} \quad \mathbf{P}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \quad (3)$$

$$\text{cubic} \quad \mathbf{P}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \quad (4)$$

4.1 Ways of thinking about Bézier curves

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

Linear interpolation. Equation 2 is obviously a linear interpolation between two points.

Equation 3 can be rewritten as a linear interpolation between linear interpolations between points:

$$\mathbf{P}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \quad (5)$$

Equation 4 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 Bézier 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 Bézier curve is guaranteed to lie within the convex hull of its control points.

Refinement of the control polygon. A Bézier curve can be seen as some sort of refinement of the polygon made by connecting its control points in order. The Bézier 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.

R&A list the properties of the Bézier curve on page 291.

4.2 Continuity

One of the most important problems of using Bézier curves (and surfaces) is getting different pieces of curve (or patches of surface) to connect smoothly together, that is: with continuity of position (C^0), slope (C^1) and curvature (C^2). These are continuity of the function, its first and its second derivatives, respectively. Much of the ensuing discussions consider how to achieve such continuity.

You should note that each Bézier curve is independent of any other Bézier curve. If we wish two Bézier 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 Bézier 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 Bézier curves of the same order: $P(t)$, defined by (P_0, P_1, \dots, P_n) , and $Q(t)$, defined by (Q_0, Q_1, \dots, Q_n) . C^0 -continuity (continuity of position) can be achieved by setting $P(1) = Q(0)$. This gives a formula for Q_0 in terms of the P_i s:

$$Q_0 = P_n. \quad (6)$$

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

$$Q_1 - Q_0 = P_n - P_{n-1} \quad (7)$$

Combining Equations 7 and 6 gives a formula for Q_1 in terms of the P_i s:

$$Q_1 = 2P_n - P_{n-1} \quad (8)$$

$$= P_n + (P_n - P_{n-1}) \quad (9)$$

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

$$Q_2 - 2Q_1 + Q_0 = P_n - 2P_{n-1} + P_{n-2} \quad (10)$$

Combining Equations 10, 7, and 6 gives a formula for Q_2 in terms of the P_i s:

$$Q_2 = 4P_n - 4P_{n-1} + P_{n-2} \quad (11)$$

$$= P_{n-2} + 4(P_n - P_{n-1}) \quad (12)$$

4.3 Bézier surfaces

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

$$\mathbf{P}(s, t) = \sum_{i=0}^n \binom{n}{i} (1-s)^{n-i} s^i \sum_{j=0}^n \binom{n}{j} (1-t)^{n-j} t^j \mathbf{P}_{i,j} \quad (13)$$

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

4.4 Exercises

1. Explain what C_0 -, C_1 -, C_2 -, C_n -continuity mean.
2. Equations (7) and (10) give the constraints on control point positions which ensure that Bézier curves join with C_1 - and C_2 -continuity. Derive these equations for two quartic Bézier curves by using the fact that the end-points must be identical, $\mathbf{Q}_0 = \mathbf{P}_n$, and then setting $\mathbf{P}'(1) = \mathbf{Q}'(0)$ and $\mathbf{P}''(1) = \mathbf{Q}''(0)$.

5 B-splines

B-splines are covered in some detail below and also in **R&A** Section 5-9. An extract from this Section of **R&A** is included in the handout. Please read that extract before continuing here. Beware that none of the worked examples are in the handout. You may find these helpful but 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 Bézier curves. In a B-spline each control point is associated with a *basis function*, $N_{i,k}$.

$$\mathbf{P}(t) = \sum_{i=1}^{n+1} N_{i,k}(t) \mathbf{P}_i, t_{\min} \leq t < t_{\max} \quad (14)$$

There are $n + 1$ control points, $\mathbf{P}_1, \mathbf{P}_2, \dots, \mathbf{P}_{n+1}$. 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 not dependent on the number of control points (which it is for Bézier curves, where k must always equal $n + 1$).

Equation 14 defines a piecewise continuous function. The $N_{i,k}$ are defined by a *knot vector*, $(t_1, t_2, \dots, t_{k+(n+1)})$, which we will consider in detail below. The knot vector specifies the values of the parameter t at which the pieces of curve join, by analogy to knots joining bits of string. It is necessary that:

$$t_i \leq t_{i+1}, \forall i \quad (15)$$

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:

$$\begin{aligned} N_{i,1}(t) &= \begin{cases} 1, & t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases} \\ N_{i,k}(t) &= \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) \end{aligned} \quad (16)$$

This is essentially a modified version of the idea of taking linear interpolations of linear interpolations of linear interpolations. Note the convention that $0/0 = 0$, which is justified formally by considering limits as the spacing between knots approaches zero.

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:

1. Each $N_{i,k}(t)$ depends only on the $k + 1$ knot values from t_i to t_{i+k} .
2. $N_{i,k}(t) = 0$ for $t < t_i$ or $t \geq t_{i+k}$ so P_i only influences the curve for $t_i \leq t < t_{i+k}$.
3. $P(t)$ is a polynomial of order k (degree $k - 1$) on each interval $t_i \leq t < t_{i+1}$.
4. Across the knots $P(t)$ is C^{k-2} -continuous.
5. $P(t)$ is continuous in all its derivatives *between* the knots.
6. A weighted sum of points only makes sense if the weights sum to one. $P(t)$ is therefore validly defined only where

$$\sum_{i=1}^{n+1} N_{i,k}(t) = 1. \quad (17)$$

This is the range $t_{\min} \leq t < t_{\max}$ where $t_{\min} = t_k$ and $t_{\max} = t_{n+2}$.

Those are the key properties of B-splines. Even more properties are described in **R&A** pp. 306–7.

5.1 The knot vector

The above introduction shows that the knot vector is important. The knot vector can 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

$$t_{i+1} - t_i = \text{constant}, \forall i \quad (18)$$

For example:

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

All of the basis functions, $N_{i,k}$ for a given k , are just shifted versions of one another and so implementation is relatively easy.

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

$$\begin{aligned} t_i &= t_1, & i &\leq k \\ t_{i+1} - t_i &= \text{constant}, & k &\leq i < n + 2 \\ t_i &= t_{k+(n+1)}, & i &\geq n + 2 \end{aligned} \quad (19)$$

For example:

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

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

Non-uniform. This is the general case, the only constraint being the standard $t_i \leq t_{i+1}, \forall i$ (Equation 15). For example:

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

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 does not 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.

5.1.1 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 $[1, 2, 3, \dots, k + (n + 1)]$ and Equation 16 simplifies to:

$$\begin{aligned} N_{i,1}(t) &= \begin{cases} 1, & i \leq t < i + 1 \\ 0, & \text{otherwise} \end{cases} \\ N_{i,k}(t) &= \frac{t - i}{k - 1} N_{i,k-1}(t) + \frac{i + k - t}{k - 1} N_{i+1,k-1}(t) \end{aligned} \quad (20)$$

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 **R&A** Figure 5-36.

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 other 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 (**R&A**, 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. (**R&A**, Figure 5-44).

Joining the ends. You can join the ends of the curve to make a closed loop. Say you have M points, P_1, \dots, 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): $P_1, \dots, P_M, P_1, \dots, P_{k-1}$. Your knot vector will thus have $M + 2k - 1$ uniformly spaced knots.

5.1.2 Open uniform knot vector

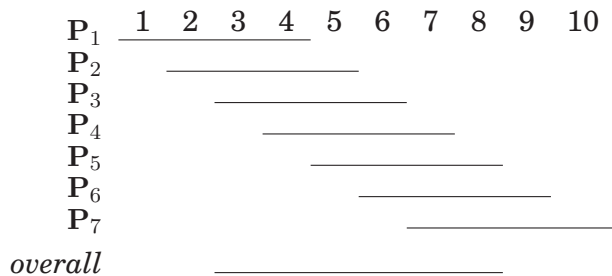
The final paragraph in the previous section tells you that uniform B-splines can be used to describe closed curves: all you have to do is join the ends as described above. If you do not want a closed curve, and you use a uniform knot vector, you find that you need to specify control points at each end of the curve which the curve does not go near (e.g. **R&A**, 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. **R&A**, 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 Bézier 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 $[0, 0, 0, 1, 1, 1]$.

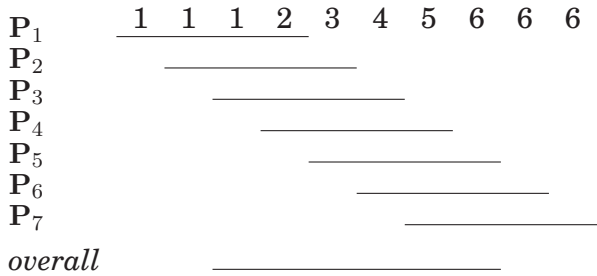
5.1.3 The difference between uniform and open uniform

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 P_i is non-zero. The B-spline itself (the *overall* line in the diagram) is defined over the range $t_3 \leq t < t_8$, i.e. over the range $3 \leq t < 8$.

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



The B-spline itself is defined over the range $t_3 \leq t < t_8$, i.e. over the range $1 \leq t < 6$. By the definition of an 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 $t_k + n + 1$.

5.1.4 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. Owing to the knot vector being scale- and translation-invariant, there are only three cases to consider: (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.

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 seems little point in doing so. However, there are some who advocate using some form of non-uniform knot spacing where knots are spaced relative to the spacing of the associated control points. Having said that, the main use of non-uniform B-splines seems to be 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.

5.1.5 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.

5.1.6 What should the defaults be?

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

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

5.2 B-spline patches

We generalise from B-spline curves to B-spline surfaces in the same way as we did for Bézier patches. Take a tensor product of two versions of Equation 14.

$$\mathbf{P}(s, t) = \sum_{i=1}^{m+1} \sum_{j=1}^{n+1} \mathbf{P}_{i,j} N_{i,k}(s) N_{j,l}(t), \quad s_{\min} \leq s < s_{\max}, t_{\min} \leq t < t_{\max} \quad (21)$$

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 $(m+1) \times (n+1)$.

5.3 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 invariant with respect to affine transforms

5.4 Exercises

1. How many control points are required for a quartic Bézier 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. Work out $N_{1,1}$, $N_{2,1}$, $N_{3,1}$, $N_{1,2}$, $N_{2,2}$, $N_{1,3}$ for the knot vector $[0, 2, 3, 6]$. Draw the graphs of these functions. [This is the exercise on page 18.]
5. [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]
6. [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 $n \geq 2k - 3$? [6 marks]
 (b) Explain what is different in the cases where $n < 2k - 3$ compared with the cases where $n \geq 2k - 3$. [3 marks]
 (c) Sketch the different basis functions for $k = 2$ and $k = 3$ (when $n \geq 2k - 3$). [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 Bézier curve. [7 marks]
7. [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 NURBS

NURBS are covered below and in some detail in **R&A** Section 5-13. An extract of this Section of **R&A** are included in the handout. Please read that before continuing here.

Non-uniform rational B-splines are the industry-standard for computer-aided design (CAD). 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.

NURBS surfaces are usually rendered by converting them to lots of small polygons and then using polygon scan conversion. They can also be 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, Bézier 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.**

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 22). The antonym of *rational* is *non-rational* (i.e. a non-rational B-spline is just a polynomial, see Equation 14). 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 14) 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:

$$(x', y', z', w) \rightarrow \left(\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w} \right) \quad (22)$$

Last year we said that the inverse transform was:

$$(x, y, z) \rightarrow (x, y, z, 1) \quad (23)$$

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

$$(x, y, z) \rightarrow (xh, yh, zh, h) \quad (24)$$

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:

$$\mathbf{P}_H(t) = \sum_{i=1}^{n+1} N_{i,k}(t) \mathbf{C}_i, t_{\min} \leq t < t_{\max} \quad (25)$$

Compare Equation 25 with Equation 14 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 22 to Equation 25. In order to better explain what is going on, we

first write Equation 25 in terms of its individual components. Equation 25 is equivalent to:

$$x'(t) = \sum_{i=1}^{n+1} x_i h_i N_{i,k}(t) \quad (26)$$

$$y'(t) = \sum_{i=1}^{n+1} y_i h_i N_{i,k}(t) \quad (27)$$

$$z'(t) = \sum_{i=1}^{n+1} z_i h_i N_{i,k}(t) \quad (28)$$

$$h(t) = \sum_{i=1}^{n+1} h_i N_{i,k}(t) \quad (29)$$

Equation 22 tells us that, in 3D:

$$x(t) = x'(t)/h(t) \quad (30)$$

$$y(t) = y'(t)/h(t) \quad (31)$$

$$z(t) = z'(t)/h(t) \quad (32)$$

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

$$\mathbf{P}(t) = \frac{\sum_{i=1}^{n+1} N_{i,k}(t) \mathbf{P}_i h_i}{\sum_{i=1}^{n+1} N_{i,k}(t) h_i}, t_{\min} \leq t < t_{\max} \quad (33)$$

This looks a lot more fierce than Equation 25, 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 33 becoming one, and the NURBS equation (Equation 33) therefore reducing to the non-rational B-spline equation (Equation 14).

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 **R&A** Figure 5-58 for an example, and play with an on-line NURBS tutorials such as the one mentioned above.

6.1 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 Bézier curve cannot exactly represent a circle. An interesting exercise is to place a cubic Bézier 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 Bézier 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. **R&A** 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, the easiest thing to do is set $\beta = 1$. 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:

$$\mathbf{P}(t) = \frac{(1-t)^2\mathbf{P}_0 + 2\alpha t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2}{(1-t)^2 + 2\alpha t(1-t) + t^2}, 0 \leq t < 1 \quad (34)$$

Assume now that $\mathbf{P}_0 = (0, 1)$, $\mathbf{P}_1 = (1, 1)$, and $\mathbf{P}_2 = (1, 0)$. Insert Equation 34 into the equation for the unit circle $(x(t)^2 + y(t)^2 = 1)$. The resulting equation is:

$$\frac{((1-t)^2 + 2\alpha t(1-t))^2 + (2\alpha t(1-t) + t^2)^2}{((1-t)^2 + 2\alpha t(1-t) + t^2)^2} = 1, 0 \leq t < 1 \quad (35)$$

Now solve this for α . Equation 35 is essentially:

$$\frac{a_N t^4 + b_N t^3 + c_N t^2 + d_N t + e_N}{a_D t^4 + b_D t^3 + c_D t^2 + d_D t + e_D} = 1, 0 \leq t < 1 \quad (36)$$

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 $0 = 0$. Thus² $\alpha = 1/\sqrt{2}$.

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.

6.2 Exercises

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

²If we had not set $\beta = 1$ above, then we would find that $\alpha = \beta/\sqrt{2}$.

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 Bézier curves? (i.e. why have NURBS, in general, replaced Bézier curves in CAD?)
4. Show that you understand why NURBS includes Uniform B-splines, Non-Rational B-splines, Béziers, 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. It would not be set as an exam question today.
 (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

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. In computer graphics, 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. This means that subdivision need only iterate until the polygons are about pixel-sized.

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.

Subdivision schemes are now the industry-standard modelling approach in computer animation and visual effects. NURBS remain the industry-standard approach in computer-aided design. The two approaches have the same foundations. The principle subdivision scheme used in practice is the *Catmull-Clark* scheme, invented in 1978, by Ed Catmull and Jim Clark, and commercialised in 1998 by Tony DeRose’s team at *Pixar*. In the limit, as we subdivide infinitely finely, the *Catmull-Clark* scheme produces exactly the same surface as the uniform cubic B-spline, except around the so-called *extraordinary points* of the surface.

W&W and **P&R** both survey the field and the related mathematical tools.

The course handout contains a slide presentation that presents the concepts from this Section of the notes.

7.1 Subdivision curves

Take an arbitrary control polygon, comprised of a set of vertices connected in sequence. We use the positions of the current vertices to determine the location of the new vertices in a new, refined, more detailed, control polygon (see Figure 10). The standard approach is for each old vertex to give 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 good approximation to the curve that you would get if you were able to subdivide infinitely many times. For practical purposes, we need subdivide only so enough times that we cannot tell the refined polygon from a curve. For computer graphics, that is when the polygon segments have become a little smaller than a pixel.

Let the initial control polygon be defined by the sequence of control points:

$$\mathbf{P}^i = (\dots, \mathbf{p}_{-1}^i, \mathbf{p}_0^i, \mathbf{p}_1^i, \mathbf{p}_2^i, \dots)$$

Subdivision maps this sequence of control points to a new sequence, \mathbf{P}^{i+1} by applying subdivision rules. This process doubles³ the number of points, and there is one rule for the odd numbered points and one for the even.

Chaikin's corner-cutting method introduces new points one-quarter and three-quarters of the way between each adjacent pair of old vertices. Informally, it "cuts the corners off" the original control polygon. Mathematically, it can be described thus:

$$\mathbf{p}_{2j}^{i+1} = \frac{3}{4}\mathbf{p}_j^i + \frac{1}{4}\mathbf{p}_{j+1}^i \quad (37)$$

$$\mathbf{p}_{2j+1}^{i+1} = \frac{1}{4}\mathbf{p}_j^i + \frac{3}{4}\mathbf{p}_{j+1}^i \quad (38)$$

Figure 10 shows a polygon defined by four points subdivided three times using Chaikin's corner cutting. The difference in the polygons in the final two iterations is already small. Figure 11(a) shows a larger example with four levels of subdivision.

In the limit, after infinitely many steps of subdivision, Chaikin's corner-cutting method produces a curve identical to the quadratic uniform B-spline. As you will notice from Figure 11(a), it does not take many steps to produce a curve that is extremely close to the limit curve. In practice, therefore, we might need to take five or six steps of subdivision to produce a curve that looks smooth on a computer screen (say 100 dpi), and only three or four more steps to produce a curve that looks smooth on a printout (say, 1200 dpi).

Chaikin's corner-cutting method can be generalised to surfaces, producing the *Doo-Sabin* surface subdivision scheme (see Section 7.2).

³It doesn't quite double the number of points when the sequence is open and of finite length. If we are dealing with a finite-length vector $\mathbf{P}^i = (\mathbf{p}_1^i, \mathbf{p}_2^i, \dots, \mathbf{p}_m^i)$, then the subdivided vector, using the rules in equations 37 and 38, would have $2m - 2$ points.

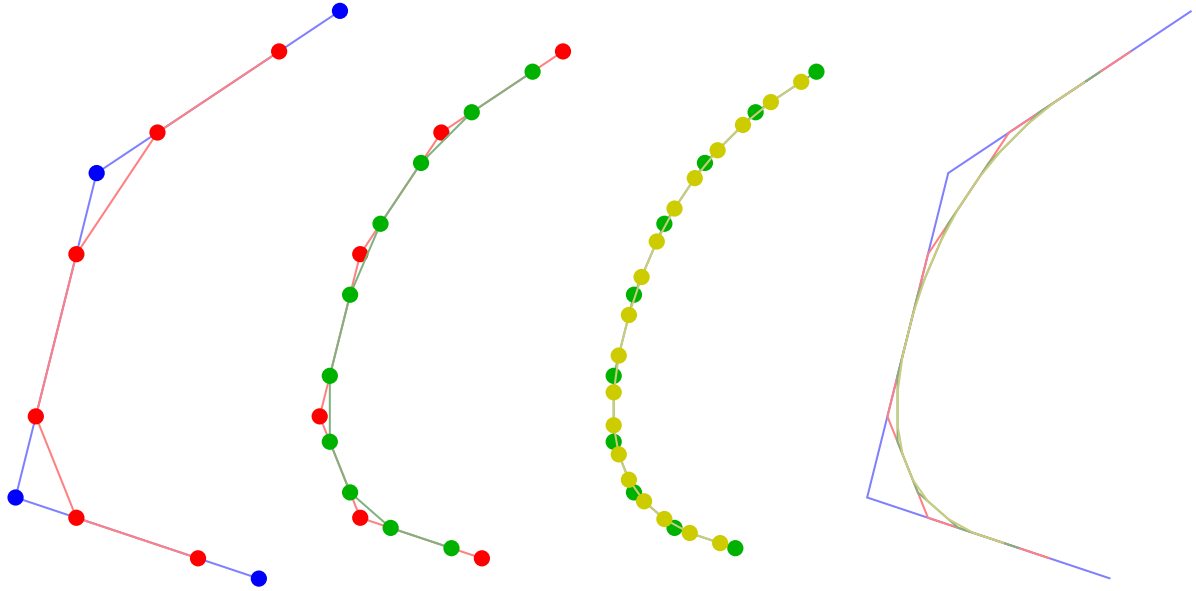


Figure 10: Chaikin's corner-cutting method. The first three diagrams show an original polygon with the subdivided version superimposed. The output polygon from the left hand diagram becomes the input to the second diagram, and its output becomes the input for the third. The right hand diagram shows all four polygons superimposed. The final two are very similar.

The industry-standard surface subdivision method is *Catmull-Clark* subdivision. The curve subdivision rules on which the Catmull-Clark surface method is based are only slightly more complicated than the ones we have seen above. They are:

$$\mathbf{p}_{2j}^{i+1} = \frac{1}{8}\mathbf{p}_{j-1}^i + \frac{6}{8}\mathbf{p}_j^i + \frac{1}{8}\mathbf{p}_{j+1}^i \quad (39)$$

$$\mathbf{p}_{2j+1}^{i+1} = \frac{4}{8}\mathbf{p}_j^i + \frac{4}{8}\mathbf{p}_{j+1}^i \quad (40)$$

An example of a polygon subdivided by these rules can be seen in Figure 11(b). The limit curve generated by these rules is the cubic uniform B-spline.

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

$$\mathbf{p}_j^{i+1} = \sum_{k=-\infty}^{\infty} \alpha_{2k-j} \mathbf{p}_k^i \quad (41)$$

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:

$$\alpha = \frac{1}{4}(\dots, 0, 0, 1, 3, 3, 1, 0, 0, \dots) \quad (42)$$

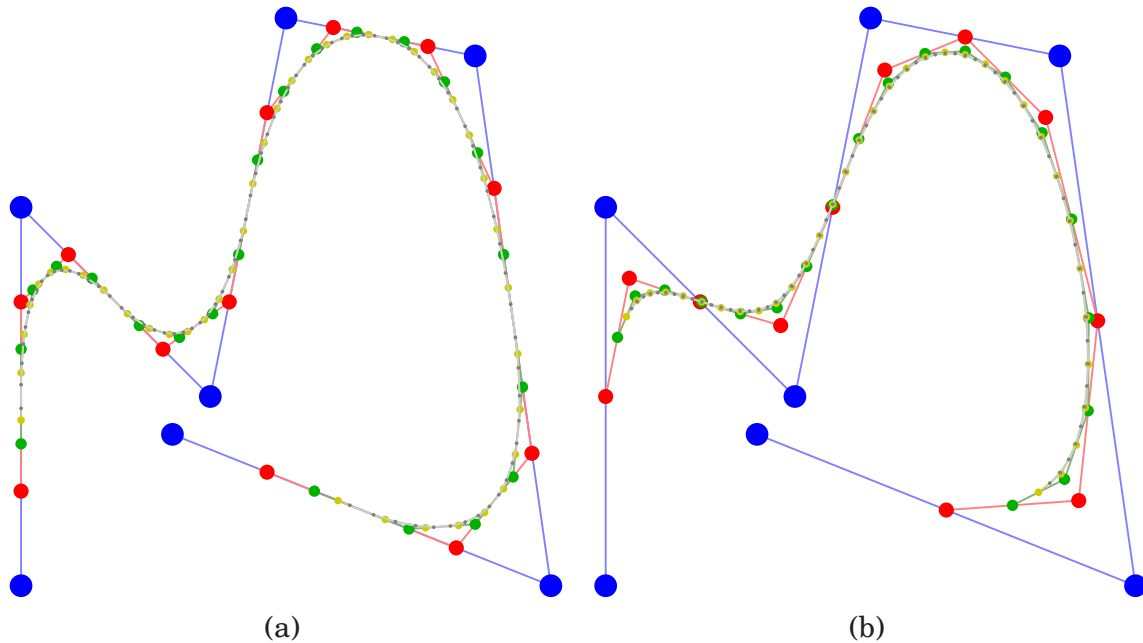


Figure 11: (a) An example of Chaikin's corner-cutting method, whose limit curve is the quadratic uniform B-spline. (b) An example of the subdivision method whose limit curve is the cubic uniform B-spline. In both cases, the original polygon is blue and there are four levels of subdivision: red, green, yellow, grey.

and

$$\alpha = \frac{1}{8}(\dots, 0, 0, 1, 4, 6, 4, 1, 0, 0, \dots) \quad (43)$$

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

7.2 Subdivision 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 for surfaces is a generalisation of Chaikin's corner cutting method for curves. It 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 12.

This all works beautifully for quadrilateral meshes. In this case it produces a limit surface that is identical to the uniform quadratic B-spline surface, defined with the same control points. The surface has C^1 -continuity everywhere, as this is a property of the uniform quadratic B-spline surface.

Now, suppose we have a quadrilateral mesh that contains *extraordinary polygons*, that is, polygons that are not quadrilateral, such as the pentagon in Figure 13. The Doo-Sabin scheme can be modified to cope with meshes that have such polygons. The

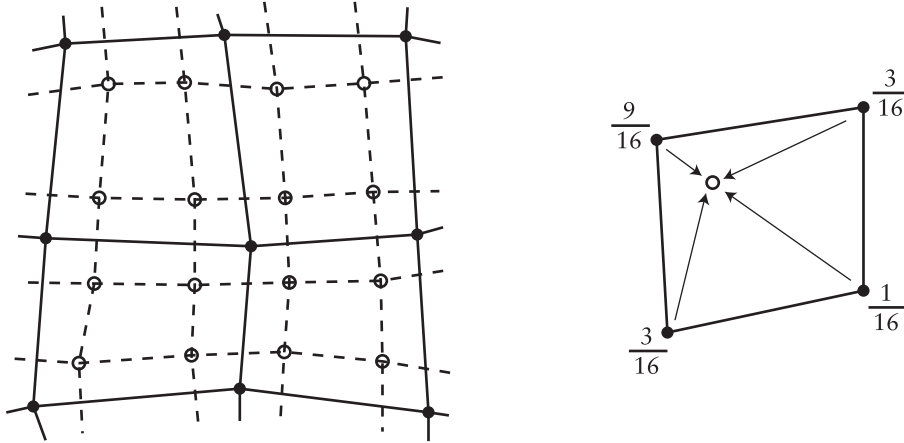


Figure 12: Doo-Sabin subdivision. On left a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). At right the weights used to generated one of the refined vertices.

resulting limit surface can be shown to be C^1 everywhere.

For a k -sided polygon, the weights, α_k on the k vertices need to be chosen. Doo and Sabin demonstrated that good results are achieved if we let:

$$\alpha_0 = \frac{1}{4} + \frac{5}{4k} \quad (44)$$

$$\alpha_i = \frac{1}{4k} \left(3 + 2 \cos \frac{2i\pi}{k} \right) \quad (45)$$

Now consider *extraordinary vertices*, that is, vertices with other than four immediate neighbours. You will see in Figure 13 that such vertices become extraordinary polygons in the first step of subdivision. It is impossible for the Doo-Sabin to introduce new extraordinary vertices: every vertex in the subdivided mesh has four neighbours. It is also impossible for the Doo-Sabin method to introduce new extraordinary polygons after the first step. At each step, every extraordinary polygon shrinks to a smaller polygon of the same type. All the new polygons are quadrilaterals. As we subdivide further, every extraordinary polygon becomes surrounded by a “sea” of quadrilaterals. Thus you can see that the limit surface is the uniform quadratic B-spline defined by all of these quadrilaterals, except at the centres of the extraordinary polygons, where additional mathematical analysis is needed to demonstrate that the limit surface is also C^1 at these points.

Catmull-Clark subdivision is a surface method that generalises the $\frac{1}{8}(1, 4, 6, 4, 1)$ subdivision curve method. It is similar in spirit to Doo-Sabin, with some important differences. The first difference is that there are three types of new vertex in the normal regions of the mesh: a vertex is introduced in the centre of each quadrilateral (a *face vertex*), in the centre of each edge (an *edge vertex*), and near to each old vertex (a *vertex vertex*). Each of these three types of vertex has a different set of weights as illustrated in Figure 14.

Catmull-Clark subdivision needs special rules for *extraordinary vertices*. The edge vertex and face vertex rules remain unchanged, but we need a new rule for the vertex

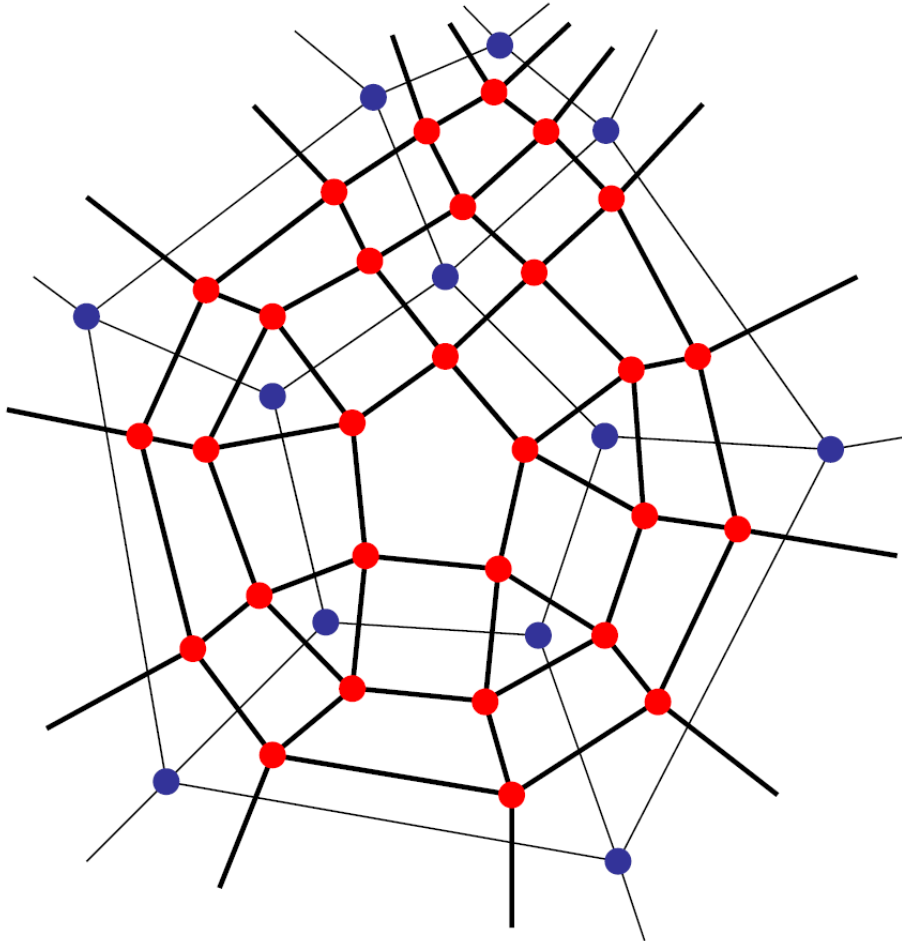


Figure 13: A Doo-Sabin mesh before (blue points, thin lines) and after (red points, thick lines) one level of subdivision. Notice that every polygon produces a smaller polygon of the same type (quadrilaterals produce quadrilaterals, pentagons produce pentagons), every edge generates a new quadrilateral, every vertex generates a polygon of the same valency, and that every new vertex is of valency four.

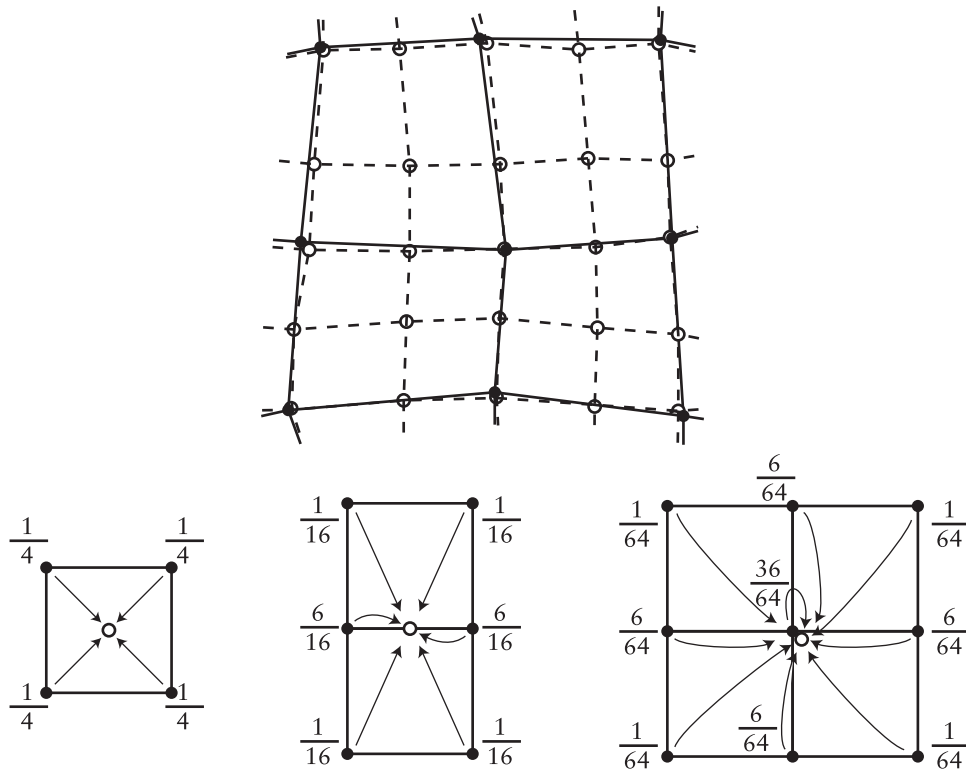


Figure 14: Catmull-Clark subdivision. Above: a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). Below: the weights used to generated each type of refined vertex: centre, edge, and modified old vertex.

vertex generated by the extraordinary vertex. Catmull and Clark solved this 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). There has since been considerable work on finding the optimal weights to give the best result. For example, instead of weights of $1/64$, $6/64$, and $36/64$ you can use Denis Zorin's weights of $1/4n^2$, $3/2n^2$, and $1 - 7/4n$, where n is the valence of the vertex.

The limit surface of the Catmull-Clark scheme is C^2 *almost* everywhere. It is, however, only C^1 at the limit positions of the extraordinary vertices. It has been proven that the only way to make a Catmull-Clark limit surface C^2 at the extraordinary vertices is for its curvature to be zero at those points. This produces unsightly “flat spots” that we would rather avoid. This means that we either have to accept “flat spots” or have to accept that the curve is not everywhere C^2 .

There are other subdivision schemes. The most notable are the *Butterfly* scheme, which interpolates the control points and the *Loop* scheme (named after Charles Loop), which works on triangular meshes.

7.3 Exercises

1. Draw an arbitrary control polygon and perform a couple of subdivision steps using the first of the two subdivision schemes above. Once you feel happy that you understand what is going on, you may like to try the second scheme.
2. Draw an arbitrary control polygon and consider what happens if you try to use the previous row from Pascal's triangle $\frac{1}{2}(1, 2, 1)$. You will find that $\frac{1}{2}(1, 2, 1)$ has a minimal effect on the *shape* of the control polygon.
3. What happens if you try to use the next row of Pascal's triangle, $\frac{1}{16}(1, 5, 10, 10, 5, 1)$? Which uniform B-spline do you think this produces in the limit?
4. Explain how Doo-Sabin subdivision works for an arbitrary polygon mesh.
5. Explain how the Catmull-Clark scheme must be modified to handle extraordinary polygons.