Lecture 12: Radiosity - Principles

Reflectance

Earlier in the course we introduced the reflectance equation for modelling light reflected from surfaces:

$$I_{reflected} = k_a + I_i k_d \mathbf{n}.\mathbf{s} + I_i K_s(\mathbf{r}.\mathbf{v}))^t$$

Where I_i is the incident light intensity and the constants represent:
k_a the amount of ambient light
k_d the amount of diffuse reflection
k_s the amount of specular reflection
We used this lighting model for calculating shading values for polygons using both Phong and Gouraud shading. We used the same equations when calculating the illumination at a ray object intersection while ray tracing. In both cases we assumed that there was a small number of point light sources, or if light was distributed then it came from a point source at infinity.

However, according to the reflectance equation, every surface in a graphics scene is emitting light. We have considered the emitted light travelling in the viewing direction, and neglected the emitted light travelling in other directions. This light will contribute to the illumination of neighbouring objects. In practice we did not attempt to calculate this, but rather chose a constant k_a to represent the ambient light. We will now attempt to model it more accurately through the use of radiosity.

A better approximation to the reflectance equation is to make the ambient light term a function of the incident light as well:

$$I_{reflected} = I_i k_a + I_i k_d \mathbf{n}.\mathbf{s} + I_i K_s(\mathbf{r}.\mathbf{v}))^t$$

or more simply to write (for a given viewpoint)

$$I_{reflected} = I_i R$$

where R is the viewpoint dependant reflectance function.

Radiosity

For any given surface (polygon) of our model we can define the term Radiosity as the energy per unit area leaving a surface. It will not be constant over the surface of a polygon. It is the sum of the emitted energy (if any) and the reflected energy. For a small area of the surface dA (where the emitted energy can be regarded as constant) we have:

$$BdA = EdA + RI$$

We are now treating each polygon of our scene as a distributed light source. The incident energy at any patch is collected from all other patches, in particular for patch i:

$$I_i = \int B_j F_{ij} dA_j$$

where the integral is taken over all patches except i, and Fij is a constant that links patch i and patch j called the form factor. For computer graphics we cannot expect to compute a continuous solution, so we divide all polygons up into patches and replace the integral with a sum:

$$B_i = E_i + Ri \sum B_j F_{ij}$$

where the sum is taken over all patches except i (or alternatively we can sum over all the patches setting F_{ii} = 0). If we can solve this for all B_i then we will be able to render each patch directly with a correct light model. The B value in the equation is that actual colour that is used to render the patch, so B, E and R are all three dimensional vector quantities for an rgb colour image. The form factors are the same for each RGB dimension. We can formulate the problem as the following matrix equation:

$$\begin{bmatrix} 1 & -R_1F_{12} & -R_1F_{13} & . & . & -R_1F_{1n} \\ -R_2F_{21} & 1 & -R_2F_{23} & . & . & -R_2F_{2n} \\ -R_3F_{31} & -R_3F_{32} & -1 & . & . & -R_3F_{3n} \\ . & . & . & . & . & . \\ -R_nF_{n1} & -R_nF_{n2} & -R_nF_{n3} & . & . & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ . \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ . \\ E_n \end{bmatrix}$$

Where n is the number of patches in the scene. The solution is not so easy to do since the form factors are not known. Moreover, the matrix will be big 50,000 by 50,000 may be typical.

When considering the computation of the form factors the specular reflection will be seen to cause problems. The difficulty is that unlike the diffuse reflection which is uniform, the specular reflection is very much direction dependent and involves the vector to the light source v. But now, as we have noted, every patch is a light source! There will be problems with specularities as well since all the light sources are no longer points, so we have to integrate incident light over a specluar cone. All this means that computing specularities will be very difficult, so for the moment we will consider only diffuse radiosity.

As previously mentioned, we need to divide our graphics scene into patches for computing the radiosity. If our graphics scene consisted of small polygons we can perhaps use the polygon map as a set of radiosity patches, but for large polygons, such as might make up a wall, we need to subdivide to make the patches small enough. This is because the emitted light will not be constant across a large polygon, and if the patches are too large we will see them as subdivisions of the polygon. In normal circumstances large polygons may have shading differentials, or shadows thrown across them. Since we calculate just one radiosity value for each patch, so the patching pattern may form an unwanted visual artefact become visible. There are two ways to get round this: (i) make the patches small enough to project to (sub) pixel size, or (ii) smooth the results (eg by interpolation similar to Gouraud shading).

The Form Factors

The form factors couple every pair of patches, determining the proportion of radiated energy from one that strikes the other. The coupling is illustrated by figure 1, and uses the following equation.

$$F_{ij} = \frac{1}{|A_i|} \int_{A_i} \int_{A_j} \frac{cos\phi_i \; cos\phi_j}{\pi r^2} dA_j dA_i$$

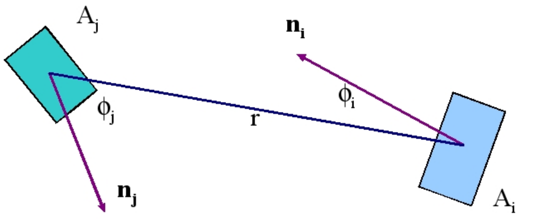


Figure 1: A form factor couples each pair of patches

where $|A_i|$ is the area of patch A_i. The two cos terms effectively compute the projection of the two patches in the direction normal to the line joining them. (If they were at right angles then there would be no light transmitted from one to the other. If they are facing each other then they are maximally coupled. The $1/r^2$ is the normal inverse square law for the decay of light intensity over distance. The equation can be simplified if we consider A_i to be small. If r is large, compared with the dimensions of A_i, the cos terms and the $1/r^2$ can be considered constant over A_i. Thus the outer integral evaluates to $|A_i|$ times the constant inner integral, and the equation reduces to:

$$F_{ij} = \int_{A_j} \frac{cos\phi_i \; cos\phi_j}{\pi r^2} dA_j$$

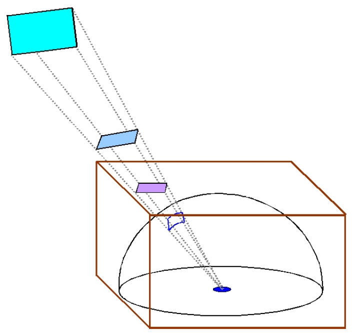
And, of course, we can make the same assumption for patch A_j. Thus this integrand can also be treated as a constant, to give the approximate solution:

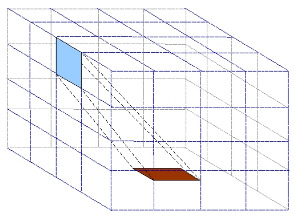
$$F_{ij} = \frac{cos\phi_i \; cos\phi_j |A_j|}{\pi r^2}$$

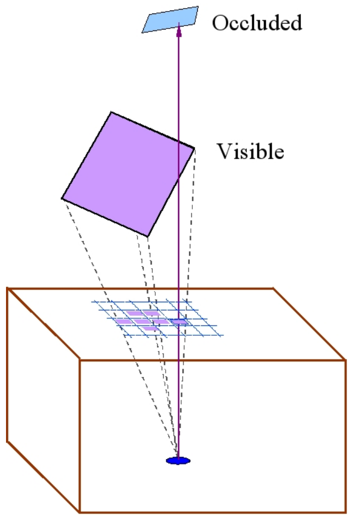
The Hemicube method

Although we have simplified the form factor equation, it would still be expensive to evaluate on a patch by patch basis. Accordingly, a fast algorithm was devised which makes the computation of form factors uniform. Using a bounding hemisphere it can be shown that all patches that project onto the same area of the hemisphere have the same form factor. This is illustrated by figure 2(a), where all four patches have the same form factor. In particular, the patches on the hemicube are used in the algorithm.

A hemicube of side 1 unit is placed over the centre of a patch whose form factors are to be computed. Each of the five faces of the hemicube is divided regularly into a set of square patches called hemicube pixels. An example is given in figure 2(b) where each face is divided into sixteen. There will be a trade off here between speed and accuracy. The larger the size of the hemicube pixels, the worse the estimate of the form factors, but the faster the algorithm.


(a) The Hemisphere and Hemicube


(b) Delta form factors



(c) Ray casting for radiosity

Figure 2: Calculating Radiosity with the hemicube

It will be observed that the form factors between the hemicube pixels and patch under consideration, called the delta form factors, will be the same whichever patch we are computing. Moreover, they will be simple to compute since the geometry is highly regular. For example, suppose that the centre of a hemicube pixel shown in figure 3 is at the coordinate $[x_p, y_p, 1]$. Thus a unit vector from the patch towards the origin can be written as $[-x_p, -y_p, -1]/r$ where $r = \sqrt{x_p^2 + y_p^2 + 1}$. The unit surface normal to the patch is $[0, 0, -1]$, and thus taking the dot product of the two vectors we find that $cos\phi_i = 1/r$. Similar reasoning shows us that $cos\phi_j = 1/r$ If the area of a hemicube pixel is ΔA, then its form factor is:

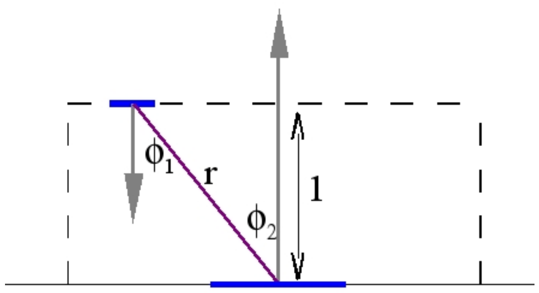


Figure 3: Computing the Delta Form Factors

$$cos\phi_i cos\phi_j \Delta A/\pi r^2$$

Thus the delta form factors on the top plane are given by the equation:

$$\Delta A/\pi r^4$$

By similar reasoning we can deduce that the form factors of the pixels on the sides are given by:

$$\Delta A z_p/\pi r^4$$

Values can therefore easily be computed and stored for these pixels, and similarly a simple equation can be derived for the delta form factors of the sides of the hemicube.

We have previously noted that all patches that project onto the same area on the hemicube have the same form factor. Thus if a patch were to project exactly to a hemicube pixel, its form factor would be the same as the delta form factor for that hemicube pixel. If a patch projects to several hemicube pixels, its form factor will be simply the sum of the delta form factors of those hemicube pixels. We use this to find an approximate value for the form factors using a ray casting operation which is shown in figure 2(c). For each hemicube pixel we cast a ray through its centre and find the nearest intersection with another patch of the scene. We assume that this is the patch that projects entirely to that hemicube pixel, and all other patches are occluded by it. The smaller the hemicube pixels the more likely this is to be true, and the better the estimate of the form factors. The ray casting can be done using the techniques described in the ray tracing lecture. For each hemicube pixel we project a ray from the centre of the patch whose form factors we are computing, through the centre of the pixel and out into the scene. We find the nearest patch that it intersects with. All the previously elaborated methods can be used to establish coherence and minimise the ray patch intersection calculations. Notice that all we need to determine is which patches are visible at each hemicube pixel. We do not need to generate any secondary rays after the nearest intersection has been found.

We can do do the same computation by the alternative means of polygon rendering. To do this we need to transform the scene. The origin of the transformed scene will be the centre of the patch that we are calculating, and, for the top face of the hemicube, the viewing direction will be through the centre of the face, verticaly upwards in figure 2(c). Each patch vertex can then be projected onto the top plane with one matrix multiplication, and the pixels it projects to can be determined by a raster filling algorithm. We need to find the closest patch that projects to a hemicube pixel, and all others can be considered occluded. Essentially we have the same choices to make as we had when removing hidden parts when rendering a scene. We could make use of a z-buffer, and allocate a patch to a pixel only if it is closer than any other previous allocation. Alternatively we can use the painter's algorithm, and sort the patches by distance before projecting them onto the hemicube. The last patch to be allocated to a particular pixel displaces all others. When the allocation process is complete, the form factors with each patch of the scene are found by summing the delta form factors of the hypercube pixels to which they project. If a patch is not allocated to any pixel its form factor is zero, which is generally the case.

In summary, the radiosity method is as follows:

1. Divide the graphics world into discrete patches
2. Compute form factors by the hemicube method
3. Solve the matrix equation for the radiosity of each patch.
4. Average the radiosity values at the corners of each patch,

5a. Compute a texture map of each point on the patch (for walkthroughs), or

5b. Project to the viewing window and render with interpolation shading.

Radiosity Images

Much of the early work on radiosity was carried out at Cornell University, and images and tutorial material can be found on their web site.

http://www.graphics.cornell.edu/online/research/

Lecture 13: Computational Issues in Radiosity

The radiosity method has stages that require intensive use of cpu time, and involve approximate methods that can lead to unwanted visual artefacts. The problems that are associated with each stage of the method are sumarised as follows:

1. Divide the graphics world into discrete patches
   Meshing strategies, meshing errors
2. Compute form factors by the hemicube method
   Alias errors
3. Solve the matrix equation for the radiosity of each patch.
   Computational strategies
4. Average the radiosity values at the corners of each patch
   Interpolation approximations
5. Compute a texture map of each point or render directly
   At least this stage is relatively easy

Alias Errors

Computation of the form factors will involve alias errors. These are equivalent to those errors that occur in texture mapping, due to discrete sampling of a continuous environment. However, these errors are perhaps the least of the problems we will encounter. Errors in the form factors are a secondary effect, as each patch radiosity will be determined by a large number of other patches. Thus each form factor will make only a small contribution to the result. The alias errors can be reduced, if necessary, by increasing the sampling of the hemi-cube, but this in turn puts up the computational demand.

Form Factor Computation

Form factors have a reciprocal relationship:

$F_{ij} = \cos \phi_i \cos \phi_j \text{ Area}(Aj) / \pi r^2$

$F_{ji} = \cos \phi_i \cos \phi_j \text{ Area}(Ai) / \pi r^2$

$F_{ji} = F_{ij} * \text{Area}(Aj) / \text{Area}(Ai)$

Thus, providing we can store the form factors only half the patches need be computed. However, the number of form factors will be very large, and in early solutions for radiosity it was not possible to pre-compute and save them. To give an idea of the size of the problem, we note that for 60,000 patches, there are 3,600,000,000 form factors. We only need to store half of these (reciprocity), but we will need four bytes for each, hence 7Gbytes are needed. This memory requirement is going up with the square of the number of patches. As many of them are zero we can save space by using an indexing scheme. An index could be created with one bit per form factor. If a bit is zero this implies that the form factor zero and not stored. All non zero form factors can be stored in a one dimensional array. Using such a scheme reduces the storage requirement, possibly to one quarter. However, it is easy to conceive of scenes where it is not possible to pre-compute and store the form factors.

Matrix Inversion

Inverting the matrix can be done by the Gauss Siedel method. This is not really a matrix method as such, but rather an iterative scheme based on the form factor equations. Each patch equation has the form:

$B_i = E_i + R_i \Sigma B_j F_{ij}$

We use the iteration

$B_i^k = E_i + R_i \Sigma B_j^{k-1} F_{ij}$

The initial values, Bi^0, may be set to zero. At the first iteration the emitted light energy is distributed to those patches that are illuminated, in the next cycle, those patches illuminate others and so on. The Gauss Siedel inversion is stable and converges fast since the Ei terms are constant and correct at every iteration, and all Bi values are positive.

Progressive refinement

The nature of the Gauss Siedel method allows a partial solution to be rendered as the computation proceeds. Without altering the method we could render the image after each iteration, allowing the designer to stop the process and make corrections quickly. This may be particularly important if the scene is so large that we need to re-calculate the form factors every time we need them.

If the image is progressively refined in this way, it will start dark and gradually illuminate as the iterations proceed. For this reason it is sometimes thought advantageous to add in a constant ambient term which is reduced as each iterative epoch is completed.

Shooting and Gathering

The Gauss Siedel inversion can be modified to make it faster by making use of the fact that it is essentially distributing energy around the scene. By choosing a good order of evaluation we can improve the quality of a partial solution and reduce the number iterations required for convergence.

The evaluation of one B value using one row of the matrix in the normal Gauss Siedel way is like a process of gathering. The value of B is estimated from the B values of all the other patches:

$$B_i^k = E_i + R_i \Sigma B_j^{k-1} F_{ij}$$

Suppose, in one iteration, Bi changes by ΔBi. Every other patch in the scene will change (providing its form factor is not zero). The magnitude of this change can be written:

$$\Delta B_j = R_j F_{ji} \Delta B_i$$

Which can be simply changed into an iterative updating scheme:

$$B_j^k = B_j^{k-1} + R_j F_{ji} \Delta B_i^{k-1}$$

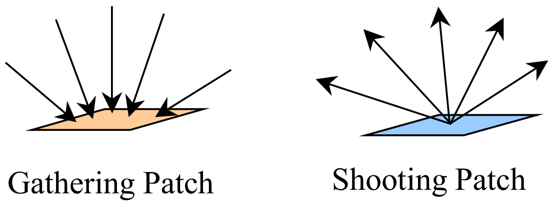


Diagram 1: Shooting and Gathering

This is the process of shooting, and is equivalent to evaluating the matrix column wise. The use of shooting allows us to choose an evaluation order that ensures fastest convergence. The patches with the largest change ΔB (called the unshot radiosiy) are distributed first. In an iteration, the unshot radiosity of a chosen patch is reduced to zero, and that of other patches is incrementally increased. The process starts by shooting the emitting patches, since at the first iteration ΔBi = Ei.

Interpolation Strategies

Visual artefacts do occur with interpolation strategies, but may not be significant for small patches. Diagram 2 illustrates the different interpolation strategies

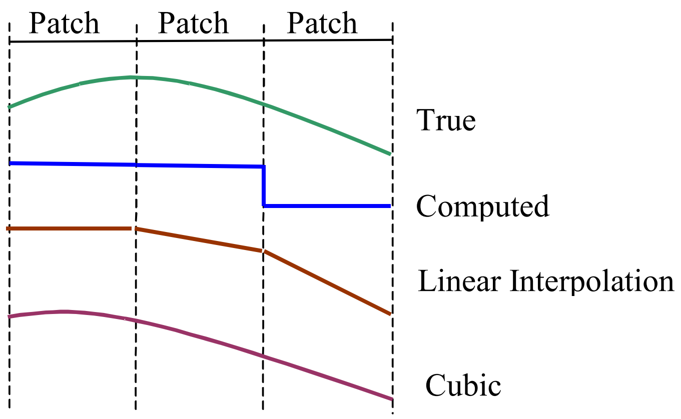


Diagram 2: Interpolation Strategies

and the results they will produce. As noted previously the computed values alone will produce undesirable visual artefacts. However, the use of a cubic interpolation scheme, though more accurate than a purely linear scheme, will probably not produce any significant improvement.

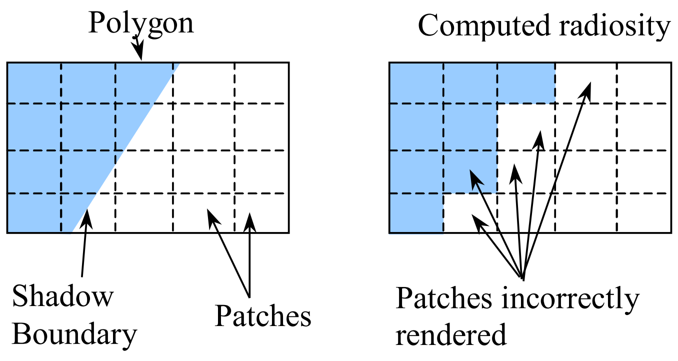


Diagram 3: D^0 Meshing Problems

Meshing

Meshing is the process of dividing the scene into patches, and causes the worst problems in terms of visual quality. Meshing artefacts are scene dependent. The most obvious are called D^0 artefacts, and are caused by caused by discontinuities in the radiosity function. These could be due to shadows or to surface albedo (ratio of incident to reflected light) or texture. Discontinuities of this kind cause problems when accompanied by bad patching. Diagram 3 illustrates the problem. Here a single polygon is divided into regular square patches. A shadow boundary crosses, resulting in a serious visual alias in the computed radiosity. Interpolation will reduce the visual effect of the alias, but will smooth out the hard edge of the shadow.

Discontinuity Meshing (a priori)

The idea behind discontinuity meshing is to compute discontinuities in advance, and align the patches with them. Additionally, the discontinuities are made known to the interpolation procedure so that hard boundaries are not smoothed out. Many of the discontinuities are scene dependent. These in particular will include object boundaries, which we would naturally expect to be patch boundaries as well, and albedo discontinuities, possibly specified in texture maps. Shadows can also be determined, and this can be done to a process similar to ray tracing. Books contain a lot of algorithmic detail on this subject, but we will not cover it here.

Adaptive Meshing (a posteriori)

The idea of adaptive meshing is to recompute the mesh as the radiosity calculation proceeds. Places where there are strong discontinuities in the radiosity can be found by comparing values at adjacent patches. There are essentially two approaches:

(i) put more patches (elements) into areas with high discontinuity or

(ii) move the mesh boundary to coincide with the greatest change.

Subdivision of Patches (h refinement)

Using the h-refinement method w compute the radiosity at the vertices of the coarse grid of patches. If the discontinuities exceed a threshold we subdivide the patch into elements, and recompute those elements. They can be checked for discontinuity and subdivided

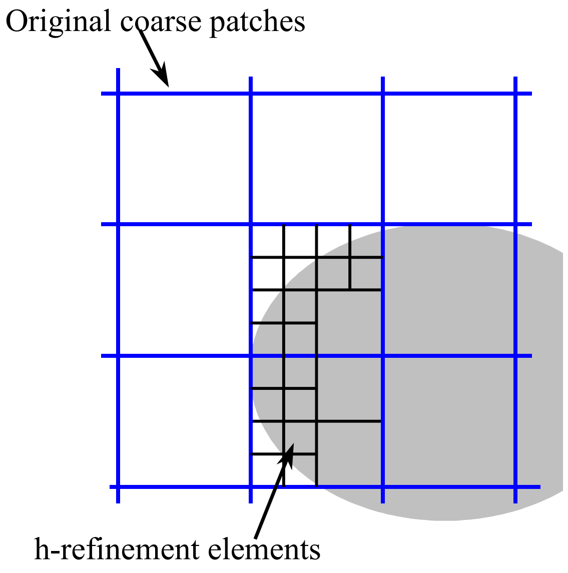


Diagram 4:
Adaptive meshing by h-refinement

further. When a patch is divided into elements the radiosity of each element is computed using the original radiosity solution for all other patches. The assumptions behind this are that:

(i) the radiosity of a patch is equal to the sum of the radiosity of its elements, and,

(ii) the distribution of radiosities among elements of a patch do not affect the global solution significantly

These assumptions are a reasonable, and they save re-computing the complete radiosity solution after each subdivision.

Patch Refinement (r refinement)

Here we compute the radiosity at the vertices of the coarse grid, and move the patch boundaries closer together if they have high radiosity changes. Unlike the h-refinement solution it is necessary to recompute the entire radiosity solution each refinement. However the method should make more efficient use of patches by shaping them correctly. Hence a smaller number of patches could be used. In diagram 5 we see that the uniform patches tend to get larger, and the ones around the discontinuities tend to get smaller, hence some other forms of re-patching may be required as the algorithm proceeds.

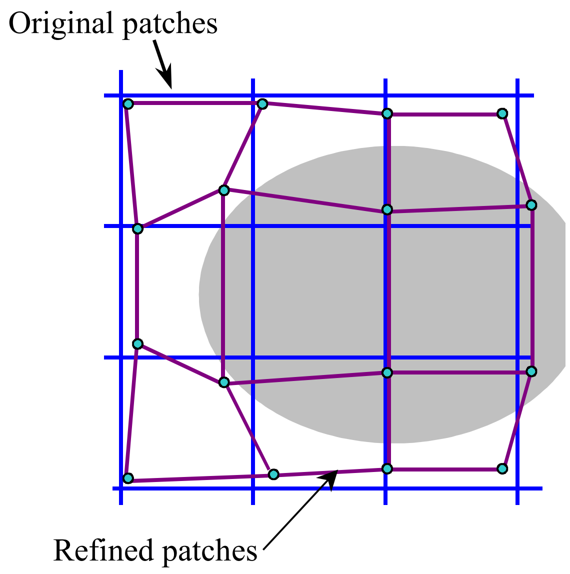


Diagram 5:
Adaptive Meshing using r-refinement

Adding Specularities

We noted that specularities (being viewpoint dependent) cannot be calculated by the standard radiosity method. However, they do form an important aspect of visual realism, and it is desirable to add them where possible. This can be done after the radiosity solution has been obtained, by means of ray tracing. The complete ray tracing solution is not required, just the specular component in the viewpoint direction. Thus only one primary ray per pixel and one secondary ray in the reflected direction are required to achieve this. Further ray tracing can be used to enhance radiosity solutions with effects such as reflections and translucency.

Tutorial 6: Spline Curves and Surfaces

1. A four knot, two dimensional Bezier curve is defined by the following table

	(x, y)
$\mathbf{P}_0$	(0, 0)
$\mathbf{P}_1$	(2, 3)
$\mathbf{P}_2$	(3, -1)
$\mathbf{P}_3$	(0, 0)

a. Use de Casteljau's construction to sketch the curve.

b. Calculate the coefficients $\mathbf{a}_0$, $\mathbf{a}_1$, $\mathbf{a}_2$ and $\mathbf{a}_3$ of the corresponding cubic spline patch:

$$\mathbf{P}(\mu) = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

c. Differentiate the spline patch equation to find $\mathbf{P}'(\mu)$ and hence show that the gradient at $\mathbf{P}_3$ is the same as the gradient of the line joining $\mathbf{P}_3$ to $\mathbf{P}_2$.

2. A Coons surface patch is to be drawn using the following array of points:

		μ			
		-1	0	1	2
ν	-1	(0, 0, 0)	(0, 10, 5)	(0, 20, 10)	(0, 30, 20)
	0	(10, 0, 5)	(10, 10, 20)	(10, 25, 30)	(15, 35, 40)
	1	(20, 0, 10)	(20, 12, 40)	(20, 30, 50)	(25, 40, 30)
	2	(30, 0, 5)	(35, 15, 30)	(40, 35, 40)	(50, 50, 20)

We are interested in the patch constructed on the centre knots, $\mathbf{P}(0, 0)$, $\mathbf{P}(0, 1)$, $\mathbf{P}(1, 0)$ and $\mathbf{P}(1, 1)$.

a. Find the equations of the four cubic spline patches that bound the Coons Patch:

$\mathbf{P}(\mu, 0)$, $\mathbf{P}(\mu, 1)$, $\mathbf{P}(0, \nu)$, $\mathbf{P}(1, \nu)$

These are each parametric cubic splines of the form:

$$\mathbf{P} = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0 \quad \text{or} \quad \mathbf{P} = \mathbf{a}_3\nu^3 + \mathbf{a}_2\nu^2 + \mathbf{a}_1\nu + \mathbf{a}_0$$

The parameters for either form can be found using:

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P}'_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}'_{i+1} \end{pmatrix}$$

b. Find the point at the centre of the patch using the equation:

$$\begin{aligned} \mathbf{P}(\mu,\nu) \quad = \quad & \mathbf{P}(\mu,0)(1-\nu) + \mathbf{P}(\mu,1)\nu + \mathbf{P}(0,\nu)(1-\mu) + \mathbf{P}(1,\nu)\mu \\ & -\mathbf{P}(0,0)(1-\mu)(1-\nu) - \mathbf{P}(0,1)(1-\mu)\nu - \mathbf{P}(1,0)\mu(1-\nu) - \mathbf{P}(1,1)\mu\nu \end{aligned}$$

NB: This numerical calculation is rather tedious unless you use a programmable calculator, spreadsheet or software such as MatLab (which is available on the lab machines).