MANCHESTER
1824
The University of Manchester

COMP37111: Advanced Computer Graphics

Raytracing and Radiosity

Steve Pettifer

November 2020

Department of Computer Science
The University of Manchester

1 Ray Tracing

The first of the techniques that we'll look at for creating photo-realistic images is Ray Tracing. Interestingly, the idea of generating scenes by tracing the effect of rays of light predates the rendering equation by some twenty or so years, and was first described by Arthur Appel in 1968 (Appel, 1968); but it is still in many ways a computational approximation to the solving the rendering equation for a scene.

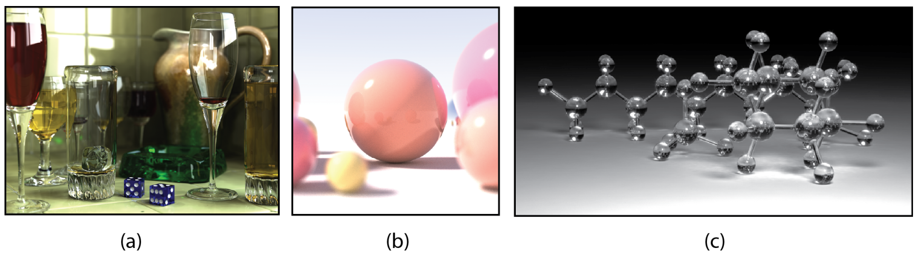

Figure 1: Examples of Ray Traced images. Note the prevalence of shiny and hard surfaces, and therefore of reflection reflection and transparency effects. The figure is a composite of images by Tim Babb, Gilles Tran and Purpy Pupple.

In nature, a light source emits a light which travels though space and eventually interacts with a surface that interrupts its progress. One can think of light as being a 'ray'—a stream of photons if you like—traveling along the same path[1]. In a perfect vacuum this ray will be a straight line (ignoring relativistic effects). In reality, any combination of four things might happen with this light ray: absorption, reflection, refraction and fluorescence. A surface may absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light. It might also reflect all or part of the light ray, in one or more directions. If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the color). Less commonly, a surface may absorb some portion of the light and fluorescently re-emit the light at a longer wavelength colour in a random direction, though this is rare enough that it can be discounted from most rendering applications. Between absorption, reflection, refraction and fluorescence, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract 50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, reflective and fluorescent properties again affect the progress of the incoming rays. Some of these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image.

Translating this idea into programmatic implementation would see us casting rays of light from every light source in a scene, and following their effects as they bounce around the environment, being variously redirected, absorbed and so on until some eventually pass through the virtual viewplane and into our virtual eyepoint. The problem with this approach from a computational point of view is that it's very wasteful—the vast majority of the rays of light that originate at a source don't actually end up going through the viewplane at all, and are therefore all the maths that went into following their route is wasted. The simple fix to this problem is to reverse the process, and to think of rays as originating at the eye point, and then moving through the environment until they eventually reach a light source or leave the scene without ever reaching a light.

[1]Without getting too **Brian Cox**[W] about it all, we will play rather fast and loose with the notion of **wave/particle duality**[W] in these notes, sometimes talking about the wavelength of light, and at other times treating it more as a stream of particles. Messy, but necessary for now.

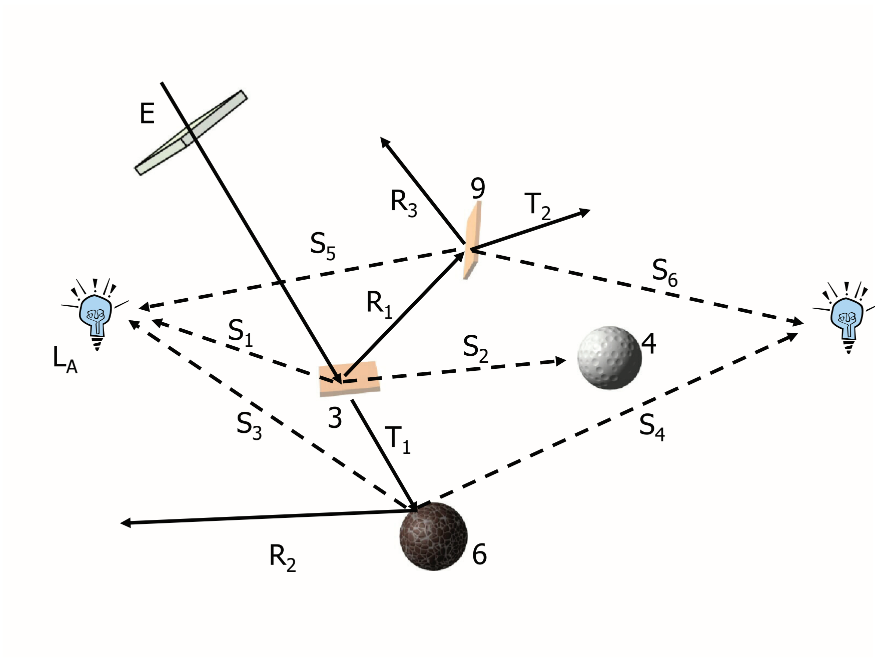

Figure 2: Example of rays traced through a scene containing various objects.

A practical recursive algorithm for basic Ray Tracing was described by Turner Whitted in 1979 (Whitted, 1979). With reference to Figure 2, the algorithm goes like this:

1. Fire a 'primary' ray from the eye point, through the viewplane (E) and into the scene. If the ray shoots straight through the scene without hitting any surfaces, then it is lost and does not contribute to the final image.
2. Otherwise, at the point where the ray interacts with a surface, check to see whether any light could have arrived directly at that surface from any of the environment's light sources. This is achieved by sending out 'shadow feeler' (S_n) rays in the direction of each of the light sources (L_A and L_B)—if a shadow feeler ray can reach a light source without being interrupted by some other object, then light can travel from that light source to the point in question to directly illuminate it; otherwise the point is in a shadow caused by the blocking object.
3. If the surface is transparent, we need to then create a 'refraction ray' (T_n) that passes through the surface, taking into account any change in direction caused by the density of the material, and also any change in colour or intensity of the ray depending on the opacity and colour of the material. Recursively follow the path of this ray through the scene, much as if it were a primary ray, but remembering where it came from so that the effect can be contributed back to its originating ray.
4. If the surface is reflective, we need to create a 'reflection ray' (R_n) that 'bounces' off the surface in an appropriate direction. As with refraction rays, we then trace this ray recursively through the scene.
5. When a ray reaches a light source, runs out of 'energy' or exits the scene, then the process for that ray stops, and the effect it has had on its originating rays is calculated, resulting in a pixel being plotted on the viewplane.

6. Repeat for every pixel on the viewplane (or if you want a really good result, several times for every pixel by creating virtual 'sub pixels' and blending the results together).

For any single primary ray, a large number of secondary rays (shadow feeler, reflection and refraction) are generated; and the recursive nature of the algorithm means that the number of rays can grow very quickly even for relatively simple scenes.

You can see from this algorithm that the Ray Tracing approach inherently takes into account (some) surface properties, the position of light sources and the eye point; and it approximates the 'integration over the hemisphere' by using lots and lots of rays. One of the main limitations of this approach as an approximation to the ideal solution is also easy to spot from the algorithm. You'll recall that incident light does one of two basic things when reflected; it either bounces off a surface as a 'high intensity' beam in a particular focussed direction for shiny surfaces, or gets scattered in all directions as lots of 'low intensity' rays for matt surfaces (or, some combination of these things). If we were to allow for the 'scattering' of rays due to matt surfaces, every ray/surface interaction would generate an even greater number of secondary rays, and very quickly the number of rays being traced would expand beyond what is computationally tractable. So Ray Tracing typically assumes that all surfaces are essentially specular.

Since Ray Tracing is based on a simplified model of physics, it naturally reproduces several important visual effects including **umbra shadows**W, reflections and refraction, and deals easily with textured objects and 'participating media' such as fog or smoke. Relatively minor modifications to the algorithm allow for optical 'camera effects' including **depth of field**W and shape of **aperture**W. On the down side, its inability to model diffuse interactions means that soft lighting and matt surfaces are not rendered convincingly, and shadows tend to appear harsher than they would in reality (Ray Tracing doesn't recreate **penumbra shadows**W very well), though various techniques that try to identify regions of the image that could be improved by selectively firing extra rays into 'areas of interest' are common (e.g. finding a 'shadow boundary' and arranging for extra rays to be traced to its vicinity results in more realistic, softer shadows).

The computational complexity of Ray Tracing is highly sensitive both in 'image space' and 'scene space'. The size of the viewplane and thus the resolution of the resulting image determines how many primary rays are fired, and the number, position, size and material type of the objects in the scene determine how many secondary rays are generated. Additionally, since Ray Tracing deals well with reflections and specular highlights, the resulting images are viewpoint dependent and not ideally suited to interactive applications such as computer games or virtual environments. On the plus side, the calculations associated with every primary ray are independent of every other ray, so Ray Tracing is **embarrassingly parallel**W, and there are numerous implementations that use multiple CPUs, GPUs or cores to improve performance.

2 Radiosity

Whereas Ray Tracing treats light transport as though it consists of 'infinitely thin rays'; Radiosity goes to the other abstract extreme and treats it as a general exchange of energy between parts of the scene. And as you might expect, where Ray Tracing excells at dealing with transparency and reflection of light, but does rather badly at the 'softer' effects (such as matt/diffuse surfaces or subtle shadowing), Radiosity has the opposite qualities and is based on the assumption that all surfaces are perfect diffusers (opaque, ideal **Lambertian surfaces**W, see Figure 4). It generates results like those in Figure 3.

Radiosity methods were first developed in about 1950 in the engineering field of **heat transfer**W. They are based on the idea of the conservation of energy (or 'energy equilibrium') and

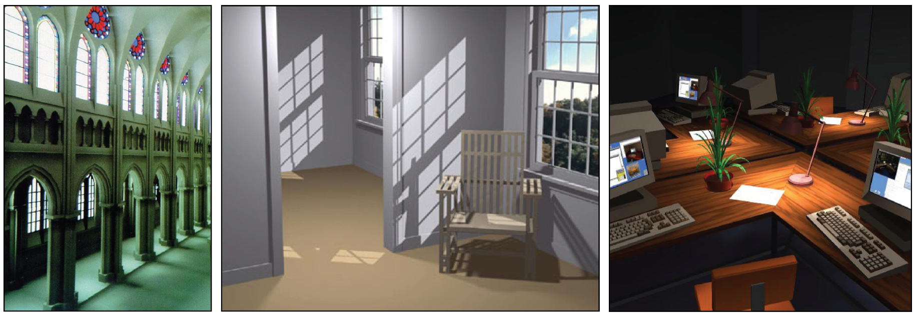

Figure 3: Three scenes rendered using the Radiosity technique. Left, by John Wallace and John Lin; centre by D. Lischinski, F. Tampieri, D. P. Greenberg; and right by Simon Gibson.

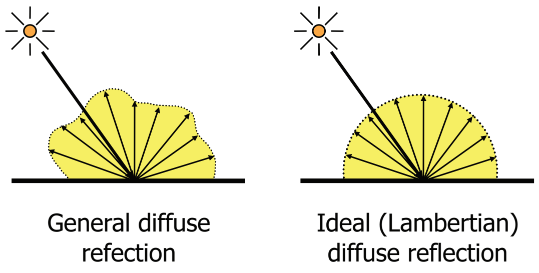

Figure 4: In the real world, the diffuse reflection of light from a surface results in a complex scattering pattern. In computer graphics its common to assume that surfaces are ideal Lambertian diffuse reflectors, where light is scattered equally in all directions. Figure by Roger Hubbold.

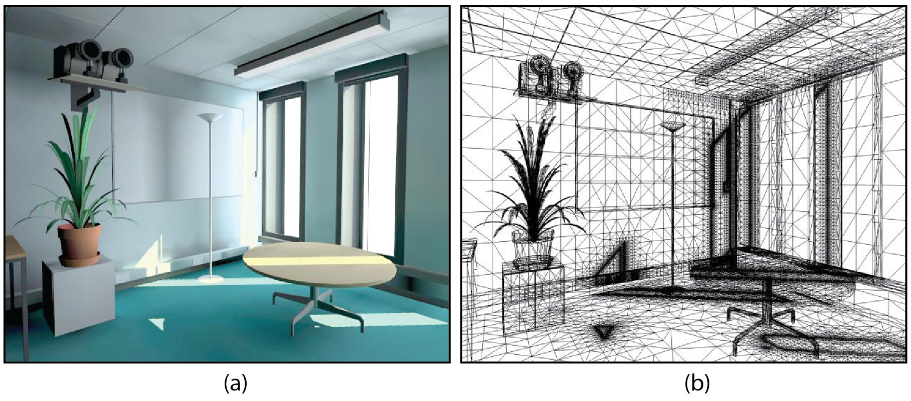

Figure 5: To the left, a scene rendered using the Radiosity approach, and to the right the 'patches' used to calculate the solution. Note that regions that in the original model were probably modelled as a single large polygon (such as the whiteboard or the floor) have been broken up into many much smaller patches. Images by Simon Gibson.

were later refined specifically for application to the problem of rendering computer graphics in 1984 by researchers at Cornell University (Goral et al., 1984; Immel, Cohen, and Greenberg, 1986). The term Radiosity in Computer Science usually refers to the technique for computing diffusely reflected illumination that we'll be exploring here; but the term originally comes from the field of thermodynamics where **thermodynamic radiosity**[W] is the **flux**[W] leaving a surface a point **x**, which is the counterpart of **irradiance**[W] which is the flux arriving at that point. Here, 'flux' is just a term that means 'a flow of particles' (where in this case the particles are photons).

The surfaces of the scene to be rendered are each divided up into one or more smaller surfaces called patches, since a polygonal mesh that's ideal for modelling a scene isn't necessarily the good for Radiosity (see Figure 5, but we'll explore this in more detail later). A 'form factor' (also known as a 'view factor') is computed for each pair of patches; this value is a measure of how well the patches are visible to one another. Patches that are far away from each other, or oriented at oblique angles relative to one another, will have smaller view factors. If other patches are in the way, the view factor will be reduced or zero, depending on whether the occlusion is partial or total. A procedural version of the Radiosity process can be thought of as working as follows (see Figure 6):

1. Identify patches that are associated with light sources
2. Shoot light energy into the scene from the sources, and consider the diffuse-diffuse effect on any patches that are visible to the light source. These target patches accumulate light energy
3. Repeat the process, starting with the patches that have the most unshot energy
4. Stop when a high percentage of the initial energy is used up

In practice, the energy values and form factors are used as values in a linearized form of the rendering equation, which yields a **linear system of equations**[W]. Solving this set of equations gives the Radiosity, or brightness, of each patch, taking into account diffuse interreflections and soft shadows. In 'classic Radiosity' we notionally solve all the equations simultaneously (which is a lot of computation, which for a complex scene is quite slow even on modern hardware—remember we've taken the original scene's mesh and split it up into even smaller polygons) and end up with a final solution.

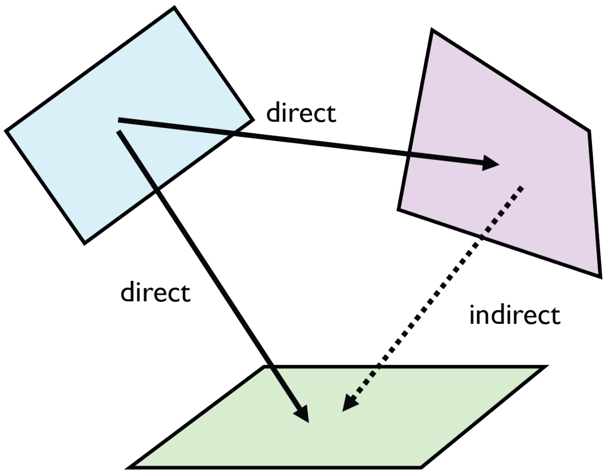

Figure 6: The process starts by considering the light energy sources, which transfer energy directly to other patches. Those other patches in turn then have energy to share with the rest of the scenes. The process continues until equilibrium is reached.

Progressive Radiosity (e.g. Yu, Ibarra, and Yang (1996) and see Figure 7) solves the system iteratively in such a way that after each iteration we end up with intermediate Radiosity values for the patch. These intermediate values correspond to bounce levels. That is, after one iteration, we know how the scene looks after one light bounce, after two passes, two bounces, and so forth. Progressive Radiosity is useful for getting an interactive preview of the scene. Also, the user can stop the iterations once the image looks good enough, rather than wait for the computation to numerically converge (we'll look at some of the things that can go wrong with Radiosity in Section 2.2).

Another common method for solving the Radiosity equation is 'shooting Radiosity', which iteratively solves the Radiosity equation by 'shooting' light from the patch with the most energy at each step. After the first pass, only those patches which are in direct line of sight of a light-emitting patch will be illuminated. After the second pass, more patches will become illuminated as the light begins to bounce around the scene. The scene continues to grow brighter and eventually reaches a steady state (this is the 'procedural' version of Radiosity described earlier).

Let's look at the process in a bit more detail, including some of the underlying maths (again don't worry—you won't need to reproduce this in the exam, you just need to understand the principles and the role of the various components).

Radiosity B is the energy per unit area leaving a patch surface per discrete time interval. It is the combination of emitted and reflected energy, and can be described by the following equation:

$$B(x)\delta A = E(x)\delta A + \rho(x)\delta A \int_S B(x') \cdot F(x,x')\delta A' \tag{1}$$

where

- $B(x)_i \delta A_i$ is the total energy leaving a small area δA_i around a point $\mathbf{x}$ (see Figure 8).
- $E(x)_i \delta A_i$ is the emitted energy.
- $\rho(x)$ is the reflectivity of the point, giving reflected energy per unit area by multiplying by the incident energy per unit area (the total energy which arrives from other patches).

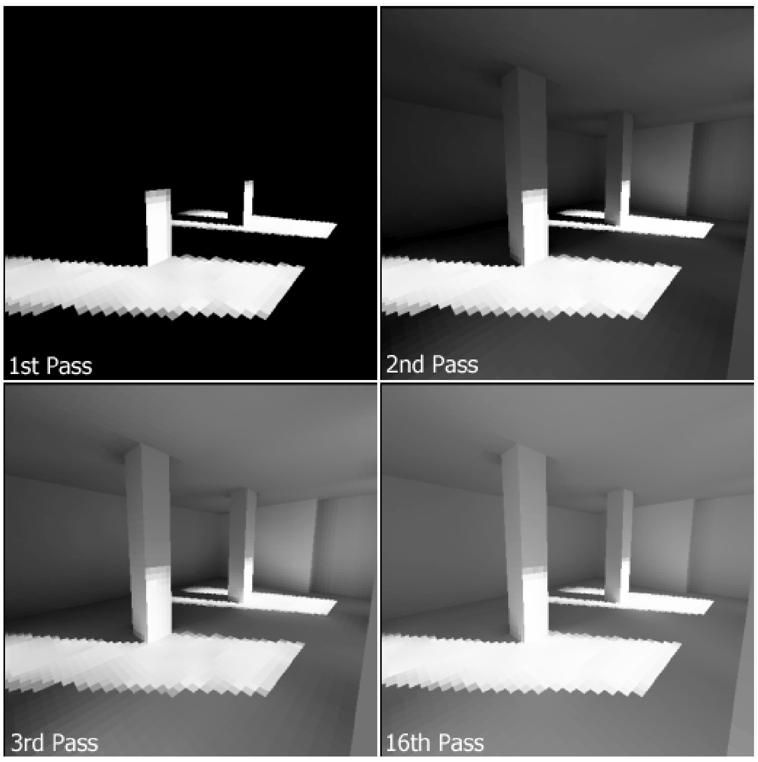

Figure 7: Progressive Radiosity stages. As the algorithm iterates, light can be seen to flow into the scene, as multiple bounces are computed. Individual patches are visible as squares on the walls and floor. Image modified from an original by Hugo Elias.

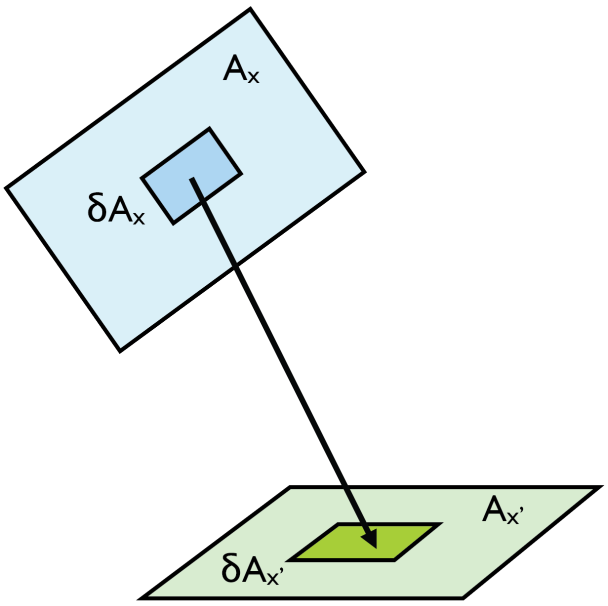

Figure 8: Two patches, A_x and $A_{x'}$ and their respective 'elemental areas' δA_x and $\delta A_{x'}$. Mathematically to calculate the effect of the whole of patch A_x on patch $A_{x'}$ we would integrate overall elemental areas. In practice, computationally we do this by treating the elemental areas as just being lots of 'tiny areas'.

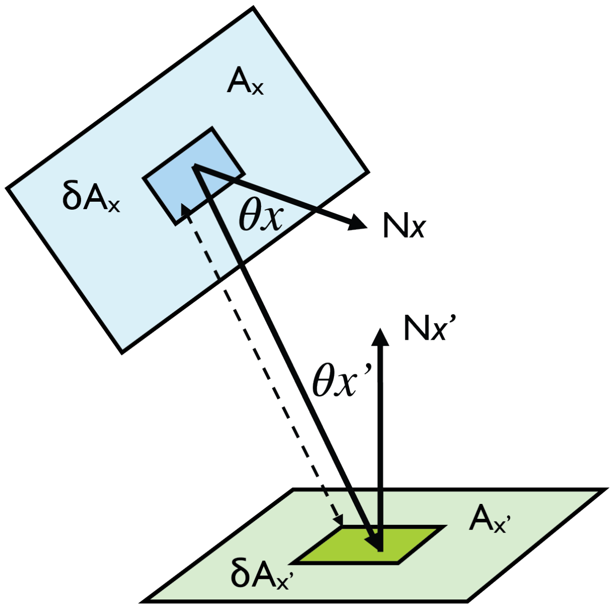

Figure 9: The $\frac{1}{\pi r^2}\cos\theta_x\cos\theta_{x'}$ component of the Radiosity equation arises from the geometric relationship between the two elemental patches under consideration. We need to take into account the falloff in energy transfer to to the distance between them (the 'inverse square law' $\frac{1}{\pi r^2}$ part) as well as their orientation with respect to one another (the $\cos\theta_x\cos\theta_{x'}$ part).

- S denotes that the integration variable x' runs over all the surfaces in the scene
- $F(x, x')$ is the form factor of x to x', defined to be 1 if the two points x and x' are completely visible to each other, and 0 if they are not.

Now of course, as with the rendering equation, we can't deal sensibly with mathematical integration, so we'll need to convert the 'continuous' integral part of the equation into something more discrete that we can represent in code. If the surfaces are approximated by a finite number of planar patches, each of which is taken to have a constant Radiosity B_i and reflectivity ρ_i, the above equation gives the discrete Radiosity equation:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \tag{2}$$

where F_ij is the form factor for the radiation leaving j and hitting i. This equation can then be applied to each patch in the scene. This means that for n patches, we end up with n simultaneous equations to solve. This can be represented as a matrix equation, and solved using one of a variety of techniques such as **Jacobi iteration**[W] or the **Gauss Seidel method**[W] for dealing with matrix solutions; you don't need to know the details of these methods for this course unit, beyond that the results are a set of Radiosity values for each patch.

2.1 Calculating the Form Factor

The form factor F_{ij} between patches i and j needs to take into account three main things:

1. The distance between the two patches, and their orientation with respect to one another.
2. The 'shape' of the two patches when projected onto one another.

3. The effect of any objects that occlude the transfer of light between the two patches (i.e. objects that get in the way and may cause shadows or a reduction in the energy transfer).

The first and second of these are purely geometric, and rely only on properties of the two patches. For the first, we need to know the relationship between the surface normals of the patches and their relative orientations. This can be represented by the expression:

$$\frac{1}{\pi r^2} \cos\theta_x \cos\theta_{x'}$$

which is illustrated in Figure 9.

The second 'projected shape' aspect can be calculated in a number of ways, but perhaps the most intuitive is that called the **Nusselt analogue**W (shown in Figure 10). The form factor between a differential element δA_i and the element A_j can be obtained projecting the element A_j first onto a the surface of a unit hemisphere, and then projecting that in turn onto a unit circle around the point of interest in the plane of A_i. The value we need is then equal to the differential area δA_i times the proportion of the unit circle covered by this projection. Rather conveniently, the projection onto a hemisphere takes care of the factors $\cos\theta'_x$ and $\frac{1}{r^2}$ we've just looked at for us, and the projection onto the circle and the division by its area then takes care of the $\cos\theta_x$ and the normalisation by π, making the previous step essentially redundant.

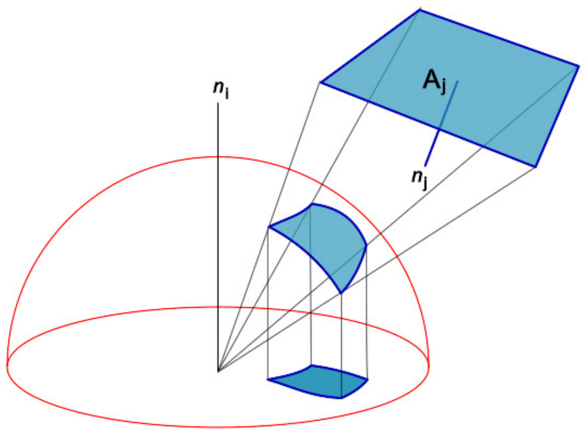

Figure 10: The projected **solid angle**W between patch i and j can be obtained by projecting the element A_j onto a the surface of a unit hemisphere, and then projecting that in turn onto a unit circle around the point of interest in the plane of A_i. The form factor (excluding the 'occlusion' part) is then equal to the proportion of the unit circle covered by this projection. Form factors obey the reciprocity relation $A_iF_{ij} = A_jF_{ji}$. Image by **Jheald**W.

The third and final component, the 'occlusion' part is more complex and mathematically less elegant since it needs to take into account not just the geometry of the two patches, but that of every other object/patch in the scene (Figure 11). Early methods at calculating this computationally used a hemicube (an imaginary cube centered upon the first surface to which the second surface was projected, devised by Cohen and Greenberg in 1985, shown in Figure 12). The surface of the hemicube was divided into pixel-like squares, for each of which a form factor can be readily calculated analytically. The full form factor could then be approximated by adding up the contribution from each of the pixel-like squares. The projection onto the hemicube, which could be adapted from standard methods for determining the visibility of polygons, also solved the problem of intervening patches partially obscuring those behind (akin to the z-buffer technique that you'll have encountered in the second year course unit; or you could imagine using 'ray casting' techniques to work out whether occlusion has occured by intersecting a ray shot from the source patch towards one of the hemicube's 'pixels' and seeing if it intersects with any other polygons in the process).

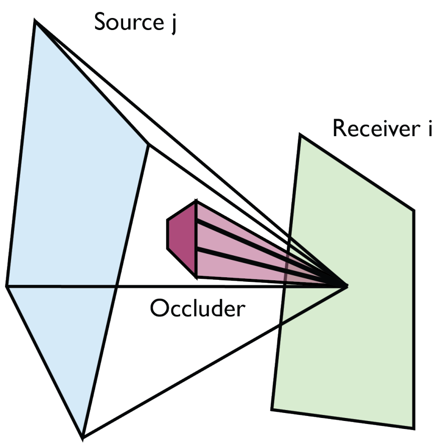

Figure 11: The form factor needs to take into account the effect of occluding objects, since in reality some high proportion (probably over 90%) of patches won't be visible to one another because of occlusion. We need to calculate what percentage of the energy is absorbed by the occluder and does not reach the receiving patch.

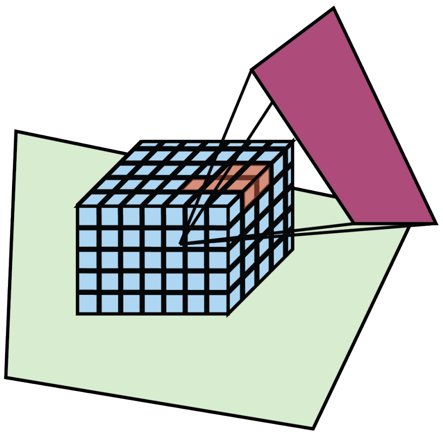

Figure 12: A hemicube placed on a surface; each face of the hemicube is covered with pixels that are treated rather like a z-buffer. Projecting the patch onto the hemicube's pixels gives an indication of the visibility of the patch to the surface, rather like a crude approximation to the Nusselt hemisphere.

However all this was quite computationally expensive, because ideally form factors must be derived for every possible pair of patches, leading to a quadratic increase in computation as the number of patches increases. This can be reduced somewhat by using a technique called a binary space partitioning tree (we'll see this in more detail later in the course when we come to look at Spatial Enumeration) to reduce the amount of time spent determining which patches are completely hidden from others in complex scenes; but even so, the time spent to determine the form factor still typically scales as O($n\log(n)$). Methods of adaptive integration (G. Walton, 2002) have been used to improve this, but are outside the scope of this course unit.

2.2 Issues with the basic Radiosity technique

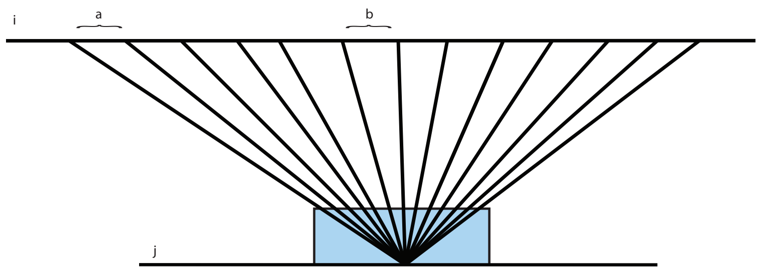

Figure 13: Issues with hemicube aliasing; polygons projected onto the flat surface of the hemicube interact with differing numbers of pixels, depending on the angle of projection.

The hemicube approach is of course a discrete approximation to what in the 'real world' is a continuous phenomenon, and like all approximations it has limitations. One of the biggest is the regular division of the pixels on the hemicube's surface, and the assumption that patches will project onto an integer number of pixels. In Figure 13 although regions a and b of patch i are of equal width, they project onto very different sized areas of the hemicube on patch j; region b is likely to 'hit' a greater number of pixels than region a, giving inaccurate results. The problem gets worse the closer the two surfaces are (e.g. a sheet of paper resting on a table). Of course increasing the resolution of the hemicube helps, but at a computational and memory cost, and like most aliasing problems these brute force techniques cannot ever guarantee to give perfect results).

Another issue is that of generating the patches in the first place. Because we don't know in advance where shadows and patches of brightness are going to end up, its impossible using raw Radiosity techniques to make sensible decisions up-front about how to split the scene's original polygonal mesh into smaller patches. If we create too many, we slow down the solution. If we create too few, we miss subtle lighting effects. One solution to this approach is to use a ray casting approach early on to work out where interesting effects are likely to occur; shooting rays from light sources past the edges of potential occluders and onto large surfaces gives us a hit as to where 'discontinuities' in the final result are likely to happen, allowing us to create a greater number of patches in those areas, whilst leaving 'boring' areas relatively untouched.

Finally, since basic Radiosity rendering assumes energy equilibrium in a closed environment, tiny flaws in the original polygonal mesh can give 'holes' in the environment that allow energy to leak out, giving darker results than one would expect.

2.3 Pros and cons of Radiosity

Radiosity solutions produce excellent results for scenes containing primarily diffuse surfaces (which is a large proportion of real-world scenes!). Its shadows are generally softer and more subtle than those generated by Ray Tracing, and can often look more realistic than the crisper umbra shadows of basic Ray Tracing; and it deals with other visual with effects such as **colour bleed**[W]. It cannot cope with transparent or shiny objects.

Unlike Ray Tracing which results in a view-dependent image-space result (i.e. an image projected onto a viewplane as though seen from a particular position in the scene), Radiosity produces view-independent world-space results (i.e. a mesh of polygons each with their own colour, which have that diffuse colour independent of where they are viewed from). This makes it more applicable to interactive applications (games etc.). It's also straightforward to map the colours from the resulting patch mesh onto textures which can then be mapped back on to the model's original and usually simpler polygonal mesh to improve realtime rendering performance.

References

Appel, Arthur (1968). "Some techniques for shading machine renderings of solids". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, pp. 37–45. DOI: `10.1145/1468075.1468082`.

G. Walton (2002). *Calculation of Obstructed View Factors by Adaptive Integration*. Tech. rep.

Goral, Cindy M. et al. (1984). "Modeling the interaction of light between diffuse surfaces". In: *SIGGRAPH Comput. Graph.* 18.3, pp. 213–222. ISSN: 0097-8930. DOI: `10.1145/964965.808601`.

Immel, David S., Michael F. Cohen, and Donald P. Greenberg (1986). "A radiosity method for non-diffuse environments". In: *SIGGRAPH Comput. Graph.* 20.4, pp. 133–142. ISSN: 0097-8930. DOI: `10.1145/15886.15901`.

Whitted, Turner (1979). "An improved illumination model for shaded display". In: *SIGGRAPH Comput. Graph.* 13.2, pp. 14–. ISSN: 0097-8930. DOI: `10.1145/965103.807419`.

Yu, Yizhou, Oscar H. Ibarra, and Tao Yang (1996). *Parallel Progressive Radiosity with Adaptive Meshing*. Tech. rep. Santa Barbara, CA, USA. DOI: `10.1006/jpdc.1997.1309`.