Advanced Graphics Notes
http://www.matthew.ath.cx/notes/

Matthew Johnson
Trinity Hall
mjj29-notes@srcf.ucam.org

November 13, 2003

1 Modelling Methods

There are several methods of modelling: *implicit*, *explicit* and *parametric*.

1.1 Explicit

Here you define lists of vertices and lists of polygons made from the vertices, and these polygons make up your objects. The problems here are how you store, represent and alter them.

1.2 Implicit

Ray-tracing is an example of implicit modelling. Here a primitive may be a sphere, with the equation $x^2 + y^2 + z^2 = r^2$, or an infinite plane. You don't define the points explicitly, any solution to the formula is a point.

1.3 Parametric

Parametric representations represent the points in terms of several variables.

2 Rendering

2.1 Polygon-Scan Conversion

This is a nice fast application we saw in graphics, which is also very fast. It tends, however, to produce 'obviously' computer-generated images. It has simplistic lighting, and is easy to produce in hardware.

2.2 Ray Tracing

Ray-tracing can produce very realistic images, and has a simple algorithm, but the number of times it is applied means that it is very slow (minutes to hours per scene - not real-time!).

The time that ray-tracing takes means that ray tracing is used for single-image advertising, or for hobbiests, and movie or other uses tend to use polygon-scan for almost everything and reserve ray-tracing for particular special effects.

3 Drawing Polygons

To draw polygons you need to store a list of the polygon's vertices, the order in which they are connected, and the orientation it is in. The last can be inferred from the vertices if you have a consistent ordering of the vectors, and you can you the cross-product to get the normal vector (which should always point towards the viewer). There are several caveats, many graphics cards only draw triangles (some will down-convert), and of course with more than 3 vertices they should all be co-planar.

We also need to know about the surface properties, the colour, texture maps etc. That is important, but won't be covered much in this course.

Simply storing a list of vertices in your polygon structure seems like a good idea for drawing (all polygons in a polygon mesh are drawn separately), however, if you want to edit your structure, where several vertices are shared, then this becomes a lot harder. Keeping lists of polygons, edges and vertices which each point up and down to the joined vertices from polygons, and polygons from vertices. This is called a winged-edge data structure. This is particularly good, since we are always dealing with surfaces (manifold objects) where an edge will only have two polygons joining it.

We have an edge list (linked):

$e_1(*p_l, *p_r, *v_1, *v_2) \rightarrow e_2(*p_l, *p_r, *v_1, *v_2) \dots$

a polygon list:

$p_1(surface, *e_1, *e_2, \dots, *v_1, *v_2, \dots) \rightarrow \dots$

a vertex list:

$v_1(x, y, z, *e_1, *e_2, \dots, *p_1, *p_2, \dots) \rightarrow \dots$

This works very well until you want to add a point in the middle of a polygon, since you have to update a lot of the data structure in unusual places.

4 Hardware Rendering (Graphics Cards)

Modern graphics cards have several processing units, and quite a lot of RAM. The CPU will send geometry objects to the card, which will then apply a lot of transformations and reduce it to a series of triangles. Another processing unit takes the triangles, and performs PSC to get a series of pixels rendered with the textures. Both of these are now programmable on the newest cards.

There are also a few optimisations for drawing triangles very quickly, known as triangle strips and triangle fans. This allows you to send each vertex once, and not three times. We now also have a vertex cache of twenty vertices, which allows the CPU to do less organisation of the vertices before sending.

5 Ray Tracing

Ray tracing takes a database of all the objects and lights in a scene. You specify an eye point and a screen of pixels between it and the scene. To find the colour of each pixel you trace a ray from the eye through each pixel (which is defined by the eye-point $\bar{E}$, the ray vector $\bar{E}$, so the ray is $\bar{P}(t) = \bar{E} + t\bar{D}$ with $t \geq 0$. The computer needs to compare the line with each object to find the closest intersecting (non-negative) object.

Once you have found the object, you need to calculate the colour. We have a vector back to the eye $\bar{V}$, but you also need to calculate the normal $\bar{N}$. In addition to that to calculate shadows you trace a ray from the intersection to each light source, and if there is an intersection you discount the light source. This calculation also requires a full object search. For reflective and refractive objects you have to recursively call the ray tracing algorithm, redefining the eye point to be the intersection.

5.1 Calculating intersects

To calculate the intersection for each object you need a formula for the object in terms of x, y & z (e.g. $x^2 + y^2 + z^2 = 1$ for a sphere) and substitute one in the other to solve for t. This give you a quadratic in t. If t is complex then there is no intersect, if it is a single then the ray touches the sphere, and if it is a double, you take the nearest point. You also need to need to consider negative values - these are behind the eye point, and if only one of them is negative then you are inside the object, and you need to invert the normal.

Other shapes are more challenging than spheres.

5.1.1 Cylinders

An infinite cylinder on the z-axis of radius 1 gives you an equation $x^2 + y^2 = 1$, and you can substitute in before. For a finite length cylinder, you define the condition $z_0 < z < z1$. You do the calculation as before, then extrapolate z from t, and check that the condition holds. If it doesn't you throw that intersection away. This still only applies to open-ended cylinders. To model end points you can catch the case where the two intersection points cross z_0 or z_1. In that case you can easily calculate the intersection parameter from the z endpoint.

5.1.2 Cones

An infinite double-code is defined as: $x^2 + y^2 = z^2$. To restrict the length you clip the z value as with cylinders, and handle end-caps in the same fashion.

5.1.3 Torus

Tori can be defined as
$(\sqrt{x^2 + y^2} - R)^2 + z^2 = r^2$
or

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

This, however, produces a quartic to solve.

5.1.4 Planes & Polygons

Non-curved surfaces require vector calculations. A plane is $\bar{N} \cdot (\bar{P} - \bar{Q}) = 0$. This can be done by substituting $\bar{P}$ in the ray into this formula. To apply this to planar polygons, you first calculate the plane of the polygon ($\bar{Q} = \bar{P}_1$, $\bar{N} = (\bar{P}_3 - \bar{P}_2) \times (\bar{P}_1 - \bar{P}_3)$), then take a line from the point, and calculate intersections with the edges to decide whether you are inside the polygon.

5.1.5 End-caps

The alternative method to implement end-caps, is to convert the cylinder into 3 objects, a cylinder and two ends. This is simpler to implement, however, will never be better than the above method, since it has to intersect all three, even if one of them succeeds. There are also problems with floating point errors, rays can slip 'between' the cylinder and the caps.

5.1.6 Non-regular primitives

The above examples are considered to be positioned at the origin. This is not the case is real scenes, however, it is useful to store the objects as a unit object at the origin, and a set of transformations for the particular object you are considering. A method for doing ray intercepts on these objects is to calculate the inverse transforms for the particular instance you are considering, and apply that to the ray, This produces a standardised ray which can be intersected with the standard primitive. However, in the case of a non-isotropic scaling, the normal vectors do not transform correctly.

5.1.7 Converting primitives to polygons

You may wish to run your ray-tracing primitives through a polygon-scan algorithm. This is trivial in the flat primitive case, however, curved primitives are more tricky.

The standard approximation is to convert the curved profiles into polygons, and joining the vertices along the curved surface. Conversion to triangles can be done by joining the diagonals trivially.

Spheres are more tricky still, and you have to use the parameterised equation for the sphere, and take discrete values for θ and ϕ, which gives you a set of triangles and quadrilaterals which are larger towards the equator than to the poles. You can perform a more even approximation by starting with a tetrahedron, finding midpoints on the edges of the tetrahedron, and pushing them onto the surface of the sphere. Toruses can be handled in a similar fashion.

6 Smooth 3D Surfaces

Smooth 3D surfaces are used in things like CAD, originally for car and airplane design, and now computer animation. We want to be able to handle any surface while guaranteeing both C1 and C2 continuity (C2 continuity is needed for smoothly reflecting surfaces and aerodynamics). We do need to allow for discontinuities where necessary, and finally it needs to be easy to use.

6.1 NURBS Curve

A curve is defined parametrically, its shape determined by the control points P_i, and the NURBS basis functions $N_{i,k}$. The curve formula is $P(t) = \sum_{i=1}^{n+1} H_{i,k}(t)P_i$. Any section of curve is affected by at most 4 of the control points, and the positions on the curve where the curve changes which points it depends on are called *knots*. The basis functions are defined as a sum to 1 for any value of t between t_{min} and t_{max}. The individual functions are calculated from the *knot vector*, which is just a non-decreasing sequence of real numbers. Therefore the definition of a NURBS curve is just the knot vector and the location of the control points. They have an additional property that if the basis functions are C_m continuous at t, then $P(t)$ is also C_m continuous at t. Continuity only depends on the basis functions, not the locations of the control points (although careful positioning of the control points can help).

6.2 NURBS Surfaces

This is a simple generalisations of the curves case, and you have a 2-dimensional array of points, and a double sum to calculate $P(t)$. The main problem with this is with multi-surface joints, which don't fit a quadrilateral grid of points.

6.3 Beziér curves

Beziér curves are a subset of NURBS curves. The simplest Beziér algorithm is $P(t) = (1-t)P_0 + tP_1$, which is a linear Beziér (2 control points). Beziérs can be defines with more control points, and the resulting formulae are quadratics, cubics, quartics etc. The degree of the function goes up one for each control point. With 3 control points, we have a quadratic. This curve can be defined in terms of basis functions known as Bernstein polynomials. The equations for calculating the curve from the basis is

$$P(t) = \sum_{i=0}^{n} P_i \cdot (\frac{n}{i})(q-t)^{n-i}t^i$$

. Using this you can get high degree functions, but moving each control point alters the entire curve, and in fact adding a point changes the whole basis function.

Therefore, the third-order (4 point) Beziér has be decided on as standard, and more complicated shapes are made from several cubic Beziérs.

The third way of calculating the Beziér curve is using linear interpolations. You take the linear interpolation of the lines between each control points, and join adjacent ones. You the interpolate over these new lines, until you only have one interpolation, and the interpolate point on that is the location of the point at this parameter.

6.3.1 Joining Beziérs

You can get join Beziér curves by connecting the end points, and this gives you c_0 continuity, however you cannot guarantee c_1 continuity. As seen in the previous course, you can line up the secondary control points, and that will give you c_1 control points. Mathematically, this is:

$$c_0 : Q_0 = P_3$$
$$c_1 : Q_0 = P_3\hat{Q}_1 - Q_0 = P_3 - P_2$$

You can also create Beziér patches, simply by having bi-variate joined Beziér curves, as with NURBS surfaces.

6.4 B-spline curves

A solution to solve the C_1 continuity problem without having to use high-degree Beziér, and are also a lot more general.

The basis functions are the same all the way along the curve regardless of order, and the basis functions are local to each point, and the order is independent of the number of point. This now means that moving a point only has local effect (the higher the order the larger the effect, and the smoother the curve).

6.5 The knot vector

From above, $P(t) = \sum_{i=1}^{n+1} N_{i,k}(t) \cdot P_i$. We also have a defined *knot vector* - $(t_1, t_2, \ldots, t_{n+1+k})$ where k is the *order* of the b-spline (the order is the degree + 1, so cubics are order 4). This is further restricted as being a non-decreasing series ($t_i \leq t_{i+1} \forall i$).

The function $N_{i,k}$ is defined recursively.

The base case is $N_{i,1}(t) = \{\frac{1, t_i \leq t < t_{i+1}}{0}$. The recursion is a linear interpolation of the adjacent points at the level below: $N_{i,k}(t) = \frac{t-t_i}{t_{i+(k-1)}-t_i} N_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} N_{i+1,k-1}(t)$. The values of t_i are defined by the knot vector.

This produces a similar effect to wavelets.

The knot vectors come in three types. Uniform, open-uniform and non-uniform. The first two are the most often used. Knot vector's have no effect other than their *relative* spacing. You can shift all their values by the same amount with no effect. You can also scale them with no effect.

6.5.1 Uniform

The knot vector is defined as $t_{i+1} - t_i = k \forall i$ - the spacing is constant. Since only relative spacing matters, we can define all uniform b-spline knot vectors as $1, 2, 3, \ldots$. This means that we can convert all t_i to simply i, and the recursive part of the definition of N becomes $N_{i,k}(t) = \frac{t-i}{k-1} N_{i,k-1}(t) + \frac{i+k-t}{k-1} N_{i+1,k-1}(t)$. In fact, it is very easy to calculate the N function, since different values of i are merely translations of each other. ($N_{i,k}(t) = N_{j,k}(t + i + j)$).

Uniform b-splines are always C_{k-2} continuous for an order k b-spline, each basis function is defined by $k + 1$ knots.

To alter uniform B-splines, you can obviously change the position of the control points, but you can also change the order. With a linear function, the knots are at the control points, and the 'curve' passes through all the points. With a quadratic, the knots are on the line half-way between each control point (and the curve doesn't pass through the control points), and with the cubic the knots are half way along each section of the quadratic curve. This obviously also causes the number of points on the line to decrease by one each time.

You can create closed curves using a uniform b-spline, but you have to add $k - 1$ of the points multiple times, and increase the knot vector as normal. This is because the curves do not reach the end-points with above linear function.

6.5.2 Open-Uniform

Open-Uniform b-splines fix the problem of not going through the end-points. To make this happen, you have to repeat several knot values at the start and end of the vector. This is defined as:

$$t_i = t_i, i \leq k$$

$$t_{i+1} - t_i = C, k \leq i < n+2$$
$$t_i = t_{(n+1)+k}, i \geq n+2$$

This gives you k identical values at the start, k identical values at the end, and then a uniform vector between them. The formula for calculating the basis functions is as before for the points affected by the uniform knots, but the first and last $k-1$ functions are special cases which much be computed separately.

6.5.3 Non-Uniform

Non-uniform b-splines can have arbitrary knot vectors. There are several effects this has. In general an order k curve has C_{k-2} continuity everywhere. However, when you repeat knot values, you lose one level of continuity for each repeated knot. So a cubic ($k = 4$) curve would normally have C_2 continuity, but at a double knots you have C_1 continuity, at triple knots you only have C_0 and at a quadruple knot there is a discontinuity.

6.6 Programmatic defaults

The sensible defaults for these are a cubic function ($k = 4$), with no multiple co-located points, and either a uniform or open-uniform knot vector, depending if the curve should hit its endpoints. Using open uniform is nice, but increases the number of basis functions which need to be programmed ($2k + 1$, rather than 1). If you allow other repeated knots to introduce discontinuities, then you need another k special functions for a double knot, and so on.

6.7 Surfaces

B-splines can be extended to surfaces, as normal.

6.8 Rational?

NURBS stands for Non-Uniform *Rational* Basis-Splines. Uniform, Open-Uniform and Beziérs are all special cases. So far, we have been working with non-rational b-splines. Rational b-splines are the general case of non-rational. As with integers, a rational is an integer over an integer. Non-Rational b-splines end up with a polynomial in t, non-rationals are generalised to allow the function to be a polynomial over a polynomial.

To get a NURBS, take 3D points $P_i(x_i, y_i, z_i)$, and generalise to homogeneous coordinates to $C_i = (X_i h_i, y_i h_i, z_i h_i, h_i)$. Then, the NURBS $P_H(t) = \sum_{i=1}^{n+1} N_{i,k}(t) C_i$. When you convert this back to 3D, you end up with two polynomials. The bottom of the division depends on the value of h_i. If all the h_i are 1, then this results in a non-rational.

h_i introduces another parameter at each control point, which allows the representation of things like circles, quadratics, conics, tori and so on.

7 Generative Models

Generative Models are used to create 3D objects out of simple 2D shapes and basic rules.

7.1 Sweeps

These are generated by taking a two dimensional shape, and extending it linearly into 3 dimensions. The shape could be either a polygon or a NURBS curve, and you extrude it into the z-axis.

7.2 Surface of Revolution

These are similar to sweeps, but you rotate your 2D shape around a circle. Surfaces of Revolution can be used to create a lot of different objects, for example tori.

Both of these are very common in CAD, and so they have built in primitives, for example a SoR requires storing merely the 2D shape and axis of revolution in a data-structure.

7.3 Generalising Sweeps

Sweeps can be generalised by sweeping along an arbitrary curve, or poly-line, rather than a straight line. Thus you have either a polygon or a NURBS curve as the shape, and another NURBS curve or poly-line as the sweep curve.

You can also introduce scaling and twisting as you progress along the extrusion. This gives you another two 2D parameters (more curves) controlling the extrusion.

7.4 NURBS Surfaces from Generative Models

You can create a NURBS surface from a swept shape. You combine the control points from the shape at each of the control points on the sweep curve to create $n \times m$ control points in 3 dimensions, which define your NURBS surface. You can, of course, have a NURBS surface not defined by a swept curve, since they generalise to any quadrilateral grid.

8 Subdivision Surfaces

Since NURBS surfaces must be a quadrilateral mesh, there are several circumstances you can't handle, for example, 3 surfaces meeting at a corner. Here you have a special case at the join point which cannot be handled by NURBS, but come up frequently in CAD.

8.1 Drawing a NURBS curve

NURBS curves and surfaces are always drawn on a pixelated surface, so you can approximate to lines and polygons near the resolution of the pixels, and perform clever shading to make it appear correct.

8.2 Subdivision

This representation does away with explicit parameterisation, and bases a curve on the connectivity of the control points. You start with a very rough representation, and performs repeated subdivision, calculating weighted sums of each subsection, based on the control points. This is an iterative procedure which you repeat down to the required detail level.

8.3 Chaikin curve subdivision

Chaikin's method works for uni-variates, and produces a C_1 continuous curve in the limit. You simply insert 2 extra points between each point in your current approximation, at the 1/4 and 3/4 positions on the current approximation. This results in a a quadratic uniform b-spline in the limit.

The mathematics for this gives you $P_{2i}^{n+1} = \frac{3}{4}P_i^n + \frac{1}{4}P_{i+1}^n$ and $P_{2i+1}^{n+1} = \frac{1}{4}P_i^n + \frac{3}{4}P_{i+1}^n$. (where P^n is the nth approximation level, and i is the point index). This generalises with vectors to a vector h (in this case $[..., 0, 0, \frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}, 0, 0, ...]$) which defines how you expand the vector of points. This idea is used in all subdivision surfaces.

The h vector is applied to the current points as a convolution, but at each iteration you expand the array of points with zeros before convoluting with h.

$$\begin{aligned} P^n &= [..., P_0, P_1, P_2, ...] \\ \bar{P}_n &= [..., 0, P_0, 0, P_1, 0, P_2, 0, ...] \\ P^{n+1} &= h * \bar{P}_n \end{aligned}$$

8.4 C2 approximating scheme

This is a similar technique, which is produced by inserting a midpoint, and shifting the current control points. This has the h vector $[..., 0, 0, \frac{1}{8}, \frac{4}{8}, \frac{6}{8}, \frac{4}{8}, \frac{1}{8}, ...]$. These h vectors are the rows of Pascal's triangle. The next level up indeed gives us a $C-3$ continuous curve in the limit, but that is not required for our surfaces. The row of Pascal's triangle *below* the ones we have considered also works, but this merely refine our straight line approximation

The other h vector worth considering, is $h = \frac{1}{16}[-1, 0, 9, 16, 9, 0, -1]$. This is an interpolating subdivision rule which includes the current points at each refinement level.

8.5 Doo-Sabin

This is merely a bivariate generalisation to surfaces of the Chaikin 1/4 3/4 rule.

8.5.1 Extraordinary polygons

NURBS surfaces cannot handle non 4-valent points, but subdivision rules convert these into extraordinary polygons (which have special cases in the limit at their centres). Doo-Sabin defines special co-efficients for these.

8.6 Catmull-Clark rules

This subdivision scheme is based on the $\frac{1}{8}[1, 4, 6, 4, 1]$ rules, and give C_2 continuity. The rules are generated from the *tensor product* of the univariate version. What you get, are new points in each face affected by the four surrounding points, on an edge affected by the surrounding six points, and vertices are moved by the surrounding nine.

Catmull-Clark also has special cases for non-quadrilaterals. For extraordinary polygons, they disappear after one step (an n-gon is replaced with n quadrilaterals) and every polygon is then a quadrilateral. However, extraordinary points (n-valent points where $n \neq 4$) remain in the mesh, and also have special coefficients for weighting the new points, but you can only get C_1 continuity at these points.

8.7 Subdivision *vs* NURBS

Subdivision handles Extraordinary points and polygons easily, however subdivision is a lot more memory intensive. NURBS is a very compact representation. Subdivision also has more artifacts than NURBS. However, in most cases subdivision is now used in most places, since computers have enough memory now. Special applications such as CAD still use NURBS where the artifacts matter, but this may change with research.

8.7.1 Principally used subdivision schemes

Doo-Sabin is a C_1 continuity scheme, which is easy to explain, but not often used. *Catmull-Clark* is used in a lot of places. This scheme is C_2 continuity everywhere except non-4-valent points, which are only C_1. If you want interpolation (which means you don't alter old points in the mesh), then *Kobbelt* produces this (with C_1 continuity). We may want to have triangular meshes rather than quadrilateral. There are two schemes equivalent to Catmull-Clark and Kobbelt, called *Loop*, which is C_2 approximating and C_1 at extraordinary points, and *Butterfly*, which is a C_1 interpolation. These both work by putting midpoints on each edge, and joining these up to create 4 smaller triangles in each larger one. Similar to Catmull-Clark, there are weighting schemes to position the new points and edges, and to move the current vertices. Extraordinary points with triangles are any which are not 6-valent. Butterfly is based on the $\frac{1}{16}[-1, 9, 9, -1]$ scheme for interpolation. Butterfly has the nice property that there are no special cases (in a closed surface).

There are also several new schemes, called $\sqrt{2}$ and $\sqrt{3}$ schemes, which use diagonal lines to do subdivisions

9 Constructive Solid Geometry

These are set operations on 3D space. Any closed primitive splits space into two sets, that outside the closed object, and that inside the object. You can reason with set logic about these sets.

This set login allows you to do operations such as subtracting two objects, to produce a primitive (cube) with a primitive (disc) hole in it. For example, you could produce a church window with two cylinders and a box, by forming $(C_1 \cap C_2) \cup B$. The way you calculate this and render it is using a modification of the ray-tracing algorithm

The data-structure used to store this is called a *CSG tree*. The leaves of this tree are primitive objects, and the nodes are set operations.

The ray tracing algorithm cannot be used without modification. First you find all of the intersects with all of the primitives involved in the construction. This is already known in the ray tracing algorithm. This gives you a sorted list of parameter values for the ray. If we take the case of an overlapping box and sphere, we might have:

$$\begin{aligned} S &\leftarrow (t_{S_1}, t_{S_2}) \\ B &\leftarrow (t_{B_1}, t_{B_2}) \end{aligned}$$

if we assume $0 < t_{S_1} < t_{B_1} < t_{S_2} < t_{B_2}$, then the different set operations have the boundaries:

$$\begin{aligned} S \cup B &\leftarrow (t_{S_1}, t_{B_2}) \\ S \cap B &\leftarrow (t_{B_1}, t_{S_2}) \\ S \backslash B &\leftarrow (t_{S_1}, t_{B_1}) \\ B \backslash S &\leftarrow (t_{S_2}, t_{B_2}) \end{aligned}$$

so we cannot throw away any of the points before calculating the set.

We can consider the implementation of this as a state diagram, with the transitions being the intersections with an objects, and the states being the combinations of which objects we can be inside. This means that, for example, if we are considering intersections, then we only need to consider the transitions which move me into or out of the state where we are inside all of the primitives. This applies to each of the set operations, and you can throw away all of the transitions which don't cross that boundary.

The implementation of this merely needs a sorted list of intersection points for each objects, and a state bit indicating whether we are inside that object. You process all of the points in order, and then just consider each in turn updating the state.

There are a few restrictions on what we can do. Firstly, all of the primitives must divide space into exactly two parts (this includes the infinite plane), and secondly you have to be careful of numerical inaccuracies, particularly when calculating $A - B$.

10 Implicit Surfaces

We have considered explicit definitions of surfaces, but you can also represent surfaces as particular values of functions. These iso-surfaces are very like contour lines, and vary shape a lot depending on the value of the function we take.

You have some function $f_i(\bar{P})$ which defines a field around a point. You then define a value for the function (e.g. $f_i = 3$), and the surface is calculated for that value - that contour line. If the function has several centres (such as two 'blobs'), then moving the two together can give a smooth 'merging' of the two. The *Geoff* function is an example of this. This function tails off to zero at a finite distance, and is defined as $F(r) = 1 - (4/9)\frac{r^6}{R^6} + (17/9)\frac{r^4}{R^4} - (22/9)\frac{r^2}{R^2}$. This is an effect called *proximity blending*. In general the expands to multiple points with a simple sum $F_t(P) = \sum_{i=1}^{i=n} c_i F_i(r_i)$. All you need to define for these are the centre points, and the radii.

This can also be extended with subtractive blobs ($C_i < 0$), and less obvious things like the maximum or minimum of the overlapping functions. These last two are equivalent to the CSG union and intersection operations, and the difference operation can be calculated from $MIN(f_A, 1 - f_B)$.

10.1 Polygonization of Implicit Surfaces

To actually draw an implicit function, we want to turn it into a series of polygons. This is done by evaluating $f(\bar{P})$ at the vertices of a regular cubic lattice, and label each vertex as either above or below the threshold. Each small cube is then divided into polygons depending on the state of each corner. If necessary, the function can also be evaluated at the centre of a face as well. More specifically, for each edge that crosses from above to below, you insert a vertex on that edge, and join up all the generated vertices. The position on the edge where the vertex is inserted can be gained by interpolating along the edge between the values of the lattice vertices.

10.2 Extensions

There are several extensions you can do to implicit surfaces. One of these is *warping*, where you alter the values of the points before passing them to the function. This has the effect of 'warping' space. Along with the set operations and blending, this allows you to create a lot of shapes using simple functions.

10.3 Medical Imaging

Medical imaging uses a similar method using voxels (small cubes in a cubic grid) and recording values for each voxel from something like MRI. Drawing this uses the method given above.

10.4 Smooth Shading

Shading in the method above is per-polygon. We can increase this to Gourad shading if we know the normals from each point. This is done by estimating the derivatives of the data to provide good normal vectors, by averaging the adjacent points on each axis.