The Design and Implementation of a Radiosity Renderer

Alexandru Telea

Contents

1 Overview of this Thesis **5**

2 The Radiosity Theory **7**
2.1 Radiometry . . . 7
2.1.1 Radiant energy (Q, $[J]$) . . . 7
2.1.2 Radiant flux (radiant power) (Φ, $[W]$) . . . 8
2.1.3 Radiant flux density (E, $M[W/m^2]$) . . . 8
2.1.4 Radiance (L, $[W/m^2sr]$) . . . 9
2.1.5 Radiant intensity (I, $[W/sr]$) . . . 9
2.2 Photometry . . . 10
2.2.1 Reflectance and Transmittance (ρ, $[dimensionless]$) . . . 10
2.2.2 Conclusions . . . 11
2.3 Ideal Diffuse Reflectors . . . 11
2.3.1 Conclusion . . . 12
2.4 The Radiosity Theory . . . 13
2.4.1 The Rendering Equation . . . 13
2.5 Overview of the Radiosity Process . . . 14
2.5.1 Radiosity Methods . . . 14
2.5.2 Form Factors . . . 15
2.6 The Structure of a Radiosity Renderer . . . 17
2.7 Solving the Radiosity Equation . . . 18
2.7.1 Full Radiosity and Progressive Refinement . . . 18
2.7.2 Form Factor Determination . . . 19
2.7.3 Substructuring . . . 24
2.7.4 Adaptive Subdivision . . . 25

3 The Design of a Radiosity Renderer **29**
3.1 Design Overview . . . 29
3.2 Modelling the Environment . . . 30
3.2.1 Polygons . . . 30
3.2.2 Patches . . . 31
3.2.3 Elements . . . 32
3.3 Form Factor Determination . . . 33
3.3.1 General Presentation . . . 33
3.3.2 Ray Casting Implementation . . . 36
3.3.3 Ray Occlusion Testing using Octrees . . . 39
3.3.4 Ray-Polygon Intersection . . . 41
3.4 Adaptive Receiver Subdivision . . . 42

3.4.1 A Non Uniform Element Mesh Implementation 46
3.4.2 Vertex Radiant Exitances Computation 52
3.5 The Progressive Refinement Strategy 57
3.6 An Overview of the Progressive Refinement Loop 59

4 The Close Objects Buffer 61
4.1 Modelling Sharp Shadows with a Radiosity Renderer 61
4.2 Sharp Shadows 63
4.3 The Close Objects Buffer 64
4.4 Building the Close Objects Buffer 66
4.5 Using the Close Objects Buffer 69
4.6 A Two-Level Close Object Buffer 71
4.7 Conclusions 72

5 The Radiant Flux Hit Model 73
5.1 Introduction 73
5.2 The Radiant Flux Hit 73

6 Modelling and Viewing the Environment 77
6.1 Introduction 77
6.2 WINVIEW: A 3D Viewing Application 77
6.2.1 Introduction 77
6.2.2 General Overview 77
6.2.3 User Guidelines 79
6.3 BUILD_3D: A 3D Modelling Application 80
6.3.1 Transformations 85
6.3.2 Colors 85
6.3.3 Material properties 85
6.3.4 Comments 86

A The Radiosity Renderer User Guide 87

B The 3D File Format 93

C The TAKES File Format 95

D Plates 97

Chapter 1

Overview of this Thesis

This thesis presents the design and the implementation of a radiosity renderer, starting from the theoretical assumptions that are made by the diffuse global illumination model and up to the final design and implementation steps. Besides the implementation of all the standard features of the most common radiosity renderers, some new methods for enhancing the rendering speed and quality of the result are presented, implemented and tested on sample scenes. Since most of the features of the renderer can be directly controlled by the user, the application can be used on a wide range of scenes and for obtaining results of the desired quality level.

The first part of this thesis (chapter 2) presents the theoretical basis of the radiosity method. Radiometric and photometric quantities are introduced, the rendering equation is described and particularized for the diffuse global illumination case. An overview of the radiosity process follows, presenting the basic steps performed by a radiosity renderer and introducing the various quantities and methods used within: form factors, progressive refinement, substructuring, adaptive subdivision. The advantages and disadvantages of most of the several techniques and models used in radiosity renderers are compared in order to determine a suitable model for the radiosity renderer that will be implemented.

The second part of the thesis starts with chapter 3, presenting the major design lines of the radiosity renderer that will be implemented, based on the accuracy, speed and memory consumption considerations outlined in the first part. The way the environment to render is modelled is firstly presented (section 3.2), followed by a presentation of the method used for form factors computation (section 3.3), the method used for adaptive subdivision (section 3.4) and for progressive refinement (section 3.5).

The next chapter (4) introduces a new method proposed for enhancing the sharp shadow rendering done with a radiosity method: the close objects buffer. Both the theoretical motivation and the implementation of this technique are presented. Chapter 5 presents a postprocessing technique used for controlling the shadow smoothness. Chapter 6 presents the methods and the software implemented for modelling the scene to be rendered as well as the ones used for viewing the results.

Specific data about the implementation of the various pieces of software used in the radiosity pipeline are presented in the Appendices, as well as plates showing several rendered scenes.

Chapter 2

The Radiosity Theory

2.1 Radiometry

There exist several photometric and radiometric terminologies used in computer graphics, sometimes being different from the ones encountered in illumination engineering. In order to avoid ambiguities in naming the physical quantities, the set of definitions introduced by [RP-16-1986, 1986] will be used. The radiometric and photometric magnitudes that will be subsequently used in this paper are briefly presented here.

Radiometry measures light in any portion of the electromagnetic spectrum. From radiometric point of view, the light is considered as radiant energy that travels through space. The radiometric magnitudes that will be presented are radiant energy, radiant flux, radiant flux density (irradiance and radiant exitance), radiance and radiant intensity. These magnitudes have a physical significance (they can describe surfaces that emit,reflect and absorb light). Alternatively, the radiometric field theory describes light by means of a three-dimensional or five-dimensional 'photic field' that gives the value of a quantity describing light in any point of the space and for any orientation [Moon and Spencer, 1981].

The main importance of the photic field model resides in the fact that it describes light totally independent from any physical surfaces. The physical quantities describing light are now intrinsic properties of the photic field. Consequently the knowledge of the photic field in all points of the space gives a complete knowledge of the lighting of that space for any position and orientation of the observer. The following definitions are valid both for the model considering physical surfaces as for the radiometric field model:

2.1.1 Radiant energy $(Q, [J])$

Radiant energy is the energy carried by light seen as electromagnetic waves. It is usually measured in joules or kilowatt-hours. Considering the light as electromagnetic waves, we can define the spectral radiant energy as being radiant energy per unit wavelength interval:

$$Q_\lambda = \frac{dQ}{d\lambda} \tag{2.1}$$

For different wavelengths one gets different spectral radiant energies (that can be regarded as density of radiant energy with respect to wavelength)(see equation (2.1)).

2.1.2 Radiant flux (radiant power) (Φ, $[W]$)

$$\Phi = \frac{dQ}{dt} \tag{2.2}$$

Radiant flux is the time rate of radiant energy flow. It is measured in watts. Similar to the spectral radiant energy, one can define the spectral radiant flux as being the radiant flux per unit wavelength interval (or, in other words, density of radiant power with respect to wavelength)

$$\Phi_\lambda = \frac{d\Phi}{d\lambda} \tag{2.3}$$

2.1.3 Radiant flux density (E, $M[W/m^2]$)

Radiant flux density is the radiant flux per unit area at a point on a real or imaginary arbitrarily oriented surface. Two cases are distinguished: the flux can arrive at the surface or can leave the surface (for a real surface, the flux can leave it due to reflection or emission). In the case the flux arrives at the surface, the radiant flux density is called irradiance, while in the case the flux leaves the surface the radiant flux density is called radiant exitance. Irradiance is defined as:

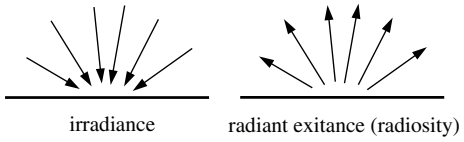


Figure 2.1: Irradiance and radiant exitance

$$E = \frac{d\Phi}{dA} \tag{2.4}$$

where dA is the differential area receiving the radiant flux $d\Phi$.
Radiant exitance is defined as:

$$M = \frac{d\Phi}{dA} \tag{2.5}$$

where, similarly, dA is the differential area from which the radiant flux $d\Phi$ leaves. Both irradiance and radiant exitance are measured in watts per square meter. Similarly to the spectral radiant flux, we can define the spectral irradiance E_λ and the spectral radiant exitance M_λ as irradiance respectively radiant exitance per unit wavelength interval:

$$E_\lambda = \frac{dE}{d\lambda} = \frac{\partial^2 \Phi}{\partial A \partial \lambda} \tag{2.6}$$

$$M_\lambda = \frac{dM}{d\lambda} = \frac{\partial^2 \Phi}{\partial A \partial \lambda} \tag{2.7}$$

A large number of computer graphics texts use the term *radiosity* to denote radiant exitance. The two terms are perfectly identical, the only difference being that the term radiant exitance was adopted by illumination engineering [RP-16-1986, 1986] while radiosity has become mostly used in thermal engineering. Therefore a so-called radiosity-based renderer has to be able to determine the radiant exitances for any position and orientation of the observer in the space and ideally for any wavelength.

2.1.4 Radiance ($L, [W/m^2 sr]$)

Radiance is the quotient of the radiant flux leaving, passing through or arriving at a differential surface, propagated through an elementary cone with a given direction and the apex on the surface, by the product of the solid angle of the cone and the surface's area orthogonally projected on the given direction. (figure 2.2)

$$L = \frac{\partial^2 \Phi}{\partial A \partial \omega \cos\theta} \tag{2.8}$$

Radiance is measured in watts per square meter per steradian. In other words, radiance

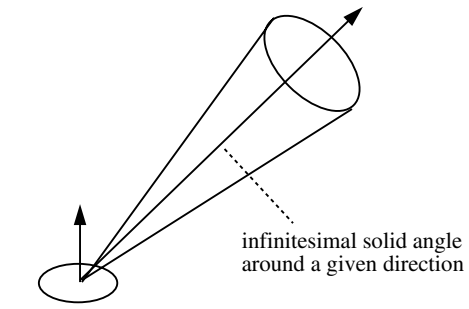


Figure 2.2: Radiance on a differential surface

is radiant flux density per unit solid angle.

2.1.5 Radiant intensity ($I, [W/sr]$)

Radiant intensity equals the radiant flux proceeding from a point light source per unit solid angle in a given direction. It is measured in watts per steradian. The definition of radiant intensity is applied for point light sources since only for these sources one can define a solid angle with the apex being the light source. In practice however one can use the same definition for radiant intensity for finite size light sources, if their sizes are small in comparison with the distance from which they are observed. Returning to the definition of the radiant intensity we have:

$$I = \frac{d\Phi}{d\omega} \tag{2.9}$$

If we want to express the irradiance on a surface dA placed at distance d from a point light source and whose normal makes an angle θ with the direction of the ray to the light source, we have:

$$E = \frac{d\Phi}{dA} \tag{2.10}$$

$$d\omega = dA \frac{\cos\theta}{r^2} \tag{2.11}$$

hence:

$$E = \frac{I \cos\theta}{r^2} \tag{2.12}$$

This last expression is called the *inverse square law* for point sources.

2.2 Photometry

Photometry measures visible light in units that are related to the human eye's sensitivity. In other words, it gives a measure of the light brightness as it is perceived by a human observer. The sensitivity of the human eye to light is a complex function of light's wavelength, amount of radiant flux, time variation of light's radiant intensity and other factors. However, an average response of the eye under 'normal conditions' can be evaluated, thus producing a statistical mapping between the radiometric and photometric quantities. This mapping (the so-called CIE photometric curve [CIE,1924]) delivers the relative perceived brightness of a light source for different wavelengths (sometimes called photopic luminous efficiency). Therefore, there exists a one to one correspondence between the radiometric and photometric quantities. Once we have computed the first ones we can directly evaluate the others and conversely.

The photometric quantities are luminous energy, luminous flux, luminous flux density, luminance and luminous intensity. They will be briefly presented in the following as defined by [RP-16-1986, 1986]:

- Luminous energy

  Luminous energy is photometrically weighted radiant energy. It is measured in lumen seconds.

- Luminous flux (luminous power) (Φ, $[lumen]$)

  Luminous flux is photometrically weighted radiant flux. It is measured in lumens.

- Luminous flux density ($d\Phi/dA$, $[lumen/m^2]$)

  Luminous flux density is photometrically weighted radiant flux density. The photometric equivalent of irradiance is illuminance (sometimes called illumination), while the photometric equivalent of radiant exitance (or radiosity) is luminous exitance (formerly called luminosity). Luminous flux density is measured in lumens per square meter or lux.

- Luminance (L, $[lumen/m^2/sr]$)

  Luminance is photometrically weighted radiance. Luminance is important since the human eye perceives luminance. Intuitively, the luminance of a surface seen from a given angle and a given direction tells how bright the human eye directly will perceive that surface.

- Luminous intensity (I, $[cd, lumen/sr]$)

  Luminous intensity is photometrically weighted radiant intensity. As for radiant intensity, it is defined for point light sources and can be practically used for sources which can be approximated as being pointlike.

2.2.1 Reflectance and Transmittance (ρ, $[dimensionless]$)

Reflectance is an intrinsic property of a physical object, perfectly independent of the photic field that surrounds the object. In fact, it can be regarded as an intrinsic property of the object, similarly to the radiometric and photometric quantity that are intrinsic properties of the photic field, independent of any physical object.

The reflectance of a surface is a dimensionless number that indicates the ratio between the flux incident to that surface in a point and the flux reflected from the surface in

that point. It can be a function of point, if the surface reflects at different points different amounts of the incoming flux:

$$\rho = \frac{\Phi_{reflected}}{\Phi_{incident}} \tag{2.13}$$

The reflectance is generally a function of wavelength too (that is, a surface reflects differently the incident spectral radiant flux of different wavelengths). It is commonly associated with the idea of 'color of a surface' : if a surface is illuminated with white light, the perceived surface color will be determined by its spectral reflectances for different wavelengths. We shall assume that the reflectance of a surface is sufficiently accurately described by its spectral reflectances for red, green and blue wavelengths. Consequently, we shall use three spectral reflectances for any surface: ρ_{red}, ρ_{green} and ρ_{blue}. They all take values in the range [0, 1].

As a remark, for a general surface, reflectance is not only function of point and wavelength but also of incident and reflected directions of the incoming and reflected fluxes, thus giving a bidirectional reflectance distribution function (BRDF). BRDFs can model any type of reflection on a surface (e.g. specular or semispecular surfaces). Moreover, a surface can be transparent of translucent. For ideal transparent objects, the transmittance (amount of transmitted flux relative to the incident flux) can be expressed by a bidirectional transmittance distribution function (BTDF), which is also function of point and incident and reflected directions. We shall limit ourselves to perfect diffuse opaque objects, for which the reflectance will be function of wavelength solely, and the transmittance will be zero.

2.2.2 Conclusions

Radiometry and photometry measure light from energetical, respectively human eye's response point of view. There exists a statistical mapping between the two systems, which makes the use of a single system of quantities sufficient. The human eye's notion of brightness is primarily described by luminosity. Using the above observation, it can be expressed in terms of radiance as well. The field theory describes the light as a photic field giving the magnitudes of the above quantities (radiance, radiant intensity and the others) at any point and for any orientation in space, regardless of the presence of a physical surface at that point. Summing up, in order to model a real environment the radiance photic field must be evaluated for all the points and orientations where an observer has to be placed. For the physical objects in this field, the incident and reflected radiances on their surfaces are related by the reflectance and transmittance of the objects.

2.3 Ideal Diffuse Reflectors

An ideal diffuse reflector is a surface that has a radiance (or luminance) independent of the viewing direction. In other words, the reflectance of the surface is not dependent on the incident or reflected directions of the flux, so the surface looks equally bright from any viewing direction. Such a surface is called also a *Lambertian reflector*. Taking a differential area dA on a Lambertian surface, we have: For the differential area dA, the radiance L is constant. Hence:

$$L = \frac{dI}{dA\cos\theta} = \frac{dI_n}{dA} \tag{2.14}$$

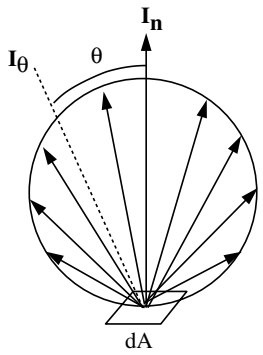


Figure 2.3: Reflection from a Lambertian surface

where I_n is the intensity of a ray leaving the surface in the normal direction. From the above expression and the definitions of radiance and irradiance we get the expression of the intensity of a ray leaving the area dA at an angle theta:

$$I_\theta = I_n \cos\theta \tag{2.15}$$

This expression is called the Lambert's cosine law. For a Lambertian surface, there exists a proportionality relation between the radiance and radiant exitance in an point:

$$M = \pi L \tag{2.16}$$

We have seen that the quantity perceived by the human eye is luminosity or radiance. Using the above relation, radiant exitance can be computed as well and used for display purposes (since it is proportional with the radiance). This explains why certain radiosity renderers are described as computing and displaying 'radiosity' (radiant exitance). Moreover, since $M = d\Phi/dA$ by definition, we get:

$$L = \frac{M}{\pi} = \frac{1}{\pi}\frac{d\Phi}{dA} \tag{2.17}$$

which tells that the eye perceives the density of flux per surface unit. For an equilibrium state, (i.e. a time-independent photic field) the eye perceives the density of radiant energy per unit area. This explains why energy is sometimes used in the computations of some radiosity renderers and energy times area is used for displaying purposes. Moreover, the relation between the BRDF of a perfect diffuse reflector and its reflectance is:

$$f_{BRDF} = \frac{\rho}{\pi} \tag{2.18}$$

2.3.1 Conclusion

An ideal diffuse reflector (Lambertian surface) exhibits an incident and reflection direction independent radiance. The radiance is proportional to the radiant exitance and to the radiant energy per unit area. Therefore, any of the previous quantities can be computed and used for display purposes, all being the same up to a scaling constant.

2.4 The Radiosity Theory

2.4.1 The Rendering Equation

It has been described in the previous sections that the knowledge of the photic field in any point of the space provides all the necessary information for rendering an arbitrary view of the environment. Since the value of radiance (or luminosity) in a point depends on the values in other points of the environment, the radiances must be computed out of a global set of equations that describe the dependencies between the values of the light field in different points. Such a set of equations is generally called a *global illumination system* or model. All global illumination models are based on the *rendering equation*. Starting from a flux balance equation expressed in terms of radiance:

$$L_{out}(x, \Theta_{out}) = L_{emit}(x, \Theta_{out}) + L_{refl}(x, \Theta_{out}) \tag{2.19}$$

which gives the radiance leaving a point **x** in a direction Θ_{out} as the sum of the radiance emitted from the point **x** in the direction Θ_{out} and the radiance reflected from point **x** in the direction Θ_{out}, we obtain an equation giving the radiance for any point and direction in terms of the radiances of all points of all surfaces in the environment, called the rendering equation [Kajiya 86]:

$$L_{out}(x, \Theta_{out}) = L_{emit}(x, \Theta_{out}) + \int_{all\ points\ x'} \delta(x, x')\, f(x, \Theta_{in}, \Theta_{out})\, L_{out}(x', \Theta_{in}) \frac{\cos(\theta_{in})\cos(\theta'_{out})}{\|x - x'\|^2}\, dA' \tag{2.20}$$

where the integral is done over all points **x'** (all differential areas dA') on all surfaces in the scene, θ_{in} and θ'_{out} are the angles formed by the normals at the surfaces in points **x**,**x'** with the vector **xx**' and $f(x, \Theta_{in}, \Theta_{out})$ is the BRDF at point **x** for directions Θ_{in} and Θ_{out}. The term δ(**x**, **x'**) is the so-called occluding (or visibility) term : it equals one if point **x'** is visible from point **x**, and zero otherwise. It is typically impossible to solve the rendering equation even for a rather simple environment. There are several particularizations of it, that reduce the number of paths the light can travel between surfaces and the number of points for which it is solved. These particular forms can simulate realistic lighting up to different degrees.

Ray tracing, for example, is based on an equation in which all BRDFs model perfect specular surfaces (that is, they are zero in all directions but the specular reflection direction). A more advanced ray tracing method will model more complex BRDFs that are different from zero for several directions around the specular reflection direction, thus modelling rough specular surfaces. An even more advanced ray tracing technique will assume that the BRDFs are non zero for several directions over the hemisphere around x different from the incident and specular reflection directions (these directions can be stochastically distributed over the hemisphere using Monte Carlo methods [Kajiya et al., 1986]. The Phong illumination model [Phong, 1975] is based on an equation similar to the one used by ray tracing, but this time the only incoming radiance will be the one of the light sources. Moreover, the light sources are assumed to be pointlike. Therefore, the integral becomes a sum taking into account the radiances of the light sources. No inter-surface reflections are taken into account. This model is called a *local illumination* model, since the illumination of a point is determined only by the light sources.

The techniques usually called *radiosity methods* are based on a rendering equation which models all surfaces as perfect Lambertian reflectors. Therefore, the radiance is a function solely of point (it is independent on the direction). Moreover, the reflectance is assumed to be independent on both incident and reflected directions, being also a function of position only. Using the relation (2.18) between the BRDF and the reflectance of a diffuse surface, the rendering equation becomes:

$$L(x) = L_{emit}(x) + \rho(x) \int_{allpointsx'} L(x') \frac{\cos\theta_{in} \cos\theta'_{out}}{\|x - x'\|^2} \delta(x, x')\, dA' \tag{2.21}$$

2.5 Overview of the Radiosity Process

This section will give an overview description of the radiosity method and the structure of a radiosity renderer. The radiosity method is firstly presented with its particularization of the rendering equation. Form factors and their main properties are then presented. Finally the outline of a radiosity renderer is described with the several assumptions and requirements it has to meet. The techniques used to implement the assumptions and satisfy the requirements are briefly reviewed.

2.5.1 Radiosity Methods

Taking into account the facts described in the previous section, a radiosity method is based on an algorithm that solves the equation (2.21) for any desired point of the environment. The equation can be solved only for a *finite* number of points **x** and, for a point, the quantity under the integral can be evaluated only for a *finite* number of points **x'**. The common approach is to discretize the environment by dividing all surfaces in small *patches* (called also elements) and evaluate the radiance only once per patch, supposing that it has a constant value for all points on the same patch and that the reflectance has a constant value per patch as well. The rendering equation becomes:

$$L_i = L_{i_{emit}} + \rho_i \sum_j L_j \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} \delta_{ij}\, dAi\, dAj \tag{2.22}$$

where L_i,L_j are the (constant) radiances of patches i, j, A_i,A_j are the areas of these patches, δ_{ij} gives the visibility of dA_j from dA_i, r is the distance from dA_j to dA_i and the integral is now performed over the patches i and j. This equation can be rewritten as:

$$L_i = L_{i_{emit}} + \rho_i \sum_j L_j F_{ij} \tag{2.23}$$

where:

$$F_{ij} = \frac{1}{A_i} \int_{A_j} \int_{A_i} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} \delta_{ij}\, dAj\, dAi \tag{2.24}$$

The quantity F_{ij} is called the *form factor* between patches i and j and it equals to the amount of radiance caused by patch j on patch i over the total radiance of patch j. There exists a reciprocity relation between form factors (easily derived from (2.24)):

$$A_i F_{ij} = A_j F_{ji} \tag{2.25}$$

Since radiances are proportional with radiant exitances and radiant fluxes for diffuse surfaces, we can rewrite the equation (2.23) in terms of radiant fluxes:

$$\Phi_i = \Phi_{i_{emit}} + \rho_i \sum_j \Phi_j F_{ji} \tag{2.26}$$

or in terms of radiant exitances as well:

$$M_i = M_{i_{emit}} + \rho_i \sum_j F_{ji} \tag{2.27}$$

A radiosity renderer must therefore *a*) solve the radiosity equation (in one of its previous forms), and for solving it must *b*) have a mean of evaluating the form-factors F_{ij} between any two pair of patches i, j.

2.5.2 Form Factors

There exists an alternative derivation of the expression of the form factor F_{ij} between two finite-sized patches i and j. Firstly, a *differential form factor* is defined, expressing the fraction of the flux emitted by differential area dA_i which is received by differential area dA_j:

$$dF_{dA_i dA_j} = \frac{d\Phi_{dA_i -> dA_j}}{d\Phi_{dA_i}} \tag{2.28}$$

The solid angle subtended by dA_j at dA_i is:

$$d\omega = \frac{dA_j \cos\theta_j}{r^2} \tag{2.29}$$

The differential flux leaving dA_i in a direction θ_i is:

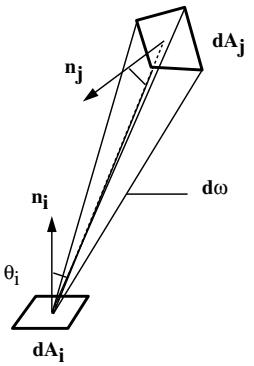


Figure 2.4: Differential form factor between two infinitesimal areas

$$d\Phi(\theta_i) = L(\theta_i)\cos\theta_i dA_i d\omega = d\Phi_{dA_i -> dA_j} \tag{2.30}$$

But dA_i is a Lambertian surface. Therefore, $L(\theta_i)$ is independent on θ_i, $L(\theta_i) = L_i = ct$ over dA_i. From equation (2.30) we get:

$$d\Phi_{dA_i -> dA_j} = L_i \frac{\cos\theta_i \cos\theta_j}{r^2} dA_i dA_j \tag{2.31}$$

Since dA_i is a Lambertian surface, the total flux emitted by dA_i is :

$$d\Phi_i = M_i dA_i = \pi L_i dA_i \tag{2.32}$$

Using (2.32) in (2.28) we get the expression of the differential form factor $dF_{dA_i dA_j}$:

$$dF_{dA_i dA_j} = \frac{\cos\theta_i \cos\theta_j}{pir^2} dA_j \tag{2.33}$$

The form factor between the differential area dA_i and the finite area A_j will equal the infinitesimal fraction of flux emitted by dA_i and received by A_j:

$$dF_{dA_i A_j} = \int_{A_j} dF_{dA_i dA_j} \delta_{ij} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{pir^2} \delta_{ij}\, dA_j \tag{2.34}$$

Finally, the form-factor between patches A_i and A_j will be:

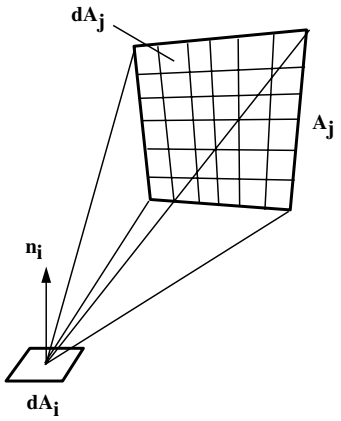


Figure 2.5: Fig.2.8, p43: determining dFdAiAj by integration over Aj

$$F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_i} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} \delta_{ij}\, dA_j\, dA_i \tag{2.35}$$

which is exactly equation (2.24). Remark that the increase in flux of A_j due to A_i can be expressed as:

$$\Phi_{ij} = F_{ij}\Phi_i \tag{2.36}$$

Sometimes it is desirable to express the increase in radiant exitance of A_j due to A_i as:

$$M_{ij} = F_{ji} M_i \tag{2.37}$$

In a closed environment, the flux emitted by a patch i will be entirely received by the other patches, therefore the relation:

$$\sum_{j=1}^{n} F_{ij} = 1 \tag{2.38}$$

As a final remark, the form factor from a patch A_i to itself (F_{ii}) is always zero for convex or planar patches, since the term δ_{jk} is zero for any two areas dA_j,dA_k on a planar or convex surface A_i.

2.6 The Structure of a Radiosity Renderer

As described in section 2.5.1, a radiosity renderer must solve the equation (2.27) and determine the radiances (or fluxes or radiant exitances) for each patch in the environment. Solving such an equation is equivalent to achieving a flux (or energy, for the time-independent case) balance for all patches in an environment, as described by equation (2.27). In the equilibrium state, any patch will have an incident flux equal to the reflected plus the absorbed one. After the solution has been found, the environment can be viewed from any point and in any direction, since the determined patch radiances describes the photic field in any point. The radiosity renderer that will be described here will be based on the following assumptions:

- all environment's surfaces are perfect diffuse reflectors
- all surfaces are discretized in a mesh of elements
- all elements are planar polygons
- for each element, the radiant exitance, radiance, irradiance and flux are constant over all its points
- the global illumination model is given by equation (2.27).

The requirements a radiosity renderer should comply with can be summarized as:

- *accuracy* The computed solution should represent as close as possible the real light field in the environment.
- *speed* The solution should be obtained in a reasonable amount of time. Moreover, it should be possible to get the most accurate solution for an arbitrary allocated amount of computing time.
- *low memory consumption* The renderer should use a reasonable amount of memory. It should deliver the best solution obtainable for the given amount of memory.

The requirements are listed in priority order. One can try to satisfy individually each requirement but since they are closely inter-related a better method is to globally optimize the radiosity renderer's strategy. Given these assumptions and requirements, the basic tasks that a radiosity renderer must perform will be:

- discretize the environment's surfaces into patches
- solve the radiosity equation (2.27) determining a radiance for each patch. In order to solve the equation, form factors between patches must be evaluated.
- display the environment from any viewpoint and viewing direction.

These three steps are not always perfectly separated. Sometimes the actions of a step are interlaced with the ones of another step. Each of these steps will be outlined in the following.

2.7 Solving the Radiosity Equation

2.7.1 Full Radiosity and Progressive Refinement

The radiosity equation has to be solved delivering a radiance or radiant exitance for each element in the environment. The set of equations of the form (2.27) for all elements in the environment form a linear systems of n equations, where n is the number of elements in the discretized environment. The solution of such a system would provide the radiance and radiant exitance for all the elements. Solving such a system by Gaussian elimination can be quite slow. Iterative techniques like the Jacobi and Gauss-Seidel methods [Golub and Van Loan 1983] can be applied since they will always converge (the matrix is strictly diagonal dominant for planar convex elements having reflectances smaller than 1). These methods can deliver a sufficiently good solution in about eight iterations [Cohen and Greenberg, 1985].

Directly solving the system is impractical: about $n^2/2$ form factors must be determined and stored for n elements. Moreover, one iteration has a complexity of $O(n^2)$. Both these disadvantages are removed by the *progressive refinement* method which is also based on the equation (2.27). While the "full radiosity method" attempts to evaluate the radiant exitance of an element as function of the exitances of all other elements (hence it is said that an element *gathers* radiant flux), progressive refinement is based on *shooting* the radiant flux of an element to all the others.

Using equation (2.27) we can express the increase in radiant exitance ΔM_j of an element j due to element i's radiant exitance:

$$\Delta M_j = \rho_j M_i F_{ji} \tag{2.39}$$

Using the above equation the amounts of radiant exitances for *all* elements j (due to the radiant exitance of a given element i) can be determined in $O(n)$ time. We say that element i *shoots* its radiant exitance (or its radiant flux) to all other elements. Shooting radiant exitance will typically start with the elements having the largest amount of radiant flux (which will cause most of the environment's illumination). Each element will keep two values: its current radiant exitance and its *unshot* radiant exitance. At the beginning all elements have their radiant exitance equal to their unshot radiant exitance and equal to their initial radiant exitance (this value is different from zero only for the so-called primary light sources, i.e. the elements that emit light). After the element having the largest value of *unsent* radiant flux has been found, equation (2.39) is used to shoot radiant exitance from this element i to all other elements. The unshot radiant exitance of this element is now set to zero and the next shooting iteration proceeds.

The progressive refinement can stop after any desired number of iterations and the scene with the radiant exitances computed so far can be viewed. This allows the rendering of progressively better images without the need of waiting for the completion of the solving of the full system of equations. Shooting from the element having the largest unshot amount of radiant flux ensures that the solution will converge to a good approximation of the real light field after a smaller number of iterations than if we shoot radiant flux from randomly selected elements. The complexity of this method is $O(Nn)$ where N is the number of iterations we wish to perform (number of shots) and n the total number of elements in the environment. Moreover, there are only n form factors needed to be stored and computed per iteration, therefore the memory requirements drop down to a reasonable level (with the observation that form factors may need to be recomputed if they are discarded after an iteration). [Cohen et al., 1988] have shown that an acceptable result was delivered by the progressive refinement method in a much smaller time

than it was required for the full radiosity algorithm to perform its *first* iteration.

2.7.2 Form Factor Determination

In order to shoot the radiant exitance of an element to all the other elements, form factors from this element to all others have to be determined. Since this process consumes most of the time of radiosity rendering (it can be about 90 percent of the total processing time) there are several methods that attempt to speed up form factor computations by making several approximations.

The form factor between two elements i and j is defined by equation (2.35). The main problem with the computation of $F_{A_iA_j}$ is that the occlusion term δ_{ij} has to be estimated for all pairs of differential areas dA_i, dA_j over the elements i, j. Several methods that approximate the double area integral with sums over small finite areas of elements i, j exist. They are essentially based on numerical quadrature techniques.

The Hemicube Method

The *hemicube method* [Cohen and Greenberg, 1985] is based on the fact that two elements subtending the same solid angle from an emitter (and being unoccluded with respect to it) will receive the same amount of flux from that emitter, hence will have the same form factor with respect to it. In order to determine the form factor $F_{dA_iA_j}$ between a differential area dA_i and a finite area A_j a *hemicube* is placed on dA_i, having its five rectangular faces subdivided into small *cells*. The form factor $F_{dA_iA_j}$ will be:

$$F_{dA_iA_j} = \sum_{all\ cells\ j} \Delta F_{dA_iA_{cell_j}} \tag{2.40}$$

where $\Delta F_{dA_iA_{cell_j}}$ is the form factor between the area dA_i and hemicube's cell j and the sum is done over all the cells j that are covered by the projection of A_j on the hemicube's faces. The delta form factor $\Delta F_{dA_iA_{cell_j}}$ is approximated by:

$$\Delta F_{dA_iA_{cell_j}} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} \Delta A_j \tag{2.41}$$

where all the quantities have the same meanings like the ones from equation (2.33) (both dA_i and cell j are approximated as being infinitely small areas). Occlusion is taken into account when projecting an element over a hemicube for each cell: that is, a cell will be *totally* covered or uncovered by an element's projection and there is only *one* element which is retained as covering a hemicube cell. Occlusions are therefore evaluated only for a number of times equal to the number of hemicube cells (the number of hemicube cells is sometimes referred to as the hemicube's resolution). Projection of an element over the hemicube is done by projecting the element on each of the hemicube's faces in turn: clipping to the viewing pyramid determined by the hemicube's center and the face we project on is implemented using a perspective transformation that maps this pyramid into an axis-aligned parallel view volume.

There are two main approximations that are done by the hemicube approach:

- occlusion is estimated only for the *fixed* set of hemicube cells and only once per cell. Aliasing can occur due to the regular nature of the hemicube subdivision into cells which doesn't take into account the relative sizes and orientation of the elements in the scene. This problem can be partially solved by randomly rotating

the hemicube around the emitter's normal for each shooter [Wallace et al., 1987]. Another problem is that projections of small elements will most frequently cover less than a hemicube cell so they will be simply discarded as radiant flux receivers during the shooting process. This phenomenon is sometimes referred to as 'leaking' of small elements between the hemicube's cells. This is a more serious problem since it can't be generally alleviated by simply increasing the hemicube's resolution.

- form factors between the emitter and the hemicube cells are approximated by considering the hemicube cells as infinitely small areas. The form factor between a differential area and a finite element is then the sum of the delta form factors corresponding to the cells covered by the finite element's projection on the hemicube. Again, approximating the integral (2.34) with this sum can be inaccurate if the hemicube's resolution is too low.

The Cubic Tetrahedron Method

The *cubic tetrahedron method* is based on the same assumptions as the hemicube. It represents a triangular pyramid placed on the top of the differential area dA_i and having its three triangular faces subdivided into cells in a similar manner with the hemicube. In order to compute the form factor $F_{dA_iA_j}$, we proceed exactly as for the hemicube: the area A_j is projected on the three tetrahedron's faces and the delta form-factors of the covered tetrahedron's cells are summed up. There are a couple of advantages of using a cubic tetrahedron instead of a hemicube for computing form factors:

- all the three tetrahedron's faces will have the same geometry for evaluating their delta form factors, while a hemicube must use different expressions for its top face and its four lateral faces.
- the finite areas A_j have now to be projected on just three triangular faces instead of five rectangular faces as required by the hemicube algorithm.
- less memory (as compared with the hemicube method) is needed to store the delta form factors for a cubic tetrahedron due to the symmetry of its faces.

It has been shown however [Beran-Koehn and Pavicic, 1992] that a hemicube samples its environment with twice the number of cells as a cubic tetrahedron with the same resolution and that the average delta form factors are the same for the two geometries when they have the same number of cells. A cubic tetrahedron has therefore to have a resolution of $\sqrt{2}nx\sqrt{2}n$ cells to be equivalent with a hemicube of nxn cells.

The Single Plane Method

The hemicube or cubic tetrahedron can be replaced by a single plane placed above the desired area dA_i. This plane (being in fact a finite rectangular area) is divided into cells similarly to the hemicube's or cubic tetrahedron's faces. The finite areas A_j are projected on this plane and the covered cells' delta form factors are summed up to compute the form factor $F_{dA_iA_j}$. The single plane's advantage is that the projected polygons have now to be clipped against a single plane (instead of the 5 planes required by the hemicube or 3 planes for the cubic tetrahedron): speedups can reach 100 percent compared to the hemicube [Recker et al., 1990]. There are however a couple of drawbacks appearing in the single plane algorithm:

- the single plane does not cover the whole hemisphere above the differential area dA_i. In other words, the radiant flux emitted by this area will be underestimated (polygons close to dA_i's horizon will not have any projection on the plane, therefore will not receive any radiant flux from the emitter. This underestimation can be acceptable if the plane is close enough to the emitter: [Sillion and Puech, 1989] showed that the amount of flux 'escaping' under the plane is approximately $2(H/S)^2$, where H is the emitter-plane distance and $2S$ the size of the plane's edge. A ratio S/H of 14:1 will limit the error to 1 percent of the emitter's radiant flux.

- the plane's cells delta form factors will now vary largely: the cells that are faraway from the differential emitter dA_i will have small form factors while the cells right above the emitter will have comparatively very large form factors. In order to avoid aliasing, the delta form factors have to be kept in a range comparable to the ones provided by the hemicube. Doing this by directly subdividing the single plane into an uniform grid of cells would result in a very large number of cells compared to the hemicube.

  Sillion and Puech (1989) used variable-sized cells that they called *proxels* over the plane such that all these cells had approximately the same delta form factor. The disadvantage of this approach is that the cells' item-buffers can no longer be determined using a classic z-buffer scan conversion since the cells have a variable size. Warnock's area subdivision algorithm was used to determine the cells covered by the polygons' projections on the plane. Recker et al. (1990) showed that it is possible to reduce the number of cells by nonuniformly subdividing the single plane into two zones: large cells are used near the plane's edges and smaller ones over the area right above the emitter. Two clipping passes (one against the large plane and one against the smaller central area) and usual z-buffer based scan conversion can be used. Recker's method uses only two sizes for the plane's cells but is faster than the method using proxels since hemicube-like algorithms can still be used for a zone containing cells of the same size.

Stochastic Methods

Stochastic (Monte Carlo) techniques can be used to determine the form factor $F_{dA_iA_j}$. Several points are randomly distributed over the emitter's surface. A hemisphere is placed over this area and rays are shot from its center through the points where the normals to the emitter's surface emerging from the random points will intersect the hemisphere's surface. This method will practically create a distribution of random shooting rays over the hemisphere's surface which is consistent with the magnitude of the form factors between the emitter and areas with different projections over the hemisphere: surfaces near the emitter's horizon have small form factors (few rays will be shot towards the emitter's horizon) while surfaces seen 'above' the emitter will have larger form factors (more rays will be shot in the direction of the emitter's surface normal) [Maxwell et al., 1986],[Malley, 1988].

The emerging rays will be traced until they encounter a surface. The surface's form factor will be finally equal to the fraction of the rays intercepted by it out of the total number of rays shot through the hemisphere. The advantages of stochastic form factor methods are that they can easily be applied to both planar and curved surfaces, they don't require a complex data structure like the hemicube and the other previously described algorithms, can be speeded up by traditional ray tracing accelerating techniques, minimize aliasing artifacts, don't require perspective projection and clipping algorithms and

are simple to implement. The major disadvantage (as compared to a hemicube method) is that stochastic approaches are accurate only for a very large number of rays. Moreover, there is a greater chance for small polygons to be missed by the rays than it is for them not to cover any hemicube cell, since rays are shot *randomly* in the scene while the hemicube will project the scene's surfaces over its faces. Consequently, small objects may be almost entirely missed from the process of gathering radiant flux. This results in the fact that they can appear almost dark in the final image. A corollary of these problems is that small receivers that are neighbours in the scene are not guaranteed to receive the same (or even a similar) amount of rays (not even in the case of a totally unoccluded environment). The variations of light intensity in the rendered scene can therefore exhibit a very large amount of noise. This noise can sometimes be less desirable than the amount of aliasing produced by hemicube algorithms.

The stochastic ray casting form factor determination method (sometimes called radiant flux or energy ray tracing) is a case of *undirected shooting* form factor determination method: rays are shot in the scene in random directions without aiming (directing) them at a certain surface. The first surface that will intercept the ray will account for the radiant flux 'carried' by that ray. In contrast to this, *directed shooting* methods will shoot a certain number of rays towards *each* polygon in the scene (this number is determined by the estimated interaction magnitude between the source we shoot from and the target we shoot at). The estimate can be done by computing the solid angle subtended by the target at the source. Directed shooting needs generally less rays than undirected shooting. It is interesting to notice that hemicube algorithms can be regarded as using a *directed shooting* method that cast rays through the hemicube cell's centers. It is both the fixed regular nature of these rays and the fact that a hemicube cell is either fully occluded or not that causes aliasing.

Target to Source Ray Casting Methods

Target to source ray casting methods reverse the previous approaches that attempt to compute the form factors $F_{dA_iA_j}$ between the emitter dA_i and all receivers A_j. Instead of computing this form factor, the reciprocal form factor between a point on the receiver A_j and the source will be determined.

The classical approach (Wallace et al. 1989) computes the form factors between each vertex of each receiving element (modelled as a differential area dA_j) and a finite area modelled as a circular disk:

$$F_{dA_jA_i} = \frac{a^2}{r^2 + a^2} \tag{2.42}$$

where a is the radius of the disk of area A_i, r is the distance to the receiver dA_j and the disk is parallel to the receiver [Siegel and Howell, 1992].

The emitter polygon is subdivided into small cells such that each cell can be approximated with such a circular disk. The form factor from the differential vertex area dA_j to the emitter will be the sum of the form factors for all the emitter's cells that are visible from the receiving vertex. In order to determine the occlusions, a ray is cast from the vertex to each emitter cell: if this ray is not occluded by another polygon, the vertex-to-source form factor is updated with the vertex-to-cell form factor. Finally the vertex-to-source form factor will be:

$$F_{dA_jA_i} = \frac{A_i}{n} \sum_{k=1}^{n} \delta_k \frac{\cos\theta_{jk} \cos\theta_{ik}}{\pi r_k^2 + A_i/n} \tag{2.43}$$

where A_i is the area of the source polygon, n is the number of cells (approximated with circular disks) the polygon is divided into, δ_k is the occlusion term of the cell k with respect to the vertex, r_k is the distance from the vertex to the center of cell k and θ_{jk} and θ_{ik} are the angles made by the receiver surface's normal respectively the emitter's normal with the vertex to emitter ray k.

Computing target to source form factors by ray casting has a number of important advantages:

- ray casting can determine the form factor between *any* differential area dA_j in the environment and a finite-size emitter A_i: dA_j is usually placed at the elements' vertices or at the elements' centers but it can be placed anywhere else. Using equation (2.27) we can determine the radiant exitance caused by an emitter A_i at *any* point in the environment.

- as a consequence, radiant exitance is typically evaluated for all the vertices or centers of all the elements. No element will therefore miss its incoming amount of radiant flux from a receiver (as it may happen when using hemicubes or stochastic methods). Moreover, the amount of rays cast is determined just by the precision we wish to sample the emitter: different numbers of rays can be used for different receivers dA_j to sample the same emitter, depending on their relative position. Form factors can be therefore computed very accurately. Moreover, since ray casting performs all intersections at object space precision, physically realistic point like or non-Lambertian area sources can be modelled very easily: all we need is the radiant flux distribution for any outcoming direction for such a source.

- ray casting avoids the aliasing caused by hemicube methods. Since rays are cast in the precise direction of the sources, we can say that target to source ray casting is a type of *directed shooting*.

- target to source ray casting can benefit from the same advantages that stochastic ray casting uses: ray-object intersections (performed at object space precision) can be accelerated using a large variety of techniques (including, for example, accurate intersections with implicit surfaces). Moreover, vertex normals can be used to approximate curved surfaces.

- target to source ray casting can be thought as a case of *light source area sampling techniques*. The light source is sampled with a number of rays that can be distributed in any desired way over the source's area. Random distributions can be used to minimize aliasing effects caused by *source*'s regular sampling patterns (as stochastic methods were used to minimize aliases due to *receivers*' regular sampling). A less expensive variant of this involves jittering of a regular light source sampling pattern. Nonuniform light source sampling can be used in occluded environments in order to adaptively sample the areas of high gradients of the light source, delivering very accurate results and minimizing the number of rays to be used (for example, Wallace et al. (1989) used as few as 16 rays per vertex with very good results. More complex light source sampling techniques are performed by wavelet and eigenvector radiosity methods).

- target to source ray casting determines one form factor at a time: the form factor from the target point to the source. Compared to this, all hemicube approaches determine *all* form factors from the source to all receivers. Besides the lower memory requirements, this allows us to determine only the form factors for the interest points.

- a final advantage is that radiosity computations based on vertex-to-source form factor determinations will directly deliver vertex radiant exitances for Gouraud shading the environment. This might not be a very important advantage though since element radiant exitances may be needed anyway for adaptive subdivision computations.

The only potential disadvantage of target to source form factor ray casting is that, while a hemicube has a $O(n)$ complexity for determining the form factors from an emitter to all other n polygons in the scene, target to source ray casting has a $O(in)$ complexity for the same task, where i is the average number of occlusion tests per source-target ray. In other words, a ray must be theoretically tested against *all* objects in the scene for occlusion. Spatial partitioning techniques, ray tracing acceleration methods and implicit surface equations can however upper bound this average intersection number to a very small value.

The Five-Times Rule

We have seen that both the hemicube, cubic tetrahedron, single-plane and target to source form factor determination methods assume that the source can be approximated by a differential area. In the case the emitter is too large for such an approximation to hold, it has to be subdivided into smaller cells (or several rays are to be used in the case of a ray casting based method) and form factors are to be evaluated from each of these cells to the desired target point.

[Murdoch, 1981] demonstrated that if a Lambertian rectangular emitter is approximated by a point like source and the distance from the illuminated (target) point to the rectangle is at least five times the rectangle's maximum projected width then the absolute illumination error will be less than 1 percent. This rule, known as the *five times rule*, allows us to quantitatively determine the level of subdivision that must be applied to an emitter in order to accurately evaluate the form factor between it and any point in the space: the emitter element will be subdivided such that the distance from any of the resulting subdivision cells to the illuminated point is at least five times greater than the maximum projected cell width (the cell's width is projected on a direction normal to the ray between the cell and the illuminated point).

2.7.3 Substructuring

There are two reasons for subdividing the surfaces of a scene in a radiosity renderer: firstly, radiant exitance variations have to be accurately captured for rendering a realistic final image and secondly, emitters have to be subdivided into small areas such that form factor computations from the emitter to the receivers are accurately done (according to an accuracy criterion similar to the five-times rule previously described).

Subdivision for accurately capturing the radiant exitance over the scene's surfaces will generate the *elements*. It is generally known as *adaptive subdivision* or *adaptive meshing* and will be described later. Subdivision for accurately computing form factors mainly involves choosing an efficient light source meshing: the techniques performing this meshing are generally known as *substructuring* methods.

An emitting surface has to be subdivided such that all the resulting cells can be approximated with point light sources. The resulting cells will generally be larger than the *elements* created for accurately sampling the radiant exitance function, due to the usual lightsource-target distances. The subdivision of a light source into cells (usually

called *patches*) is therefore independent of a receiver's subdivision into elements. Since these patches are generally larger than the elements, a substructuring scheme can be designed where all the surfaces are subdivided into patches and the patches are subdivided in their turn into elements. Form factors will be computed between patches (radiant flux shooters) and elements (radiant flux receivers). For a scene with N patches and M elements, where $N < M$, there will be $O(NM)$ form factors to be compute instead of $O(N^2)$ that would have been needed if we had evaluated element to element form factors [Cohen et al., 1986]. A simple substructuring method will use a *uniform* emitter subdivision into patches: in such a case, there might still be the need of subdividing each patch when shooting radiant flux from it to some elements if these receiving elements are close enough to the patch such that the five-times rule does not hold. A more complex substructuring scheme will subdivide an emitter to *different* levels, according to the distance from it to the receiver. The same emitter may then be subdivided up to different levels when computing the form factor between it and different receivers. [Hanrahan et al., 1991] and [Cohen and Wallace, 1993] have implemented such a hierarchical substructuring scheme, where a quadtree is used to subdivide each emitter into a hierarchy of patches. The level an emitter will use to shoot its radiant flux is dynamically determined by the expected interaction magnitude with each individual receiver. Other forms of hierarchical scene substructuring have been used for accelerating form factor computations: [Xu et al., 1989] and [van Liere, 1991] have subdivided the scene into *subscenes*, using virtual walls that are subdivided into patches and act as radiant flux transmitters between the subscenes. Radiance computations (using progressive refinement) is locally done for each subscene. When all subscenes have computed their local solution, radiant flux is exchanged between the neighbouring subscenes through the virtual walls. This scheme attempts to minimize the number of patch-to- element interactions, since a patch will directly interact only with elements in the same subscene and with the subscene's virtual walls. Another substructuring method is *grouping* [Kok, 1993], [Rushmeier et al., 1993]. Small neighbouring surfaces are grouped and radiant flux is received by the group as a single entity, after which it is distributed to the surfaces contained in that group. This method minimizes the number of patch-to-element interactions since a patch will now interact with a group and not with each individual (presumably small) element of that group.

2.7.4 Adaptive Subdivision

As previously outlined, a receiving surface has to be subdivided in order to accurately capture the value of the radiance field over it. Since the final result of a radiosity rendering is a discretization of the environment into elements with a single radiance value computed per element, these elements should be small enough such that the variations of the real radiance field over such an element should be reasonably small for viewing purposes. Nonuniform subdivision comes as a natural alternative: smaller elements will be used in areas of rapid variation of the radiance field and larger ones in areas where this field exhibits a slower variation. The methods that attempt to automatically discretize an environment such that the above constraint is obeyed are generally known as *adaptive subdivision* or adaptive refinement techniques. The requirements that a subdivision (meshing) method must obey are:

- the radiance solution has to be accurately captured. This is the most difficult requirement, since the solution is not known beforehand but is to be computed on a given mesh. Several prediction methods have been designed in order to determine

the areas of high radiance gradients (which will be refined). The most efficient of them do not rely on the existing mesh when trying to predict refinement areas (are said to be *mesh-independent* or working in *object space*. The others may use the solution already computed on the current mesh to initiate refinement over the computed solution's high gradient areas.

- vertex-to-vertex radiance differences are to be minimized. Since linear interpolation of the radiance solution will be finally used for displaying the solution (Gouraud shading), radiance differences among the vertices of an element has to be minimized in order to eliminate unpleasant (e.g. Mach banding) effects.

- element aspect ratios are to be minimized. Besides the reasons presented above, an element is typically characterized by its center when it is involved in form factor computations. Approximating it with a differential area will give the best results when the element's shape is closest to a square (for quadrilateral elements) or to an equilateral triangle (for triangular elements).

- the total number of elements must be kept at minimum: refinement must be carried on only in the areas where it is really necessary in order to minimize the time and memory consumption incurring when having a large number of elements.

- the mesh must be smoothly graded: a mesh is said to be *smoothly graded* if the difference in areas of neighbouring elements is kept to a minimum. A smoothly graded mesh will generally have elements with a small aspect ratio. The overall advantages of smoothly graded meshes are Mach banding minimization and more accurate form factor determinations (by approximating an element with its center) [Baum et al. 1989].

- the mesh must be balanced: the elements of a *balanced mesh* will have no more than two neighbour elements on any of their edges [Baum et al., 1991]. Having a balanced (or almost balanced) mesh together with a small element aspect ratio ensures that the mesh will be smoothly graded.

There are several adaptive subdivision methods, differing mainly on the type of information they use in order to detect the high radiance gradient areas that need to be refined:

- *user-defined nonuniform meshing:* the zones of high radiance gradient are supposed known in advance by the user that will indicate the areas to be finely meshed. This is the simplest method for generating a nonuniform mesh but it has the obvious disadvantage of transferring the task of mesh refinement entirely to the user.

- *adaptive repositioning meshing:* the renderer starts with an initial mesh. After computing a solution on this mesh, the vertices of the elements exhibiting high radiance gradients on their surface will be *repositioned* such that the elements' boundaries become aligned with the shadow boundaries (they become normal to the radiance function's gradient over that element). The method provides just a limited improvement (no element subdivision takes place) to the initial mesh, is dependent on the initial mesh's resolution and may produce thin, poorly shaped elements [Aguas and Muller, 1993]. The method can be used as a postprocessing step that repositions the final mesh before viewing the solution.

- *discontinuity meshing:* the shadow boundaries (areas where a sharp variation of the radiance field is visible) are detected and the elements intersected by these

boundaries are subdivided until they do not contain any *discontinuity* in the radiance value (i.e. they are not intersected by a shadow boundary) or they are considered small enough for viewing purposes [Heckbert 1992, Lischinski et al. 1992, Cohen and Wallace 1993]. Shadow boundaries due to primary light sources can be detected by using a preprocessing step that shoots shadow rays from these light sources to determine where shadows will be the most likely to occur [Nishita and Nakamae, 1985], [Campbell and Fussell 1990]. The disadvantage of discontinuity meshing is that full-fledged shadow casting algorithms working in object space can be very slow (especially if we try to use them for casting shadows from the secondary light sources as well) and that keeping track of complex shadow boundaries crossing elements can be a very delicate process.

- *gradient-based meshing:* this simple method examines the gradient of an already computed solution over the current mesh. The areas where a high gradient is found will be refined and, in case of a progressive refinement method, radiant flux will be reshot towards the newly created elements. The refinement process stops when the gradient of the solution over an element is below a desired threshold or the element is small enough for viewing purposes. This method is relatively simple to implement, requires no object space computations and is typically applied after each radiant flux shooting iteration in a progressive refinement approach. The main disadvantage it has is that it can not guarantee that fine radiance variations will be detected if the initial mesh was too coarse to completely miss these details (the fine shadow of a pencil on a table may be completely missed if the initial mesh's elements are so large that they never intersect this shadow). This is essentially a sampling problem and the best gradient-based subdivision can do is to use an initial mesh fine enough for capturing the desired shadow detail level.

- *other methods:* there exist other refinement methods that use more or less elaborate prediction functions for determining the radiance gradient over the scene's surfaces in order to find out the areas to refine. Different types of information are involved in these prediction functions (scene's geometry, primary and secondary light source positions and orientations, etc). Although the prediction methods can be very elaborate and expensive, the obtained meshing and solution can have a very high quality, unattainable by other methods (for example, wavelet radiosity [Gortler et al. 1993], eigenvector radiosity [DiLaura and Franck 1993]).

The dynamic management of a general nonuniform mesh (element subdivision, insertion and removal) can be quite complicated. The classical approach is to use a winged-edge data structure, that records information about all the vertices, edges and elements over the meshed surface [Baumgart 1974, Glassner 1991]. The advantage of such a structure is that it provides all desired connectivity information for a vertex, edge or element in a constant time. The disadvantage is that is rather complicated to be maintained and that it takes a considerable amount of memory. Other less general structures can prove to be more efficient for the strict purpose of encoding a nonuniform mesh used for a radiosity renderer's adaptive subdivision.

Chapter 3

The Design of a Radiosity Renderer

This chapter will present the full design of a radiosity renderer. The first section 3.1 will recall the requirements a radiosity renderer must satisfy and will outline the models we have used in order to satisfy these requirements. The section 3.2 presents the model that was used to describe, store and manipulate the tridimensional environment to be rendered. Section 3.3 presents the method chosen for computing form factors (ray casting) and various issues concerning its implementation. Section 3.4 presents the methods used for local receiver mesh refinement. Section 3.4.1 describes the practical structure used to implement a dynamic nonuniform mesh. Section 3.4.2 describes the way the vertex exitances finally used for displaying the rendered scene are obtained out of the computed element exitances.

3.1 Design Overview

The design of a radiosity renderer has to take into account the three requirements presented in section *The Structure of a Radiosity Renderer*:

- *ACCURACY:* The radiant exitance field has to be accurately evaluated for all the points on all the surfaces of our environment. In order to accurately capture this field, adaptive subdivision of the receivers will be used, based on several types of criteria. In order to accurately determine the radiance caused by an emitter at a point in space, the emitters will be adaptively sampled as well. The presented renderer will evaluate form factors using ray casting from receivers to the light sources.
- *SPEED:* The renderer attempts to deliver the best obtainable solution for a given amount of time. Progressive refinement will be used, based on directed shooting from sources towards receivers and on shooting from source patches having the highest unshot radiant flux value. A two-level substructuring scheme will be used: patches are the shooting units and elements are the receivers. Several elements are grouped in a patch and the radiance of that patch is shot as a whole, thereby minimizing the number of form factor computations. Several techniques will be employed to accelerate the ray tracing form factor determinations and culling the elements occluded from the radiant flux shooters. The nonuniform

mesh will be encoded using a structure that will attempt to be simpler and faster than the classical winged-edge structure.

- *MEMORY REQUIREMENTS:* The renderer attempts to be very efficient in terms of used memory. Adaptive source subdivision for accurate form factor computations will be performed on the fly (in contrast to the hierarchical radiosity technique) thereby minimizing the memory requirements and still providing a good quality of the solution. The mesh encoding structure is more efficient in memory terms than the winged-edge structure, providing a similar access time and element dynamic subdivision time.

Summarizing, a radiosity renderer based on progressive refinement, using directed shooting with ray-cast form factors, a two-level patch and element substructuring scheme, adaptive element subdivision and patch sampling will be described. The renderer accepts as input a polygonal environment featuring perfectly diffuse reflecting polygons and light sources and delivers the radiant exitance solution as a set of elements with radiant exitances at vertices that can be displayed using Gouraud shading. Several techniques that were developed in order to satisfy the accuracy, speed and low memory requirements will be presented as well.

The next sections present the way in which the previous design requirements have been incorporated in the renderer's implementation.

3.2 Modelling the Environment

This section explains the model used by the radiosity renderer for describing and manipulating the tridimensional scene that has to be rendered. In decreasing order of priorities, here are the requirements that a model of the 3D scene must comply with:

- *completeness:* the model has to contain all the data that the renderer needs about the scene (supplementary informations might be computed on the fly out of the initial data).

- *access speed:* the model has to allow a fast (possibly random) access to all the information about the scene. Random access is particularly important since it is quite frequently performed by different parts of the renderer.

- *flexibility:* parts of the data structure may need to change while the rendering evolves in time. Adaptive subdivision (possibly up to an arbitrary level) requires that elements are generated, subdivided and destroyed on the fly. The environment's modelling structure must be able to cope with such operations.

- *low memory requirements:* the data structure must consume a reasonable amount of memory, proportional to the environment's complexity. This requirement has been considered to be the least important in comparison to the speed and flexibility ones.

3.2.1 Polygons

The environment we shall use consists of an unstructured list of polygons modelling the scene to render. Each polygon is described by its list of 3D vertices and its *RGB color* which is actually the RGB reflectivity of the surface it describes (the reflectivity

is considered to be a constant for all its points and all incoming and outcoming directions). Besides these perfectly diffuse reflecting polygons, initial (called also primary) light sources are present into the environment. They are modelled by polygons whose RGB color values will be interpreted as initial RGB radiant flux values rather than reflectivities.

The environment described as a list of polygons together with the light sources specification is the input of the radiosity renderer and it fully describes the scene's geometry. The radiosity renderer will perform a subdivision of this polygonal environment. The next sections outline the two levels of the performed subdivision: patches and elements.

3.2.2 Patches

A polygon is subdivided into patches. A *patch* will be the radiant flux shooting unit during the progressive refinement process. The subdivision of polygons into patches is the first of the two essential hierarchy levels of the 3D world subdivision used by the radiosity renderer. It is the first level of subdivision performed during the preprocessing of the 3D polygonal world read as input by the radiosity renderer. Since a patch is used as a progressive refinement shooter, it will be characterized by:

- a radiant flux value, equalling the total unshot radiant flux presently held by the patch, for each of the 3 color components (R,G,B). This is the value that the patch will shoot when selected.
- the patch's area, used in radiant flux and radiant exitance computations.
- the patch's center, which is used to model the patch when computing some preliminary values during form factor determinations.
- the patch's vertices.

Patches are created from polygons via a subdivision process done in the preprocessing phase of the rendering. This subdivision is done only once, is uniform in the sense that a polygon is subdivided in a uniform mesh of patches and is controlled by the user by specifying the desired patch area for the patches to be obtained as result of the subdivision. Subdivision of polygons into patches is done for the only purpose of creating a set of finite area radiant flux shooters during the progressive refinement process.

The polygon to patch subdivision algorithm that was implemented produces two kind of patches: triangular and quadrilateral. The subdivision algorithm (currently accepting only convex polygons) attempts to subdivide a polygon into patches that are as close as possible to the user given patch area and have an aspect ratio as close as possible to 1, by creating equally spaced points on the edges of the polygon to subdivide. In other words, triangular patches will be close to equilateral triangles and quadrilateral patches to squares respectively. Four sided polygons will be subdivided into quadrilateral patches. All other polygons will be subdivided into triangles (any n sided convex polygon can be subdivided into triangles by first splitting it into n triangles created by joining its gravity center with all its n vertices.

Concluding we can say that the polygon to patch subdivision attempts to create a smoothly graded patch mesh (all patches are close to a user given patch area and their aspect ratios are close to 1).

The next section presents the second subdivision level: patches to elements.

3.2.3 Elements

A patch is subdivided into elements. The subdivision of patches (and consequently of polygons) into elements is the second essential subdivision level performed by the radiosity renderer. As described previously, elements will represent the level that gathers the radiant exitance information used for visualizing the scene.

An element is characterized by:

- the gathered (received) radiant exitance for all 3 color components (R,G,B). This value will be computed by the renderer during the progressive refinement process.
- the element's area, needed for radiant flux calculations.
- the element's center, used to represent this element in radiant exitance calculations.
- the element's vertex list. The element's vertices will be used when this element will be subdivided into other elements and for computing the radiant exitances used for display purposes. Remark that an element's *vertices* are not identical to its *corners*: in the case of a nonuniform element mesh, an element can have several (different sized) neighbour elements over its edges. We shall call corners the points at the ends of the element's edges (for example, a triangular element has 3 corners, a quadrilateral one will have 4) and vertices all the points on its edges (there can exist vertices between the corners in the case an element has more than one neighbour on one of its edges. Sometimes these vertices are called *midpoints*. The actual data structure will not impose a limit to the number of vertices an element has (i.e. it will be allowed to have as many neighbour elements as necessary). The implementation may limit however this number (if the smooth mesh grading requirement is to be obeyed, an element will tend to have a small number of neighbours, therefore a small number of vertices).
- the element's parent patch. Since polygons are not directly subdivided into elements but the patches are subdivided into elements, each element will have a parent patch.

The subdivision of patches into elements is a process less strictly controlled by the user than the polygon to patch subdivision. In the preprocessing phase, each patch is uniformly subdivided into elements up to a user-indicated element size but the subdivision can proceed further than this during the rendering phase, producing a nonuniform element mesh. The initial patch to element subdivision is very similar to the polygon to

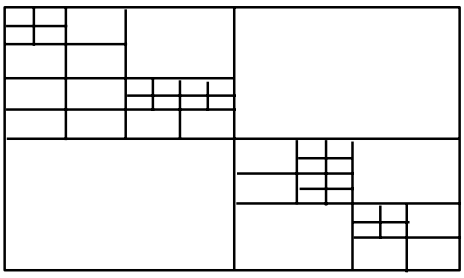


Figure 3.1: A nonuniform element mesh

patch subdivision. It will process each patch at a time, firstly creating an element equal to the patch and then subdividing this element into one or more elements. Since patches are either triangles or quadrilaterals we can have that the elements preserve the same aspect ratio as the patches they originate from by using a subdivision scheme that will join the element's edges middle points and create 4 new elements:

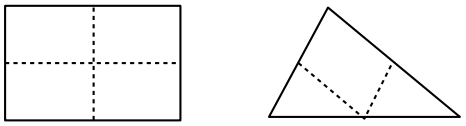


Figure 3.2: Quadrilateral and triangular elements subdivision preserving aspect ratio

If the original patch has a good aspect ratio, all elements created over this patch by recursively applying the above subdivision scheme will have the same aspect ratio too.

As previously mentioned, element subdivision can appear also during the rendering process and the resulting element mesh can be a nonuniform one. The data structure used by the radiosity renderer to maintain this dynamic nonuniform element mesh will be described later in this section.

3.3 Form Factor Determination

3.3.1 General Presentation

Since the presented radiosity renderer is based on progressive refinement, form factors have to be computed from *shooters* to *receivers*. Shooters will be called patches and receivers will be called elements. At the lowest level of the progressive refinement algorithm, a patch will have to shoot its radiant flux to all elements in the scene, therefore patch to element form factors will be evaluated. Practically, equation (2.39) will be used: the increase in radiant exitance of an element j due to a patch i is evaluated using the form factor F_{ji}.

In order to evaluate element-to-patch form factors, an element to patch ray casting approach will be used. Such an approach offers several advantages over the methods that attempt to compute patch-to-element form factors (e.g. the hemicube) (see *Target to Source Ray Casting Methods*). Target to source ray casting methods usually compute the form-factor between an infinitesimal area on the target and the whole source (see, for example, vertex-to-source form factors). We shall use the same approach, computing the form factor between the *center* of the receiving element and the whole source patch. This approach is perfectly similar to the one that computes form factors between receiver's *vertices* and the source patch. The reasons for which we preferred to use the centers of the receivers instead of their vertices are strictly related to low level efficiency issues in the renderer's implementation. Besides this, the receivers' sampling resolution offered by evaluation of radiant exitance at each element's center is the same as the one offered when evaluating the radiant exitance at elements' vertices (local differences can appear but we can't say that one approach offers a better sampling than the other one in the general case). A frequent statement in radiosity methods is that vertex to source form factors are more advantageous since vertex exitances and not element exitances are ultimately needed for displaying purposes. The last part of the assertion

is true but it doesn't imply that a vertex-to-source form factor renderer will be faster than an element-to-source one: element exitances have to be computed anyway in the case adaptive subdivision is used (an element will have to be subdivided and the exitances of the new elements are generally evaluated on the basis of this element's exitance and not on the basis of its vertices' exitances). Overall, we can say that the vertex-to-patch and element center-to-patch form factor approaches are roughly similar in speed and accuracy.

As for the vertex-to-patch form factor determination, the element center-to-patch approach will compute the form factor $F_{dA_jA_i}$ between a differential area dA_j at the center of receiver j and the whole source patch A_i, using a formula similar to (2.43).

The five-times rule (see *The Five-times Rule*) will be used to determine if a patch i has to be subdivided when computing $F_{dA_jA_i}$ (in order to keep flux transfer errors below 1 percent). Actually, the source patch will not be really subdivided, but several rays will be cast form the receiver to several points on its surface. The situation is very similar to the one depicted in figure 2.5. The five-times rule will be applied as following: when computing a form factor from a finite area source patch, the source patch has to be subdivided such that all the subdivision cells will subtend an angle smaller than 0.2 radians at the receiver point. Firstly, the 'coarse' unoccluded form-factor between the whole source i and the target point j will be computed:

$$F_{ji} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} dA_i \tag{3.1}$$

where θ_i and θ_j are the angles the normals of the surfaces i, j make with the ray from point j to i's center and r the length of this ray. If the five-times rule holds for the whole patch (i.e. $A_j \cos\theta_j / r^2 < 0.2$) this means that $F_{ij} < 0.2/\pi$. Testing F_{ij} against $0.2/\pi$ is therefore equivalent to applying the five-times rule for the patch. This can be stated intuitively as: if an unoccluded form factor from a whole source to a receiver point exceeds $0.2/\pi$, then the source must be subdivided. If subdivision is determined to be necessary, the shooting patch will be subdivided up to the smallest level that ensures that the five-times rule is obeyed for all the subdivision cells. The receiver to source form factor will be computed as the sum of the receiver to source-sudivision-cell form factors (for each source subdivision cell, a ray will be cast from the receiver to it in order to estimate both the receiver to source subdivision cell 'delta' form factor and the occlusion term).

For a reasonably fine initial environment subdivision into patches, it will be enough to subdivide a source patch into 4 up to 16 cells (i.e. it is enough to use 4 up to 16 rays from the receiver's point to the source). In our approach, uniform on the fly source subdivision is used: a source will be subdivided uniformly into 4 or 16 cells (depending on the value of the unoccluded coarse form factor) and one ray will be cast towards each of the subdivision cells in order to estimate its occlusion as seen from the receiver. This is a quite simple method that gives good estimates for the form factors, uses no additional memory or data structures (the source patch subdivision used for casting rays to the source is done on the fly, hence there is no need to store it in memory) and offers a very good speed in comparison to other methods (source patch subdivision is *uniform*, therefore it is very fast to be computed).

There exist other alternative approaches: as mentioned previously, Wallace et al (1989) have subdivided the source in *variable-sized cells* such that each cell has approximately the same analytic form factor with respect to the receiver point. Their algorithm was very similar to Sillion and Puech's (1989) single plane algorithm presented previously. Again, up to 16 subdivision cells (or rays) were used per source patch with very

good results. The subdivision scheme is however very complicated, therefore Wallace et al. also used uniform source subdivision for more complex scenes.

Another approach is to select a number of randomly distributed points over the source patch (this number is determined again using the five-times rule) and to cast rays form the receiver's point to them. Wallace et al. (1989) have shown that this approach may minimize the aliasing effects that can occur due to the regular subdivision (ray casting) of the source (these aliases are especially visible at shadow boundaries). Instead of generating a random point distribution over the receiver, we can alternatively jitter the points generated by the previously described uniform distribution. This method seems better since we can tune the jittering factor choose between a totally uniform ray casting approach (jitter factor equal to zero) and a random ray distribution.

Our approach using on the fly subdivision contrasts with the hierarchical radiosity method [Hanrahan et al., 1991] where a quadtree is used to keep the several subdivision levels of a source in memory (when needing to compute a form factor as seen from a receiver's point, the appropriate level is chosen by a method using a rule similar to the five-times rule). The advantage of such a hierarchy should be that subdivision can be done only once and then reused whenever necessary. The disadvantages are that a more complex and memory-consuming scheme will be required and that the management of this scheme may be as slow as on the fly subdivision when there are only 1 or 2 subdivision levels. For a fine enough initial environment subdivision into patches, on the fly subdivision can be at least as fast as the hierarchical approach.

There exists a special case that is worth treating separately: the initial (primary) light sources that have a very high radiant flux. The overall scene's illumination computation will be heavily influenced by the accuracy with which the patches of these initial light sources shoot their radiant flux. Sometimes we would like to relax the five-times rule (that is, use a less finer source subdivision than the one imposed by the five-times rule) in order to increase the renderer's speed. Doing this directly for all the light sources may result in visible artifacts caused by a small number of rays cast towards the initial light sources' patches. An additional refinement was used in the actual implementation, allowing different polygon to patch subdivision levels for the normal polygons and the light source polygons. The results obtained by subdividing the normal polygons into a coarse patch mesh and by subdividing the light source polygons into a patch mesh a couple of times finer were almost as good as the ones that used a fine uniform initial patch mesh, but the rendering speed was a couple of times higher (since the areas covered by initial light sources are typically just a small fraction of the total scene's area, therefore the increase in the total number of patches due to their fine subdivision was relatively small).

As a final remark, computing element center to patch form factors has another advantage with respect to vertex to patch form factors: the receiver element's center can be jittered in the receiving surface's plane in order to account for a nonuniform sampling of the radiance function over that surface (this jittering may diminish aliasing due to the uniform sampling produced by a regular element mesh). This is very simple to be done when the element's sampling point is its *center* (since a jittered element center is still a point in the element) but may pose problems if we'd desire to jitter the element's *vertices*. Notice that this jittering is different (and essentially independent) on the jittering of light source sampling points. This reminds again the fact that source and target subdivisions are independent.

Conclusions

A target to source ray casting approach has been implemented for form factor determination. Rays are cast from the receiver to the source, occlusion tests are performed and the form factor is built by summing up contribution of each ray. Sources are uniformly subdivided in order to obey the five-times rule for form factor estimations. Receiver centers are used rather than receiver vertices for sampling the receiving surface. Jittering is used over the source subdivision points and the receiver's center in order to minimize aliasing. The source's subdivision is done on the fly in order to minimize memory requirements.

3.3.2 Ray Casting Implementation

Ray casting is at the core of the radiosity renderer: for each form factor computation, up to 16 rays will be cast from the receiver's center to the source patch. Each of these rays must be tested for occlusion against all polygons in the scene. The first direction for optimizing the renderer should be the development of a very efficient ray casting algorithm.

There are several levels at which the ray casting algorithm has been optimized. In order to proceed with their presentation, the scene's structure will be reminded: a scene is composed of polygons, each polygon is subdivided into a number of patches having an edge aspect ratio close to 1, each patch is nonuniformly subdivided into elements that maintain the same aspect ratio as the patch. Rays are cast form the receiver's center to the subdivision points on the source. When, in the following, we say that 'rays are cast from a source patch to a receiver element', we actually mean that rays are cast between the subdivision points on the source and the receiver's center.

A source patch must shoot its radiant flux to all elements in the scene. A naive approach would be to iterate over all elements on all polygons and shoot at them from the source patch. However, even in the absence of occluding objects between a source and a receiver, there are many elements that are not visible from the source: this can be simply determined by checking the dot products of the source and receiver normals with the source-receiver ray: Element e on surface s is visible from patch p if:

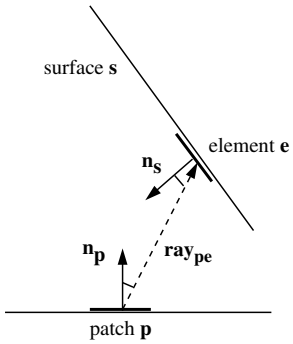


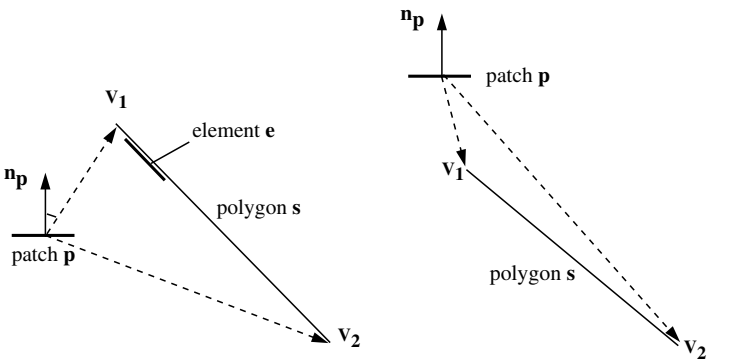
Figure 3.3: Visibility relation between a patch and an element

$$\mathbf{n_p ray_{pe}} > 0 \; \textit{and} \; \mathbf{n_s ray_{pe}} < 0 \tag{3.2}$$

The above equation refers to the *intrinsic* mutual visibility of element e and patch p. It refers to the fact that the two items are oriented in such a way so they can see each other

and depends only on the relative orientations of the two items with respect to each other. The other kind of visibility can be called *extrinsic* visibility and it refers to the fact that two items may see or not each other depending on the presence of occluding objects in between. Extrinsic visibility doesn't depend at all on the relative orientations of the two items but only on the positions of the scene's objects with respect to the source-target ray. In order to have visibility between the two items, both intrinsic and extrinsic visibilities must be present.

It is clear that the two visibility tests are independent (hence they can be independently optimized). The intrinsic visibility test is however much cheaper than the other (since it involves just the source and the target, so its complexity is basically $O(1)$ while the other test has a complexity $O(N_{in})$ where N_{in} is the average number of occluding objects to be tested between a source and a receiver), so it will be performed the first in order to directly reject all items not visible from the source. We shall firstly try to design an efficient intrinsic visibility testing procedure. A major observation is that in the general case of small elements compared to the source-element distance there is an *intrinsic visibility coherence* over the elements of a polygon: neighbouring elements of the polygon have the same value of intrinsic visibility with respect to the source. This observation tells that we can design some tests that may reject all elements of a receiving polygon *as a whole* as being invisible from the source without needing to perform individual visibility tests for each of them (thus saving a considerable number of the dot products required by equation (3.2)): Two tests will be developed in order to determine



a) TEST 2 succeeds on vertex $\mathbf{V_1}$ so there is at least one potential visible element on polygon **s** ($\in$ for example).

b) TEST 2 fails on both vertices $\mathbf{V_1}$ and $\mathbf{V_2}$ of surface **s**. TEST 1 fails too, so all elements of **s** are invisible from patch **p**.

Figure 3.4: Testing the intrinsic visibility of a patch and a polygon

if a polygon, seen as a whole, is invisible from a source patch. If any of the tests fails, we skip all elements of the polygon at once.

- *TEST 1:*

  If $\mathbf{n_s ray_{pc}} < 0$ then patch p is in the visible halfplane of the polygon p (the visible halfplane is determined by the polygon's plane and its normal, since polygons are considered to be one-sided surfaces). **c** is a point in the polygon's plane (one of its vertices, for example). Moreover, the converse is valid too: if TEST 1 fails, then the patch is in the invisible halfplane of s and therefore all elements e of polygon s will be invisible from p. The proof of these assertions is rather simple:

if $\mathbf{n_s ray_{pc}} < 0$ for a point $\mathbf{c}$ of plane s then this will be valid for *any* point $\mathbf{c2} = \mathbf{c} + \mathbf{v}$ of plane s, since for any vector $\mathbf{v}$ in s we have $\mathbf{vn_s} = 0$ (the plane's equation).

Conclusion

TEST 1 will be firstly used on plane s and patch p. If it fails, then p can see no point of s, therefore all elements of s are skipped. TEST 1 accounts for the second term of equation (3.2).

- *TEST 2:*

  We shall use the following assertion ASSERT: If there exists at least 1 element e of s such that $\mathbf{n_p ray_{pe}} > 0$, then there exists at least 1 vertex $\mathbf{c}$ of s such that $\mathbf{n_p ray_{pc}} > 0$. By negating this assertion we find that, if for all vertices $\mathbf{c}$ of s we have $\mathbf{n_p ray_{pc}} < 0$, then no element of s is visible from p, therefore we can skip all elements of s at once.

  Proof of ASSERT:

  Let $ray_{pc} = ray_{pe} + \mathbf{v}$, where $\mathbf{v}$ is the position vector of a vertex c of s with respect to element e. In order to prove now that $\mathbf{n_p ray_{pc}} > 0$, we must prove only that $\mathbf{n_p v} \geq 0$. If $\mathbf{n_p} \perp s$ then the above dot product is zero, so ASSERT is proved. If $\mathbf{n_p}$ not orthogonal s, then consider $\mathbf{n'_p}$ the projection of $\mathbf{n_p}$ on s. We have now to prove that there exists a vertex $\mathbf{c}$ of s with position vector $\mathbf{v}$ with respect to element e such that $\mathbf{n'_p v} \geq 0$.

  Looking at the 2D plane s, we can see that element e's center is a point *inside* polygon s. Therefore if we consider the halfplane of s determined by the line passing through e and orthogonal to $\mathbf{n'_p}$, this line will intersect polygon s so at least 1 vertex of s must be in each of the two halfplanes. But for any point $\mathbf{c}$ in the above mentioned halfplane, such that $\mathbf{v}$ is its position vector with respect to e, we have $\mathbf{n'_p v} \geq 0$.

  Conclusion

  TEST 2 will be used in the case TEST 1 has passed. If it fails, then p can see no point of s, therefore all elements of s are skipped. TEST 2 accounts for the first term of equation (3.2). TEST 2 is more expensive than TEST 1, so it will be used after TEST 1 has passed.

Tests 1 and 2 can eliminate a large amount of elements from the next stages of the form factor computations. They are somehow similar to the backface culling tests used for 3D viewing. Using them in the context of ray casting form factor computations is based on the intrinsic visibility coherence of elements over a polygon (which is essentially the coherence of the **nray** dot product, **n** being an arbitrary vector and **ray** being the position vectors of all points of the polygon with respect to a given 'shooting' point).

If both TEST 1 and TEST 2 pass, there exists at least one element e of polygon s that is intrinsically visible from the source patch p. For all elements of s, the second term $\mathbf{n_p ray_{pe}} > 0$ of equation (3.2) is evaluated again. All elements e that pass this test are considered to be intrinsically visible from p.

Extrinsic visibility must now be evaluated for these elements e. In other words, the ray from p to e must be checked to see if it is occluded by any objects in the scene. An efficient strategy for this occlusion testing is presented in the next section.

3.3.3 Ray Occlusion Testing using Octrees

A visibility ray must be tested against all polygons in the scene (and not against all elements or patches, since there are far fewer polygons to check than patches or elements and we are interested only in a yes or no answer to the occlusion question).

A spatial subdivision scheme is extremely useful for accelerating the ray-polygon intersection tests: instead of checking the ray against all polygons in the space (which would be prohibitively slow), it is better to check the ray only against those polygons placed 'close' to the ray's path.

For this purpose, the space was subdivided using an octree partitioning scheme that records per each leaf octree-cell all the polygons intersecting it. The bounding-box of the whole 3D world is therefore recursively divided into octree-cells until we obtain either leaf-cells with less polygons/cell than a user-given amount or until a maximum subdivision depth has been reached. Recording items (polygons) per octree-cell was preferred to the voxel-subdivision scheme, in which a voxel is either full or empty, since the latter requires much more storage-space (even for the nonuniform case when different sized voxels are used, since one must subdivide a voxel until is either empty or totally occupied by a polygon).

In order to check the occlusion, the ray is traced through the octree, cell after cell, starting with the cell of the source patch, ending in the cell of the receiver element, testing occlusion only against the polygons listed in the cells we pass through. At the first occlusion the ray tracing process can stop and return 'occluded' without further investigation.

Several delicate problems appear while building the octree and while tracing a ray through it. All are caused by 'boundary phenomena', i.e. polygons that happen to fall very close to (or even coincide with) octree-cells faces and rays that travel on an octree-cell face/edge. Special attention has to be paid to such cases since tracing a ray from cell to cell might be problematic and also polygons coinciding with octree-cell faces might be recorded/tested incorrectly. An incorrect occlusion test decision might result in visibly wrong illumination or shadow patterns.

The solution to such problems comprises the following elements:

- When distributing polygons in octree-cells, we consider that a cell is actually a bit *larger* than its exact geometrical size. In this way, polygons close to any face of a cell will be recorded as belonging to both cells sharing that face, eliminating frequent problems appearing due to numerical inaccuracies (e.g. two neighbour cells that might not have the common vertex coordinates perfectly equal, so problems may appear if a polygon has exactly these coordinates: it might arrive to 'leak' between the two cells).

- When tracing the ray from cell to cell, the following algorithm is used:

  First, we intersect the ray with all cell's faces that the ray 'sees' (out of 6 faces, a ray can see maximum 3 faces at a time), remembering the closest intersection to the ray's origin. This is the face the ray exits from. Now we generate a point on the ray a little outside the cell (this is trivially done if the ray-faces intersections are computed parametrically) and determine the octree-cell this point belongs to, by a depth-first traversal of the octree. When determining the point in cell containment, we use the *exact* geometrical dimension of the cell, so that a little shifting of a point outside the current cell will surely 'throw' it in another cell. However it is interesting to note that there are a few cases when this shifting of a

point outside the current cell fails to move it in the next cell (when the ray is almost tangent to the cell-wall, so when a little shift on this ray will be numerically too small to represent a significant amount). These few cases (about 1 at 10000 ray traces) can be easily handled: if the new cell determined is the same with the current cell (ray failed to exit), we determine which cell-face the ray tried to exit through and force a transversal shift outside:

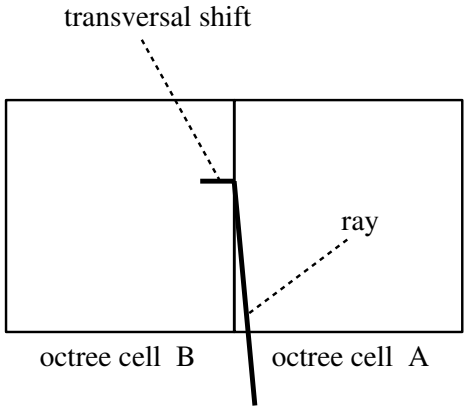


Figure 3.5: A transversal shift is necessary for the ray to pass from cell A to cell B

Using the above ideas has several benefits. All the possible cases in which a ray and the cell-structure can relatively be were examined. We tried to prove that the ray tracing algorithm used keeps the following two properties:

- if a polygon will occlude the ray, then the algorithm will surely find it.
- the ray will follow the correct path through the cells up to the destination cell in the shortest time.

Another useful and simple enhancement for the ray casting method was implemented, based on the shadow caching described by Haines and Greenberg (1986). There exists a certain amount of occlusion coherence for rays shot from the same source towards elements on the same receiver. Therefore, when a ray was intercepted by a polygon, a pointer to that occluding polygon is stored (cached). When tracing the next rays, occlusion against the cached polygon (if any) is firstly tested. There are many cases when this will indeed be the polygon occluding the new ray, so this will be found without the need for tracing the ray through the octree.

There are several other enhancements to the basic ray-polygon occlusion procedure: axis-aligned bounding boxes (Ng and Slater, 1993) are computed for all the polygons. When checking for intersections with a polygon, its bounding box will be firstly tested against the ray. Only in the case this bounding box intercepts the ray the more expensive ray-polygon test is performed. Using bounding boxes is very useful when the objects are not polygonally described and ray-object intersections are expensive.

A *BSP tree* can be used instead of an octree (Sung and Shirley 1992). In this case, each node in the tree will represent a 3D subspace while its children will represent half-spaces of that subspace, separated by a plane. It is simpler to use planes parallel with the coordinate-planes (mainly for tracing a ray through such a tree). The BSP tree might be more advantageous than an octree if the division planes can be found in such a way

to 'balance' the tree better than the octree. This will surely imply that the planes have to be drawn such to separate an equal number of polygons on their two halves. The process of tracing a ray through the BSP tree might be also a bit more complicated than ray tracing through the octree.

The final step of the ray casting process is the ray-polygon intersection itself, that will be described in the following section.

3.3.4 Ray-Polygon Intersection

As for the previous stages of the ray casting algorithm, there are various types of speedups for the ray polygon intersection algorithm. All such algorithms working in object space take the following approach:

- Check the ray-polygon plane intersection. This is done by testing the intersection of the polygon's plane with the ray seen as a *segment* (starting from the source patch and ending to the receiver element), since:
  - clipping the ray to the octree cell we pass through is useless (we don't gain anything by this clipping since the only polygons we test the ray against are the ones in the octree cell).
  - the ray must *not* be tested against polygons in the start octree cell that are *behind* the start point and against polygons in the end cell that are *after* the end point.

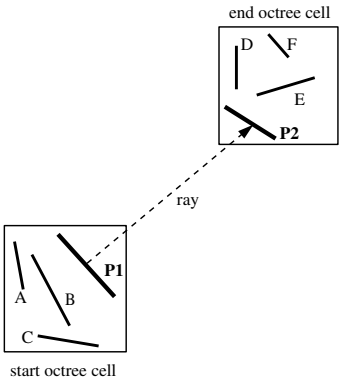


Figure 3.6: Polygons A,B,C in octree cell of P1 and D,E,F in octree cell of P2 must not be tested against the P1-P2 occlusion ray

- if the first test succeeds, check that the intersection-point obtained is indeed inside the polygon. For this, the polygon is projected on the xyz-system's plane on which it has the largest projection (in order to minimize errors) and we check that the intersection-point's projection falls inside the 2D polygon projection, using a 2D point-in-polygon algorithm. There are several such algorithms offering different advantages:
  - some algorithms check each polygon's edge against a horizontal line starting at some faraway point and ending at the point we want to check if it is inside the polygon or not. Such an algorithm can be carefully designed to

work for concave polygons as well, in which case it has a linear complexity with the number of polygons' edges. We used such an algorithm in the current implementation.

- some algorithms are designed only for convex polygons. Badouel (1990) proposes an algorithm that will divide the convex polygon into a number of triangles equal to its number of edges and check the point in triangle containment using a simple parametric approach.

An efficient solution is to use different algorithms depending on the polygon's type: if the polygon is a triangle, a very simple and fast intersection algorithm can be used. If the polygon is convex, the faster convex point in polygon algorithm can be used. If the polygon is concave, the general (more complex) algorithm will be employed. An efficient and elegant procedure would be an object-oriented implementation using class-specific containment functions for each type of polygon derived from an abstract polygon base class.

3.4 Adaptive Receiver Subdivision

Adaptive receiver subdivision attempts to capture accurately the radiance function over the scene's surfaces (see section 2.7.4). There are two main information sources that can be used to make this process automatic:

- the already computed solution can be scanned for high gradient areas. these areas are likely to contain sharp variations of the function so they have to be subdivided and the solution has to be recomputed over them. This approach uses the current solution as a predictor for the high gradient areas of the final solution.

- various object space techniques can be used to predict which are the areas where a sharp illumination gradient will occur. The predictions do not depend on the computed solution but try to directly estimate the radiance function (or its gradient).

This section presents the subdivision based on the first type of criteria. The next section will present a subdivision technique that uses the second type of criteria.

As previously described, the elements' mesh can be nonuniform: any element can be subdivided up to practically any desired level in order to capture illumination gradients. Regardless of the actual subdivision criterion used, the goal is to obtain elements over which the illumination solution varies as smoothly as allowed by the imposed minimum element size. Adaptive subdivision should proceed only in the highly nonuniform solution areas, in order to minimize the number of resulting elements. The main reason for having a smooth varying solution over an element is that it will be rendered as a Gouraud-shaded polygon using its vertices' radiant exitances. Such a polygon might appear totally incorrect if its vertices' solutions are not varying in a plane (such that the Gouraud shading might linearly interpolate them).

The adaptive subdivision process is done in several steps:

Firstly, the vertex radiant exitances are evaluated out of element radiant exitances (which are computed during the radiant flux shooting process). After this, the vertex radiant exitances are normalized to the display range (that is, the vertex radiances ultimately used to display the image, therefore perceived by the observer, are computed). All elements are then scanned in order to determine if they have to be subdivided or not.

For an element, the subdivision decision is taken by examining the radiant exitances previously computed in its vertices. There are several criteria that can be used:

- the *gradient criterion*: for any edge of the element, the absolute difference in radiant exitance of its two vertices is divided by the geometrical length of the edge. The obtained quantity equals the derivative of the radiant exitance solution on the element's edge direction. If this quantity (which can be thought as being the gradient of the radiant exitance solution projected on the element's edge direction) exceeds a given threshold, the element is subdivided. The advantage of such a scheme is that the threshold can be established as a dimensionless, scene-independent value. A finer threshold will produce smaller elements that will render a smoother final image, at the expense of an increased number of elements and processing time.

- the *delta criterion*: it is similar to the gradient criterion: for any edge of an element, the absolute difference (or delta value) between its vertex radiant exitances is computed and compared against a threshold. If this threshold is exceeded, the element is to be subdivided. This criterion will generate a considerably larger number of subdivisions that the previous criterion. A similar feature with the gradient criterion is that the threshold is again scene-independent: it just indicates the maximum visual illumination difference that is allowed to exist between two adjacent vertices.

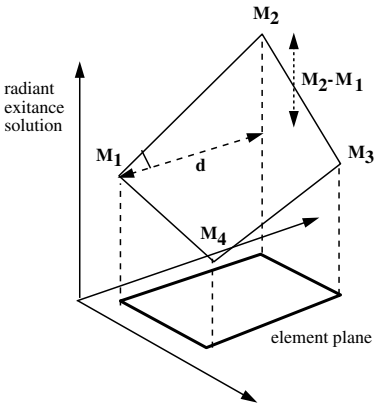


Figure 3.7: Computing quantities for the gradient and delta criteria

$$grad_{element}(M) = \frac{M_2 - M_1}{d} \tag{3.3}$$

$$delta_{element}(M) = M_2 - M_1 \tag{3.4}$$

Another alternative would be to compute the real gradient vector length for the radiant exitance function defined over a given element. The problem is that an element can have many vertices (in the case the elements over its borders are smaller than it) and the radiant exitances values in these vertices can be highly different. Therefore the gradient of the solution will *not* be constant over this element. Attempting to evaluate an *average*

gradient over the element (by approximating the solution over the element with a plane) can also give a highly incorrect estimation of the variations of the radiance function.

A good method would be to fit a linear radiant exitance solution through the vertices' solutions and compute the sum of the distances from these solutions to the plane. The resulting value is a good measure of the difference between a linear variation solution and the actual solution. The main problem with such a method is that it is rather expensive to be computed.

Both the delta and gradient criteria have been implemented and tested. As a general rule, the gradient criterion performs better (i.e. produces a good nonuniform mesh with less elements). They share another good property: if the radiant exitance's variation is determined to be too large over an edge, *both* elements sharing that edge will be subdivided, thus preserving a smooth mesh grading over the subdivision process.

After all elements have been checked for high solution variations, the ones marked as such are subdivided: new elements emerge out of the subdivision process while the old ones are destroyed. The *element* radiant exitance solution of the initial elements has to be mapped on the newly created elements. There are two methods that have been tried:

- *uniform mapping:* the new elements are assigned a radiant exitance equal to the original element's one.
- *redistribution:* the radiant exitance of the original element is nonuniformly distributed to the new elements. There are several possibilities of doing this, all having to obey the following constraints:
  - the sum of the radiant flux values of the new elements must equal the radiant flux of the original element.
  - the new elements must interpolate the variation of radiant exitance, defined as function of the original element's vertices exitances. In other words, the exitances of the new elements must somehow interpolate the exitances of the original element's vertices.

.

Redistribution of the original element's exitance gives far better results than the uniform mapping, even for a highly unsmooth graded mesh. The exitance redistribution function that has been used assigns to each of the new elements an exitance that is a weighted average of the original element's vertex exitances, depending on the distances to these vertices. For example, for a four-sided element subdivided into 4 new elements, we have:

$$M_1 = \frac{9}{16}M_{v1} + \frac{3}{16}M_{v2} + \frac{1}{16}M_{v3} + \frac{3}{16}M_{v4} \tag{3.5}$$

$$M_2 = \frac{3}{16}M_{v1} + \frac{9}{16}M_{v2} + \frac{3}{16}M_{v3} + \frac{1}{16}M_{v4} \tag{3.6}$$

$$M_3 = \frac{1}{16}M_{v1} + \frac{3}{16}M_{v2} + \frac{9}{16}M_{v3} + \frac{3}{16}M_{v4} \tag{3.7}$$

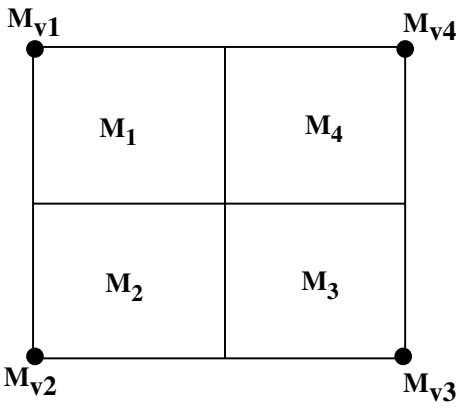


Figure 3.8: Redistribution of exitance using vertex exitance averaging

$$M_4 = \frac{3}{16}M_{v1} + \frac{1}{16}M_{v2} + \frac{3}{16}M_{v3} + \frac{9}{16}M_{v4} \tag{3.8}$$

The above formulas obey both requirements formulated above. Similar redistribution formulas have been used for triangular elements.

It is important to remark that the above redistribution formulas use the non normalized vertex exitances and *not* the normalized ones that are created for displaying purposes. The reason for this is that the sum of the new elements' fluxes must equal the original element's flux (in order to obey the flux conservation requirement). The normalized vertex exitances are obtained via a user-defined nonlinear mapping and have a totally different range and distribution so they can not be used for evaluation of the new elements' exitances.

After the required elements have been subdivided, a *reshooting* operation takes place: the source patch that last shot will reshoot towards all the newly created elements. This process can be seen as the *resampling* of a function after a first sampling has been done and the high gradient areas have been detected. The check for high gradients is again done for all elements at which the source reshot and a new subdivision of some of these elements can occur. The process stops when there is no element that is subdivided any more.

This process is easily implemented using only two queues of elements, q_1 and q_2. At the beginning of a shooting iteration, all elements of the receiving polygon are put in q_1. The source patch shoots then at all the elements in q_1. After the shooting is done, all elements in q_1 are scanned and the high gradient ones are placed in q_2. When scanning has proceeded, all elements in q_2 are subdivided and the new resulting elements are placed in q_1. At this moment, q_1 contains all elements at which reshooting has to be done, so the algorithm loops back to the shooting phase. The process will stop when q_1 is empty.

The advantage of this approach is that reshooting and rescanning for high gradients is done only over the *new* elements resulting out of the subdivision phase and not on all other elements on that polygon. Moreover, the process can be done on a per polygon basis, hence the queues q_1 and q_2 need to be only as long as the maximum allowed number of elements per polygon. This allows implementing them as arrays, ensuring maximum speed for all queue operations.

Three criteria for starting, respectively stopping adaptive subdivision have been used:

- *threshold criterion:* when the test value (gradient or delta value) is below a certain threshold, the subdivision stops.

- *area criterion:* when the area of an element is below a given threshold, the element is considered to be small enough for visual purposes and subdivision stops. The threshold area is automatically determined as a fraction of the average element area over the scene or it can be alternatively user specified.

- *grading criterion:* when the ratio between the area of an element and a neighbour element's area exceeds a given threshold, the element is subdivided. This criterion proves itself to be a very effective way to control the mesh grading. By specifying a low threshold, we ensure that no two neighbouring elements will be allowed to have a too large area ratio. The mesh will be smoothly graded. Although this means more elements will be generated, tuning the mesh grading can ensure that many visual artifacts that appear due to too high local mesh nonuniformities (discontinuities in the mesh grading) will be removed. Again, the threshold is scene independent (it represents just an element area ratio).

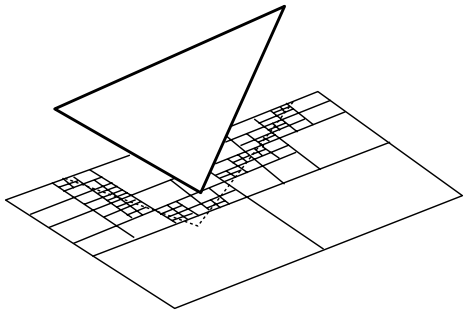


Figure 3.9: A triangle casting a shadow over a quadrilateral. Adaptive subdivision on a gradient-based criterion was used over the quadrilateral

3.4.1 A Non Uniform Element Mesh Implementation

The reasons for having a non uniform element mesh have been presented in the previous section. Section 3.1 has outlined the creation of the element mesh: an element is firstly created for each patch and a recursive element subdivision scheme will proceed during the progressive refinement phase over the zones where it is determined to be necessary. Besides its quality, the element mesh must offer the following features:

- arbitrary subdivision depth: it should be possible to subdivide an element up to any desired level, regardless of the sizes of the neighbouring elements. The mesh implementation must remain consistent after such arbitrary depth element subdivisions.

- vertex sharing: vertices have to be shared between neighbouring elements. this will ensure that a geometrical vertex will finally have a unique radiant exitance value that will be used for Gouraud shading. Vertex sharing should be done over a whole surface that is desired to appear smooth shaded in the final image. In our

case, this surface will be a polygon. Moreover, vertex sharing reduces considerably the amount of memory and computations to be done for evaluating vertex exitances, since there are no duplicates of the same vertex.

- vertex to element access: it should be possible to rapidly determine the elements sharing a vertex in order to evaluate the vertex's radiant exitance from the elements' exitances (as described in the previous section).

- element to vertex access: it should be possible to rapidly determine the vertices of an element (for computing that element's radiant exitance gradient, for example).

- access speed for gradient-based criteria: the elements must be rapidly scanned in order to determine which are the ones needing to be subdivided due to high radiance gradients. The mesh implementation should offer a means of rapidly iterating over all elements of a polygon as well as over all vertices created during its subdivision. It is important to notice that, while the subdivision process is performed over only a small fraction of the total number of elements, there is a comparatively much larger amount of iteration being done over these elements' vertices. The mesh implementation should therefore favor a high access speed as compared to the subdivision speed.

The mesh implementation we shall present is based on two data structures: the element and the vertex. The structure of an element, as presented in *Elements*, is rather simple: it features an array of pointers to vertices and an array of indices of the *corner* vertices in the vertex array. The vertex number can be arbitrary but there are only three or four corners for an element (depending on the element being a triangle or a quadrilateral). The vertex array will refer to all the vertices of the element in counterclockwise order. Storing the vertices pointers and the corner indexes as arrays offers a very efficient memory usage for the element mesh. The only drawback appears when an element is subdivided: since *all* vertices must be shared, the new vertices that appear in the subdivision process will have to be inserted in the vertex arrays of the neighbours of this element. The neighbours' vertex arrays may need to be reallocated to accommodate the new vertices. We have tested the array based implementation against a vertex linked list implementation (that maintains the vertices of an element in a linked list): the list-based implementation proved finally to be *slower* than the array one, due to the decreased iteration speed over a list as compared to iteration over an array as well as the overhead caused by the dynamic memory management (which caused around 20 percent slowdown). In conclusion, both the memory usage and the access speed favor an array-based vertex list as compared to a linked list solution.

A vertex structure will contain (besides the vertex's exitance for the red, green and blue colors) a set of pointers to the elements that use this vertex. The element subdivision scheme presented in section 3.2.3 shows that a vertex participating in a quadrilateral mesh may have maximum 4 elements sharing it, while a vertex of a triangular mesh may have at most 6 elements sharing it. According to this observation, a vertex will have a fixed size array of 6 pointers to elements. We shall call this array the *element array* of a vertex. Not all the cells of this array have to be filled (if, for instance, the vertex has less than 6 elements sharing it).

The element array has a special layout. In order to explain this layout, consider a quadrilateral polygon meshed into quadrilateral elements. If the polygon's corners are v_0, v_1, v_2, v_3 (in counterclockwise order) we can consider an *orientation scheme* over this polygon where v_0 will be called the *northwest* corner, v_1 the *southwest* corner, v_2

the *southeast* corner and v_3 the *northeast* corner. All elements created over this polygon will have their vertex array created in the same order: northwest vertex first, southwest vertex, southeast vertex and northeast vertex finally. In other words, for any element of this polygon, its vertices will always come in upperleft, lowerleft, lowerright, upperright order: The same idea will be applied for triangular meshes. For a triangular

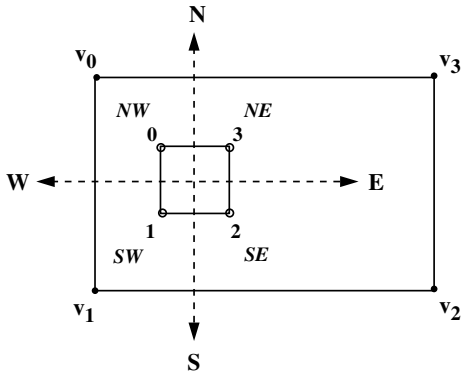


Figure 3.10: A quadrilateral polygon $v_0v_1v_2v_3$ and one of its elements. The element's corners come in the same order as the polygon's ones: NW, SW, SE, NE.

polygon, its orientation will be given by its corners as follows: if the polygon's corners are v_0, v_1 and v_2 (in counterclockwise order) then v_0 will be the *north* corner, v_1 the *southwest* corner and v_2 the *southeast* corner. An element of such a triangular polygon can however have two types of orientation with respect to the orientation scheme established by the polygon's vertices. In the next figures, for example, element E_1 will be *downwards* oriented while element E_2 will be *upwards* oriented: We shall need one

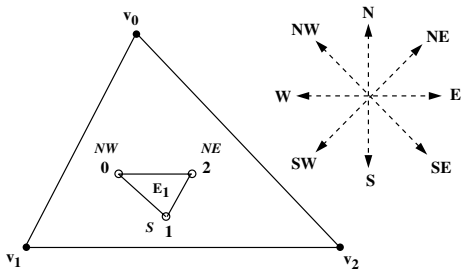


Figure 3.11: A triangular polygon $v_0v_1v_2$ and one of its elements with downwards orientation. The element's corners come in the order: NW, S, NE (with respect to the polygon's orientation scheme)

more item in order to establish a consistent orientation over a triangular polygon and that will be the elements' type (*upwards* or *downwards*). Returning to the element array for a vertex, we shall impose the following invariant structure to this array: its first element will point to the element at south from the vertex, the second to the element at the vertex's southeast, the third at the vertex's northeast and so on until the sixth array

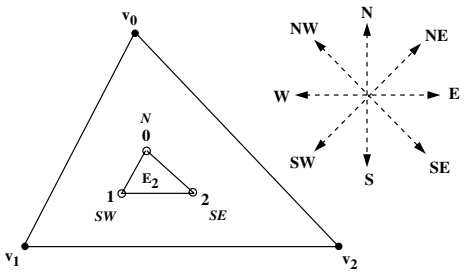


Figure 3.12: A triangular polygon $v_0v_1v_2$ and one of its elements with upwards orientation. The element's corners come in the order: N, SW, SE (with respect to the polygon's orientation scheme)

entry pointing at the element at the vertex's southwest. If a vertex has no element in a certain direction, a NULL pointer will be stored there in the element array. An element may appear several times in the element list of the same vertex (in the case this vertex is a midpoint, for example): The element array structure described above has an impor-

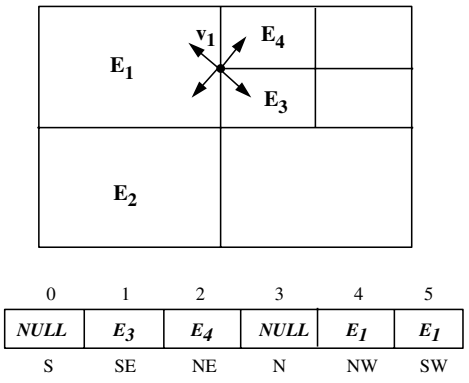


Figure 3.13: The element array for vertex v_1 for a quadrilateral polygon. Element E_1 appears twice in the array.

tant property: it allows us to retrieve the element sharing a given vertex for any given direction from that vertex. We can inquire, for example, about the element at northeast of a vertex v just by dereferencing the third position of v's element array (third position corresponds to northeast direction). The fast access over the elements sharing a vertex is ensured: we can iterate over all the non NULL positions of the vertex's element array. Moreover, we shall link all vertices over a polygon in a single list so that iterating over all the polygon's vertices can be done very fast and each vertex will be accessed just once during the iteration. Another desirable feature is the possibility to rapidly find the *corners* of an element: since we store the corners indices in the element's vertex array, this can be done directly by iterating over The above mesh structure must preserve all its properties during an arbitrary nonuniform subdivision. The element subdivision algorithm (illustrated for a quadrilateral element) will proceed as follows:

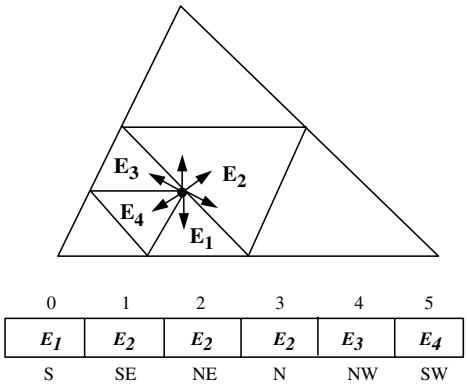


Figure 3.14: The element array for vertex v_1 for a triangular polygon. Element E_2 appears three times in the array.

- *STEP 1:* the element's corners (c_0, c_1, c_2, c_3) are found using the element's corner index array.
- *STEP 2:* a new point c (coincident with the element's center) is created and added to the vertex list of the polygon who owns the element to be subdivided.
- *STEP 3:* four new points i_0, i_1, i_2, i_3 being the points at the middle of the edges c_0c_1, c_1c_2, c_2c_3, c_3c_0 are created (if they don't exist already) or retrieved out of the element's midpoints (if they exist). If new points are created, they are inserted in the vertex arrays of the neighbouring elements of the element $c_0c_1c_2c_3$ (see figure 3.15).

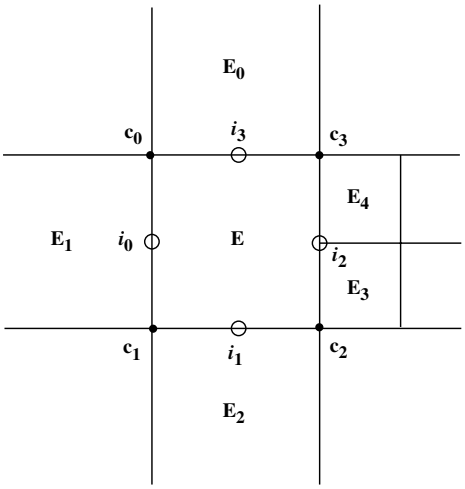


Figure 3.15: The element E is subdivided into four new elements. Midpoints i_0, i_1, i_3 are created and added to E and to its neighbours E_0, E_1, E_2. Midpoint i_2 exists (from the neighbours E_3, E_4) so it will be used.

- *STEP 4:* four new elements are created: element e_0 having vertices c_0,i_0,c,i_3 and similarly for the other three elements. If the subdivided element had other mid-

points on its edges, they are inserted on the edges of the new elements $e_0..e_3$ in the correct (counterclockwise ordered) positions.

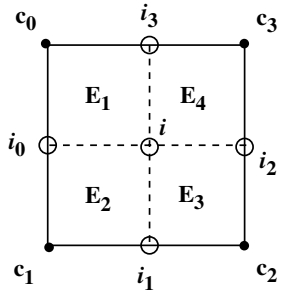


Figure 3.16: Element $c_0c_1c_2c_3$ is subdivided into elements $E_1(c_0i_0ii_3)$, $E_2(i_0c_0i_1i)$, $E_3(ii_1c_2i_2)$, $E_4(i_3ii_2c_3)$.

- *STEP 5:* the old element is destroyed.

A similar algorithm is used for subdividing triangular elements (the only supplementary feature is that the orientation (*upwards* or *downwards*) has to be determined for the 4 new triangular elements created at subdivision). This is trivially done by noticing that out of the 4 elements created when subdividing a triangle, the three ones using its vertices will always have the same orientation as the original element while the fourth (central) one will have the opposite orientation (figure 3.17). Notice that, while

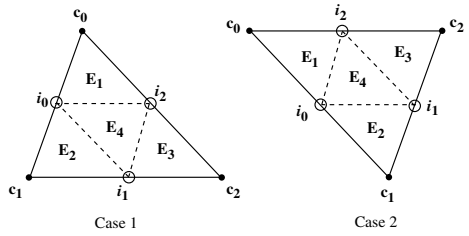


Figure 3.17: Case 1: Elements E_1, E_2, E_3 are UPWARDS oriented and central element E_4 is DOWNWARDS oriented. Case 2: Elements E_1, E_2, E_3 are DOWNWARDS oriented while central element E_4 is UPWARDS oriented.

elements are destroyed and created during an adaptive subdivision process, vertices are never destroyed: they will always be reused in the new elements if possible. Therefore, the vertex list of a polygon will always keep strictly increasing (so it can be directly implemented by a singly linked list).

Conclusions

The implementation presented above seems to be a very efficient way of encoding a nonuniform mesh of triangles or quadrilaterals. Vertex sharing is fully done over a whole polygon. Array based representation rather than linked lists has been used wherever

possible to minimize memory requirements and to maximize traversal speed without trading the flexibility of the structure: the mesh can have still an arbitrary subdivision depth (there is a practical limit of 256 vertices per element but this limit should never be reached since a good mesh grading imposes a small number of midpoints for any element). Using the concept of orientation over a polygon provides a very fast way of determining the elements using a given vertex in a given direction. This allows a rather simple and efficient element subdivision algorithm. For a smoothly graded mesh (allowing not more than 3 or 4 midpoints per edge) an element's subdivision is practically done without any search over the data structures. In comparison to this, a winged-edge data structure needs considerably more searching for determining which element uses a given vertex in a given direction (besides the increased amount of memory needed for a classical linked list based implementation and the storage of edges as separate entities).

3.4.2 Vertex Radiant Exitances Computation

The Problem of Vertex Radiant Exitances Computation

Computing *vertex radiant exitances* out of element radiances is essentially an interpolation problem. If the set of vertex radiant exitances is well interpolated out of the set of element radiances, the final image will exhibit far less discretization artifacts than otherwise. The main requirement of the final step of vertex radiant exitance computations can be stated as obtaining a set of vertex radiant exitances for all the existing element vertices that will minimize the interpolation artifacts appearing during the display phase due to the linear nature of the Gouraud shading method. A supplementary remark is that most artifacts appear due to the fact that the usual Gouraud shading algorithms assume that *all* vertices of an element have their solutions (radiant exitances) lying in the same plane in the solution space. In the case this is not true (which mostly happens for large elements having many intermediate points between their corners, i.e. having smaller elements as neighbours), the linear interpolation scheme provided by the Gouraud shading might produce sharp transitions of the radiant exitance over the element, resulting in visible Mach banding. The goal can therefore be formulated as obtaining a set of vertex radiant exitances such that all vertex radiant exitances for any element are lying in the same plane in solution space (or are as close as possible to this constraint)(figure 3.18).

A vertex radiant exitance can be computed as the direct average of the radiant exitances of elements it is shared by. This method is fairly simple but has the major disadvantage that it might produce undesired results when computing the radiant exitance of the midpoints shared by elements having largely different areas: the smaller elements' radiant exitances will have a too large influence on the shared midpoints' radiant exitances. Since these midpoints will be used when displaying the large elements as well, the perceived result could be that the large element's radiant exitance is severely diminished (or increased) by some very small neighbouring element. The shading discontinuities that were captured for element centers using the nonuniform meshing can be lost when passing to vertex radiant exitances if a proper mapping from element centers to vertices is not used.
A more involved solution would be to compute a vertex's radiant exitance as the area-weighted average of the radiant exitances of the elements sharing that vertex. This will ensure that small elements on the border of a relatively large element will not sensibly affect the illumination of the large element. The area-weighted interpolation works better on meshes exhibiting a higher nonuniformity than the plain averaging approach.

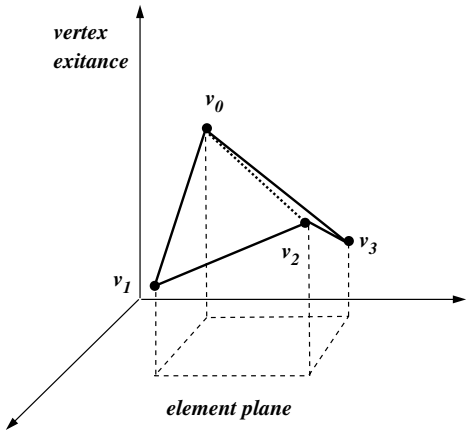


Figure 3.18: A noncoplanary quadrilateral in the element plane-exitance space. Since exitances of vertices v_0, v_1, v_2, v_3 are not in the same plane, visible interpolation artifacts will appear along the v_0v_2 line.

Still, if the mesh is too coarsely graded, shading artifacts can become visible when using linear interpolation (Gouraud shading) between vertex radiant exitances.

An additional technique was suggested by Cohen (1989): all vertex exitances are computed for all elements' vertices but an element will be displayed as a Gouraud shaded polygon, only its *corners* are used as vertices for display. This practically means that all midpoints lying on an edge of an element will be ignored when shading that element. They will be used when shading those elements for which they act as corners (figure 3.19). This technique diminishes the chance for an element to have its vertices'

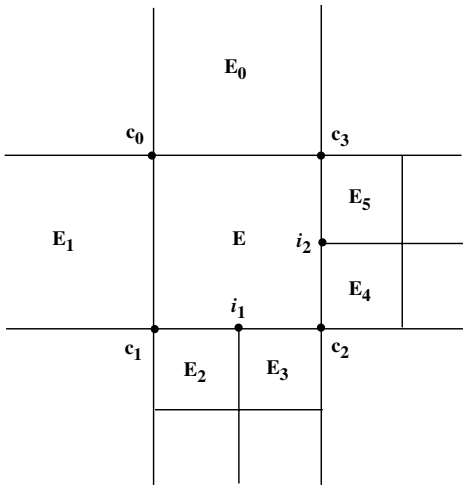


Figure 3.19: Using corners for Gouraud shading: Although E has midpoints i_1, i_2, it will be displayed as the Gouraud-shaded polygon $c_0c_1c_2c_3$. i_1 will be used only for E_2, E_3 and i_2 for E_4, E_5.

radiant exitances not lying in the same plane.
Overall, ensuring a smooth grading of the mesh strongly diminishes these interpolation

problems. Smooth mesh grading together will the above techniques were implemented. The actual implementation of the renderer gives the possibility to tune several parameters affecting the interpolation process: the mesh grading for the automatic adaptive subdivision process, the averaging method (selecting between area averaging or plain averaging) and the introduction or rejection of an element's midpoints in the display phase. Moreover, there are three types of output that the renderer can generate:

- *Flat shaded elements:* The scene's elements are output as flat shaded polygons (i.e. there is a single exitance value for all points of such a polygon. The element's exitance, as computed by the renderer, is used for this purpose and the computed vertex exitances are discarded). In this case, it doesn't matter if midpoints are output together with the corners of an element or not, since they will all have the same exitance (the element's exitance).

- *Gouraud shaded elements:* The scene's elements are output as polygons having vertex exitances. The above presented issue of keeping the midpoints or not is now relevant. Polygons output in this manner should be Gouraud shaded (to achieve linear exitance interpolation between the computed vertex exitances).

- *Hit modelled elements:* Each element in the scene is regarded as a hit (see section 5). A polygon in the scene is output as having a list of hits given by all the elements that it was meshed into. The output will therefore be a list of polygons, each polygon having a list of hits. See section 5 for a presentation of the hit radiosity model.

Radiant Exitance Normalization

The last step of the radiosity pipeline is the mapping of the computed radiant exitances in the displayable range. As previously described, radiant exitances can be directly used for displaying purposes (therefore they can be directly assimilated with the more common notion of 'pixel intensity' of a pixel being displayed in the viewed image). However, pixel intensities lie in a display device-dependent range (the most common range is the [0..1] interval, 0 meaning 'perfect darkness' while 1 is the maximum brightness achievable by the given device. The radiant exitances delivered by the radiosity renderer are in general some 'abstract', device-independent numbers, whose magnitudes might be solely related to the radiant exitances of the initial (primary) light sources that were placed in the scene. What it matters is the *ratio* of these radiant exitances, that tells the relative brightness of a point in the scene with respect to another point. It is rather hard to estimate beforehand which should be the initial radiant fluxes (or radiant exitances) that an user should assign to the primary light sources to have the final vertex radiant exitances conveniently spread over the 0..1 range, since this depends on the number of shooting iterations performed, the scene's average reflectivity, the positions of the objects in the scene with respect to the light sources, the relative size and position of the light sources with respect to the objects in the scene and other factors.
There are methods that attempt to automate this mapping process. The most commonly used method will firstly calculate an estimate of the *radiant flux unshot* after the last iteration (i.e. the sum of unshot radiant fluxes of all patches in the environment) and divide it by the *environment average reflectivity* in order to obtain the *environment average radiant exitance*. All vertex radiant exitances are then increased by this average exitance to yield the value of the exitance used for display purposes. This method is commonly used in order to estimate an ambient value to be added to the computed exitances, at the

end of the radiosity process.
In order to map the computed exitances (with added ambient term or without) to the display range, a mapping function has to be established, taking values in the computed exitances space and delivering values in the display intensities range. We shall call this mapping *exitance normalization* process. It will depend on the following factors:

- the *range* of computed exitances to be mapped to the display [0..1] range. The exitances in this range are mapped by the function in the [0..1] range. The exitances outside this range are clamped by mapping the ones below the range to the value 0 and the ones above the range to the value 1.

- the *mapping function* itself.

The main problem with the automation of such a normalization process is that the computed exitances range $[M_{min}..M_{max}]$ that is to be mapped on the [0..1] range can not be determined on an entirely automatic basis. At a first look, it seems that M_{min} can be taken as being 0 and M_{max} can be automatically found by scanning all vertex exitances and taking the maximum value. This is potentially dangerous since:

- the initial light sources have by far the highest exitances. Using any type of normalization with their exitance being taken as M_{max} would be similar to aiming a photographic camera at a light bulb: the light bulb will appear as a white spot on an almost dark background, due to the limited intensity range of such a camera (similar to the limited display intensity range of [0..1]).

- depending on the nature of the scene, number and positions of light sources, there are many cases when the user would have certain areas of interest in the scene. Thus, he will prefer to normalize the exitances according to the *desired area*'s average exitance and not taking a *global*exitance average as a reference level. In this way, he will use the limited display range with a maximum efficiency, by mapping it over the computed exitances' range for a *given* view.

- the user might desire to try several different mapping functions. For the same view, there might be *different* $[M_{min}..M_{max}]$ ranges that will perform the best. The $[M_{min}..M_{max}]$ range is therefore dependent on the mapping function as well.

A possible solution seems to be for the radiosity renderer to output a set of unnormalized exitances so that the user might try then different normalizations over this set to obtain the desired image. The exitances normalization is however not entirely a postprocessing stage: exitance normalization is used also after each shooting iteration when computing vertex exitances in order to check for high gradients (the gradient or delta-based adaptive subdivision algorithm). We have to apply these algorithms on *normalized*(display) exitances rather than on *computed* exitances, since a large range of exitances might map to a smaller range of display exitances (or even to a single value, via clamping), so there is no need to adaptively subdivide that area: even though the *computed* exitance's gradients are high, they will all be lost via a nonlinear normalization mapping. It is therefore useless to adaptively mesh these high-gradient areas since the normalized image will have no gradients at all. We therefore need to apply exitance normalization during the radiosity pipeline as well, so we need to know the normalization mapping beforehand. Consequently, the program will demand this mapping's parameters at the beginning.

Normalization functions

Three types of normalization functions (mappings) have been tried. They are described in the following.

- *Linear mapping:* the $[M_{min}..M_{max}]$ range is linearly mapped into the [0..1] range. Values outside the $[M_{min}..M_{max}]$ interval are clamped to M_{min}, respectively to M_{max}. Hence the normalization function:

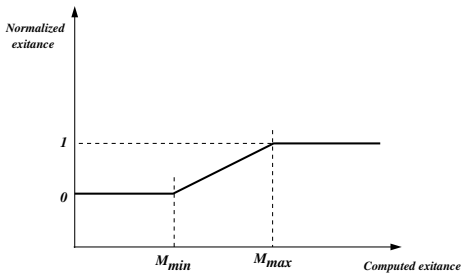


Figure 3.20: Linear normalization

$$M_{mapped} = \frac{M - M_{min}}{M_{max} - M_{min}} \tag{3.9}$$

This has proven to be by far the most useful normalization function, giving the best results. In almost all cases, M_{min}, was taken as zero and M_{max}, was somewhat greater than the average of all computed vertex exitances in the scene. Choosing M_{max} a bit lower creates an effect of 'highly burning light sources' (due to the clamping, the very bright zones get saturated to white while the rest of the scene gets relatively a bit brighter).

- *Logarithmic mapping:* the $[M_{min}..M_{max}]$ range is mapped to [0..1] using a logarithmic function:

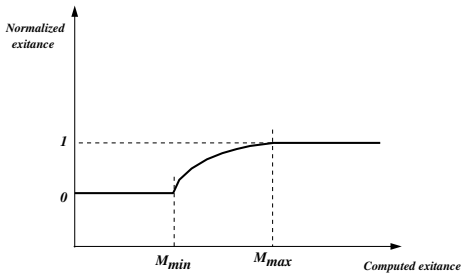


Figure 3.21: Logarithmic normalization

$$M_{mapped} = \frac{\log(M/M_{min})}{\log(M_{max}/M_{min})} \tag{3.10}$$

Actually not the value of an exitance M matters, but its ratio to M_{min}. For scenes exhibiting just a few bright spots this normalization is better than the previous one since it preferentially increases the low-intensity range while leaving the already

high exitances rather unchanged. Its effects resemble a little the effects of gamma correction. Still, care has to be taken for using this mapping over scenes having a balanced exitance distribution since the high exitance values get saturated while the low exitance range gets depleted.

- *Square root mapping:* the $[M_{min}..M_{max}]$ range is firstly linearly mapped to the [0..1] interval and then square root is applied to the result (the values keeping themselves in [0..1]):

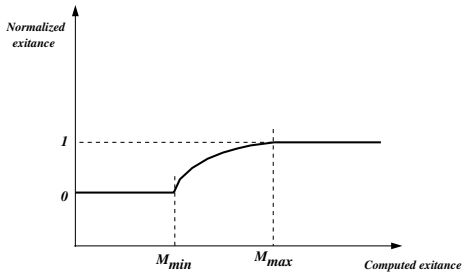


Figure 3.22: Square root normalization

$$M_{mapped} = \sqrt{\frac{M - M_{min}}{M_{max} - M_{min}}} \tag{3.11}$$

Square root mapping was tried as a replacement for the logarithmic mapping for dark scenes. It is comparatively much more stable, i.e. the M_{min} and M_{max} values can be chosen from a wider range and its effect of preferentially amplifying the low intensities is not so dramatic. In practically almost all cases it successfully replaced the logarithmic normalization. Square root mapping mostly resembles a gamma correction process (the square root) applied to a linearly normalized scene.

Conclusions

Element exitances have to be mapped to vertex exitances by an interpolation process that strongly depends on the mesh's quality. After vertex exitances have been determined, a normalization maps them to the device range. The normalization process is done during the radiosity rendering as well, in order to determine *visible* high gradient areas to be refined. The user establishes beforehand the normalization's parameters among several normalization functions and specifying a desired exitance range to map. Since the whole normalization process takes a very short time even for large scenes, the best seems to be to let the user experiment with several values until he obtains the desired illumination. An automatically-computed average exitance can be a good starting point for the user. The process resembles the exposure time tuning done for a photographic camera in order to get a desired illumination range for a given view.

3.5 The Progressive Refinement Strategy

The progressive refinement method is based on a method of selection of patches that will shoot their radiant flux into the environment. The goal of a 'patch selection' (shooter selection) method should be to select first the patches that will have the greatest influence

on the environment's lighting. There are several approaches that attempt to do this (for an overview of the methods, see [Kok and Jansen 91],[Chen et al. 91], [Shirley 90]). A simple approach is to always shoot the patch having the largest amount of unshot radiant flux (since we suppose that this will help the radiant flux balancing process to converge faster).

The detection of the patch having the maximum amount of radiant flux (out of all the environment's patches) has to be done as fast as possible since an usual environment can have thousands of patches and this selection process has to be done at each shooting iteration. There are two basic approaches for this:

- maintaining a totally sorted list of patches (sort on the amount of unshot flux). The patch to shoot will be easily found (it is the patch at the list's head). Maintaining the order relation on the list is time consuming since after a patch has shot its radiant flux, most of the patches in the list will change their unshot fluxes (so the list has to be reordered).

- maintaining a totally unsorted list of patches. When a patch shoots its flux, there's no need to reorder the list (as it was in the previous approach). Finding the patch having the maximum unshot flux is slow though: the whole patch list must be traversed (this is exactly $O(number\ of\ patches)$).

In other words, finding the patch with maximum unshot flux is fast if the patch list has a certain invariant (like the total order for example) but maintaining this invariant overt the shooting process is slow (especially for a 'strong' invariant like total ordering) so this may cancel the speed gain given by the $O(1)$ retrieval of the maximum flux patch out of a totally sorted list. Another approach that would fit between the two extremes described above would be to 'relax' the invariant maintained over the patch list: do not maintain a perfectly ordered patch list but reorder the list each N shooting iterations. The patch to shoot is still picked as the first of the list (even though the list is now not perfectly ordered). Therefore it can happen that this patch is not the one having the maximum unshot flux. This is not a problem if the list is reordered frequently (each 5 or 10 steps, for example): the first patch in the list will still have a high unshot flux, therefore the fast convergence of the progressive method will still be preserved. A supplementary improvement should be to include a limited search phase instead of selecting the first patch in the list: search the maximum unshot flux patch among the first few (10..15) patches. This will still keep the algorithm fast, while there is a greater chance to find a patch with a larger unshot flux than the first patch in the list. The complexities of the selection methods presented above are:

$$C_1 = O(NP \cdot I) \tag{3.12}$$

for NP patches and I shooting iterations, without reordering and with a full search of the patch list at each iteration.

$$C_2 = O(I \cdot NP \cdot \log_2 NP) \tag{3.13}$$

for NP patches and I iterations and full patch list reordering after each iteration (first patch in the list is taken as the maximum flux patch).

$$C_3 = O(\frac{I}{n} NP \cdot \log_2 NP) \tag{3.14}$$

for NP patches and I iterations and full patch list reordering after each n iterations (first patch in the list is picked to be shot).

$$C_4 = O(I(\frac{NP \cdot \log_2 NP}{n} + n_1)) \tag{3.15}$$

for NP patches and I iterations and full patch list reordering after each n iterations (patch to be shot is found searching the maximum unshot flux among the first n_1 patches in the list at each iteration). For a large number of patches NP and a large number of iterations I, the fourth method (complexity C_4) might be more advantageous than the plain search method (complexity C_1).

3.6 An Overview of the Progressive Refinement Loop

The progressive refinement loop is at the core of the radiosity renderer. It starts with the selection of the patch to shoot. After this patch has been found, all the visible elements of all the visible patches of all the visible polygons in the world are chosen in turn and the patch shoots its flux towards them (visibility culling tests and ray casting are used during this step). After all elements have received radiant flux, normalized vertex exitances are computed and subdivision tests (delta or gradient tests) are applied in order to determine which elements must be subdivided. For each of such elements, four new elements are created and their radiant exitance is evaluated as a function of the original element's vertex exitances. The algorithm loops now to the next shooting iteration (see figure 3.23).

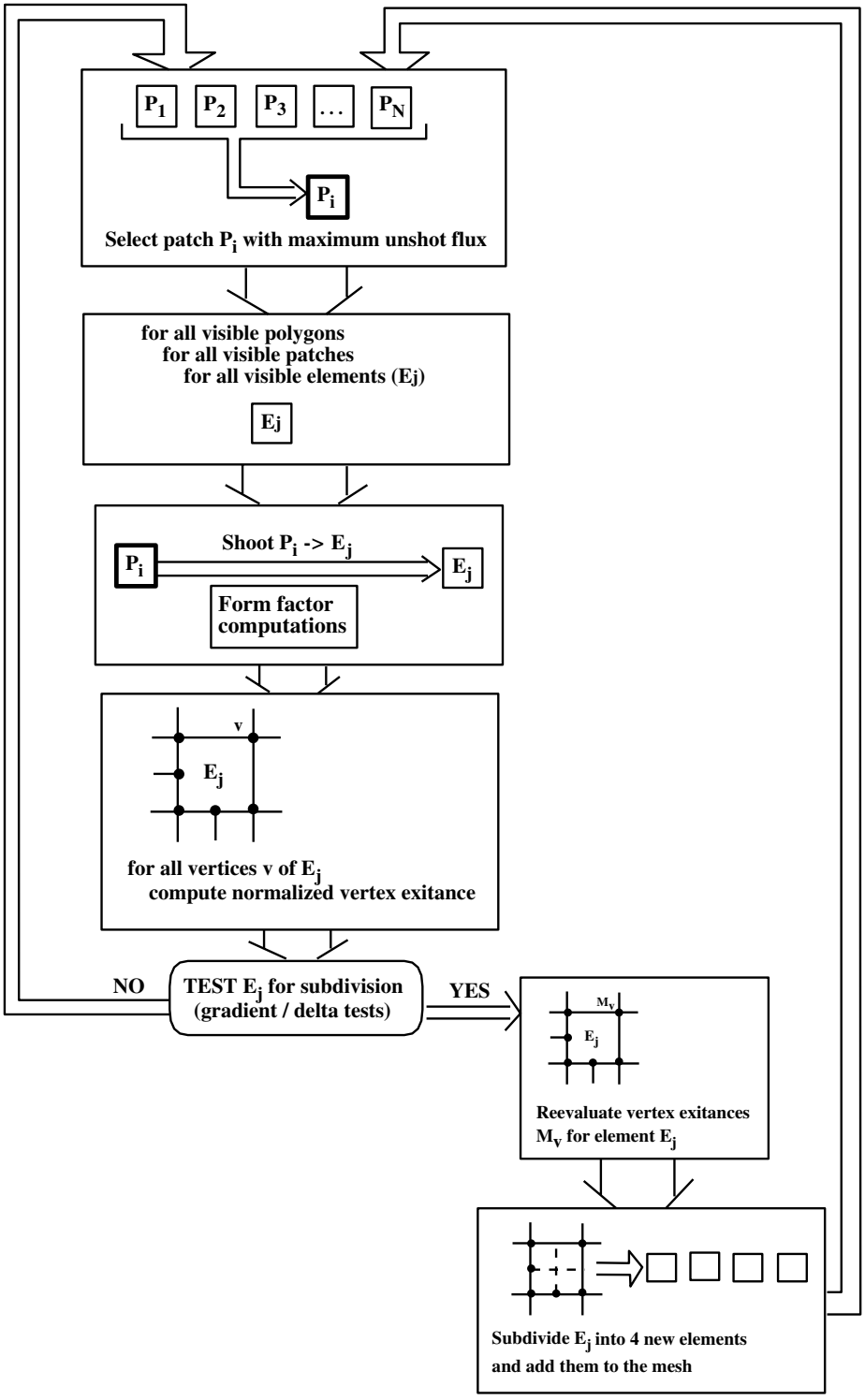


Figure 3.23: The progressive refinement main loop

Chapter 4

The Close Objects Buffer

4.1 Modelling Sharp Shadows with a Radiosity Renderer

A radiosity renderer computes the illumination of an environment by discretizing it into elements. There is only one radiance value computed per element or per vertex (one of the main assumptions of the radiosity renderer is that the radiance is constant per patch). The discretization of the environment into elements is therefore very important for the accuracy with which the shading is captured. This discretization (called also meshing or subdivision) will depend on two factors: the *initial meshing* (which is performed as a preprocessing step, before any radiosity computations have been done) and the *adaptive subdivision* strategy (which locally refines the meshing while the radiosity computations are going on). An accurate sampling of the environment depends therefore on both the initial mesh (which is the start point for the adaptive subdivision process) and the strategy used to detect and adaptively subdivide different zones. A good subdivision of the environment should use small elements in the zones of rapid variations of the radiance and larger elements in the other zones. In other words, an element should ideally cover an area over which the value of the radiance is almost constant. Visually one should notice this by the fact that there will be more small-size elements on the shadow or highlight boundaries than in the rest of the environment. The main problem resides in finding these zones of high radiance gradient and mesh them appropriately *before* having the *final* radiance result.

There is an inherent practical limit for the accuracy of the solution a radiosity-based method can deliver. If the method attempts to discretize the environment and compute one solution per element for displaying it from any viewpoint and viewing direction then there will always be viewpoints from which the artifacts caused by the sampling of the real radiance solution will be visible. This phenomenon can not be avoided since there is a fixed number of elements to be displayed for an infinite set of viewpoints and viewing directions. The most visible artifacts appear in areas where sharp shadows or small details have to be displayed. View-dependent radiosity renderers or two-pass radiosity and raytracing renderers can be used in order to produce more accurate images (and also for modelling specular reflections). These methods finally render a view in image space (at pixel accuracy level), therefore they do not deliver a view-independent solution any longer.

The best that a view-independent radiosity renderer can do is to accurately determine the shading details of the environment and sample the solution as precisely as the

maximum number of elements desired by the user permits. This strongly requests a performant meshing strategy for the environment.

There are several possibilities for determining a meshing strategy for a radiosity renderer. They can be grouped in two cathegories, depending on the moment the information that is used for meshing is collected and on the type of that information:

- *preprocessing strategies:* These strategies attempt to find the areas of high radiance gradient *before* computing the radiance solution. The information used for finding these areas consists in the relative position of the light sources with respect to the surfaces. Shadow rays techniques can be used to predict a part of the shadows that will be generated during the radiance computations. The information collected (typically sharp shadow boundaries similar to those generated by ray tracing renderers) is used for generating a non-uniform initial mesh, having refinement zones around the detected sharp shadow boundaries. Sometimes this information is saved and used subsequently during the adaptive refinement performed during the radiosity computations.

- *adaptive refinement strategies:* These strategies refine an initial mesh *while* the radiosity computations are performed. They are generally used with a progressive refinement radiosity method. The information used for deciding the refinement consists in the solution computed so far. After an iteration has been done, the current solution is mapped on the elements and a search is done for elements having high radiance gradients. These elements are then subdivided and the same iteration is performed again (this is called *reshooting* in the case of a progressive refinement method) or alternatively the next iteration proceeds.

A radiosity renderer can use both types of methods: the preprocessing will generate a nonuniform initial mesh that has a certain degree of local refinement due to the detection of the 'primary' shadowed areas while the adaptive refinement will keep on improving this mesh in the high radiance gradient zones detected while progressively computing the solution.

Both strategies have some drawbacks:

- a shadow-detection preprocessing strategy will be able to detect just the shadows generated by the *primary* light sources. Although in many scenes these are the most pronounced ones, there can be many cases when a secondary light source (like a reflection of a bright lamp shining directly on a white wall nearby) can cause pronounced shadows. These shadows will be missed by the preprocessing algorithm. This problem can be partially solved by performing *several* shadow detection passes, not only for the initial light sources but also for the next patches that will shoot into the environment (the secondary light sources).

- the shadow-detection algorithm is well suited for modelling sharp shadows for ray tracing purposes (in which case the light sources are supposed to be pointlike). Many of the shadows cast by such an algorithm might be irrelevant for the radiosity process, since the light sources modelled by patches are *area* light sources, casting generally soft shadows. Therefore the usage of such an algorithm might generate a mesh refinement in some areas where there don't really exist sharp shadows.

- a large scene can contain a very large number of groups of small objects (e.g. pencils on a table, books in a bookcase,etc). It is very expensive to use the shadow-

detection algorithm against *all* objects in the world, especially if this is done several times for several light sources since a shadow-casting algorithm working entirely in object space is quite time consuming.

- an adaptive refinement strategy does not perform well if its initial mesh is not fine enough in the areas of interest. This is basically a sampling problem: if the initial sampling frequency (given by the initial mesh) is too low (elements are too large) then relevant shading detail can be completely missed. It is practically impossible to determine afterwards that such shading detail exists just by examining the computed solution. Adaptive refinement performs well when the initial meshing is fine enough to capture a sharp variation that can be *refined* further. In many scenes there are however numerous small shading details that can not be trapped by the initial meshing (due to the preprocessing strategies' problems listed above). They might be accidentally detected during adaptive refinement but in most cases they will be ignored. This problem can be slightly alleviated by relaxing the gradient criterion that starts the adaptive refinement of an area, so that the subdivision will start faster. A major drawback is that this will cause refinement to be done over large areas where it is not really necessary, generating a very large number of elements. The meshing effort is clearly not directed in the areas of interest.

In conclusion a good environment meshing must:

- capture the shading details as accurately as allowed by the maximum number of elements permitted by the user. This information can *not* be provided by the radiance solution computed so far but has to be obtained by other means. Preprocessing techniques involving the primary light sources may provide partially this information at rather high costs.

- exhibit smooth radiance transitions between neighbouring elements in order to allow a smooth shading of the result. This information can be extracted by analyzing the computed solution's gradient.

4.2 Sharp Shadows

As it was previously described, adaptive refinement strategies can not (and should not be used to) provide the refinement level required for capturing shading detail missed by the initial meshing resolution. This section presents a method for detecting the existence of such shading detail and for adaptively refining the mesh in the areas of interest. Looking back at the rendering equation (2.21) one can see two main reasons that cause visible illumination detail to appear over an area illuminated by a light source:

- the points on that area are unequally illuminated by an unoccluded light source. This happens if the lightsource and the illuminated area are facing each other at sharp angles and the distance between each other has a sufficiently large variation over their points (the cosine and r terms vary rapidly between neighbour points of the illuminated area). The variation of illumination over the receiver will be rather smooth and continuous (these are the 'smooth shadows').

- the points of the illuminated area have different visibility terms with respect to the light source. This happens if the light source is occluded for some points of the

receiver and not occluded for other ones. Since occlusion is zero or one, the variation of the illumination over the receiver may exhibit sharp, irregular transitions between neighbour points. These are the 'sharp shadows'.

Smooth shadows (smooth radiance variations more exactly) are therefore treated properly by the adaptive refinement techniques: the areas exhibiting too sharp variations are subdivided and the radiance is recomputed over the new elements. Adaptive subdivision using gradient criteria works very well for producing a good mesh in which the radiance differences between neighbouring elements are smaller than the threshold required for display purposes.

Sharp shadows are mainly generated by partial occlusion of light sources. Looking again at the rendering equation we can see that there is a high probability for a sharp shadow to occur over an element receiving radiance if:

- the light source is very bright (has a high radiance or flux value). The difference between areas exposed to light and areas occluded from it will be larger.
- the light source *directly* illuminates the object. The angles between the source's normal and rays leaving it to the receiver are very small, hence the receiver will get a large amount of the source's radiance. The same applies for the receiver.
- the light source is close to the receiver. Both previous criteria and this one are actually stating that the receiver gets an important amount of the light source's flux (so the potential shadows might be sharp, if any).
- the lightsource has a small area. The smaller the area, the sharper the shadow is. At the limit, a pointlike light source will create a shadow with no penumbra.
- there is an object between the source and the receiver, partially occluding the receiver. If this object is *closer* to the receiver, then it is very probable that the shadow it will cast will be sharper. Therefore, the distance between the occluder and the receiver influences the sharpness of the shadow.
- the receiver's total reflectance (the sum of ρ_R, ρ_G,ρ_B) is large enough such that its radiant exitance is sufficiently high for a sharp difference in illumination between the shadowed and the non shadowed areas to be visible (shadows are not well visible on a very dark coloured receiver).

4.3 The Close Objects Buffer

The *close objects buffer* attempts to solve the problem of detection of areas where a sharp shadow can occur. As we have seen, there is a high probability of sharp shadow appearance when a receiver (an element) is partially occluded from a light source by an object which is very close to it and when the light source casts a strong illumination over the receiver. For each element in the environment, a close object buffer will be used that stores references to all the objects that are above and close to the visible surface of the element. Practically, this will be an item buffer storing references to the polygons close enough to the element's surface and partially occluding the element: When a light source is about to shoot its radiance towards an element, the close object buffer of that element is investigated: if it is empty, then there aren't any objects that might cast sharp shadows over the element, so the radiance is received by the element. If the buffer is not

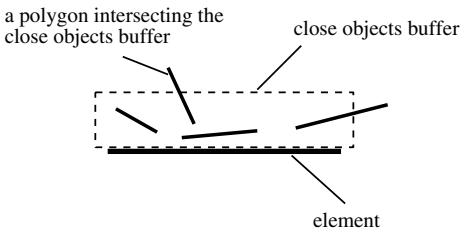


Figure 4.1: An element and its close objects buffer

empty, then there are objects partially occluding light coming from the source and potentially casting sharp shadows. The element is then directly subdivided (without shooting at it anymore) and the polygons held into its buffer are redistributed in the buffers of the newly created elements. The close objects buffer proves itself efficient in two

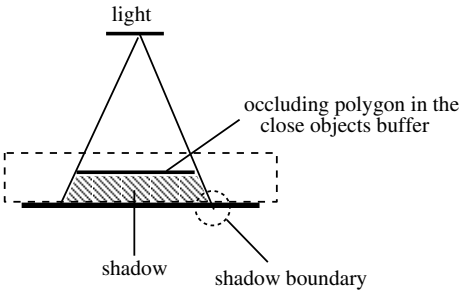


Figure 4.2: Shadows created by occluding polygons in an element's close objects buffer

situations:

- small objects are located on or nearby the surface of large objects (like pencils or cups on a table top). It is very likely that the initial meshing of the large object (the table top) will be coarse enough such that it won't be able to capture the small but sharp shadows cast by the objects placed on its surface (like the pencils and the cups). The close objects buffer of the table's elements will detect the presence of the close objects and start the adaptive sudivision that will ultimately capture the detail shadows.

- arbitrary objects that have common edges (like two walls sharing an edge or the inside faces of an open box). If the initial meshing of such objects is coarse, it is very likely that the subtle shadows appearing sometimes near an edge shared by two faces will be missed. Again the small objects buffer will detect such situations and initiate a local mesh refinement over the area close to such edges.

The close objects buffer is unique per element: it stores all the polygons being in the proximity of its surface independently on the lightsources. The reason for this is that for objects very close to a surface it is assumed that they will cast a *sharp* shadow if illuminated by *any* light source above that surface. This is true in most cases since the distance receiver-occluding object is much smaller than the distance source-occluding object (the buffer's height is very small compared to the usual inter-object distance). The cases when this assertion doesn't hold will be treated separately.

The close object buffer is not built by default, in a preprocessing stage (as the one used for some shadow-detection algorithms). Initially, all elements have an empty buffer. When a light source is about to shoot to an element, the light source is firstly checked to see if it potentially could cast a sharp shadow over the receiver (using a combination of the above criteria including light source's power, distance to the receiver, mutual orientation of receiver and source and others). If the criteria succeed, then the receiver's close objects buffer is checked: if it has been already built, it is used as described above. If not, this means there hasn't been any lightsource bright enough that had shot to this element so the buffer is built now and then used. If the criterion fails, then the light source is not able to cast a sharp shadow on the element, so we can directly shoot its radiance at the element without subdividing.

This strategy ensures that the close objects buffer will be built *only* for the elements for which there is a strong probability to get a sharp shadow. Many elements will never have such a close object buffer (therefore the memory usage increase is negligible for them - an empty buffer is 5 bytes per element in the current implementation). Moreover, the buffer is built *on demand*, that is only when there's really a need for it so this saves also computing time since we don't have to generate the close objects buffers in a preprocessing stage, but rather on the fly, during the radiance computations. The process of generating the buffer fits therefore directly in a progressive refinement method.

The subdivision of an element is triggered by the fact that its close objects buffer is not empty and that there's a light source shining on it for which the above criteria have decided that it is able to create sharp shadows on this element. We could use a simpler version of this buffer (requiring just a one-bit flag per element) in which an element's buffer has the states full or empty. There are several advantages of using an item buffer instead (i.e. storing all polygons intersecting the close objects buffer):

- when an element is subdivided, we can compute the buffers for the new elements quite rapidly, by distributing the polygons in the original element's buffer into the new buffers. This is typically done very fast since there aren't many occluding polygons in an element's buffer.

- the close objects buffer can be used in a more advanced way than just checking if it is full or empty. The distribution of the polygons inside it (their relative positions, sizes, distances from the element) can provide valuable information about the kind of occlusion they will generate when lit by a given lightsource. A more elaborate way to use this information will be presented further.

4.4 Building the Close Objects Buffer

The buffer will contain, as previously described, references to polygons being in the proximity of an element's visible surface. Geometrically speaking, the buffer is a prism having the base a bit larger than the element and a given height δ. In the simplest case, one can use a simple rectangular box placed on the element's surface (similar to the hemicube used in form factor determinations)(see figure 4.3).

The box has been made a bit larger than the element itself in order to trap also the polygons located in the vicinity of the element. This is especially useful for elements near an edge shared by two polygons. These are typical cases when a shadow strip occurs and they are directly detected using the close objects buffer (see figure 4.4).

As described previously, the buffer holds all polygons that potentially cast sharp shadows over the receiver. A simple approach would be to store in the buffer all poly-

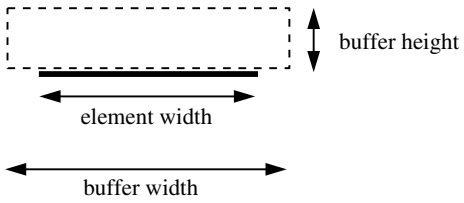


Figure 4.3: Sizes of the close objects buffer

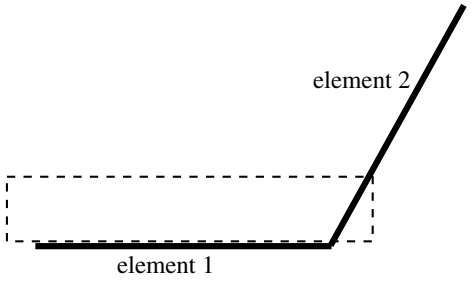


Figure 4.4: Two elements sharing an edge: element 1 is in element 2's close objects buffer and conversely

gons intersecting the element's box. However not *all* the polygons intersecting this box will cast sharp shadows over the element, so a more elaborate strategy would try to test the polygons intersecting the box (basic candidates for the close objects buffer) and select only those that may indeed cast a sharp shadow. Several criteria attempt to do this:

- A first important one is the *total occlusion criterion*: a polygon that covers the whole box (i.e. intersects all the box's edges that are normal to the element's surface) will occlude the whole element rather than cast a sharp shadow on it. Indeed, if illuminated from above, it will occlude practically completely the element from any light source. This is a special situation in which the element beneath can be skipped during the shooting process: since there's practically no chance for a light source to reach it, there will be no shooting towards this element anymore. This can save an important amount of time (consider, for example, the case when a table is covered with several sheets of paper: most of the elements of the table are invisible from a light source placed above the table since they are completely occluded by the paper sheets. There will be just one attempt to shoot at them and then they will be found to be totally occluded, hence it will be skipped from subsequent shootings (see figure 4.5).
- The total occlusion criterion may be invalidated in two cases:
  - if the light source illuminates tangentially the element, then the element might not be totally occluded from the light even though its close objects buffer is totally occluded as seen from above. This situation is handled by default since a tangential illumination that might creep under the occluding polygon will probably influence very little the element's final radiant exitance, so it can be safely ignored (figure 4.6).
  - if there exists a *primary* light source intersecting the close objects buffer of an element that is totally occluded, this lightsource might be placed *below*

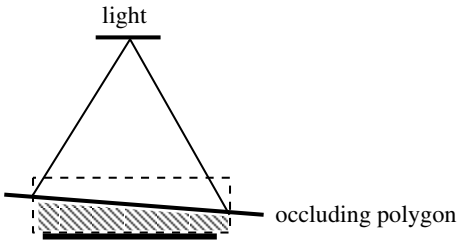


Figure 4.5: Totally occluded element

Figure 4.6: Tangential illumination of a *totally occluded* element

the occluding polygon. Therefore this light source will not be totally occluded when illuminating the element so we can't completely skip shooting at this element since the source chosen to shoot might be exactly the one in the element's buffer (figure 4.7).

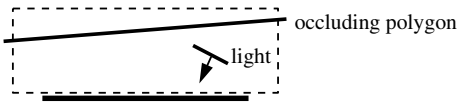


Figure 4.7: Light illuminating under a *totally occluded* element

– if the buffer is totally occluded and there's no primary light source inside as described above, then we practically don't store anything in the element's buffer but just mark it as 'totally occluded'. This saves memory and time and also is consistent with the fact that a totally occluded element will never need to be further subdivided so no new buffers have to be computed out of the current one. Another good alternative would be to remove this element out of the polygon's elements list so that the renderer will never try to shoot at it in the future. This can save an important amount of time since there typically are many quite large areas in the scenes to render that are totally occluded from any light source and therefore it is useless to try to shoot at them or even to test these elements for total occlusion. For example, a box put on a table totally occludes the elements on the table top under it from any light source.

The element can be marked as totally occluded only in the case there doesn't exist a light source in its close objects buffer or in the case such a source does exist but it is oriented in such a direction that it doesn't illuminate *directly* the element. The reason for this is that a such light source illuminates indirectly the polygon, hence it has a smaller chance of casting a sharp shadow than a directly illuminating source.

It is important to notice that this criterion should be applied only for *primary* light sources since they are the only lights that have a reasonable chance of casting a

strong illumination on an element whose close objects buffer is totally occluded. Other (primary or secondary) light sources can not practically illuminate this element so the element can be regarded as totally occluded from any light source except a primary lightsource intersecting its small object buffer and shining directly towards it.

Such tests are very fast and they can be done when the small objects buffer is built for an element with virtually no time penalty.

- In the case the total occlusion criterion fails but a polygon still intersects the small objects buffer of an element, this polygon will be regarded as a potential shadow caster and will be added to the element's buffer. There are a few cases when such polygons do not really cast a sharp shadow over the element but they are still added to the buffer. A subdivision process will proceed even though after several progressive refinement iterations the element may be fully illuminated , therefore there will be no sharp shadows over it. Since these cases are statistically very few (the time penalty being paid due to the unnecessary subdivision being consequently very small) we can ignore them (i.e. refrain from using special tests to reject them).

Summarizing the above:

- the small object buffer is built only on demand, when there is a high probability of a sharp shadow to occur on an element.
- the buffer can be modelled with a rectangular box similar to the hemicube placed on the top of an element, with the height proportional with the element's edge and the width a bit larger than the element. A polygon is said to be in the small objects buffer if it intersects this bounding box.
- a special function of the buffer is to detect cases of total occlusion of elements from all light sources. This can be done by checking the intersection of a polygon with all buffer's edges normal to its base and an additional test concerning the existence of a primary light source in the buffer directly illuminating the element. Totally occluded elements are completely excluded from the flux gathering process.
- at an element's subdivision, the small object buffers of the new elements an be easily computed out of the original buffer.

We can say that the construction of the buffer attempts to detect the possibility of a sharp shadow from the *receiver's point of view*.

4.5 Using the Close Objects Buffer

The close objects buffer is used within the progressive refinement process, when an element is selected to receive radiant flux. There are several steps performed during this process:

STEP 1: The buffer is first checked to see if it is totally occluded or not. If it is, then the chance of a sharp shadow is small, so we simply skip shooting to this element. This step might be skipped if the totally occluded elements are removed from the polygon's element list as described above.

STEP 2: Depending on the estimated interaction between the shooter and the element, the buffer will be used or not. A criterion that attempts to estimate this interaction can be based on the unoccluded form-factor between the receiver and the shooter. This factor has to be computed anyway for other purposes so there is no additional time cost to be paid.

Recalling the necessary conditions for having a visible sharp shadow on the receiver, we can evaluate:

$$\Delta M_i = (\rho_{R_i} + \rho_{G_i} + \rho_{B_i}) \frac{\cos\theta_i \cos\theta_j}{\pi r^2} A_j \tag{4.1}$$

where i is the receiver element and j the shooter. ΔM_i will be an approximative estimate of the increase in radiant exitance of element i due to the shooter. This increase can be compared to the actual radiant exitance of the element M_i to determine if it can indeed produce a visible shadow. Alternatively the fraction of shooter j's flux reaching element i can be evaluated: if it exceeds a certain percentage of shooter's total shot flux, then there's a high probability that a possible shadow over element i will indeed be sharp. This is justified by the fact that the shooters are picked in decreasing order of their unshot fluxes in the progressive refinement method, hence a large fraction of a shooter's flux is indeed capable of creating a sharp shadow.

Together with this criterion, the element's size is checked as well. If the element is too small, we ignore the buffer and shoot to it as usually. It is important to remark that the stop threshold for the subdivision initiated by the previous criterion is independent on the stop threshold used for the gradient-based refinement: the small objects buffer subdivision attempts to capture *sharp* shadows over rather small areas, so the minimum element size should be sensibly smaller than the one allowed for the gradient-based subdivision, since the reason for gradient-based subdivision is to accurately capture the variations of an *already calculated* radiance function and not to attempt to predict and detect sharp shadow areas.

STEP 3: If the criterion has decided that the interaction is strong, the buffer is checked to see if it has been built or not. If it has been built and it is not empty, then there are close objects to this element, therefore potential sharp shadows. The element is simply put aside to be subdivided at the end of the current iteration. If the buffer has been built but it is empty, then there aren't close objects to this element so the shooting part takes place. If the buffer hasn't been built, then this is done at this moment and step 3 is reexecuted.

Summarizing the above:

- the small objects buffer is checked at shooting time for total occlusion. A positive answer will prevent shooting to a totally occluded element from *any* light source, thus saving several expensive form-factor and occlusion tests computations. This result can not be obtained with an usual preprocessing shadow detection method which detects only the shadows cast by the primary light sources.

- a criterion estimating the strength of the interaction between shooter and receiver is used to determine if there's a high probability of having a sharp shadow. If not, the element will receive radiosity as usually.

- a non-empty buffer will trigger element subdivision. This subdivision is different from the one initiated by the gradient-based refinements and attempts to accurately capture the probable sharp shadow boundaries over the element.

- the small objects buffer subdivision and the gradient-based subdivision are independent processes; although both try to refine the mesh, they are based on different criteria, have different stop thresholds and practically perform their best in different areas of a scene.

The above algorithm causes a minimal overhead to a normal progressive refinement gradient-based adaptive subdivision process while attempting to predict the possibility of a sharp shadow using the close objects buffer's contents, as opposed to the gradient-based refinement which refines the result on the basis of the computed solution.

4.6 A Two-Level Close Object Buffer

The management of a close objects buffer is simple but it has to be done for each element in the scene, each time that it has to receive radiant flux from a shooter. If the number of elements is large this process can be slow: there will be at least a test done for each element's close objects buffer (even though many buffers will always be empty) and building a buffer involves, in a brute-force approach, intersecting *all* polygons in the scene with it. Although these intersections can be significantly speeded up by using the octree built for form factor ray tracing purposes (the polygons potentially intersecting an element's close objects buffer are a subset of the polygons contained in the octree cells over which that element spans), the method presented in the following can prove more efficient.

There exists a coherence of the set of close objects for the elements of a polygon that can be exploited to speed up the building and the usage of the close objects buffer. This coherence can be expressed by the fact that neighbour elements tend to have a similar set of close polygons - that is, a similar close objects buffer. More precisely, there usually exist large areas over the polygons of a scene for which all the elements have an empty close objects buffer.

We can take advantage of this coherence by introducing a close objects buffer for each polygon in the scene. A polygon's buffer would have the same semantics like an element's one: it stores all the polygons close to the visible surface of that polygon. All polygons' buffers are not built by default, but rather on demand, the first time we have to shoot at such a polygon (exactly like for an element's buffer). Such a two-level hierarchy of close objects buffers can speed up the whole process significantly, especially for scenes containing a large number of polygons:

- building an element's buffer is now very fast:
  - if the element's polygon has an empty buffer then *all* its elements will have empty buffers also.
  - if the element's polygon's buffer is totally occluded then *all* its elements will be totally occluded also.
  - if the element's polygon's buffer is not empty then its elements build their buffers by testing intersections only for the polygons in their father polygon's buffer. This is much faster than testing all polygons in the scene and comparably faster to using an octree.
- using the close buffer is now very fast:
  - if a polygon has an empty buffer then we can skip at once testing its elements' buffers: there are no close objects therefore no possibility of sharp

shadows over this polygon. The computations have now exactly the same speed for this polygon as a usual radiosity renderer.

- if a polygon has a totally occluded buffer then we can skip at once shooting at *all* its elements. This will give a clear speedup over an usual radiosity renderer that would still try to shoot flux to these elements.
- the buffers of a polygon's elements will start to be used only if the polygon's buffer is not empty and not totally occluded.

This two-level scheme can be extended to include a larger number of levels, similarly to the hierarchical radiosity method presented by [Hanrahan et al., 1991].

4.7 Conclusions

Adaptive gradient-based mesh refinement can provide only a smoothing of the radiosity solution over a given environment but typically fails to detect sharp detail shadows. A strategy integrating directly in a progressive-refinement radiosity method has been presented, featuring the possibility of detection of detail shadows cast by small objects or sharp shadows cast by occluding objects placed in the immediate vicinity of receivers. The presented method is virtually independent on the initial mesh's resolution (works entirely in object space), requires small amounts of memory and imposes a very small time penalty over a normal radiosity method. It doesn't need any preprocessing step but is rather applied only on demand and it can integrate and cooperate very well with a normal gradient-based refinement. The method also detects totally occluded elements, eliminating them from the radiance computations and speeding up the whole process. Furthermore, the user can easily control the maximum level of refinement of the method, being therefore able to choose the desired one for his time and memory resources. A two-level hierarchical variant of the close objects buffer offering an important speed improvement has been presented.

The method has been practically tested and the time penalties noticed were less than 2% of the total rendering time. The first iterations of the progressive refinement are perceivably slower since most of the elements' close objects buffers are not yet built now and they have to be built. After a few iterations, most buffers get computed so the process practically reaches its normal speed. Tuning the buffer's height parameter can strongly affect the level of refinement (a high buffer will presumably contain more polygons inside, so there will be a greater chance for more elements to be subdivided due to non empty buffers).

The improvements of the rendered images are quite visible. Detail shadows are now detected and captured appropriately around small objects placed on large surfaces or around edges shared by two large surfaces.

Chapter 5

The Radiant Flux Hit Model

5.1 Introduction

This section describes an optional additional stage of the radiosity rendering pipeline. At the end of the rendering phase the output will consist in a number of elements having either vertex exitances or element exitances (going to be Gouraud shaded or flat shaded respectively). The additional stage described here will take each element and model it as a *radiant flux hit*. Such a hit is practically another model for the radiant flux (and therefore radiant exitance) distribution over the element. After each element has been replaced by such a hit, the vertex exitances for all the vertices of all elements in the scene will be evaluated as a superposition of the hits' contributions. The resulting scene will be a set of elements with vertex exitances, that can be Gouraud shaded. The following will describe the procedure briefly outlined above.

5.2 The Radiant Flux Hit

As previously presented, an element is modelled as having a *constant radiant flux (or radiant exitance) distribution* over it during the shooting process, since the renderer evaluates the radiant exitance only in the element's center, assuming that this point will characterize all the element's surface in terms of radiant exitance. At the end of the rendering phase, elements are displayed using linear interpolation of the exitance (Gouraud shading). However, this procedure assumes as given the fact that the exitance has a linear variation between the elements' vertices. In fact, we don't have any information about the real exitance values in any point but the points where we compute it (namely the centers or the vertices of the elements). The problem is how to interpolate these computed values over the surface of the elements in order to get a shading that is accurate or at least is not exhibiting visible unpleasant artifacts.

The radiant flux hit model will create a radiant flux (radiant exitance) distribution over an element that will be different from the constant one. The model will assume that the radiant flux will have a Gaussian distribution over the element's plane, with the maximum in the element's center and decreasing radially from this point with the square of the distance: We call this Gaussian flux distribution a *hit* because the effect it produces is similar with having the energy (or radiant flux) concentrated in the center of the element (like a hit) and spreading out radially from there over the element plane's surface. While the constant flux distribution assumes that the final exitance of an element will

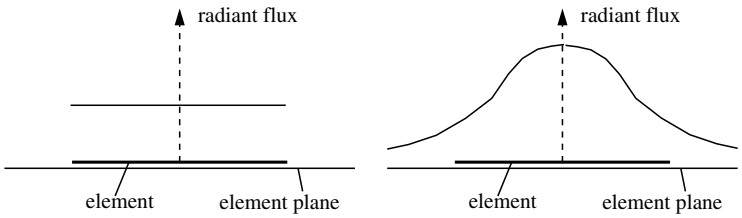


Figure 5.1: Constant and Gaussian flux distributions

be given just by its computed flux value, the Gaussian distribution will 'spread out' the computed flux of an element over its neighbouring elements. In other words, the Gaussian distribution will act like a filtering function (a convolution with a Gaussian filter having the kernel larger than the element's surface) over the computed flux values (in fact the kernel is infinite but the filtering function can be regarded as being practically zero for distances larger than twice an element's edge). It becomes now possible that the flux value of an element influences the flux values of some neighbouring elements. Compared to this, the procedure that was computing vertex exitances by averaging element exitances can be regarded as a simpler form of interpolation, where the exitance in one vertex is given just by the elements sharing that vertex. In contrast, the hit model will state that the exitance at *any* point on a polygon's surface will be function of the exitances of *all* elements of that polygon: Assuming that the radiant exitance will have a Gaussian distribution radially around the center of an element, we have the following equation for the radiant exitance at a certain distance from an element's center:

$$M(r) = M(0)\exp^{-\frac{r^2}{A}} \tag{5.1}$$

where r is the distance from the element's center, A is the element's area. The Gaussian distribution function is weighted with the element's area since we want to have the spread proportional with the element's area. Since we have $M = d\Phi/dA$ (the radiant exitance for diffuse reflectors), we get for the value of the element's flux Φ:

$$\Phi = \int_{all\ element\ plane} M(r)dA \tag{5.2}$$

Integrating with $dA = 2\pi r dr$ we get the value of the radiant exitance for $r = 0$ and finally the expression of radiant exitance in any point of the plane:

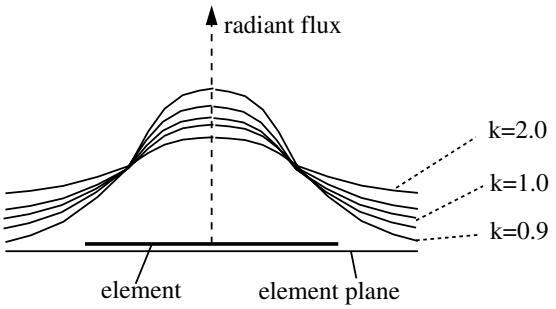
$$M(r) = \frac{\Phi}{\pi A}\exp^{-\frac{r^2}{A}} \tag{5.3}$$

This is the exitance caused by a single hit over its element's plane. If we consider all the hits for all the elements over a polygon and superimpose their effects, we get the radiant exitance at any point of this polygon due to all hits over its surface:

$$M(r) = \sum_{all\ elements\ j} \frac{\Phi_j}{\pi A_j}\exp^{-\frac{r_j^2}{A_j}} \tag{5.4}$$

where r is a point on the polygon's surface and r_j the distance from that point to element j's center, A_j is element j's area, Φ_j is element j's flux.

The above equations use a filtering function having the kernel roughly equal to twice an element's edge (since the Gaussian function is practically zero for distances larger than this). Sometimes it is useful to tune the kernel's size explicitly: a larger kernel will produce a more accentuated spreadout of the radiant flux (equivalent to an image where shadows are softened) while keeping this kernel smaller will diminish the spreadout. This can be simply achieved by replacing the element's area A in equation (5.3) with kA, where k is a real constant. Good values for k are in the interval $[0.5..3.5]$. Values larger than 1 will produce a spreadout while values smaller than 1 will produce an opposite 'flux gathering' effect. In most cases, the values smaller than 1 do not perform an interesting filtering result: After replacing each element with a hit, the hit based model



b) Gaussian flux distribution

Figure 5.2: Different Gaussian filters

will be able to evaluate the radiant exitance in any point of a polygon's surface with the above equation. The polygon can be now remeshed into elements (and vertex exitances for these elements can be evaluated with the above equation or alternatively we can use the existing element mesh and evaluate exitances at its vertices.

Other filtering functions can perform better than Gaussian ones (powers of cosine functions, triangular filtering functions, etc). In order to increase the speed of the filtering process, we shall not perform the summation described in equation (5.4) over *all* the elements of a polygon (such a process should take $O(n_{elems}n_{vertices})$, where n_{elems} is the number of elements of a polygon and $n_{vertices}$ is its number of vertices). A better approach is to start, for each vertex, from the elements sharing it and perform a breadth-first traversal of the element mesh and stop at the point where the distance from the elements to the vertex is large enough such that the filtering function is almost zero. This will give a complexity of $O(n_{vertices}N_{elems})$ where N_{elems} is the average number of elements around a vertex that influence that vertex. This number is much smaller than the total number of elements of a polygon. The hit-based model performs a radiant flux (radiant exitance) redistribution over the computed set of element exitances yielding a set of vertex exitances. It is a more sophisticated way of exitance interpolation than the linear interpolation scheme used for computing vertex exitances out of exitances of elements sharing that vertex. It allows smoothing exitance variations over a polygon up to the desired degree by tuning the filter's kernel size. The filtering step can be successfully used especially on fine element meshes (where the spreadout will not be very high).

Chapter 6

Modelling and Viewing the Environment

6.1 Introduction

The radiosity renderer is just the central part of the whole graphics pipeline. The scene to be rendered has first to be built using a modelling application as well as the rendered result has to be viewed using a 3D viewing application. Two pieces of software have been developed for this purpose. Section 6.2 describes WINVIEW, a MS Windows based 3D viewing application. Section 6.3 describes BUILD_3D, a 3D modelling application. Both applications have been used together with the radiosity renderer.

6.2 WINVIEW: A 3D Viewing Application

6.2.1 Introduction

This section will present a 3D viewing application that was designed for the final part of the radiosity pipeline. The application that will be presented can be used for viewing the tridimensional scenes that are the input of the radiosity renderer as well as the rendered results obtained at the end of the radiosity process. The viewer's features will be first described, followed by a brief presentation of its design.

6.2.2 General Overview

WINVIEW is an application offering a basic set of 3D viewing and rendering functions. It is designed mainly for viewing 3D scenes composed of any number of objects, represented in a polygonal mesh format. The scenes to be viewed can be loaded as files that are created by other 3D modelling applications.

WINVIEW supports both a hierarchical, structured 3D world description and an unstructured description. For the structured case, a world is a set of independent 3D objects. An object is described a set of surfaces together with some color and material properties that are common for all the surfaces. Each surface is modelled as a a set of polygons that share the vertices at all their common vertices over the whole surface. A polygon is finally a set of 3D vertices. Polygons inherit the material and color attributes of the object to whom they belong. The structured world is encoded in the input

file according to a format that will be described in Appendix B). The structured input is provided for importing 3D scenes modelled by applications that can output their results encoded in such an object-surface-polygon- vertex hierarchy, offering advantages of memory, speed and scene description easiness.

The unstructured world is represented by a set of independently described 3D polygons, each having some individual attributes (color, vertex normals, vertex colors, material properties, etc.). This type input is provided for viewing 3D scenes generated by applications that encode their results on a per-polygon basis. This input type places an extra overhead on memory and decreases the viewing speed but it should be more general than the structured input type.

WINVIEW is a MS Windows based application offering the following features:

- 16 colors 256 colors, 65536 colors and 16 million (true color) rendering modes for all the available resolutions.
- both color and monochrome (grey-shaded) rendering modes.
- on the fly color quantization and dithering methods are used for the 16 and 256 color mode and also the 65536 color monochrome rendering mode. Although not perfect, these options eliminate a good amount of color-banding effects, while preserving the original colors' hues.
- interactive viewpoint positioning for viewing the scene from different angles.
- interactive light-source direction positioning.
- zoom in/zoom out and viewport resizing facilities. An auto-sizing window facility is provided in order to automatically obtain a good overall viewing from any direction.
- perspective and parallel projections. Perspective projection has an option for choosing an optimum viewing distance.
- one directional colored light source is supported. Extension to several colored light light sources can be easily done.
- several rendering modes are available: wireframe, opaque hidden surfaces, flat shading, Gouraud monochrome and colored shading, Phong monochrome and RGB shading.
- supports up to 20000 polygons on screen.
- depth sort or z buffer based hidden surface removal. A special z buffer is implemented that can be adjusted at runtime to perform its best according to the available free memory.
- image grabbing facility (saving is done in the Windows BMP format, works for all resolutions and color schemes supported by Windows).
- several precalculation and data caching schemes are used in order to enhance the throughput on slower systems.

6.2.3 User Guidelines

WINVIEW is entirely menu and dialog-driven via the Windows interface. A description of the offered options follows:

- *File menu*
  - *Load world:* Loads a structured or unstructured world description file. The structured files are supposed to have the extension 3D while the unstructured ones should have the TAK extension. The formats for the two types of file are described in Appendices B,C. The loaded world replaces the current world and the viewing parameters and rendering mode are reset to the defaults.
  - *Remove world:* Simply clears the current world from memory.
  - *Save as bitmap:* Saves the current view as a Windows BMP file.
  - *Save as bitmap:* Exits the program.
- *View menu*
  - *Viewpoint window:* Creates a viewpoint-positioning window. The viewing direction can be changed around the origin using the interface provided by the viewpoint window. *Apply* should be used to apply the new viewing settings. The *Dynamic update* checkbox sets the viewer to a state where it dynamically updates the view while the user keeps pressing the view direction arrows. This option best works when a fast rendering mode (e.g. wireframe) is selected.
  - *Set auto window:* A toggle option that should normally be left on. It provides automatic view scaling to fit the displayed view in the window from any view point and view direction.
  - *Zoom in/Zoom out:* Zooms in/out view by approximately 50%.
  - *Projection type:* Opens a projection type selection dialog. Parallel and perspective projections can be selected. For the perspective one, the *Auto set* option can be used to automatically set an optimal viewing distance (computed on the basis of the viewed scene's extent)
  - *Restore defaults:* Restore the viewing defaults (black/white hidden surfaces, original viewpoint, parallel projection, etc).
- *Light menu*
  - *Light window:* Opens a light direction positioning window similar to the viewpoint-positioning window.
  - *Light colors:* Opens a dialog that offers light ambient and light color setting facilities. The user can pick any available light color light intensity for the directional and the ambient light.
- *Options menu*
  - *Set aspect:* Opens a rendering mode selection dialog. Rendering modes range from wireframe (very fast) to the Phong RGB mode. Polygon normals are used for the flat-shaded mode light computations while vertex normals are used for Gouraud and Phong rendering modes.

- *Show information:* Shows some useful information about the world and the viewer's state (number of polygons, points, vertices, surfaces, memory used by the 3D scene and by the z buffer, current color scheme, etc).
- *Set dither:* This option is enabled only in palette-based color schemes (16 and 256 color modes). It allows the user to select between two dithering modes. Both dithering modes are applied on the fly while the image is rendered.
- *Hidden surfaces:* The user can select between a depth sort hidden surface algorithm and a z buffer algorithm. The depth sort algorithm is considerably faster than the z buffer but may not produce the correct results for some cases (it assumes that there exists a total depth ordering of the polygons in the scene). The z buffer option uses a normal per pixel z buffer technique, enhanced with front to back polygon preordering. Moreover, the user can choose the height and width sizes of the z buffer he wants to use. This feature allows using a z buffer *smaller* than the size of the image to render, in which case rendering will 'tile' the whole image with the given z buffer and render all polygons in each tile at a time. The speed of such an algorithm is similar to the one achieved by using a z buffer as large as the whole image while the memory requirements can be much smaller. In the extreme case, the algorithm performs like a scan line z buffer. The user can choose if the z buffer's memory is to be 'locked' into the system or can be discarded if he switches to depth sort. If the memory is locked, it will be kept allocated even if the user switches to depth sort. This prevents an eventual memory allocation failure that might occur if the user relinquishes a very large z buffer and later attempts to allocate the same z buffer and if the system has fragmented the available memory. The option can be useful on systems having less than 4 MB of RAM.

For the strict purpose of viewing results of the radiosity renderer, color or monochrome Gouraud shading should be used. The radiosity renderer outputs its results as an unstructured set of polygons having vertex colors (the radiant exitances of the vertices) and a polygon color. WINVIEW will detect that its input file contains vertex colors so it will directly use these colors for Gouraud shading the polygons without performing any lighting. The world can be viewed alternatively as opaque polygons, in which case an overview of the element subdivision over the whole scene is obtained. The scene used as input for the radiosity renderer can be viewed as flat shaded polygons (although the user can select Gouraud or Phong lighting as well and cast a directional light over the scene).

6.3 BUILD_3D: A 3D Modelling Application

BUILD_3D is a modelling application taking as input 3D object description files written in a structured language and producing structured polygonal descriptions of these objects in the 3D file format (see Appendix C for the 3D file format). Tridimensional objects having material properties and colors can be easily created with the given language and then translated into 3D files (that contain a polygonal approximation of the objects) and used further on as inputs for the radiosity renderer. All the 3D scenes rendered were initially created using BUILD_3D.

The BUILD_3D modelling language uses objects as the primitive items in describing a scene. There are 7 types of objects implemented in the language: rotation objects, sheet objects, tubular (swept) objects, file objects, polygon objects, two-faced polygon objects and composite objects. The language is entirely a declarative language: objects are declared and then 'instantiated' in order to create a scene. The output scene will contain all the instantiated objects. Here follows the description of the objects:

- *Rotation object:* generated by rotating a given plane crossection around the z axis. The plane crossection is in fact a 2D polyline in the xy plane. Is a particular case of the tubular object described further on. Quadrilateral polygons will be generated to approximate the rotation surface.

  Language syntax :

  ROTATION_OBJ

  angle1

  angle2

  nsections

  closure

  nelems

  $\mathbf{x_1}$ $\mathbf{y_1}$
  ...
  $\mathbf{x_{nelems}}$ $\mathbf{y_{nelems}}$

  Figure 6.1: Rotation object syntax

  - $angle_1$: start angle from which the crossection starts being rotated.
  - $angle_2$: end angle to which the crossection ends rotation (both angles are in degrees, rotation is done in counterclockwise sense. They are real values)
  - $nsections$: an integer denoting how many rotation sections to generate between angle1, angle2.
  - $closure$: 'USE_LIDS' or 'NO_LIDS' . Tells if there are to be lids at the start and end of the rotation surface. A lid is a polygon identical to the 2D crossection of the rotation object.
  - $nelems$: an integer denoting the number of points on the plane crossection.
  - $x_1 y_1 .. x_{nelems} y_{nelems}$: the 2D coordinates of the crossection points, real numbers. It is important to notice that these points have to be given in clockwise order by the one who creates the file. In other words: the sense of the crossection (ran from the first to its last point) is always identical to the sense of rotation from angle1 to angle2; this can be checked using the 'right-hand rule'. If the user desires to divide the rotation surface created by the polyline into several 'regions' that will not be smoothed into each other by Gouraud shading, then he can insert in the polyline description the symbol '—-' between the points i and $i+1$ where a discontinuity on the rotation surface is desired.

- *Sheet object:* similar to the graph of $z = f(x, y)$. It represents a 3D surface meshed by quadrilateral polygons.

Language syntax:

SHEET_OBJ

m
n

$\mathbf{x_1}$ $\mathbf{y_1}$ $\mathbf{z_1}$
...
$\mathbf{x_{m*n}}$ $\mathbf{y_{m*n}}$ $\mathbf{z_{m*n}}$

Figure 6.2: Sheet object syntax

- m : number of 'rows' of the mesh, integer value.
- n : number of 'columns' of the mesh, integer value.
- $x_1 y_1 z_1 .. x_{m \cdot n} y_{m \cdot n} z_{m \cdot n}$: the 3D coordinates of the points on the sheet, real numbers. These points must be given 'row by row' such that, for example, points 1,2,$n+2$,$n+1$, in this order, will generate the first polygon. The points are coupled four-by-four in this way, therefore generating the surface from quadrilaterals.

- *Tubular (swept) object:* generated by translating a given plane crossection along a given 3D curve (directrice) and rotating it along the translation direction while translating. The crossection and directrice are respectively a plane polyline and a 3D polyline. The swept surface is generated by connecting the points on consecutive translated crossections.

  Language syntax:

TUBE_OBJ

angle1
angle2

joining

closure

ndir

$\mathbf{x_1}$ $\mathbf{y_1}$ $\mathbf{z_1}$
...
$\mathbf{x_{ndir}}$ $\mathbf{y_{ndir}}$ $\mathbf{z_{ndir}}$

nelems

$\mathbf{x_1}$ $\mathbf{y_1}$
...
$\mathbf{x_{nelems}}$ $\mathbf{y_{nelems}}$

Figure 6.3: Tubular (swept) object syntax

- $angle_1$: start angle from which the crossection starts rotation.

- $angle_2$: end angle of crossection rotation. The last section of the tube, i.e. the one corresponding to the last point on the directrice, is hence rotated with angle2. (both angles are in degrees, rotation is done in counterclockwise sense. The angles are real values).
- *joining* : 'JOIN' or 'NO_JOIN'. Tells if the start and end of the tube are to be 'joined'. This implies that closure equals NO_LIDS and that the directrice polyline is a closed- polyline (last point equals the first point). This option is useful for generating 'closed' swept objects (like a torus, for example).
- *closure* : 'USE_LIDS' or 'NO_LIDS'. Tells if there are to be lids at the start and the end of the tube. The semantics of the lids is the same as for rotation objects.
- *ndir* : an integer denoting the number points on the directrice.
- $x_1 y_1 z_1 .. x_{ndir} y_{ndir} z_{ndir}$: the 3D coordinates of the directrice, real numbers. For closed directrices: $x_{ndir} = x_1$, $y_{ndir} = y_1$, $z_{ndir} = z_1$.
- *nelems* : an integer denoting the number of points in the plane crossection
- $x_1 y_1 .. x_{nelems} y_{nelems}$: the 2D coordinates of the crossection points, real numbers. These points have to be given in clockwise order when creating the file: the sense of the crossection (ran from the first to its last point) is always identical to the sense of translation on directrice, also ran from the first to its last point; check using the 'right-hand rule'.

- *File object:* includes another object in the current file (similar to a C *#include* statement). The file to be included must contain objects in the same modelling language.

  Language syntax: where *filename* is the name of the file we want to add to the

  FILE_OBJ filename

  Figure 6.4: File object syntax

  description of the current object. The added file can contain anything a modelling language file contains, even another FILE_OBJ ('nested' definitions of objects are allowed).

- *Polygon object:* a planar convex/concave *n* vertices polygon. This is the simplest object, and is provided as a gateway for the 3D format to support any type of polygon-based modelling.

  Language syntax:

  POLY_OBJ

  nverts

  $\mathbf{x_1\ y_1\ z_1}$
  ...
  $\mathbf{x_{nverts}\ y_{nverts}\ z_{nverts}}$

  Figure 6.5: Polygon object syntax

  - *nverts* : number of vertices of the polygon.
  - $x_1 y_1 z_1 .. x_{nverts} y_{nverts} z_{nverts}$: vertex coordinates in anticlockwise order.

- *Two-faced polygon object:* a collection of two plane-parallel polygons. This object can be used to model a two-faced polygon. The two polygons are separated by a very small distance measured along the (common) normal vector direction of the two polygons.

  Language syntax:

```
POLY2_OBJ

nverts

x1 y1 z1
...
xnverts ynverts znverts
```

Figure 6.6: Two-sided polygon object syntax

  - *nverts* : number of vertices of the polygon.
  - $x_1 y_1 z_1 .. x_{nverts} y_{nverts} z_{nverts}$: coordinates of the vertices, in anticlockwise order. These are the coordinates of one of the two polygons. The other one has its vertices in exactly the converse order, to ensure an opposite normal.

- *Composite object:* a collection of other objects. Composite objects allow the design of a structured scene, where objects are described as collection of subobjects. Transformations can be used to achieve complex modelling.

  Language syntax for definition of a composite object: where *objname* is the name

```
DEFINE objname
{

  ...

}
```

Figure 6.7: Composite object syntax

under which the composite object will be known from this point further in the current file. Any kind of language constructs are allowed between the braces (object instantiations, other DEFINEs, etc).

The braces pair of such an object definition define a scope. Any kind of declaration done in a scope is valid only within that scope (and, of course, in the eventual nested scopes). Object names (introduced by DEFINEs) can be hidden by local names in scopes. If an object name is defined twice in the same scope, the first definition will be used until the place the second definition appears, then the second definition will be used until the end of that scope. A global name is a name defined at a file level (scope). A file scope ends at the end of the current file. There is no limit for the scope nesting level. Moreover, BUILD_3D is designed in such a way that it can run with high speed and a very small amount of memory.

After an object *name* is defined, it can be instantiated with:

DEF_OBJ name

Figure 6.8: Instantiation of an object

6.3.1 Transformations

Any object may have a set of 3D transformations specified right after its keyword (ROTATION_OBJ,FILE_OBJ,DEF_OBJ,etc). This transformations will be applied to the object after building it, in the order they appear listed in the file.

There are three types of transformations:

- Translation: *TRANSLATE x,y,z* translates the object with the real given values x, y, z along the axes of the 3D system it was defined in.
- Rotation: *ROTATE* α_x, α_y, α_z rotates the object with the real angles α_x, α_y, α_z (in degrees) counterclockwise around the axes of the 3D system it was defined in.
- Scale: *SCALE* s_x, s_y, s_z scales the object with the real factors s_x, s_y, s_z along the axes of the 3D system it was defined in.

An object can have any number of such transforms listed in any order desired by the user.

6.3.2 Colors

An object may have a color, exactly as it may have a set of transforms. The color must be specified right after the transforms, if any. If an object doesn't have a specified color, it takes a default gray color. If a color is specified for a DEFINEd user-defined object, then all the objects inside the composite object that do not have the color explicitly specified will take that color.

Language syntax for color definition: where r, g, b are 3 floating point numbers be-

RGBCOLOR r g b

Figure 6.9: Color definition syntax

tween 0.0 and 1.0, specifying the amounts of red, green and blue that define the object's color.

6.3.3 Material properties

An object may have material properties defined as it has a color. The material properties consist in a diffuse reflectivity coefficient, a specular reflectivity coefficient and a specular index value (the parameters of the Phong lighting model). These values must be specified right after the colors, if any. If an object doesn't have a set of material properties specified, it takes some default values. All objects inside a composite object that do not have the material properties explicitly specified will take the composed object's ones.

Language syntax for material properties definition: where kd, ks are floating point numbers in the [0..1] range, giving the object's diffuse and specular properties and $specn$ is a floating point value greater than zero.

PHONGMODEL kd ks specn

Figure 6.10: Material properties definition syntax

6.3.4 Comments

The modelling language may contain comments for human readers. They start with an asterisk '*'. Everything after it until the end of line is ignored. They can appear anywhere in the file (they resemble C++'s '//' comments).

Appendix A

The Radiosity Renderer User Guide

This section is a user guide to the current implementation of the radiosity renderer. In order to use the renderer, an *input scene* has to be created, containing the environment to be rendered as well as the description of the light sources. This is done by generating a TAKES file that will contain a polygonal description of the scene. The light sources are described as polygons having at least one RGB color component greater than one. The color of these polygons will be read by the renderer, divided by two and the result will be interpreted as the color of a lightsource geometrically described by that polygon. For example, in order to describe a light source with RGB color $(0.2, 0.3, 1.0)$, we shall create a polygon in the TAKES file with RGB color $(0.4, 0.6, 2.0)$. Such an input TAKES file can be either created by hand or it can be automatically generated out of a modelling language description file (by means of a 3D modelling application). Such a modelling application, BUILD_3D, together with a modelling language, is described in section 6.3.

The input world can have practically any number of polygons of any size. It should be however avoided to have very large and very small polygons in the same scene for accuracy reasons (the geometrical computations performed by the radiosity renderer use a tolerance that is determined as a percentage of the world's bounding box).

The radiosity renderer will read the world input file and render according to some settings. These settings can be given either directly (the renderer will prompt the user for them) or in a configuration file *xff.cfg* that has to be in the same directory as the one the renderer is started from. For any required setting that is not found in this configuration file, the renderer will prompt the user. Here are the settings used by the radiosity renderer together with the configuration file syntax to be used for them:

- *Input file:* The TAKES input file containing the scene's geometry and light source description is to be given.

  Configuration file syntax: *INFILE input_file_name*

- *Output file:* The TAKES output file containing the rendered scene. There are three types of output files (see section 3.4.2 for an overview of the output types of the renderer): flat-shaded output, vertex-exitance output and hit-based output. The renderer will select the output type according to the extension of the output file name given by the user: flat-shaded for *"taq"* extension, vertex-intensities

for *"tak"* extension, hit-based for *"hit"* extension. The vertex-intensities output is the most usual one, since it suits a Gouraud shading viewing.

Configuration file syntax: *OUTFILE input_file_name*

- *Number of patches per polygon:* The number of patches per scene polygon. If the user desires to explicitly specify the number of patches per polygon, then all polygons in the scene will be meshed into this given patch number and, since polygons' sizes can be very different, the resulting patch sizes might also be quite different. If the user desires that all patches in the scene have (almost) the same patch size, then this setting has to be left out and next setting is to be used. In most of the cases, the user will indeed desire an uniform patch size over the whole scene, so he will leave out this setting.

  Configuration file syntax: *NPATCHPERPOLY num_patches*

  where num_patches is an integer.

- *Patch area:* The renderer will attempt to mesh all polygons in the scene in patches having this area (differences from this area will appear if there are long, thin polygons in the scene but the meshing algorithm will never create patches *larger* than the user specified patch area). This option is mutually exclusive with the above option, i.e. the user will either specify a number of patches per polygon (above setting) or a patch area (this setting) but not both. In case the user has no idea of what patch size to use, he can request an evaluation of the patch size based on the total number of patches in the scene he wishes to have (this number is related to the computing power he wishes to spend). The renderer will compute an average patch size dividing the total area of the scene (the sum of all polygons' areas) by the total number of patches given by the user. This computed patch size can be used as a first guess to the patch size to be used on a certain scene.

  Configuration file syntax: *PATCHAREA area*

  where area is a floating point number.

- *Patch to subpatch subdivision mode:* When loading a scene, the renderer will firstly mesh all its polygons into patches (see section 3.2.2). After this, all patches are to be meshed into subpatches (called also elements). This is done by firstly generating a subpatch equal to the patch for each patch and then recursively subdividing this subpatch into smaller subpatches. There are two ways of doing this: either by subdividing all subpatches over a polygon up to the *same level* (this is called *uniform* subdivision) or by individually meshing each subpatch up to the point it will be smaller or equal than a user-specified subpatch size. If the patches generated by the polygon to patch meshing have approximately the same size, the two methods will produce the same result. If the patches of a polygon have different sizes, this nonuniformity will be preserved if *uniform* patch to subpatch subdivision is used, while the *individual* subdivision will mesh each patch individually and stop as soon as the resulting subpatches are smaller than the desired size. In most of the cases, the user will desire to use the *uniform* subdivision however.

  Configuration file syntax: *SUBDIV_METHOD method_name*

  where the method_name can be either *UNIFORM* or *INDIVIDUAL*.

- *Element area:* The area of an element (subpatch). After the renderer has meshed the scene into patches, the second substructuring level meshes all these patches

into elements (subpatches) (see section 3.2.3). The patch to element subdivision (either uniform or individual) will stop when the resulting subpatches are smaller or equal than the value specified by the user by this setting.

Configuration file syntax: *SUBPATCHAREA area*

where area is a floating point number.

- *Minimum subpatch area:* If adaptive subdivision (see section 3.4) is used, the renderer will attempt to subdivide subpatches (using some subdivision criterion) during the flux shooting process. There has to be a limit to this process, otherwise there will be too many subpatches generated (there might be even an infinite subdivision loop if there's no minimum subpatch area threshold). This setting imposes a minimum subpatch area: the adaptive subdivision will stop when the size of a subpatch is smaller than this given value, even if the subdivision criterion decides that the subpatch must be subdivided. If the user has no idea which should be this minimum subpatch area value, he can leave this setting out. A default value will be used (1/20 of the subpatch area in the current implementation).

  Configuration file syntax: *MINSUBPAREA area*

  where area is a floating point number.

- *Minimum subpatch area (close object buffer):* When the close object buffer is used (see section 4), subdivision will proceed due to detection of close objects potentially casting sharp shadows. Sometimes we should like to generate a *finer* subpatch mesh in the areas where subdivision is triggered by this buffer than in the areas where subdivision is triggered by the normal solution-based adaptive subdivision method, since the former areas will potentially have sharp shadows over them. Therefore it should be possible to establish different stop thresholds for the solution-based subdivision and for the close objects buffer-based subdivision. This setting gives the minimum subpatch area generated by the close objects buffer subdivision: the user can hence select a smaller minimum area than for the other subdivision method (a good value is, for example, a quarter or a tenth of the minimum subpatch area used for the solution-based subdivision). In the case this setting is left out, the renderer will use the same area as for the solution-based subdivision.

  Configuration file syntax: *MINSUBPAREA_SB area*

  where area is a floating point number.

- *Subdivision criterion:* The subdivision criterion used for the solution-based adaptive subpatch subdivision (see section 3.4). This can be the delta or gradient criterion. If this setting is left out, gradient criterion is used as a default. In most cases, the gradient criterion performs better than the delta criterion (generates less elements by detecting more accurately the areas to be refined).

  Configuration file syntax: *SUBDIV_CRITERION criterion*

  where criterion is *DELTA* or *GRADIENT*.

- *Light patches area:* The user can assign a different area for the light patches (see section 3.3). The light polygons can therefore be meshed in patches smaller than the usual patches in the scene. A good value to use is a light patch area equal to the subpatch area. Configuration file syntax: *LIGHTPATCHAREA area*

  where area is a floating point number.

- *Minimum and maximum exitances:* The user has to specify the radiant exitance range he wishes to view. Out of the full range of computed radiant exitances, there is typically a subrange that the user will desire to map to the displayable [0..1] intensity range by a normalization transformation (see section 3.4.2). It should be theoretically possible to skip passing this range at the beginning of the rendering (and perform all normalization as a postprocessing phase). However, exitance normalization is performed also *during* the radiosity rendering process, in order to compute exitance gradients *over the final displayed exitance range*. Therefore the renderer has to know the exitance range the user wishes to display in order to perform this normalization (see section 3.6). This range will be used *only* for the normalization done for adaptive subdivision computations. The usual way of getting it is by firstly deciding on an exitance range to view (for example [0..5]) and then using it as input to the renderer as well.

  Configuration file syntax:

  - *I_MIN min* where min is a floating point number.
  - *I_MAX max* where max is a floating point number.

- *Delta/gradient criteria thresholds:* The user can input thresholds for the delta and/or gradient criteria used for adaptive receiver subdivision. If the delta or gradient values (see section 3.4) of an element are larger than the user specified threshold, then that element is subdivided otherwise it is left as such. The delta threshold refers to the *visible* [0..1] exitance range, therefore a reasonable value should be around 0.1 or 0.2. The gradient threshold is essentially a tangent so its range is determined by the element sizes ranges (it is therefore scene-dependent). For example, a value of 0.1 or 0.2 will be reasonable for a scene where the average patch area is 100 (the average patch edge length will be 10, so the maximum exitance variation allowed is 0.1 per unit length for a gradient threshold of 0.1). Care has to be taken when tuning these threshold values since having them too low can generate a very large amount of elements out of adaptive subdivision.

  Configuration file syntax:

  - *GRAD_EPS eps* where eps is a floating point number.
  - *DELTA_EPS eps* where eps is a floating point number in the [0..1] range.

- *Close objects buffer usage:* The renderer can use or not the close objects buffer. In some cases, not using the buffer can increase noticeably the rendering speed and the image quality can be still very good (especially if there aren't shadows cast by 'close' objects).

  Configuration file syntax: *USE_SB usage*

  where usage is *TRUE* or *FALSE*.

- *Output element midpoints:* The user may choose to use or not midpoints when generating the output polygons that are to be Gouraud shaded (see section 3.4.2). This option is meaningful only in case the output type is polygons with vertex intensities. In most cases, there is not a sensible difference between using or not the midpoints when rendering the final image (differences can appear when the element mesh is not smoothly graded, when the user might wish not to use these midpoints).

Configuration file syntax: *OUTPUT_MIDPOINTS option*

where option is *TRUE* or *FALSE*.

- *Number of iterations:* The number of shooting iterations for the progressive refinement. The quality increases with this number but the rendering time is proportional with it as well. For most scenes, 100 iterations suffice (although highly perceivable color bleeding appears sometimes after 150..200 iterations).

  Configuration file syntax: *NITERATIONS iter*

  where iter is an integer value.

A configuration file should end with the keyword *END*.

Here is a configuration file example: After running the renderer, an output file *TAK/TAQ/HIT*

```
OUTFILE                 tt.tak
SUBDIV_METHOD           UNIFORM
I_MIN           0
I_MAX           4
DELTA_EPS               0.2
GRAD_EPS                0.25
USE_SB                  TRUE
OUTPUT_MIDPOINTS  TRUE

END
```

Figure A.1: A simple configuration file *xff.cfg*

is created. The user has to normalize the exitances of the output and then he can directly view a *TAK* or *TAQ* file with a TAKES file 3D viewer. Such a viewer, WINVIEW, was implemented for the MS Windows platform (see section 6.2). A very simple equivalent for viewing on an X platform was implemented as well, using the OpenGL graphics library. Hit-based *HIT* files should first be converted to vertex intensities *TAK* files and then normalize the exitances of the output and view the final result.

The hits to vertex intensities conversion process is described in section 5. The exitance normalization process is performed by an auxiliary filter-like program (see section 3.4.2 for a presentation of the normalization process).

The current version of the radiosity renderer has been implemented in C++ and is a platform-independent application (all input and output is done either via files or the standard input and output). Valuable reference sources for implementation-related details were found in [Foley et al., 1990], [Ashdown, 1994] and [Watt, 1990] as well as the *Graphics Gems* series.

Appendix B

The 3D File Format

The *3D files* are used to describe a structured tridimensional environment. The main purpose of 3D files is to be used as inputs for the 3D viewer WINVIEW (see section 6.2) but they can be used also as scene descriptions for other purposes. A 3D file contains a list of objects. Each object is individually described by its color, its material properties and its polygons and points. Each object has a list of points consisting in a sequence of coordinates of points in the 3D space. Besides this, it has a list of polygons. A polygon has a list of indexes in the point list (its vertices refer to points in the point list of the object it belongs to) and a surface index. Polygons belonging to the same surface are regarded as sharing vertex attributes (like vertex normals and vertex colors, for Gouraud shading purposes, for example). The surfaces of an object are numbered starting from zero. The point indices used when describing a polygon refer to the object's point list where points are numbered starting from zero. All polygons of an object share the same material properties and colors. An object's color is described by three floating point numbers between zero and one (the RGB color components). The material properties are described by three parameters of the Phong illumination model: the diffuse illumination coefficient K_d, the specular illumination coefficient K_s and the specular index n:

3D_file **:**	*(object)+*
object **:**	*rgb_color*
	material_prop
	POINTS *(point)+*
	POLYGONS *(polygon)+*
rgb_color :	**RGBCOLOR** r g b
material_prop:	**PHONG_MODEL** kd ks n
point :	*x y z*
polygon :	*surf_n nverts* $v_1 ... v_{nverts}$

Figure B.1: Grammar for the 3D files

In the above grammar description for the 3D files '+' stands for one or more occurrences of an item. r, g, b are color components in the [0..1] range, x, y, z are floating point coordinates (not restricted to a certain range), kd, ks are in the range [0..1], n is an integer greater than 0 (the so called Phong specular index), $surf\_n$ is the polygon's surface number (surfaces are numbered starting from zero), $nverts$ is the number of vertices of a polygon and $v_1..v_{nverts}$ are indices in the points list (first point in this list has index zero).

Appendix C

The TAKES File Format

The *TAKES file format* was used both as an input file format for the 3D viewer WINVIEW as well as an input and output file format for the radiosity renderer. It consists of an unstructured list of independently described polygons. The subset of the TAKES file format facilities that were used by the above applications will be described.

A polygon can have an optional color and a set of vertices given by their 3D coordinates. Besides its coordinates, a vertex may have an optional 3D vertex normal and a vertex color. Both polygon and vertex colors are RGB triplets of floating point numbers in the [0..1] range. Figure C.1 presents a grammar description of the TAKES file format.

takes_file :	*(polygon)+*
polygon :	*(vertex_prop)**
	*(color)**
	nverts $vertex_1$ ... $vertex_{nverts}$
vertex :	*x y z (nx ny nz)* (r g b)**
vertex prop :	**vertex** *vert_type*
vert_type :	**c** | **n** | **nc**
color :	**color** *r g b*

Figure C.1: Grammar for the TAKES files

In the grammar, '+' stands for one or more occurrences and '*' for zero or one occurrences of an item while '—' means one of the several listed possible items. If a polygon has only vertex coordinates as vertex attributes, then there's no *vertex_prop* item in the file. If the vertex has colors, then *vertex_prop* is **vertex c**. If it has a vertex normal, then *vertex_prop* is **vertex n**. If it has both attributes, then *vertex-prop* is **vertex nc**. If a polygon has a given RGB color, then the **color** keyword appears followed by the *r*, *g*, *b* color values. *nverts* is the number of polygon vertices (their description comes right after this keyword). *x*, *y*, *z*, *nx*, *ny*, *nz* are floating point numbers (the last three should describe the normal vector. For WINVIEW, these normals should be normalized to unit length).

Appendix D

Plates

Bibliography

[Aguas and Muller, 1993] Aguas, M. N. P. and Muller, S. (1993). *Mesh Redistribution in Radiosity*. Proceedings Fourth Eurographics Workshop on Rendering, Toronto.

[Ashdown, 1994] Ashdown, I. (1994). *Radiosity: a programmer's perspective*. Wiley.

[Baum et al., 1991] Baum, D. R., Mann, S., Smith, K. P., and Winget, J. M. (1991). *Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions*. Computer Graphics (SIGGRAPH '91 Proceedings) vol 25 no 4.

[Beran-Koehn and Pavicic, 1992] Beran-Koehn, J. C. and Pavicic, M. J. (1992). *Delta Form Factor Calculations for the Cubic Tetrahedral Algorithm*. in Kirk 92: Graphics Gems III.

[Cohen and Greenberg, 1985] Cohen, M. F. and Greenberg, D. P. (1985). *The Hemi-Cube: A Radiosity Solution for Complex Environments*. Computer Graphics (SIGGRAPH '95 Proceedings vol 19 no 3).

[Cohen et al., 1986] Cohen, M. F., Greenberg, D. P., Immel, D. S., and Brock, P. J. (1986). *An Efficient Radiosity Approach for Realistic Image Synthesis*. IEEE Computer Graphics and Applications.

[Cohen and Wallace, 1993] Cohen, M. F. and Wallace, J. R. (1993). *Radiosity and Realistic Image Synthesis*. Academic Press, San Diego CA.

[Cohen et al., 1988] Cohen, M. F., Wallace, J. R., E., C. S., and Greenberg, D. P. (1988). *A Progressive Refinement Approach to Fast Radiosity Image Generation*. Computer Graphics (SIGGRAPH '95 Proceedings vol 22 no 4).

[Foley et al., 1990] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1990). *Computer Graphics: Principles and Practice (2nd ed.)*. Addison-Wesley, Reading, MA.

[Hanrahan et al., 1991] Hanrahan, P., Salzman, D., and Aupperle, L. (1991). *A Rapid Hierarchical Radiosity Algorithm*. Computer Graphics (SIGGRAPH '91 Proceedings vol 25 no 4).

[Kajiya et al., 1986] Kajiya, J. T., Kalos, M. H., and Whitlock, P. A. (1986). *Monte Carlo Methods*. Wiley, New York.

[Kok, 1993] Kok, A. J. F. (1993). *Grouping Patches in Progressive Radiosity*. Proceedings Fourth Eurographics Workshop on Rendering, Paris.

[Malley, 1988] Malley, T. J. (1988). *A Shading Method for Computer Generated Images*. Master's thesis, Department of Computing Sicence, University of Utah.

[Maxwell et al., 1986] Maxwell, G. M., Bailey, M. J., and Goldschmidt, V. W. (1986). *Calculations of the Radiation Configuration Factor using Ray Casting*. Computer-Aided Design 18(7).

[Moon and Spencer, 1981] Moon, P. and Spencer, D. E. (1981). *The Photic Field*. MIT Press, Cambridge, MA.

[Murdoch, 1981] Murdoch, J. B. (1981). *Inverse Square Law Approximation of Illuminance*. Journal of the Illuminating Engineering Society.

[Nishita and Nakamae, 1985] Nishita, T. and Nakamae, E. (1985). *Continuous-Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection*. ACM SIGGRAPH '85 Proc. in Ian Ashdown 'Radiosity: A Programmer's Perspective',Wiley.

[Phong, 1975] Phong, B. T. (1975). *Illumination for Computer Generated Pictures*. ACM Communications, vol 18 no 6.

[Recker et al., 1990] Recker, R. J., George, D. W., and Greenberg, D. P. (1990). *Acceleration Techniques for Progressive Refinement Radiosity*. Computer Graphics 24(2), Symposium on Interactive 3D Graphics.

[RP-16-1986, 1986] RP-16-1986, A. (1986). *Nomenclature and Definitions for Illuminating Engineering*. New York, Illuminating Engineering Society of North America.

[Rushmeier et al., 1993] Rushmeier, H., Patterson, C., and Veerasamy, A. (1993). *Geometric Simplification for Indirect Illumination Calculations*. Proceedings on Graphics Interface '93, Toronto.

[Siegel and Howell, 1992] Siegel, R. and Howell, J. R. (1992). *Thermal Radiation Heat Transfer*. Hemisphere Publishing, Washington DC.

[Sillion and Puech, 1989] Sillion, F. and Puech, C. (1989). *A General Two-Pass Method Integrating Specular and Diffuse Reflection*. Computer Graphics 23(3), ACM SIGGRAPH '89 Proc.

[van Liere, 1991] van Liere, R. (1991). *Divide and Conquer Radiosity*. Proceedings Second Eurographics Workshop on Rendering, Barcelona.

[Wallace et al., 1987] Wallace, J. R., Cohen, M. F., and Greenberg, D. P. (1987). *A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods*. ACM SIGGRAPH '87 vol 21 no 4.

[Watt, 1990] Watt, A. (1990). *Fundamentals of Three-Dimensional Computer Graphics*. Addison-Wesley, Reading, MA.

[Xu et al., 1989] Xu, H., Peng, Q. S., and Liang, Y. D. (1989). *Accelerated Radiosity Method for Complex Environments*. Eurographics '89, Elsevier, Amsterdam.