

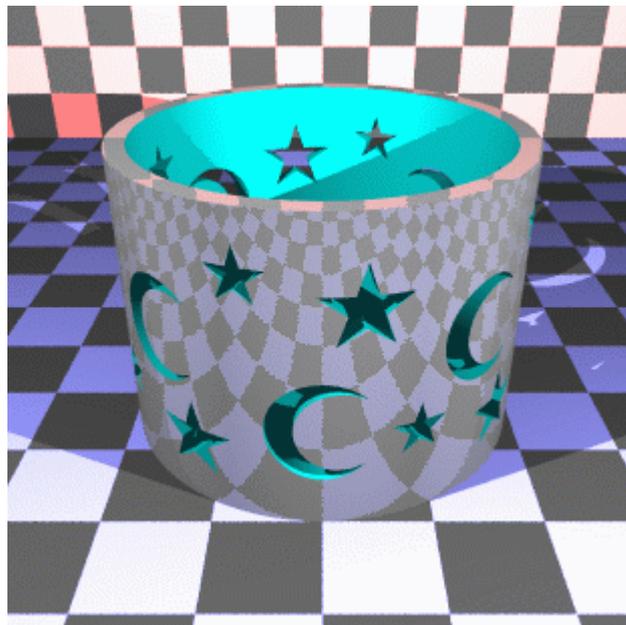
Advanced Graphics, part A



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Part II



Study Guide & Supplementary Material

Neil Dodgson, 2010

Computer Laboratory
15 J J Thomson Avenue
Cambridge CB3 0FD

E-mail: nad@cl.cam.ac.uk
<http://www.cl.cam.ac.uk/~nad/>

Contents

Advanced Graphics Lecture Notes, including exercises
N. A. Dodgson (33 pages)

An introduction to the surface representations used in CAD
N. A. Dodgson (5 sheets)

Advanced Graphics 2006: subdivision curves & surfaces
N. A. Dodgson (9 sheets)

Mathematical elements for computer graphics (extract)

D. F. Rogers & J. A. Adams (12 sheets) Not available in the online version owing to copyright

Other useful material

The course web pages:

<http://www.cl.cam.ac.uk/Teaching/current/AdvGraph/>

Advanced Graphics Lecture Notes

Neil Dodgson*

University of Cambridge Computer Laboratory

Overview

This first half of the course provides students with a solid grounding in a variety of three-dimensional modelling mechanisms.

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. Today, it is mainly concerned with 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, 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 web¹. 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 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.

- **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.

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

¹<http://www.cl.cam.ac.uk/Teaching/current/AdvGraph>

- **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 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 Bézier and B-Spline curves and surfaces (Ch. 19).
- **Buss** Buss, S.R. (2003). *3-D Computer Graphics*. Cambridge University Press. Another recent book which 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).
- **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. Explains Bézier, 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 Bézier, 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 first book on the market devoted entirely to subdivision methods.
- **GG I-V** *Graphics Gems I* (1990) to *Graphics Gems 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 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.

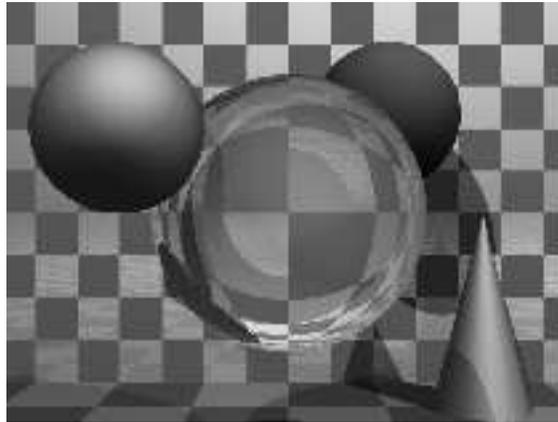


Figure 1: A basic ray traced model showing refraction and shadowing.

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 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 reasonably detailed scene. Ray tracing was first implemented in hardware by a Cambridge company, Advanced Rendering Technologies², in the late 1990s. The quality of the images that they can produce is high compared with polygon scan conversion. This

²<http://www.artvps.com/>



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

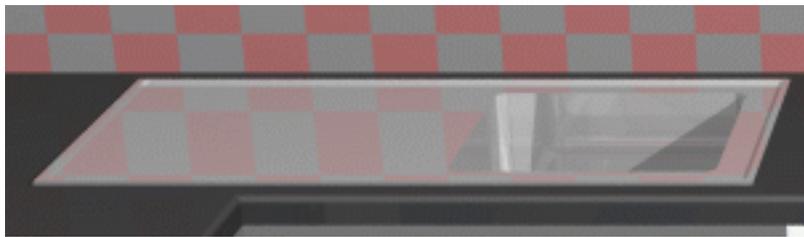


Figure 3: A close up of the kitchen sink.

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*³. *Rayshade* has not been updated for over a decade. An alternative ray tracer, which is kept up to date, is *POVray*⁴, 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 perfectly curved object (impossible to achieve with polygons). The inner curve is part

³<http://graphics.stanford.edu/~cek/rayshade/rayshade.html>

⁴<http://www.povray.org/>

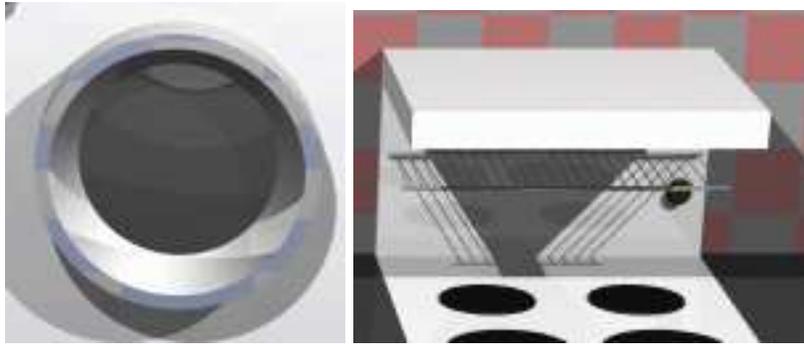


Figure 4: Close up views of the washing machine door and the grill on the stove.



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

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.

1.1.3 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, such as the A-buffer.

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

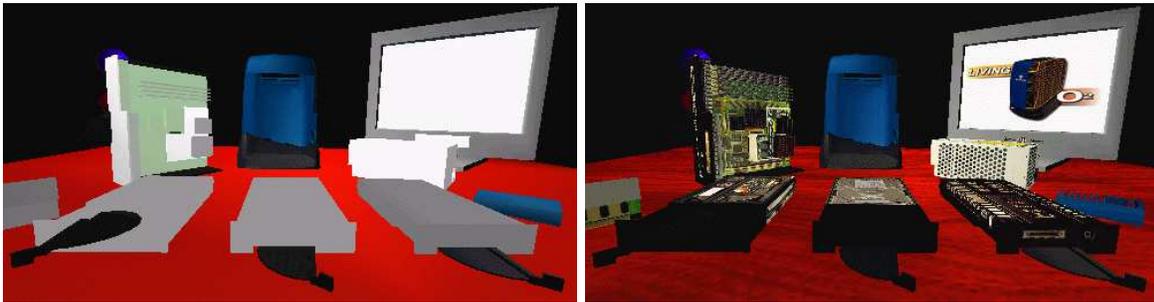


Figure 6: An SGI O2 computer and components drawn with and without texture maps.

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 Carolina⁵ 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 NVIDIA⁶ GeForce8 graphics processing unit (GPU) architecture, for example, has up to 128 parallel stream processors running at 1.35GHz. It can render up to 36.8 billion textured pixels per second, and can render scenes containing several million 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 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 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 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

⁵<http://www.cs.unc.edu>

⁶<http://www.nvidia.com/>

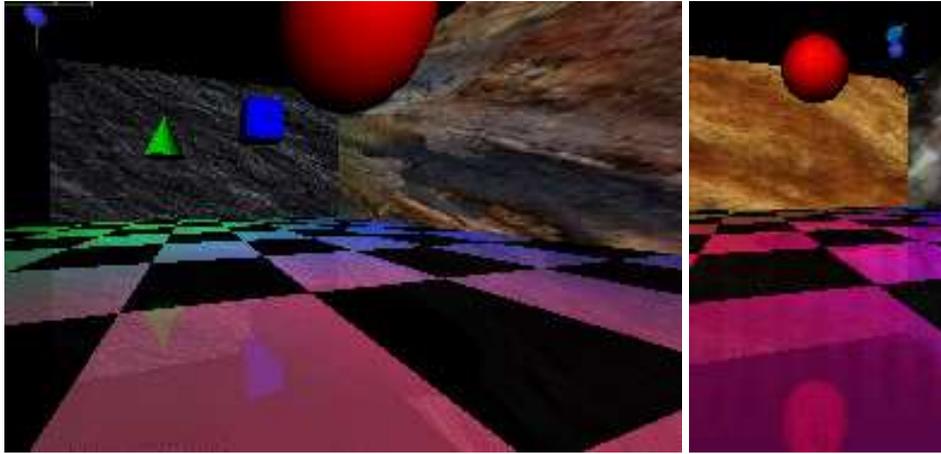


Figure 7: Left: some floating objects in a simulated environment. Right: a close up of the red ball showing the reflection of the ball in the shiny floor.



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

make 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, 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



Figure 9: Screen shots from commercial flight simulators (circa 1995).

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 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 was historically 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 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).



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

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*⁷ and *Unreal*⁸) use polygon scan conversion because it gives interactive speeds. A lot of other “3D” games (for example *SimCity*⁹, *Civilisation*, *Diablo*¹⁰) 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.

⁷<http://www.idsoftware.com/games/quake/quake3-gold/>

⁸<http://www.unreal.com>

⁹<http://www.simcity.com/>

¹⁰<http://www.blizzard.com/diablo2/>

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*¹¹ [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 twenty-fifth anniversary re-release^{25th} of the Star Wars trilogy included a number of computer graphic enhancements, all of which were composited into the original movie.

Twenty years on we saw computer graphics effects of the kind found in movies such as the (rather bloodythirsty) *Starship Troopers*¹² [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 was 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]¹³ with *King Kong* [2005]¹⁴. The plot has barely changed, 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*¹⁵ [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* movies¹⁶, 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

¹¹<http://www.starwars.com/>

¹²<http://www.imdb.com/title/tt0120201/>

¹³<http://www.imdb.com/title/tt0024216/>

¹⁴<http://www.imdb.com/title/tt0360717/>

¹⁵<http://www.titanicmovie.com/>

¹⁶<http://www.lordoftherings.net/>

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 for over a decade. *Toy Story*¹⁷ [1995] was the world's first feature length computer generated movie. Two more were released in 1998 (*A Bug's Life*¹⁸ [1998] and *Antz*¹⁹ [1998]). These were followed by *Toy Story 2*²⁰ [1999], *Dinosaur* [2000], *Shrek*²¹ [2001], *Monsters Inc*²² [2001], *Ice Age*²³ [2002], *Finding Nemo* [2003], and *Shrek 2* [2004]. The genre is now well established and there are several recent examples, with more in the pipeline. Note the subject matter of these movies (toys, bugs, dinosaurs, monsters, sea life, fairytale characters). It is still very difficult to model humans realistically and much research is being undertaken in the field of realistic human modelling. *Final Fantasy*²⁴ [2001] was the first serious attempt to represent fully human characters in a fully computer-generated movie.

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 took between one and three hours to render each frame (these frames have a resolution of 1526×922 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 the ACM SIGGRAPH²⁵ conference in 1998 I had the chance to hear about the software that some real special effects companies were using. Two of these companies used ray tracing and two were pretty happy using polygon scan conversion.

BlueSky—ViFX Ray traced everything using CGI-Studio.

Digital Domain²⁶ Used ray tracing provided by commercial software, except when

¹⁷<http://www.pixar.com/featurefilms/ts/index.html>

¹⁸<http://www.pixar.com/featurefilms/abl/index.html>

¹⁹<http://www.imdb.com/title/tt0120587/>

²⁰<http://www.pixar.com/featurefilms/ts2/index.html>

²¹<http://www.shrek.com/>

²²<http://www.pixar.com/featurefilms/inc/index.html>

²³<http://www.iceagemovie.com/>

²⁴<http://www.imdb.com/title/tt0173840/>

²⁵<http://www.siggraph.org/>

²⁶<http://www.d2.com/>

the commercial software cannot do what they want. Used MentalRay²⁷ on *Fifth Element* [1997]; used Alias²⁸ models (NURBS) passed to Lightwave²⁹ (polygons) for one advertisement; used MentalRay³⁰ plus Renderman³¹ for another advertisement.

Rhythm + Hues³² Used a propriety renderer, which was about ten years old in 1998. It has been rewritten many times. They made only limited use of ray tracing.

Station X Used Lightwave³³ 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. Eight years further on, polygon scan conversion is still the algorithm of choice.

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.

²⁷<http://www.mentalray.com/>

²⁸<http://www.aliaswavefront.com/>

²⁹<http://www.newtek.com/>

³⁰<http://www.mentalray.com/>

³¹<http://www.pixar.com/renderman/>

³²<http://www.rhythm.com/>

³³<http://www.newtek.com/>

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 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: $N = (C - B) \times (A - B)$.

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

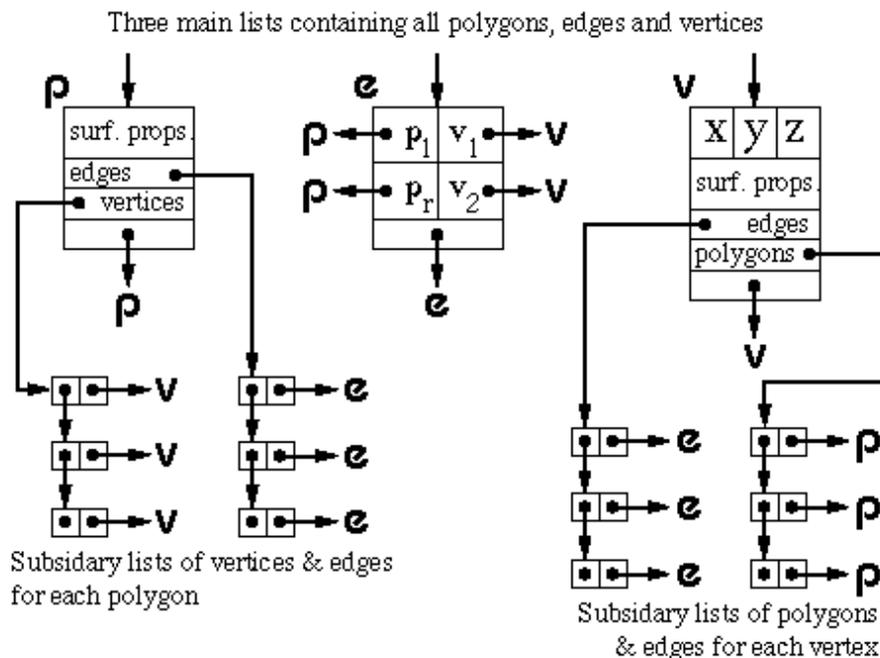


Figure 11: One version of the winged edge data structure.

(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. **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. 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 Graphics³⁴ Reality Engine or the NVIDIA³⁵ GeForce family of graphics cards, has generally consisted 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 in what he or she can program. However, the most recent graphics cards allow for a good deal of flexibility. Early generations of 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 latest NVIDIA GeForce 8 cards (2007) have 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 choose whether or not to execute any given instruction.

To give you an idea of the complexity which is possible, on the GeForce4 generation of NVIDIA cards (2002), the information that 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 size³⁶. 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-intensive computation tasks, using texture buffers for storing intermediate results.

The latest (2007) generations of NVIDIA GeForce graphics cards (the GeForce 8 family) and ATI chips (the Xenos chip used in the Xbox 360) 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.

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

³⁴<http://www.sgi.com>

³⁵<http://www.nvidia.com/>

³⁶You 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.

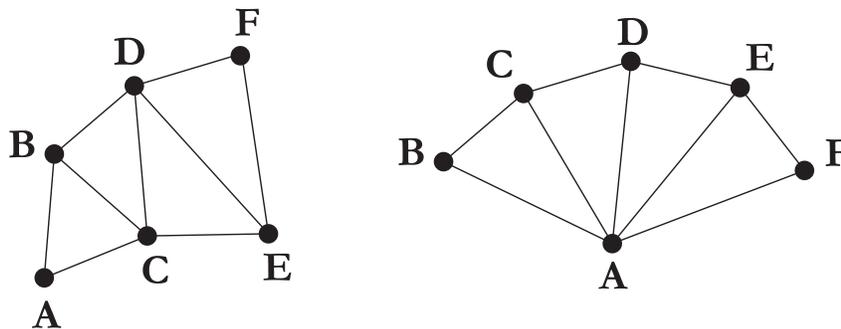


Figure 12: Left: a triangle strip set. Right: a triangle fan set.

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 (see Figure 12). 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.

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. Around 2000, a vertex cache was introduced to graphics cards. On the NVIDIA family of cards, the initial version held 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 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).

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 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: Bézier 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.

The course handout contains a slide presentation introducing the concepts in this part of the course.

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. Parts of this Section of **R&A** are included in the handout.

If you want to experiment with Bézier curves then there are a number of on-line tutorials. One such is available from the Technion in Israel³⁷.

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.

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

³⁷<http://www.cs.technion.ac.il/~cs234325/Homepage/Applets/applets/bezier/html/>

4.2 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: $\mathbf{P}(t)$, defined by $(\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_n)$, and $\mathbf{Q}(t)$, defined by $(\mathbf{Q}_0, \mathbf{Q}_1, \dots, \mathbf{Q}_n)$. 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:

$$\mathbf{Q}_0 = \mathbf{P}_n. \quad (6)$$

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

$$\mathbf{Q}_1 - \mathbf{Q}_0 = \mathbf{P}_n - \mathbf{P}_{n-1} \quad (7)$$

Combining Equations 7 and 6 gives a formula for \mathbf{Q}_1 in terms of the \mathbf{P}_i s:

$$\mathbf{Q}_1 = 2\mathbf{P}_n - \mathbf{P}_{n-1} \quad (8)$$

$$= \mathbf{P}_n + (\mathbf{P}_n - \mathbf{P}_{n-1}) \quad (9)$$

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:

$$\mathbf{Q}_2 - 2\mathbf{Q}_1 + \mathbf{Q}_0 = \mathbf{P}_n - 2\mathbf{P}_{n-1} + \mathbf{P}_{n-2} \quad (10)$$

Combining Equations 10, 7, and 6 gives a formula for \mathbf{Q}_2 in terms of the \mathbf{P}_i s:

$$\mathbf{Q}_2 = 4\mathbf{P}_n - 4\mathbf{P}_{n-1} + \mathbf{P}_{n-2} \quad (11)$$

$$= \mathbf{P}_{n-2} + 4(\mathbf{P}_n - \mathbf{P}_{n-1}) \quad (12)$$

4.3 Bézier surfaces

We learnt in the IB course that the simplest way to construct a Bézier surface is 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. Derive the constraints on control point positions which ensure that two quartic Bézier curves join with (a) C^0 -continuity, (b) C^1 -continuity, and (c) C^2 -continuity.

5 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 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 therefore 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*, $(t_1, t_2, \dots, t_{k+(n+1)})$, 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:

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

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. . .

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 $t \geq t_{i+k}$ so \mathbf{P}_i only influences the curve for $t_i \leq t < t_{i+k}$. Formally, $\mathbf{P}(t)$ is a polynomial of order k (degree $k - 1$) on each interval $t_i \leq t < t_{i+1}$. 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. A weighted sum of

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

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

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.

5.1 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

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

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 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:

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

For example:

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

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 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 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.

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 Rogers and Adams Figure 5-36.

5.1.2 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, $\mathbf{P}_1, \dots, \mathbf{P}_M$. 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): $\mathbf{P}_1, \dots, \mathbf{P}_M, \mathbf{P}_1, \dots, \mathbf{P}_{k-1}$. Your knot vector will thus have $M + 2k - 1$ uniformly spaced knots.

5.1.3 Open uniform knot vector

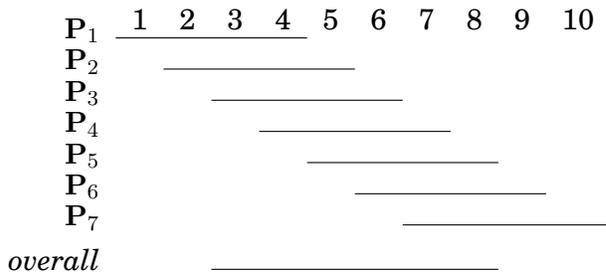
The previous section intimated that uniform B-splines can be used to describe closed curves: all you have to do is join the ends as described above. If you do not want a closed curve, and you use a uniform knot vector, you find that you need to specify control points at each end of the curve which the 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 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 $[0, 0, 0, 1, 1, 1]$.

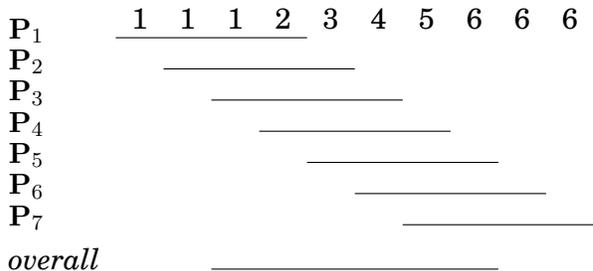
5.1.4 The difference between uniform and open uniform

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 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 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 $t_k + n + 1$.

5.1.5 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.

5.1.6 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.

5.1.7 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.

5.1.8 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.

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. [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 $n \geq 2k - 3$? [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]
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]

5.5 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 Bézier 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 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 in Israel³⁸.

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).

³⁸<http://www.cs.technion.ac.il/~cs234325/Homepage/Applets/applets/bspline/html/>

- 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 Rogers and Adams Figure 5-58 for an example, and play with an on-line NURBS tutorials such as the one mentioned above.

5.6 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 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. 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, 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 $0 = 0$. Thus³⁹ $\alpha = 1/\sqrt{2}$.

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.

5.7 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 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, Béziars, 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]

³⁹If we had not set $\beta = 1$ above, then we would find that $\alpha = \beta/\sqrt{2}$.

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.

6 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.

The course handout contains a slide presentation that presents the concepts from this part of the course in an alternative way.

6.1 Mathematical details: curves

Take an arbitrary polygon 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 doubles⁴⁰ 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:

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

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

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

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

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

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:

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

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.

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.

⁴⁰It 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.

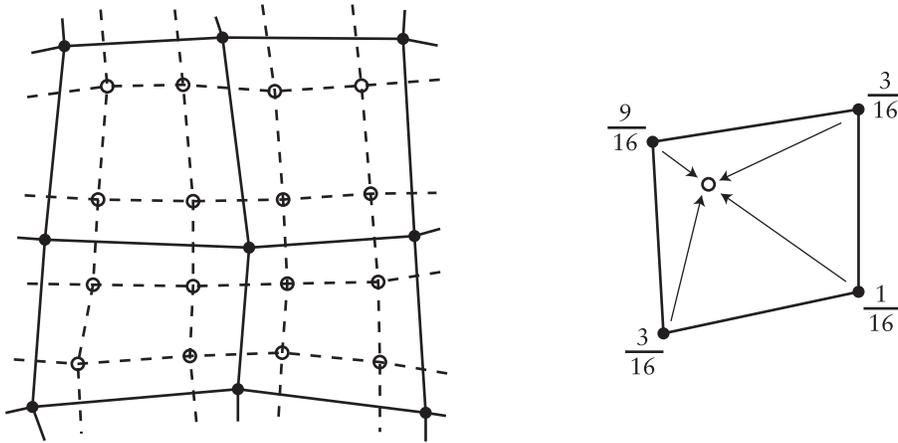


Figure 13: Doo-Sabin subdivision. On left a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). At right the weights used to generated one of the refined vertices.

6.2 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 13.

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 14.

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 14). 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,

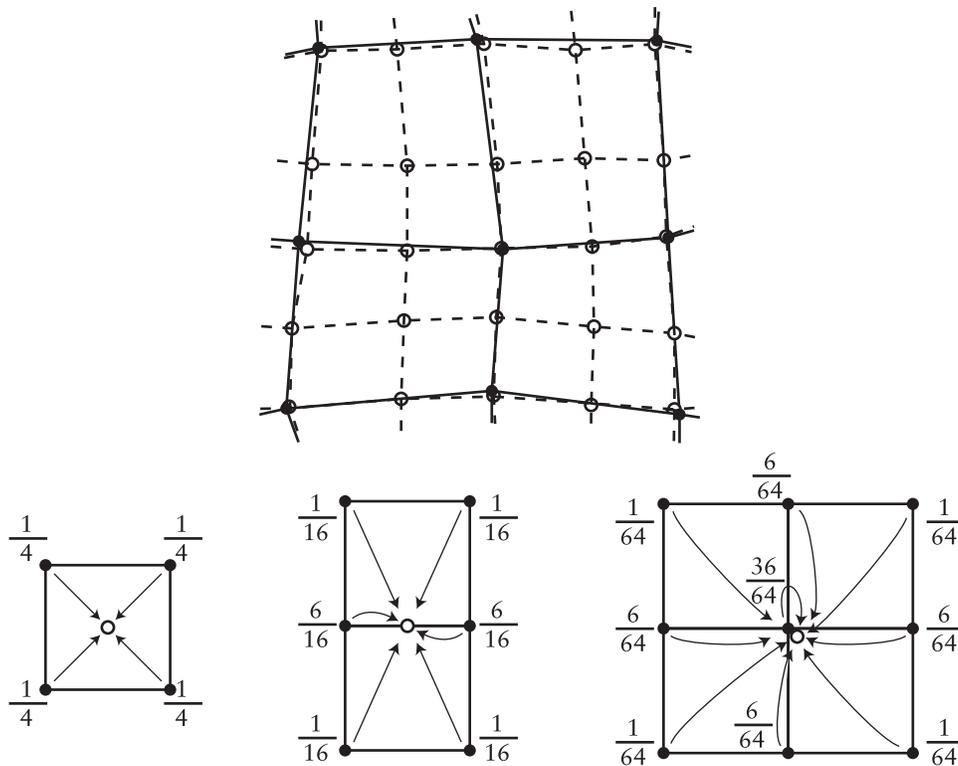


Figure 14: Catmull-Clark subdivision. Above: a mesh (solid dots and solid lines) that has been refined (open dots and dashed lines). Below: the weights used to generated each type of refined vertex: centre, edge, and modified old vertex.

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

$$\alpha_0 = \frac{1}{4} + \frac{5}{4k} \tag{44}$$

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

There are other schemes, notably the Loop scheme (named after Dr Loop) which works on triangular meshes. My research 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.

6.2.1 Exercises

1. Do the “constructive” exercises at the end of section 6.1.
2. Explain how Doo-Sabin subdivision works for an arbitrary polygon mesh.

Slide 1

An introduction to the surface representations used in Computer Aided Design

Neil A. Dodgson
University of Cambridge
Computer Laboratory

with considerable help
from Sabin, Barthe,
Ivrissimtzis, Hassan,
Gérot, Kobbelt, Albat
& Müller

Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk



Slide 2

Applications

- ✦ Computer-aided design (CAD) of artistically attractive surfaces
- ✦ Principally the design of car bodies



Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk



Slide 3

Applications

- ✦ The design of industrial artefacts and animated characters



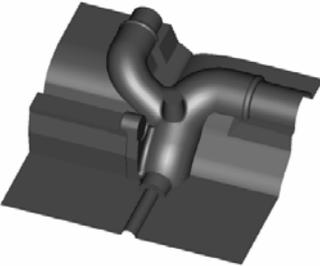
Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk



Slide 4

Applications

- ✦ Computer-aided design of industrial artefacts which do not need to be beautiful
 - ◆ Such as internal engine parts



Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk



Slide 5

Desirable features in CAD

- ✦ Need to handle *any* surface
- ✦ Need guaranteed continuity
 - ◆ Continuity of slope (C1)
 - Smooth surfaces
 - ◆ Continuity of curvature (C2)
 - Smoothly reflecting surfaces
 - Required for some aerodynamics
- ✦ Need to allow for discontinuities
 - ◆ Edges, creases and holes
- ✦ Needs to be easy to use

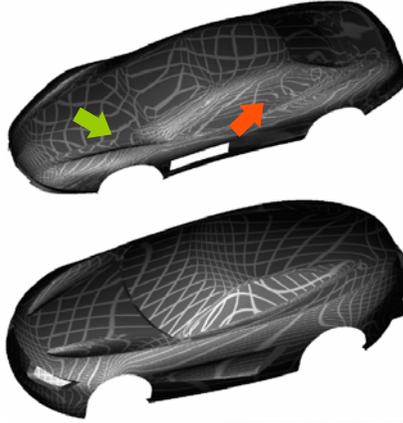


Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

Slide 6

Reflection patterns on car models

- ✦ The top car has problems
 - ◆ Some discontinuities in curvature ●
 - ◆ Some badly distorted areas owing to rapidly changing curvature ●
- ✦ Only the bottom car is a “Class A” surface
 - ◆ The reflection lines look nice



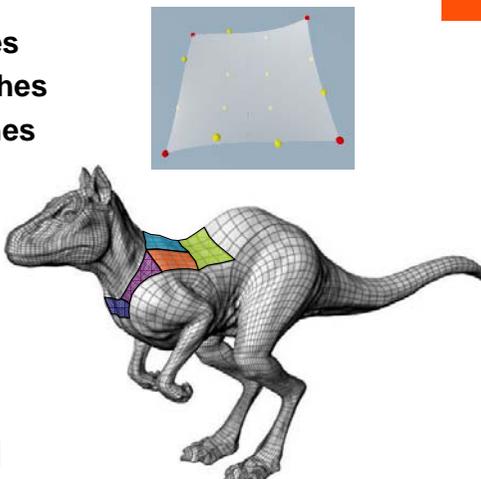
Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF
CAMBRIDGE

Slide 7

Traditional tools

- ✦ Bezier patches
- ✦ B-spline patches
- ✦ NURBS patches

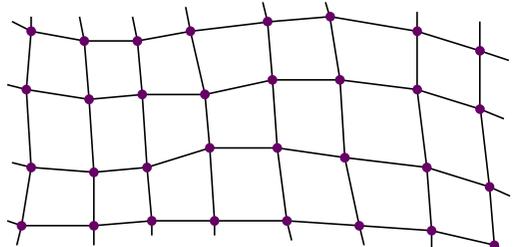


Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

Slide 8

Bezier & B-spline patches

- ✦ A rectangular array of control points
- ✦ A mathematical function determines where the surface goes based on those points
- ✦ Move a control point to change the surface



Part II
Advanced
Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF
CAMBRIDGE

Slide 9

The first problem

✦ Very few objects are made up of a single rectangular patch, so we need to join patches together

Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 10

The mathematics of joins

✦ We want to preserve certain types of mathematical continuity across joins

- ◆ C0: continuity of position
 - Prevents holes at the join
- ◆ C1: continuity of slope
 - Prevents a sharp edge at the join
- ◆ C2: continuity of curvature
 - Strongly related to aesthetics
 - Most often visible in reflections
 - Prevents sharp edges in reflected lines
- ◆ These are continuity of the zeroth, first and second derivatives

Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 11

Joining two Bezier patches

✦ C0 but not C1

- ◆ Four edge points are the same

✦ C0 and C1

- ◆ Four edge points are the same
- ◆ Next four points out in either direction are constrained

Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 12

Joining four Bezier patches

Constraints

- C0 ●
- C1 ● ●
- C2 ● ● ●

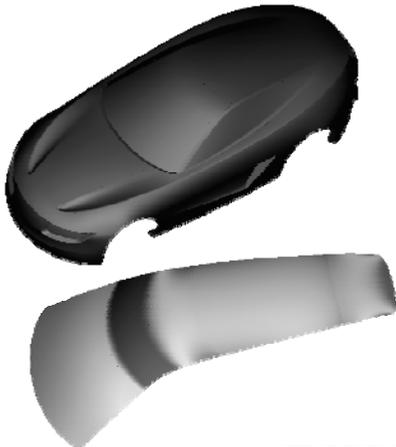
Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 13

An example: the car's roof

- ★ The car
- ★ Curvature plot of its roof



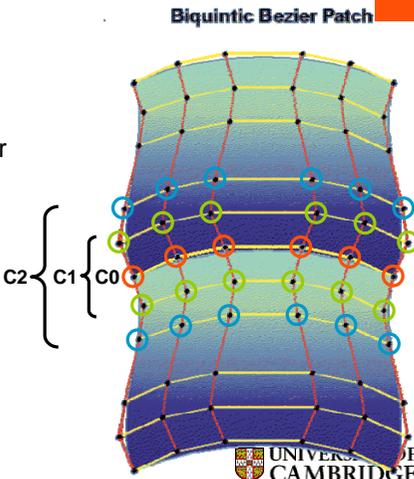
Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 14

Definition of the car's roof

- ★ 5x2 grid of biquintic Bezier patches
 - ◆ 36 control points per patch
 - ◆ 286 control points overall
 - ◆ Moving one point also moves several others to maintain C2 continuity



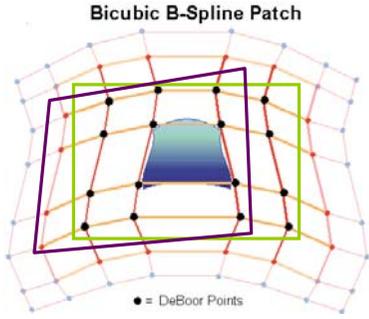
Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 15

B-spline patches

- ★ A rectangular array of points define a rectangular array of *automatically joined* patches
- ★ Example
 - ◆ The black points control the blue patch
 - ◆ All points together define a surface of many joined patches



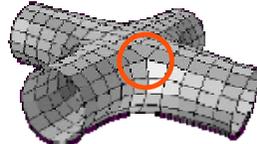
Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 16

The second problem

- ★ What do we do at special points where other than four patches meet?
 - ◆ Either we cannot get C2
 - Which means that curvature is not continuous
 - ◆ Or we get C2 by forcing curvature to be zero
 - Which produces a flat spot
 - ◆ Or we get C2 using very high degree patches
 - Which are very hard for a designer to control



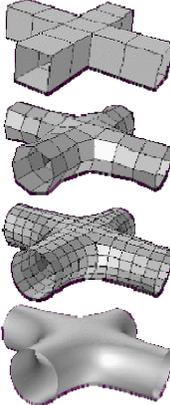
Part II
Advanced Graphics
2004
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 17

Subdivision surfaces

- ◆ Developed in the 1970s, adopted in computer animation in 1990s
- ◆ Replace the patch-based representation of B-splines and Beziers
- ◆ Base a curve or surface solely on its control points and their connectivity
- ◆ A simple mechanism produces a larger, more refined set of control points from the current set
- ◆ Iterate refinement until the appropriate level of detail is achieved



Part II
Advanced Graphics
2004

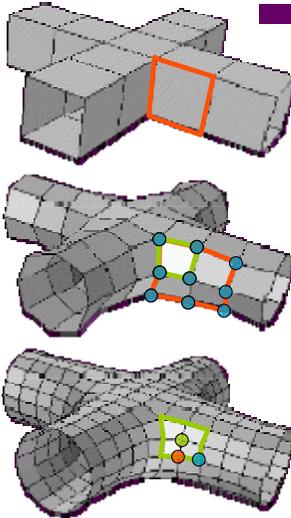
Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 18

Subdivision

- ✦ Introduce new points
 - ◆ At face-centres
 - ◆ At mid-edges
- ✦ Adjust positions of original points
- ✦ Repeat until sufficiently detailed



Part II
Advanced Graphics
2004

Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Slide 19

Subdivision

- ✦ Advantages
 - ◆ Reproduces everything which can be done by B-splines
 - ◆ Handles extraordinary points much more easily
- ✦ Disadvantages
 - ◆ Cannot get C2 unless you produce a flat spot
 - ◆ Generates other visual artefacts, not seen in B-spline surfaces
- ✦ Commercial position
 - ◆ Subdivision is replacing B-splines in computer animation
 - ◆ Subdivision is **not** replacing B-splines in CAD

Part II
Advanced Graphics
2004

Neil Dodgson nad@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

Advanced Graphics 2006

- Subdivision curves & surfaces

Beware: some slides contain multi-layer animations, which do not print well.

©2002,2003,2004,2006 Neil Dodgson

1

Modelling smooth 3D surfaces

- Where are smooth 3D surfaces used?
 - Computer Aided Design (CAD)
 - First developed for cars & aeroplanes
 - Adopted for other manufactured objects
 - Computer animation
- What mechanisms exist?
 - Bézier patches
 - NURBS surfaces
 - Subdivision surfaces

2

Desirable features

- Need to handle *any* surface
- Need guaranteed continuity
 - C1-continuity
 - Smooth surfaces
 - C2-continuity
 - Smoothly reflecting surfaces
 - Required for some aerodynamics
- Need to allow discontinuities
 - Edges, creases and holes
- Needs to be easy to use



History of 3D modelling 1/3

- Some mechanism was needed for modelling 3D surfaces
- Hermite interpolation was generalised to bivariate patches
 - ...but proved too difficult to use in practice
- Bézier patches
 - Developed for car design around 1960
 - Bézier (Renault), de Casteljaou (Citroën), de Boor (GM)

4

History of 3D modelling 2/3

- B-spline theory
 - Developed in the 1960s and '70s, led to:
- NURBS (Non-Uniform Rational B-Splines)
 - More general than Bézier patches
 - Béziers are special cases of NURBS
 - NURBS quickly became the industry standard in CAD
 - ...and remain the industry standard today
 - Adopted by the computer animation industry when it began

5

History of 3D modelling 3/3

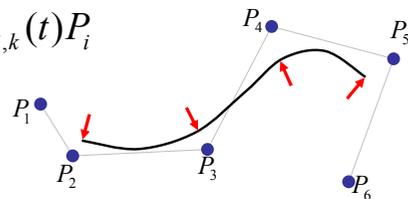
- Subdivision surfaces
 - Theory developed in 1970s and early '80s
 - Picked up by computer animation industry in late 1990s
 - Now replaced NURBS in computer animation
 - Solves one of the big problems of NURBS
 - Still under active research for use in CAD
 - Introduces new problems, not present in NURBS, which make it unsuitable for CAD in its present form

6

NURBS curve

- A curve is defined parametrically
- Its shape is determined by:
 - control points, P_i
 - and the NURBS basis functions, $N_{i,k}$

$$P(t) = \sum_{i=1}^{n+1} N_{i,k}(t)P_i$$



7

Basic properties of NURBS 1/3

$$P(t) = \sum_{i=1}^{n+1} N_{i,k}(t)P_i$$

- The basis functions must sum to 1 to produce a valid new point

$$\sum_{i=1}^{n+1} N_{i,k}(t) = 1, t_{\min} \leq t \leq t_{\max}$$

8

Basic properties of NURBS 2/3

$$P(t) = \sum_{i=1}^{n+1} N_{i,k}(t)P_i$$

- The basis functions are calculated from a *knot vector*
 - Just a non-decreasing sequence of real numbers
 - e.g. [0,0,0,1,1,1] or [1,2,3,4,5,6] or [1.2, 3.4, 5.6, 5.6, 7.2, 15.6]
 - See lecture notes or Rogers & Adams for details

9

Basic properties of NURBS 3/3

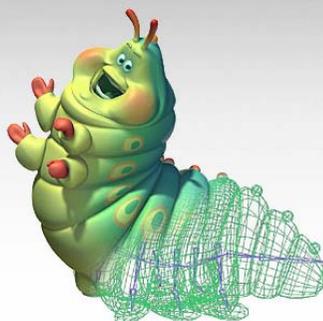
$$P(t) = \sum_{i=1}^{n+1} N_{i,k}(t)P_i$$

- If the basis functions are C_m -continuous at t , then $P(t)$ is guaranteed to be C_m -continuous at t
 - So continuity depends only on the basis functions, $N_{i,k}$
 - Continuity does *not* depend on the locations of the control points
 - you can sometimes get extra continuity by careful positioning of control points

10

NURBS surface

- A bivariate generalisation of the univariate NURBS curve



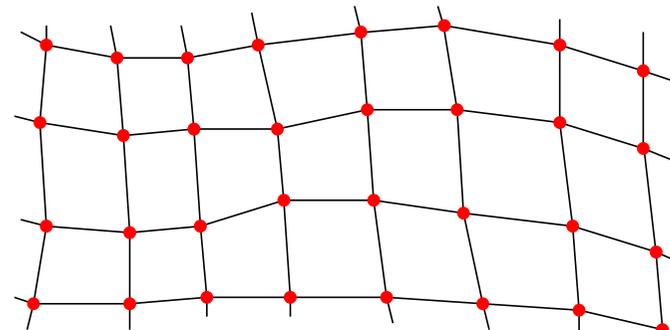
Curve $P(t) = \sum_{i=1}^{n+1} N_{i,k}(t)P_i$

Surface $P(s,t) = \sum_{i=1}^{m+1} \sum_{j=1}^{n+1} N_{i,k}(s)N_{j,k}(t)P_{i,j}$

11

The big constraint...

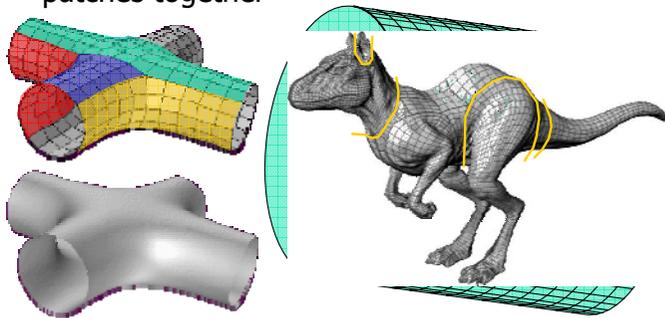
- NURBS surfaces require a quadrilateral mesh of $(m+1) \times (n+1)$ points



12

The first problem

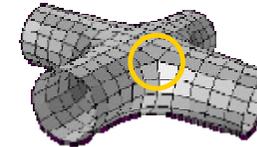
- Very few objects are made up of a single rectangular patch, so we need to join patches together



13

The second problem

- What do we do at special points where other than four patches meet?

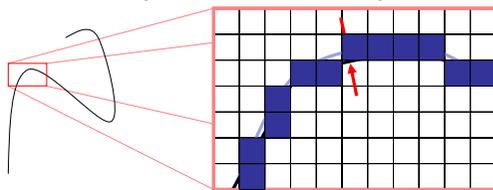


- Either we cannot get C2
 - Which means that curvature is not continuous
- Or we get C2 by forcing curvature to be zero
 - Which produces a flat spot
- Or we get C2 using very high degree patches
 - Which are very hard for a designer to control

14

Drawing a NURBS curve

- NURBS curves and surfaces are always drawn on a pixelated surface
- NURBS curves can be approximated by straight lines
 - So long as each straight line deviates from the curve by less than half a pixel



15

Drawing a NURBS surface

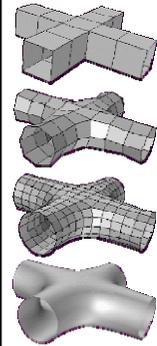
- NURBS surfaces are subdivided and drawn as a series of planar polygons
- Each polygon is only one or two pixels in area on the screen
- Shading algorithms are used to ensure that the surfaces appear to be smoothly curved



16

Subdivision surfaces

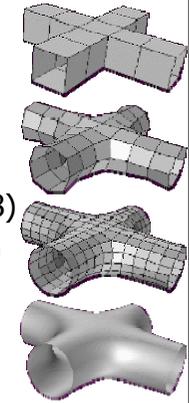
- Do away with the explicit parametric representation
- Base a curve or surface solely on its control points and their connectivity
- Provide a simple mechanism which produces a larger, more refined set of control points from the current set
- Iterate refinement until the appropriate level of detail is achieved



17

History of subdivision schemes

- A univariate (curve) scheme was described by de Rahm in 1947
- Rediscovered by Chaikin in 1974
- Extended to bivariate (surfaces)
 - Doo-Sabin bi-quadratic patches (1978)
 - Catmull-Clark bi-cubic patches (1978)
- Flurry of mathematical work in the early 1980s
 - Dyn & Levin at Tel Aviv University



18

Use of subdivision schemes

- Pixar picked up the ideas and tested them in Geri's Game (1997)
- ...then discarded its NURBS based software in favour of subdivision schemes

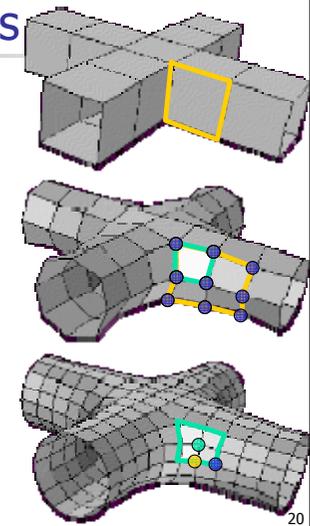


- NURBS
 - Toy Story 1995
 - A Bug's Life 1998
- Subdivision surfaces
 - Toy Story II 1999
 - Monsters Inc. 2001
 - Finding Nemo 2003

19

Subdivision basics

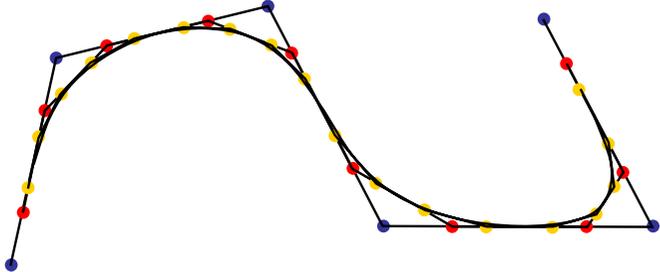
- An example: Catmull-Clark subdivision
 - Introduce new points
 - At face-centres
 - At mid-edges
 - Adjust positions of original points
 - Repeat until sufficiently detailed



20

Chaikin curve subdivision

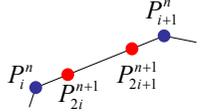
- Underlies Doo-Sabin surface subdivision
- C1-continuous in the limit
- Essentially just a $\frac{1}{4}$ - $\frac{3}{4}$ rule



21

The maths of Chaikin

$$P_{2i}^{n+1} = \frac{3}{4}P_i^n + \frac{1}{4}P_{i+1}^n$$

$$P_{2i+1}^{n+1} = \frac{1}{4}P_i^n + \frac{3}{4}P_{i+1}^n$$


$$h = [K, 0, 0, \frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}, 0, 0, K]$$

$$P^n = [K, P_0^n, P_1^n, P_2^n, K]$$

$$\overrightarrow{P^n} = [K, P_0^n, 0, P_1^n, 0, P_2^n, 0, K]$$

$$P^{n+1} = h * \overrightarrow{P^n}$$

22

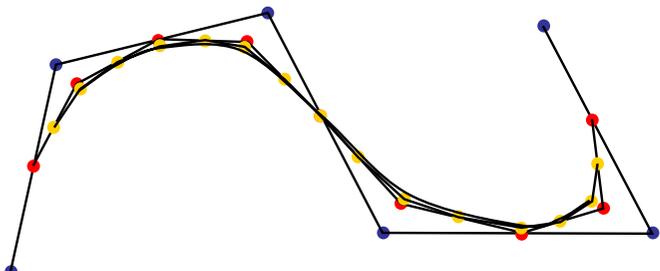
The limit curve

- It can be shown that the limit curve of the Chaikin scheme is the uniform quadratic B-spline, which is guaranteed to be C1
- When drawing curves in computer graphics, we draw a set of straight lines, so only need to subdivide until each segment is about a pixel long and we have a good enough approximation to the curve

23

C2 approximating scheme

- Underlies Catmull-Clark surface subdivision
- Can be described as: "Insert a midpoint and adjust the old control points"



24

The maths of the C2 scheme

$$P_{2i}^{n+1} = \frac{1}{8}P_{i-1}^n + \frac{6}{8}P_i^n + \frac{1}{8}P_{i+1}^n$$

$$P_{2i+1}^{n+1} = \frac{4}{8}P_i^n + \frac{4}{8}P_{i+1}^n$$

$$h = [K, 0, 0, \frac{1}{8}, \frac{4}{8}, \frac{6}{8}, \frac{4}{8}, \frac{1}{8}, 0, 0, K]$$

$$P^n = [K, P_0^n, P_1^n, P_2^n, K]$$

$$\vec{P}^n = [K, P_0^n, 0, P_1^n, 0, P_2^n, 0, K]$$

$$P^{n+1} = h * \vec{P}^n$$

25

Why this notation?

- Easy to analyse
- Allows use of the z-transform

$$h = [K, h_0, h_1, h_2, K]$$

↓

$$h(z) = \Lambda + h_0z^0 + h_1z^1 + h_2z^2 + \Lambda$$

vector
↓
polynomial

$$P^{n+1} = h * \vec{P}^n$$

↓

$$P^{n+1}(z) = h(z) \times P^n(z^2)$$

convolution
↓
multiplication

26

The analysis tools

- Generating function formalism
 - Use the z-transform on the kernel, h
 - Provides sufficient conditions for continuity
 - Essentially checks that the differences between adjacent points decrease fast enough at each refinement step to produce a smooth curve
- There is also a matrix formalism
 - Analyse stationary points
 - Provides necessary conditions for continuity
- For details see our research papers ☺

27

Useful subdivision kernels

- $h = \frac{1}{4}[1, 3, 3, 1]$ C1, approximating, limit curve is quadratic B-spline
- $h = \frac{1}{8}[1, 4, 6, 4, 1]$ C2, approximating, limit curve is cubic B-spline
- $h = \frac{1}{16}[-1, 0, 9, 16, 9, 0, -1]$
 - C1, interpolating, "four-point scheme"
 - There is also a C2 interpolating six-point scheme

28

From Chaikin to Doo-Sabin

- Doo-Sabin scheme is bivariate generalisation of Chaikin $1/4$ - $3/4$ scheme

29

Extraordinary polygons

- Need special co-efficients for these

$$\alpha_0 = \frac{1}{4} + \frac{5}{4K}$$

$$\alpha_i = \frac{1}{4K} (3 + 2 \cos \frac{2i\pi}{K})$$

(Doo-Sabin)

30

Catmull-Clark subdivision

- Catmull-Clark is based on the $1/8[1,4,6,4,1]$ univariate scheme

31

Catmull-Clark rules

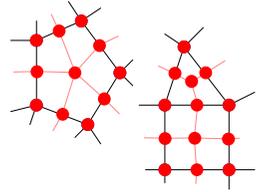
face **edge** **vertex**

- This is easy: the rules are simply the tensor product of the univariate $1/8[1,4,6,4,1]$ rules.

32

7Uha i `!7`Uf_`gdYVU`WUgYg

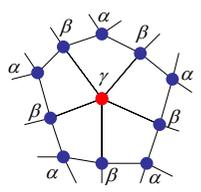
- H\jg]jg'a cfY`X]ZZW`h`k`Y`bYYX`hc`Z]bX`
Vt!YZZWYbng k \]W`a U]bU]b`Vt`b]bi]m
- -h]g'cb`mdcgg]V`Y`hc`[`Yh`7%Vt`b]bi]mUh`
h`YgY`YI`hfUcfX]bUfmdc]bng'



9I hfUcfX]bUfmdc`m]`cbg`
X]gUddYUf`UZ`Yf`cbY`ghYd`

$$\alpha = \frac{1}{4n^2}$$

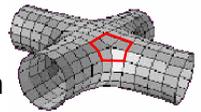
$$\beta = \frac{3}{2n^2}$$

$$\gamma = 1 - \frac{7}{4n}$$


9I hfUcfX]bUfmj`Yfh]Wg`
fYa`U]b`]b`h`Y`a`Yg`..`

Gi VXj]g]cb`j`gBI`F`6G

- 9I hfUcfX]bUfmdc]bng
 - Gi VXj]g]cb`\UbX`Yg`h`Ya``YUg]`m
 - BI`F`6G`f`YeI`]fYg`h`Y`i`g`Y`c`Z`c`h`Yf`m`d`Yg`c`Z`
g`i`f`Z`W`h`c`Z`]]`b`h`Y`\`c`Yg`
- A`Ya`cfm`f`YeI`]f`Ya`Y`bng
 - Gi VXj]g]cb`b`YYXg`U`c`h`f`a`U`b`m`A`6t`
 - BI`F`6G`]g`j`Y`f`m`V`t`a`d`U`M`h`
- 5fh]ZUMtg
 - Gca`Y`U`fh]ZUMtg`dfYg`Y`b`h`]b`V`ch`
 - Gi VXj]g]cb`\Ug`YI`hfU`U`fh]ZUMtg`



H`Y`Z`h`f`Y`

- 7ca`di`h`f`g`b`c`k`\`U`j`Y`Y`b`c`i`[`\`a`Y`a`c`f`m`
h`c`\`U`b`X`Y`g`i`V`X`j`]`g`]`c`b`Y`U`g`]`m`
- Gi`V`X`j`]`g`]`c`b`b`c`k`g`h`U`b`X`U`f`X`Z`c`f`V`t`a`d`i`h`f`
U`b`]a`U`h`c`b`
- BI`F`6G`g`h`]]`g`h`U`b`X`U`f`X`Z`c`f`7`5`8`
- Gi`V`X`j`]`g`]`c`b`k`]]`Y`j`Y`b`h`i`U`m`f`Y`d`U`W`X`
BI`F`6G`Z`c`f`7`5`8`Z`k`Y`W`U`b`g`c`f`h`c`i`h`h`Y`
U`f`h]ZUM`d`f`c`V`Y`a`g`

Ci`f`k`cf`_`U`h`7`U`a`V`f`X`[`Y`

- I`b]Z`h]b[`BI`F`6G`U`b`X`g`i`V`X`j`]`g`]`c`b`
 - "BI`F`6G`k`]h`YI`hfUcfX]bUfmdc]bng"
 - G= ;`F`5D<`&\$\$-
- 7`U`f`U`M`W`f`]g]b[`h`Y`U`f`h]ZUM`g`]b`
g`i`V`X`j`]`g`]`c`b`U`b`X`BI`F`6G`
 - 75;`8`d`U`d`Y`f`g`z`&\$\$-`U`b`X`&\$\$%
- ;`Y`c`a`Y`r`f`W`m`g`Y`b`g`]h`j`Y`g`i`V`X`j`]`g`]`c`b`
 - A`c`X]Z`h]b[`YI`g`h]b[`g`W`Y`a`Y`g`h`c`h`U`_`Y`
U`W`z`i`b`h`c`Z`[`Y`c`a`Y`r`f`W`f`Y`U`h]c`b`g`]\`d`g`

