Interactive dynamic objects in a virtual light field

Year: 2005

Author: Jamie Wither

Supervisor: Mel Slater

This report is submitted as part requirement for the MSc Degree in Vision, Imaging and Virtual Environments at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Acknowledgements

I would like to thank Pankaj Khanna and Jesper Mortensen for the many helpful discussions we had regarding their implementation of the virtual light field. I would also like to thank Mel Slater for his supervision and guidance while working on this project.

Contents

1 Introduction

Computer graphics is a young field concerned with visually representing aspects of the real world or human imagination to human users. Since the early 1960s computer technology has progressed from representing simple monochrome points, lines and shapes on a single 2D display panel to complex 3D objects in millions of colours, projected onto multiple large screens and viewed in stereo through head mounted eye pieces. As these visual display systems have progressed in complexity so have the methods of creating the data to display on them. It is now possible to synthesize static 2D images representing a view of the real world which is indistinguishable from a photograph of the real view. Such techniques are useful in visualisation applications where a user wishes to study the visual effects of changes they couldn't make in the real world without a great expenditure of time or money, for example architectural visualisation or prototyping complex mechanical products such as automobiles. More useful still would be the ability to instantly move the virtual camera around such virtual scenes and view it interactively from many different angles. This too is currently possible – but under a large set of constraints (the number of constraints is being reduced as the field progresses).

Light is the starting point for human visual perception and global illumination is the field concerned with modelling the way light propagates in the real world. These models approximate the way light would be distributed in the real world for a given scene description and allow the recreation of existing views or the creation of completely imagined scenes as they would appear in the real world.

This project extends a particular model for approximating global illumination, a Virtual Light Field (VLF)[2]. This model is well suited for interactive viewing of a static scene. This project extends the existing implementation to allow the addition of a dynamic object to the scene. This object can be moved within the scene in real-time and will be illuminated and change the illumination on existing objects within the scene according to the light distribution contained within the model. Where the object cannot be illuminated realistically at real-time frame rates approximations are made to maintain interactivity.

1.1 Global illumination

The name *global illumination* (GI) indicates that models of this type take into account light in a scene due to object interreflections in some way. Global illumination solutions include light arriving at an object directly from primary light emitters such as light bulbs (direct light) and also light arriving via reflections of light from other objects in the scene (indirect light). It is the recursive nature of the problem that makes it complex to solve. A *local illumination* solution only considers the direct effects of light emitters on objects in the scene and does not consider the light hitting objects due to reflections from other objects. This can be considered in more detail after defining a vocabulary for discussing the field. This starts with a discussion of the *radiance equation*. For further detail see [3] and [10]. The discussion below proceeds along the lines of Chapter 3 from [3].

1.2 The radiance equation

Light energy is carried by particles called *photons* which travel in a straight line with speed c ($3*10^8 ms^{-1}$) in a vacuum. The amount of energy flowing through a surface per unit time is called flux. It is measured in units of Watts (W) and denoted by the

symbol ϕ. We can think of the flux as the flow of photons per unit time. The energy carried by a photon is related to the frequency of its related light wave by the Einstein relation E=hf where h is Planck's constant. In perceptual terms the frequency corresponds to the colour of light. However for the purposes of our discussion we will consider light of just one frequency and drop the dependency from the equations.

Consider the total flux of photons in an arbitrary volume where the perceived levels of illumination are constant. Photons may travel through the volume without interaction (streaming). They may be reflected from a surface within the volume (out-scattering). They may be absorbed by materials in the volume. Photons may enter the volume from outside (in-scattering) or they may be emitted from surfaces within the volume itself. If we apply the law of conservation of energy (which is a constraint in the real world and thus applies to light) then we realise that the total energy leaving the volume must equal the energy entering the volume plus any energy emitted within the volume minus any energy absorbed within the volume. This can be formulated as:

Emission + in-scattering = streaming + out-scattering + absorption ***(Eq 1)***

Each of the terms in the equation above can be represented as integrals of probabilities of the various events occurring over the appropriate domains of volume, set of directions of interest and surface area of the volume. The common term to each integral is $\phi(p,\omega)$, the flux at point p (in the volume) in direction ω. So if we know $\phi(p,\omega)$ we can solve this integral and we have a complete description of the light in the scene (see sect 3.2 of [3] for further detail). We can then use this description to render an image of the scene from any viewpoint. Finding a set of approximations for the values of $\phi(p,\omega)$ is what global illumination solutions try to achieve. In order to simplify determining the solution a number of assumptions are made:

1. Wavelength independence. Photons of different wavelengths do not interact (i.e. no fluorescence).
2. Time invariance. Any solution remains valid as long as the content of the scene does not change. (i.e. no phosphorescence)
3. No participating media. We assume light is travelling in a vacuum. This restricts light interactions to occur only at object surfaces. However some advanced solutions allow the simulation of participating media.
4. Object materials are isotropic. The relationship between the incident and reflected direction of light is the same over the whole surface.

In computer graphics it is not flux but a derivative of flux called *radiance* which is of most interest. Radiance is defined as *flux leaving a surface, per unit projected area of the surface, per unit solid angle of direction* and is denoted by the symbol L. The unit of radiance is $W/m^2/sr$. sr is a steradian, the unit of solid angle. Radiance is constant along the length of a ray of light. Closely related is the concept of *radiosity* which is defined as *the total power leaving a point on a surface, per unit area on the surface* and is denoted by the symbol *B*. It has units of W/m^2 and can be determined by integrating the radiance in all directions over the hemisphere above a point on a surface. From equation 1 after applying the above and additional simplifying assumptions it is possible to derive a new equation expressed in terms of radiance rather than flux. This equation is known as the radiance equation as is expressed as follows:

$$L(p,\omega) = L_e(p,\omega) + \int_{\Omega} f(p,\omega_i,\omega) L(p,\omega_i) \cos\theta_i d\omega_i$$

Where the terms are:

$L(p,\omega)$ the radiance leaving a point p in direction ω

$L_e(p,\omega)$ the radiance emitted by the surface at point p in direction ω

$\int_{\Omega}(...)d\omega_i$ an integral over the hemisphere of incoming directions above point p

$f(p,\omega_i,\omega)$ the bi-direction reflectance distribution function (BRDF). A term which relates the reflected radiance at point p in direction ω to the incoming radiance in direction ω_i

$L(p,\omega_i)$ the incoming radiance from direction ω_i

$\cos\theta_i$ the cosine of the angle between the surface normal and the incoming direction. Due to the projection of the surface area along the radiance direction.

The equation neatly expresses that radiance is the sum of two terms. The radiance emitted at that point (if any) $L(p,\omega)$ and the amount of radiance reflected at that point. The reflected radiance is computed as the sum over all rays arriving at point p multiplied by the BRDF of the surface. This is the second term on the right hand side of the radiance equation.

The term we wish to compute – the radiance L - appears on both sides of the equation, thus making it difficult to solve. In practice it can rarely be solved analytically and so numerical methods are used. There follows a summary of four approaches to solving the radiance equation and a discussion of their merits.

1.3 Classifying solutions

When we consider the process of creating a series of 2D images from the solution to the radiance equation we realise that for any particular image in the series we only require the solution for those rays which enter the eye. Such a solution is *view-dependent*. If the eye looks elsewhere in the scene then the equation needs to be resolved for the new set of eye rays. This can lead to a variable frame rate as some sets of rays will be more expensive to solve than others. *View-independent* solutions attempt to solve the equation for all surfaces in the scene in as many directions as possible ahead of time. So when an image is generated the solution for the required rays has been pre computed and can be looked up. This has the advantage of providing a constant frame rate – but requires a long

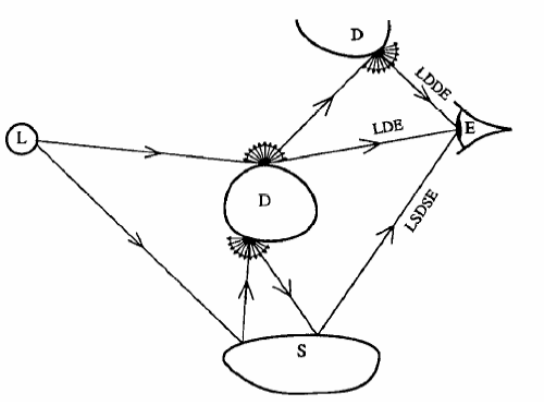

Figure 1 From Heckbert [1]. (L)ight. (S)pecular. (D)iffuse. (E)ye. See text.

resource intensive pre-computation step.

1.3.1 Heckbert notation

Heckbert [1] introduced a useful notation for discussing light paths in GI solutions, illustrated in Figure 1. The notation is a shorthand indicating events along the complete path of a particular ray. For example a ray which left a light source, bounced off two diffuse surfaces and then entered the eye would be LDDE (Figure 1). L – emitted from light, D – bounce off diffuse surface, S – bounce off specular surface. This can be combined with regular expression notation to provide expressions such as L(S|D)*E: a ray which leaves the light and encounters 0 or more specular or diffuse surfaces in any order before entering the eye. L(S|D)*E solutions are the ideal for GI solutions because they model both specular and diffuse rays. However they are also the most costly.

1.4 GI methods of solving the radiance equation

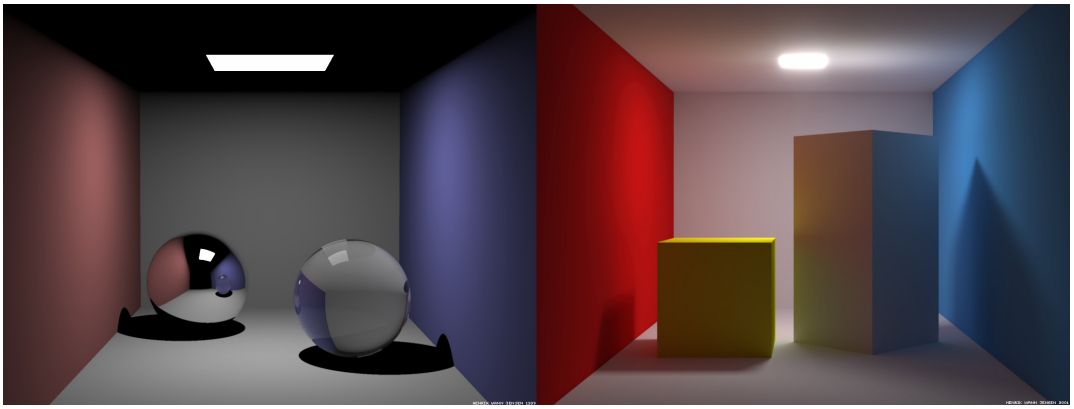

Figure 2 Ray tracing (left). Radiosity (right)[1].

1.4.1 Ray tracing

Ray tracing restricts the solution to consider only perfect specular surfaces (in a global sense) and point light sources. This reduces the number of rays the solution must calculate. In scenes with only one light source each point on each specular surface in the scene will have only one incoming (from the eye) and one outgoing (to the light) ray direction. Diffuse surfaces are only approximated using direct (no inter-reflections accounted for) light. Rays are traced from the eye into the scene and bounce recursively from specular surfaces until some pre-determined depth or until the contribution to the light perceived by the eye is negligible. This is an ES*DL solution and is view dependent. Ray traced scenes (Figure 2, left) have a classic 'computer generated' look about them as real scenes don't contain large numbers of perfect specular surfaces and they don't account for diffuse inter-reflective effects such as colour bleeding and soft shadows.

1.4.2 Radiosity

Radiosity can be considered as the complement to ray tracing in that it restricts the scene to containing perfect diffuse surfaces only and no specular surfaces (Figure 2, right). This eliminates all directional aspects of the radiance equation so it can be

[1] Images courtesy of Henrik Wann Jensen. (http://graphics.ucsd.edu/~henrik/images/cbox.html)

reformulated in terms of radiosity. A time consuming pre-computation step then begins. The geometry of the scene is subdivided into patches. The light emitters in the scene are given initial radiosity values. The method iterates through every patch in the scene emitting the radiosity energy of the patch to all the other patches in the scene until the solution converges. The energy is distributed among all the surfaces of the scene and doesn't depending on viewing position and so the method is view independent. This means that once the pre-computation is complete these scenes can be rendered in real time for walk through purposes. However with this solution in its basic form no part of the scene can be changed without re-computing the entire solution again. The light paths represented are LD*E.

There are a large number of variations to basic radiosity and the VLF approach (explained in detail later) can be considered to be one of them.

1.4.3 Path tracing

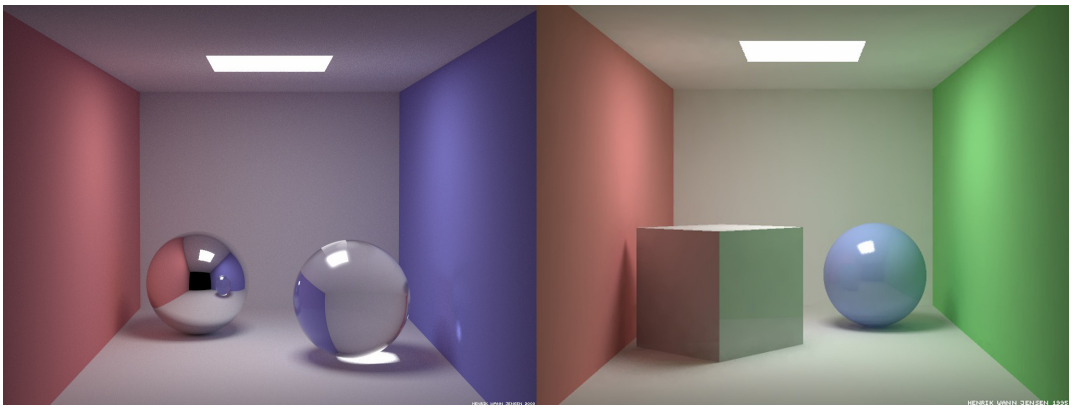

Figure 3 Path tracing (left). Photon Mapping (right)[2].

Path tracing is similar to ray tracing. Rays are traced into the scene from the eye, but instead of terminating on diffuse surfaces the BRDF function of each surface is sampled and suitable reflected direction is chosen to recurse along. Path tracing thus allows both diffuse and specular surfaces and mixtures of the two. It is a stochastic solution to the radiance equation and requires many primary rays to be traced per image pixel, to reduce sampling error to the point where the image isn't overly noisy. It is view-dependent and more expensive than ray tracing per image but produces much more realistic images than either ray tracing or radiosity alone. The paths traced are E(S|D)*L. The solution includes both soft shadows and colour bleeding (Figure 3, left).

1.4.4 Photon mapping

One of the most complex issues in radiosity solutions is how to efficiently divide the scene geometry into patches for best computation time and visual effect. The VLF avoids this issue (see later) as does photon mapping (Figure 3, right). Photon mapping [40] adopts the photon rather than the light ray as the basic unit of energy transport. In a pre-computation phase photons are emitted into the scene from the light sources and reflected off both diffuse and specular surfaces until they are absorbed. The position and direction of the photon is then recorded in an efficient spatial subdivision structure called a KD-tree. Different density maps are used to record different types of path. In

[2] Images courtesy of Henrik Wann Jensen. (http://graphics.ucsd.edu/~henrik/images/cbox.html)

particular, caustics (the effect of a sharp beam of light hitting a diffuse surface and being reflected into the eye) are caused by LS*D paths and require a high density photon map for their representation. A lower resolution global map is used to store L(S|D)* paths. During the rendering phase the contributions of different types of radiance is calculated separately for the eye position using path tracing and information from the photon maps. The solution is thus ultimately view-dependent, as some calculation is left until the rendering phase. Research demonstrating real-time dynamic GI solutions based on photon mapping is available [40].

1.5 Motivation for and contribution of this project

The Virtual Light Field implementation for global illumination [2] provides a good approximation of the physically correct distribution of direct and indirect light rays in a static scene consisting of perfect diffuse and perfect specular surfaces. The existing implementation allows walkthough of this scene at interactive rates (15-20 frames per second (FPS)). However it has no support for changing the geometry of the scene without having to re-propagate all of the light.

A useful addition would be the ability to add movable objects to the scene, so as to provide a basic mechanism for interaction. An example application for this would be in Virtual Reality experiments, where the subject may have to interact with the scene in some way, perhaps by picking up and moving an object. Studies suggest that in such an experiment a higher level of immersion[3] can aid the subject in their task [12]. This makes global illumination a natural choice for rendering static textures for scenes in such virtual experiments. Researchers often desire that a Virtual Environment induces a sense of presence [11] in a subject, so that the subject's responses to the environment are the same as they would be in the real world. A greater level of immersion may contribute to sense presence [12].

By extending the GI walkthrough implementation of the VLF to support realistically shaded dynamic objects we can allow a greater degree of realistic interaction within the scene. However frame rate is more important to presence than shading realism [13]. To approach interactive frame rates the changes to the light in the scene due to object motion will have to be approximate. Also for the purposes of our example VR experiment the changes should be those most useful for understanding the relationship between objects in the scene. In particular shadows [15] and inter-reflections are good clues to depth and contact between objects in a scene. These are useful even when they are very crude approximations [14].

Section 2 provides a review of literature covering dynamic changes in globally illuminated scenes. Many of these techniques try to find the nearest approximate solution to the correct one for a general scene in the least time (or for a desired frame rate). This is a difficult and computationally expensive task. Rather than starting with an attempt to find a complete general solution to this problem for the VLF, my approach is to work towards greater realism in stages. This way we gain insight into how much processing time different techniques take, how much realism a technique adds and how much is required for a 'believable' moveable object. Also, given the time constraint of the project, this method ensures that code with some degree of usefulness is available upon completion.

[3] Immersion is a description of the fidelity of a virtual environment to physical reality and is a function of the technology used. Realistic visuals in a VE contribute to a higher level of immersion.

The contribution of this project is to extend the existing VLF implementation to support additional movable diffuse geometry, which casts and receives shadows and is shaded to different degrees of physical correctness using information from the VLF. The implementation can achieve interactive rates for simple geometry on a modern desktop PC.

1.6 Approach

A set of progressively more complex objectives were planned and executed for this project:

1. Render a simple flat shaded dynamic object (a cube) in the VLF by mixing z-buffer information using OpenGL.
2. Add keyboard controls for moving this object within the scene.
3. Allow the object to cast shadows into the scene. This is a significant depth cue.
4. Shade the object as though it has a diffuse material using a simple approximation of the emitters in the scene.
5. Shade the object as though it has a diffuse material using light incident on the object taken from the VLF.

Potential alternative approaches and the justification for each choice of approach are examined in the relevant sections of this report. The scope is of the project is restricted to adding dynamic objects to the scene – not changing the position of existing static geometry. These dynamic objects will be translated and rotated only, no scaling or distortion of the geometry is considered.

1.7 Scope

The constraints of the existing VLF implementation are:

1) Geometry consists entirely of planar polygons

2) Surfaces have either perfect diffuse or perfect specular properties

3) All surfaces are static during walkthrough

The existing implementation provides between 1-20 frames per second during walkthrough on a dual processor PC[4] depending on the content of the VLF scene and the rendering mode. In the fastest mode (coherent see Section 2.4.2.2.4) 20 FPS is possible. My aim will be to add at least one user controlled dynamic object which conforms to constraints 1 and 2 to existing VLF scenes. Such objects should render within a small fraction (10-20%) of existing frame render times[5] to be considered ‘interactive’. These objects should affect the appearance of the scene in a realistic manner (shading, shadows). Approximations of the ‘correct’ global illumination solution are acceptable where the approximation doesn’t cause an observer to judge the scene as too unrealistic. This is a subjective judgement discussed further in the results section.

[4] Dual Xeon 1.7 GHz workstation

[5] The test platform is of a lower specification than footnote 4, and so we consider frame times relative to the rendering times on the test platform. The test platform is a single processor CPU PC specified in the results section.

1.8 Report structure

In Section 2 the background literature leading to this work is discussed to place the problem in context. Section 2.4.2 outlines in detail the functionality of the pre-existing C++ code base for propagation and walkthrough of VLFs which provides the context for the further code modifications I will present. Section 3 describes each step of the implementation, from an initial high level of detail down to the algorithms and code used for each. Section 4 is a summary of the results achieved with the new code and includes screenshots and timing details of the implementation. Section 5 draws together the conclusions and further work arising from this report. Section 6 is the bibliography followed by appendices of detailed code listings. The full code of the project is supplied on the attached CD-ROM disc.

In this section we have outlined the basics of global illumination and the motivation for and contribution of this project. The next section expands upon the basic methods of calculating global illumination, the approaches taken to allow changes to global illumination solutions, methods of approximating shadows and the existing work on Virtual Light Fields.

2 Background literature

The virtual light field and its historical context is explained in detail in [1]. An application of the VLF to non-real-time dynamic ray tracing is outlined in [4]. However there is no existing literature on dynamic changes to a globally illuminated scene using the VLF approach.

As outlined in Section 1.3, GI solutions fall into one of two main categories: view-independent solutions which require an expensive pre-computation stage and yield a static solution for one arrangement of lights and geometry in the scene (such as radiosity); or view-dependent solutions which calculate the solution for rays currently visible to the eye only (such as ray tracing). Recently, more complex hybrid approaches have appeared which attempt to marry the best features from both (photon mapping).

Research which addresses the issue of real-time dynamic updates to geometry in GI solutions takes an approach appropriate to the type of solution they extend, and thus also falls broadly into two categories.

For view-independent solutions, the approach is to consider changing only the affected portions of pre-computed GI solutions, in real-time, as the scene changes. For example, only re-propagating light for affected patches within radiosity solutions.

For view-dependent solutions, the approach is to speed up the basic operations of the solution to the point where real-time frame rates may be sustained. Speed ups may be through optimization of existing algorithms for a particular hardware platform, parallelization of the operations for distribution among more processors or acceleration structures which reduce the number of operations required. Approaches in one area will often be of benefit in another area, as the basic problem tackled by both categories is the same (propagating light).

Figure 4 - In game images from Half Life 2[6] (Courtesy of gamespy.com)

The methods outlined so far all attempt to find a close approximation of the physically correct distribution of light within a scene, given certain simplifying assumptions. When the intended use of these methods is to provide a view into a virtual world for a human user, then coarser approximations can be used – sacrificing physical correctness of the view for something perceptually acceptable, which maintains an acceptable frame rate. Modern First Person Shooter (FPS) computer games are a good

[6] Copyright Valve/Sierra 2005

example of this. Figure 4 taken from Half Life 2 illustrates this. Static game geometry is pre-rendered using a GI solution and used during runtime as either flat textures or input into real time lighting equations. The soft shadow cast by the bridge is an example of a pre-rendered texture. The ripples and reflections in the water surface are updated accordingly as the viewpoint changes. In-game characters cast real-time blurred hard shadows onto the floor plane. Such approximations lead to a believable real-time dynamic environment without requiring a full GI solution in real-time.

This section of the report will consider prior work which:

- Considers the problem of dynamic geometry in pre-computed GI solutions such as radiosity
- Considers the problem of accelerating view dependent methods such as ray tracing to real-time rates
- Outline different methods of quickly approximating important GI features such as shadows

Each of these areas informs some part of the dynamic implementation of the VLF outlined later in this report. This will be noted where relevant. The existing implementation of the VLF is explained in more detail at the end of this section.

2.1 Radiosity

Radiosity [5] was introduced in Section 1.4.2. This method models only diffuse interactions and allows only static scenes. The VLF method can be considered to be a particular type of radiosity method which also allows specular surfaces. Here we discuss the major aspects of the radiosity method and how they compare to the VLF method.

2.1.1 Computation

In order to distribute light energy the scene geometry is divided into a number of patches n. Each patch has an area A and radiosity B. The formula relating the radiosity of one patch to all the other patches in a scene is:

$$B_i A_i = E_i A_i + \rho_i \sum_{j=1}^{n} B_j F_{ji} A_j$$

Where the terms are:

$B_i \mid B_j$ radiosity of patch i or j

$A_i \mid A_j$ area of patch i or j

E_i emission of patch i

ρ_i reflectivity of patch i

$\sum_{j=1}^{n}(...)$ sum over all n patches in the environment

F_{ji} the form factor between patch j and patch i

The form factor is an important geometrical quantity which represents the fraction of power leaving patch j that is received by patch i. Calculation of these form factors solves the visibility problem for each patch in a scene. This calculation is commonly solved by projecting every other patch in a scene onto the 5 subdivided faces of a hemicube placed above the patch in question. The fraction of hemicube surface area covered by the projected patch gives the form factor between the two patches. This relationship is reciprocal so that $A_j F_{ji} = A_i F_{ij}$. This solution is an approximation of the form factor suggested by the *Nusselt analogy* which recognizes that the form factor is an integral over the solid angle subtended by the sending patch on a unit hemisphere above the receiving patch. Calculation of form factors for a scene is an $O(n^2)$ operation as each patch needs to consider every other patch. We can use the reciprocity of form factors and divide the original equation by A to derive a new formula:

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

This formula can then be expressed in a matrix form and solved. The most expensive operation in radiosity (and GI in general), is determining the visibility between different points within the scene. In radiosity this translates to calculating the form factors. For the VLF this translates to finding the intersection area of projected radiance from sending cells on receiving cells via clipping (see 2.4.1.2). Because form factors are geometrical terms any change in the geometry has an effect on them, and thus changes to the geometry require expensive re-computations to the solution. Similarly, visibility calculations in the VLF are expensive and so approximations are used to allow dynamic objects (see 3.6).

2.1.2 Gathering and shooting

The matrix expression of the radiosity problem was initially solved using the Gauss-Siedel method. This method considers each patch in turn as a receiver of energy and 'gathers' the radiosity from all other patches for that patch. However this method means that in order to produce a complete image of the scene every patch in the scene must be solved at least once. For interactive purposes it would be preferable to provide the users with a scene image as soon as possible. This can be achieved through *progressive refinement (PR) [7]*. Instead of gathering the light from all patches for each patch in turn, the progressive refinement method 'shoots' radiosity from each patch in turn to all other patches in the scene. So with n patches in a scene only O(n) operations are required before an image of all patches (albeit a low quality one) is available rather than $O(n^2)$. The PR method also orders the operations so that patches which make large contributions of radiosity to the scene are calculated first – this providing a quick convergence to the correct solutions. Finally at iteration PR estimates an ambient light term that depends on the remaining unshot radiance in a scene. This ambient term is added to displayed patches to provide an approximation to the final solution before convergence.

2.1.3 Meshing

The visual quality of a radiosity solution increases with the number of patches used in the light distribution mesh. Low resolution meshes exhibit artefacts such as light and shadow leaks and staircase effects, among others.

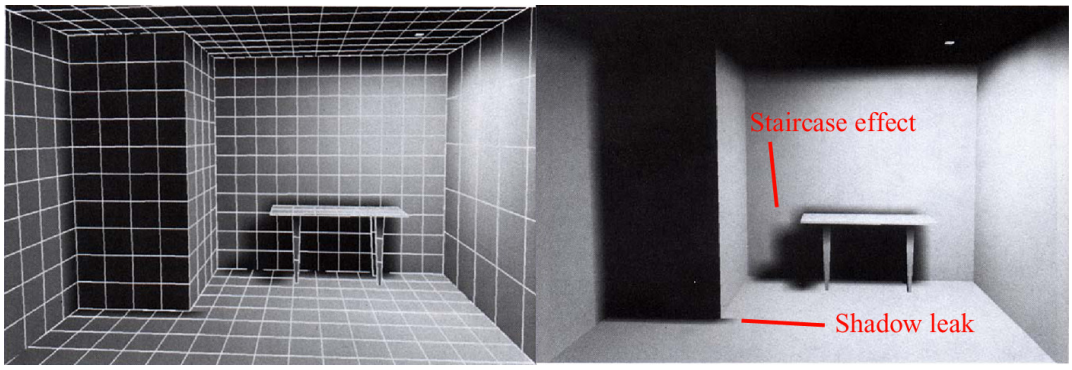

Figure 5 Uniform mesh subdivison may exhibit artifacts[7]

Increasing the resolution of the mesh by further subdivision reduces these effects, but a greater number of patches in the mesh requires much greater memory and processing overhead. Adaptive subdivision [8] is a technique which addresses this by only subdividing the mesh where the gradient in light distribution between patches is large. This provides the extra resolution where it is needed and leaves fewer, larger patches in the mesh where it is not. Discontinuity meshing [9] is a technique where the mesh is generated according to the predicted location of discontinuities. Both of these techniques add complexity to the scene description and increase the time taken to arrive at a solution but deliver a higher quality.

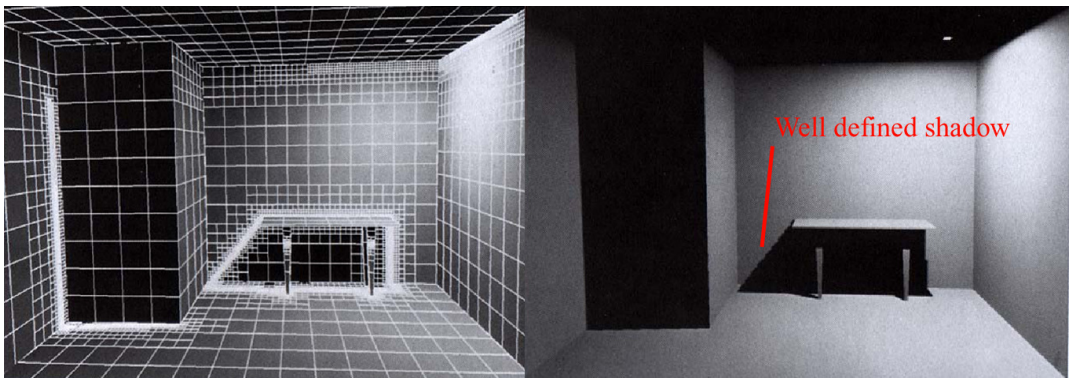

Figure 6 Adaptive subdivision addresses uniform subdivison artifacts[7]

The VLF method does not require any complex meshing calculations. Surfaces in a VLF are uniformly subdivided into cells which accumulate radiosity during the light propagation phase. These cells can be applied to the surfaces as diffuse textures during the visualisation of the solution.

2.1.4 Altering geometry in Radiosity solutions

As mentioned previously, Radiosity in its basic form models only diffuse interactions and allows only static scenes. A good summary of research addressing these limitations is available section 3 of [6]. A brief summary this research follows.

2.1.4.1 Incremental Progressive Radiosity (IPR)

IPR was proposed independently by two different groups [23], [24] and extends the concept of progressive radiosity (see above). IPR was designed to allow changes to

[7] Images from [10]. Originally courtesy of J. Wallave, 3D/Eye, Inc.

geometry in Radiosity solutions without a full re-computation of the solution. IPR extends basic radiosity by allowing attributes of patches to be changed during a continuous computation of the solution. The difference between the patches previous and new values is calculated and then propagated from the changed patch in the usual way. In this way, if a light is turned down or off then *negative* energy is propagated into the scene until convergence. Figure 7 (A) is the solution cycle for progressive radiosity. Figure 7 (B) is the new cycle for IPR.

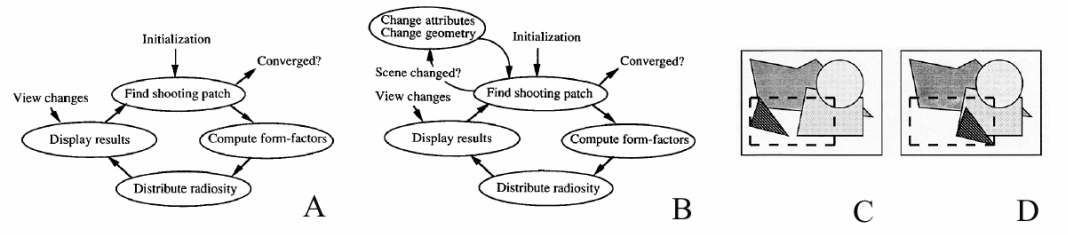

Figure 7 (A) Solution cycle for progressive radiosity. (B) Cycle changed for incremental PR. (C&D) Only the form factors for the affected portion of the hemi-cube are recalculated[8].

The geometry position can also be changed by using a similar incremental method for form factors. Only patches which can see the geometry changes need incremental form factors recalculation. As form factor calculation is expensive, only the portion of the hemi-cube affected by the change is recalculated. This is illustrated in Figure 7 (C & D). Both are views of a hemi-cube face as seen by a shooting patch. Only the dashed area needs to be considered for form factor recalculation.

2.1.4.2 Dynamic Hierarchical Radiosity (DHR)

DHR [26] is an extension of hierarchical radiosity (HR) [25]. HR extends basic radiosity by constructing a hierarchical presentation of the form factors in the scene. Polygons in the scene are adaptively subdivided into elements using a quad-tree. The form factor between elements belonging to different trees are estimated. If they fall below a user supplied threshold then a link between elements is established. These links are used to propagate radiosity between elements in the hierarchy, and radiosity is pushed up and down within each tree. The advantages of this approach are that the links have the same radiometric significance and that - at most - O(n) form factors are required (as opposed to $O(n^2)$ for the basic implementation).

[8] Images courtesy of [23]

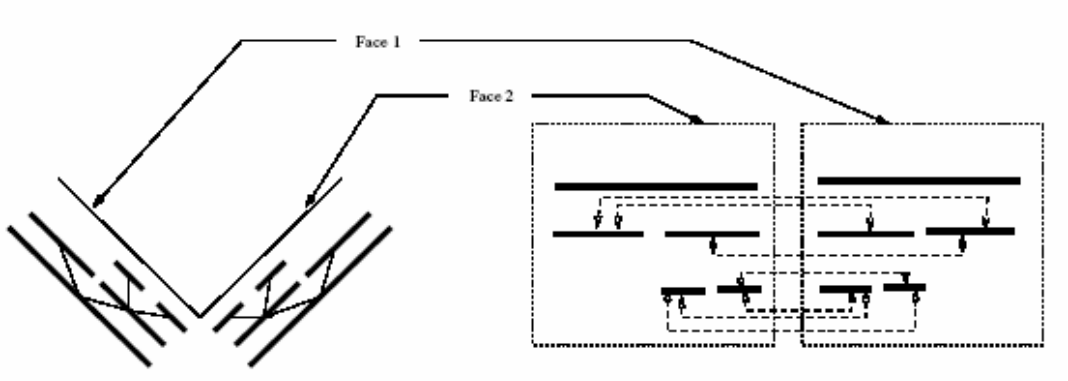

Figure 8 On the left, two plane walls at right angles (thin lines are geometry). The thick lines are elements. The hierarchy is express in the position of elements. The right diagram shows the links between elements[9].

DHR extends HR by making the assumption that objects are not created or destroyed and then considering the possible interactions between form factors as an object moves. Movement will alter the occlusion between elements, and so form factors associated with links may be modified or set to zero (but retained in case the occlusion is removed later). If polygons move towards each other, the form factors increase and so elements may need to be further subdivided and new links created lower in the hierarchy. This is known as demoting a link. If polygons move away from each other, the form factors decrease, and so links may also be promoted to higher in the hierarchy. DHR proceeds by predicting which links will be affected in the next frame of the dynamic scene and updating only those links; and then using an iterative solver based on the previous solution to solve the new situation.

Though results from this method are encouraging, it doesn't deal well with discontinuities and lacks an efficient mechanism to identify which parts of the scene changed while maintaining a quality/time tradeoff. The line space hierarchy approach outlined next addresses these shortcomings.

2.1.4.3 Line-Space Hierarchy

LSH [27] provides a mechanism to control the quality vs time trade-off in DHR by using shafts to represent the sets of line segments joining elements in the hierarchy. Traversal of this line-space allows modified links to be rapidly identified. It also reduces the amount of work required to solve the new hierarchical system at each frame.

2.1.4.4 Unified Hierarchical Algorithm (UHA)

Granier et al. enhanced the best of these approaches to include the specular effects of light in a Unified Hierarchical Algorithm [28]. This added stochastic particle shooting to determine the specular light paths and separated the diffuse and specular path information. It allows object motion by re-emitting a small percentage of particles for each new position. A space division scheme is used to quickly identify affected particle paths. Image quality is degraded during motion, but a high quality image is generated when the scene is static again. This is similar to the progressive rendering mode of the VLF walkthrough.

[9] Diagram courtesy of [26]

2.2 Raytracing

The approaches above concentrate on areas of change in a pre-computed solution to reduce processing time. The opposite approach is to speed up basic operations to the point where no pre-computation is required. In this approach all the rays reaching the eye are sampled once or more to produce an accurate rendering of the scene. The basic operation for visibility in GI methods is ray tracing. If this operation can be accelerated to the point where the set of required rays can be completely solved within the time allotted for an individual frame, then the solution can be recomputed completely each frame and no special processing is required to move scene geometry or change lighting parameters.

Similarly, reducing the number of rays which need to be calculated would reduce the processing time per frame. Ray/Space division schemes are one method of reducing the number of intersection tests required per frame. The following sections outline research along these lines.

2.2.1 Scene traversal acceleration schemes

In its most primitive form, ray tracing performs an intersection test for the ray in question with each object in the scene. Because objects occlude one another usually only the nearest intersection is required. Any method which helps identify the nearest intersecting object can reduce the number of intersection tests required per ray. One approach is to subdivide the space of the scene and record which objects occupy each partition of the space. Then the ray is traversed from its origin, testing only the objects which lie on its path. Regular grids, hierarchical grids, BSP-trees, octrees and bounding volumes are all variations of this approach.

2.2.1.1 Bounding Volumes

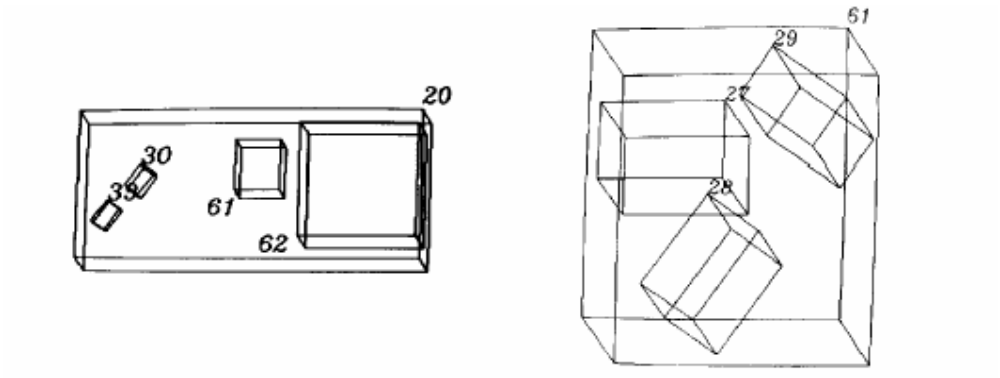

Figure 9 On the left is shown volume 20 of the hierarchy and the sub volumes it contains. On the right are the contents of volume 61, which can be seen within volume 20.[10]

Rubin and Whitted [31] Suggested representing the object space entirely by a hierarchical data structure consisting of bounding volumes. Each volume would be a parallelepiped oriented to minimize its size. All volumes at one level could be tested for the nearest intersection. That volume is then entered and also traversed. The bottom level of this tree consists of displayable objects such as planar polygons. Figure 9 illustrates this. As the orientation of each parallelepiped is just a matrix transformation

[10] Diagram courtesy of [31]

relative to its parent, the whole structure can be stored and traversed efficiently by translating the ray into each sub volume as required.

2.2.1.2 Space subdivision using grids

Regular grids divide the space equally into regions known as voxels (volume element). Objects which intersect a volume are recorded as belonging to the voxel. The ray is traversed from origin through the space. For each voxel it traverses, the intersecting objects are looked up and the relevant intersections are performed. This works well for scene where objects are spaced regularly, but performs poorly where many objects are clustered in a small area of the space. To overcome this, grids can be created which divide the space more efficiently.

2.2.1.3 Space subdivision using trees

Glassner [29] proposed using an octree data structure to recursively subdivide 3D space into voxels. If a voxel contained more than a threshold number of objects the space was further subdivided. Such a scheme leads to deeper trees to recurse, but avoids traversing many empy cells as could occur in a grid method.

A variation on this is to use a binary space partition (BSP) tree. Instead of dividing each voxel into 8 smaller voxels choose one cutting plane and divide the space into two smaller spaces. Do this recursively, until a threshold is reached. This approach leads to a simple binary decision (which is the nearest side of the tree?) at each level of the BSP tree, and can be implemented efficiently. A similar cache optimized approach was used in the VLF to accelerate ray traversal (see Section 2.2.2).

2.2.1.4 Ray classification

Another way to approach the problem is to subdivide the space of rays. Arvo and Kirk[30] describe an approach which uses the fact that a ray in a 3D space can be represented by a point in a 5D space. That is a ray is completely described by its origin (a point in 3D space) and a direction which can be recorded with 2 spherical co-ordinates (a further 2 dimensions). This 5D space can be subdivided into disjoint volumes and every object intersected by every ray in each sub-volume is recorded for that volume (this is the candidate set of objects for this sub-volume). Then when performing ray intersection first determine which sub-volume the ray belongs to and determine the closest intersection with the returned candidate set of objects (should there be one). Early results for this method suggested a factor 8 improvement for a particular scene rendering over Glassner's octree space subdivision method.

2.2.2 Optimizing ray tracing

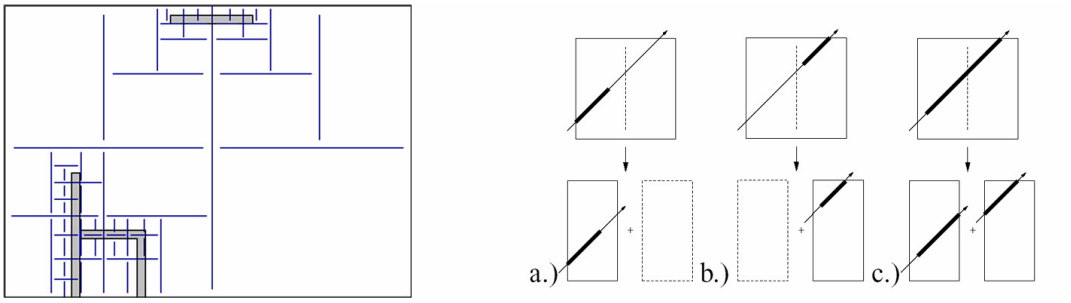

Figure 10 (Left) Scene divided by axis aligned BSP tree. (Right) ray traversal cases. a) Ray segment in front of splitting plane. b) behind splitting plane. c) ray segment intersects both sides[11].

Wald et al. [32] present a highly optimized ray-tracing implementation. Careful profiling revealed that in general ray tracers are bound by accessing main memory. When rays are shot incoherently (as in most global illumination algorithms), main memory can be accessed almost at random, and so the CPU cache isn't used efficiently and performance falls. Wald exploits the coherence of neighbouring rays traveling in the same direction to reduce main memory access and increase cache efficiency. This is achieved by considering packets of neighbouring rays traveling in the same direction. Additionally, cache performance is increased by aligning scene data to the size of cache lines used by the CPU. This is possible via the use of an axis aligned BSP tree where child object data can be placed close to the relevant parent data. SIMD (Single Instruction/Multiple Data) instruction sets (provided by modern AMD and Intel consumer processors) are used to perform 4 ray intersection tests in parallel. By restricting primitives to triangles and choosing an optimized barycentric co-ordinate test for triangle intersection calculations, further speed increases were possible. Wald was able to demonstrate between factor 11 and 15 speed increases over other popular ray-tracing engines such as POVray. These techniques were implemented in the triangle intersection code for the coherent ray-tracing mode of the VLF [2].

As mentioned most rays computed in GI solutions are incoherent. This and other restrictions imposed on ray-tracing by global illumination means that such impressive results are hard to achieve for a globally illuminated scene. Wald et al. [37] build on their previous fast ray-tracing work to present an interactive implementation of GI on a cluster of PCs. This is achieved by using fast, distributed ray-tracing engine, the idea of 'instant radiosity' [35] and a photon mapping stage to provide caustic effects. The idea behind 'Instant Radiosity' is to determine a number of secondary point light sources by tracing photons from the primary light sources and then determining the indirect illumination, by casting direct illumination from these secondary point sources. The photon mapping stage shoots photons towards specular surfaces and reflects them into the scene until a diffuse surface is hit and the photon is stored in a caustic photon map. Caustic photon tracing and illumination from point light sources is recalculated each frame. A client server approach is used for the distributed engine where samples of each image tile are computed on different machines and then combined with filtering on a master machine. This means network bandwidth and latency is also a factor in the implementation. The implementation handles dynamic scenes easily as the scene is

[11] Diagram on left from VIVE course notes. Diagram on right from [32].

recomputed each frame. Frame rates between 1-5 FPS were demonstrated on a cluster of 16 PCs[12] without using dedicated graphics hardware.

2.2.3 Dedicated graphics hardware

Recently, improvements in graphics hardware make methods harnessing the graphics processor unit (GPU) attractive. GPU methods involve identifying parts of the algorithm where operations can be parallelized and stacked in the graphics pipeline. Udeshi and Hansen [34] and Keller [35] both propose GI methods which use graphics hardware to support the CPU. These methods are parallel and require multiple processors. Recent methods harness the latest generation of consumer GPUs to allow real-time GI on a single processor supported by a GPU. Radiance Cache Splatting[36] exploits the theory that the most obvious GI effects are due to the first bounce of light to produce real-time walkthrough of a GI scene using the GPU on a high end consumer PC[13]. This extends the concept of radiance caching [38] which uses sparse sampling, caching and interpolation of incoming radiance to compute GI.

The VLF walkthrough uses the GPU to identify primary eye-ray intersections in its ray-tracing mode. Currently reading back the pixels from the GPU to the CPU is a slow operation though which can be a bottleneck to GPU approaches.

Ward et al. [39] demonstrate an interactive moving object in a light-field like holodeck ray cache structure. This supports full global illumination in non-diffuse environment rendered using a distributed network. Rays are calculated in parallel when required and results cached on the server in a holodeck (spatial grid) structure, which can supply the cached calculations when required. This method assumes ray calculations are expensive and so worth storing – an approach which differs from Wald [32], but the method used to light the dynamic object is of interest to us. The lighting is applied using standard OpenGL where the light parameters are estimated from the holodeck structure. If the object moves too far from the last light computation, its parameters need to be recalculated. This approach is used as a first approximation of the shading on the dynamic objects in this project, but the lighting parameters are set manually. It is left as further work to estimate suitable lighting parameters from the VLF.

[12] Dual-Athalon MP 1800+, 512MB RAM each.

[13] NVIDIA Quadro FX 3400 PCI-E and a 3.6 GHz Pentium 4 CPU with 1 GB RAM.

2.3 Shadows

A powerful cue to an object's position in a scene are the shadows it casts and receives. Correct soft shadows arise naturally in radiosity like solutions to GI, and the VLF is no different in this respect. Soft shadows refer to the fact that shadow has not only a 'hard' central area of constant illumination level (the umbra of the shadow) but also a 'soft' illumination gradient at the edges of the shadow (the penumbra). The static objects in the pre-propagated VLF scene cast and receive natural looking soft shadows. To produce shadows the same way for an object added to the existing scene would require re-evaluating all of the rays in the scene which intersected the object and modifying their onwards radiance values to account for the occlusion, then calculating the next object intersected along the ray and reducing the incident radiance accordingly. These 'shadow rays' could then be further propagated into the scene depending on the surface properties. This is clearly too computationally expensive to achieve in real-time on a single PC. To achieve a real-time shadow an approximation must be used.

There is a large body of research on shadows [16] and there are a variety of methods for rendering both hard and soft shadows relatively quickly. Broadly, these methods operate in either image space or object space methods. In real scenes you never see pure hard shadows, as they are generated by point light sources and in the real world all light sources have a finite area which will cause some degree of penumbra in a shadow. However hard shadow methods are easier to implement and fast soft shadow methods in general are multipass extensions to the hard shadow methods. So a natural choice for a first attempt at fast dynamic shadows is to approximate the emitters in a scene using point light sources and to implement a hard shadow method. I consider two common hard shadow methods: *shadow mapping* and *shadow volumes*.

2.3.1 Shadow mapping

Figure 11 - Scene rendered from camera (left), and light (middle). Depth buffer from the light's view (right)[14]

Shadow maps are an image based method [18]. The scene is rendered from the viewpoint of the light to create a depth buffer (z-buffer). This z-buffer is then transformed into the eye reference frame and the scene is re-rendered from the viewpoint of the eye. For each pixel in the eye frame, the transformed z-buffer is compared with the eye depth information. If a fragment is further away it is deemed to be in the shadow of another object and rendered accordingly. Shadow maps don't

[14] Images courtesy of [18]

generate any additional scene geometry and can create shadows for any entity which can be rendered to the z-buffer. They are implemented efficiently in modern graphics hardware. However they suffer from aliasing effects due to the finite nature of the z-buffer. If the camera view direction is opposed to the light view direction, the shadow's depth information is undersampled with regard to the requirements of the camera view. This problem is known as dualling-frustra and is illustrated in Figure 12.

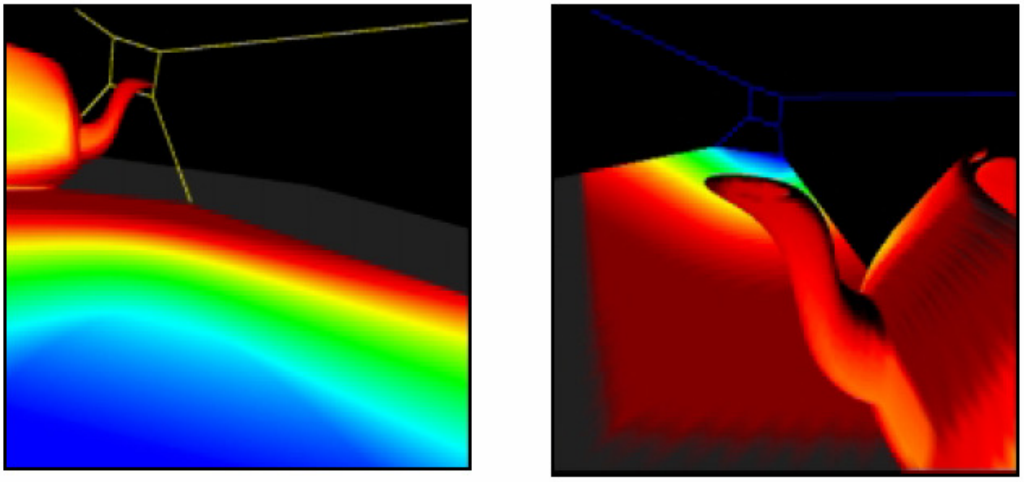

Figure 12 The eye's view (left) and light's view (right). Colours represent variation in sampling frequency of the shadow map with blue being the highest. The problem is that only a small portion of the lights image plane (blue, right) can be used where the shadow requires most detail (blue, left). Thus the shadow is undersampled and appears jaggy[15].

This aliasing can be reduced by increasing z-buffer resolution via rendering to the z-buffer in stages, using *perspective shadow maps* [20] or *percentage-closer filtering* [21]. However each approach requires more computation and is beyond the scope of this report.

2.3.2 Shadow volumes

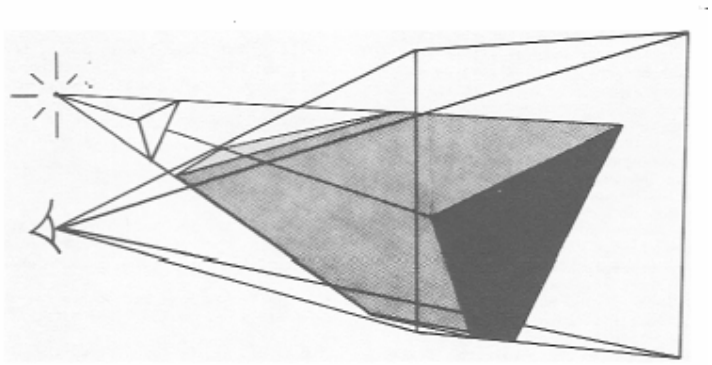

Figure 13 - A shadow volume clipped to the camera view frustum[16]

This method, first suggested by Sutherland and Crow [17], projects the silhouette of a light occluding polygon into a scene, to create a shadow volume defined by its

[15] Diagram courtesy of [18]

[16] Diagram courtesy of [17]

projected surfaces. This results in shadow geometry being created along with the usual scene description and is thus a view-independent (object space) solution. When rendering the scene from the eye, rays which terminate within the shadow volume are determined to be in shadow. The amount of computation involved increases with the square of the amount of data and, for dynamic objects, the shadow volumes would have to be recalculated each frame. This would result in slower shadow generation for a scene with many dynamic objects. For these reasons I decided to implement shadow mapping as opposed to shadow volumes to achieve hard shadows. However shadow volumes would have made calculating the reflections of dynamic shadows in static specular surfaces easier. This is discussed later. Fast implementations of shadow volumes in graphics hardware exist [22].

2.4 The Virtual Light Field

2.4.1 Theory

The detailed concepts are covered in [1]. There follows an overview of the basic concepts and the details most relevant to changing the implementation to support dynamic objects. Virtual Light Fields build on the idea on light-fields [2 from VLF] where a scene is visualised as a 2D slice of a 4D data structure. The VLF is a data structure consisting of a uniformly distributed set of l directions over the hemisphere (see Section 2.4.1.1). Each of these directions corresponds to a N by N set of rays in that direction called a parallel subfield (PSF) of rays. We index the set of rays within each PSF by considering each ray as originating at the centre of a pixel (i, j) of an N by N image at the base of each PSF each ray travels in direction l. To exploit coherence between rays the pixel space of each PSF is divided into a set of m by m tiles (where m is between 1 and N and N is a multiple of m).

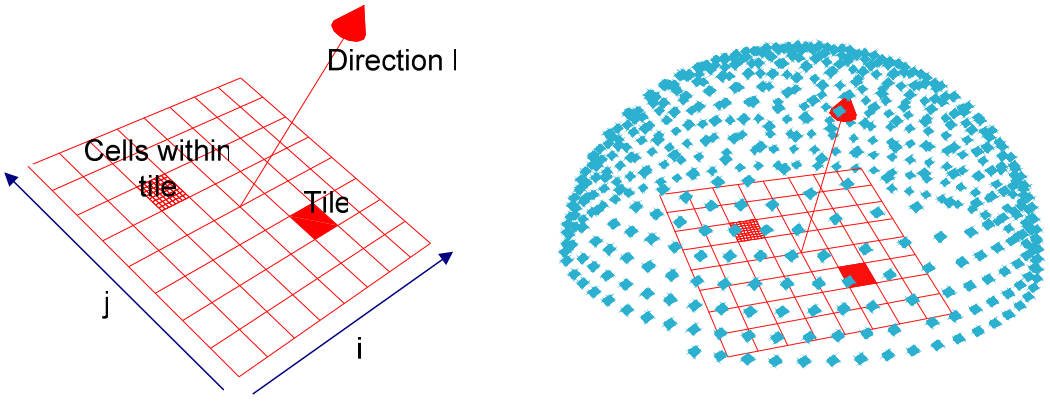

Figure 14 (Left) A parallel subfield (PSF). Each ray originates at the centre of its respective cell (i,j) and travels along direction *l*. Here *m*=8, *N*=8. (Right) A PSF inside the hemi-sphere of directions. Each cyan point represents a unique direction from the origin of the hemi-sphere.

The canonical PSF is the set of rays parallel to the z-axis (vector (0,0,1). We can chose l uniformly distributed points on the +ve hemisphere and define each PSF as a rotation of the canonical PSF into the spherical co-ordinates of each point.

The geometry of a scene exists in its own frame of reference which we will call world co-ordinates (WC). For the purposes of the VLF the scene is considered to exist

within its own bounding sphere of unit radius. The centre of this sphere is taken as the origin for the VLF co-ordinate system – which is the WC system scaled and translated into this unit sphere.

If we now consider a surface defined in the scene and the canonical PSF of rays we can imagine the surface being intersected by some of these rays. It is likely given that the rays in each tile are close to each other that other rays in the same tile will also intersect the same surface. This coherence is exploited by storing in each tile the surface identifiers of surfaces intersected by any rays in the tile. For each surface identifier in each tile, there is also a visibility map for the surface and a 2D image map (called a *radiance map*) intended to hold the outgoing radiances from each point which has a visibility map entry. Intersection depths could be stored, but given the fast rasterisation method used, they can be recomputed as needed to save space.

2.4.1.1 The distribution of directions over the hemisphere

Ideally, for the VLF, each uniformly distributed direction on the hemisphere would have an equal solid angle and the same shape. Unfortunately, there is no known solution which can provide this and allow constant time lookup. The distribution of directions is achieved via a triangular subdivision of a regular tetrahedron discussed in [33] and illustrated below. This provides solid angles with ~60% variance in solid angle and unequal ray density in different parts of the hemisphere. Varying solid angles lead to unequal amounts of diffuse radiance being propagated in different directions from a diffuse surface. During the propagation phase this has to be counter balanced by a normalization term based on the weight of the solid angle of the PSF direction. This has implications for the shading of dynamic objects and is discussed in section XXX.

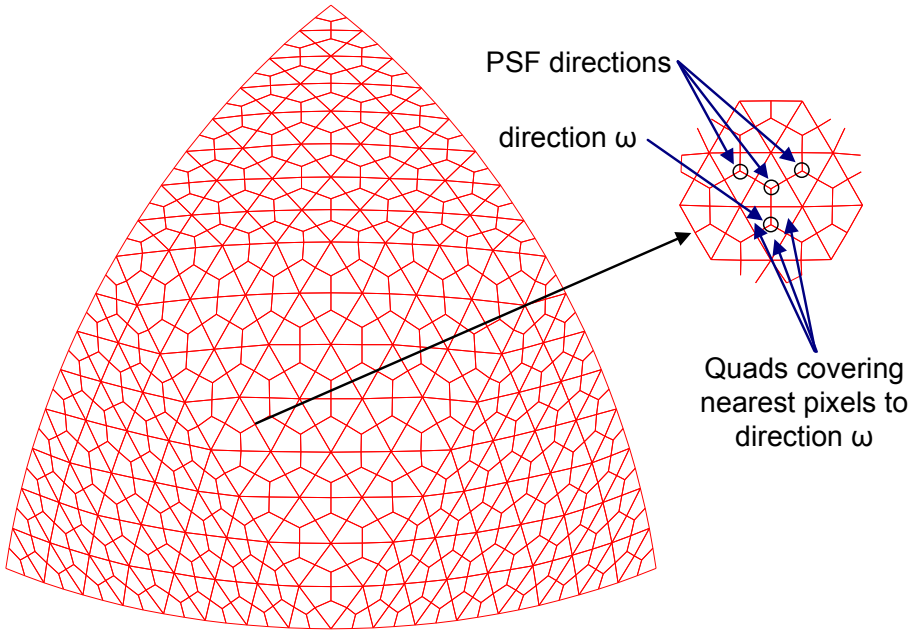

Figure 15 A Quadrant of the hemisphere used to lookup PSF directions (viewed from within). Note unequal solid angles.

To lookup the nearest PSF direction to an arbitrary direction in constant time, first assume the direction is in the first quadrant of the hemisphere. Render the quadrant to the frame buffer so that each PSF direction has an associated set of polygons

representing the directions closest to that point. The frame buffer is then read back to a 2D array and we compute the pixel position corresponding to the projection of the required direction on the hemisphere. The colour index at that buffer position gives us the index of the nearest VLF direction. Should the required direction not be in the first quadrant, we can just rotate the direction into the first quadrant and add an offset to the returned index. See [33] for further detail.

2.4.1.2 Propagation

The process of correctly populating the radiance maps in the VLF structure is covered in section 4 of [1]. It is not repeated here in detail as it is beyond the scope of the project. In short however the algorithm is:

- For each PSF direction ω
    - For each tile T
        - For each surface P
            - Consider all (visible) incoming rays
            - Compute the correct fraction of incoming diffuse and specular radiance and accumulate in the total radiance map for the sender
            - Record the incoming radiance as radiance which must be propagated diffusely in the next iteration
            - Record the outgoing specular energy in the nearest PSF to the correct reflected direction from the surface
- Repeat for a pre-specified number of iterations

The process of determining the correct fractions of energy to propagate is complex and requires calculating the intersection area of the projections of cells from sender to receiver. This is done using polygon clipping which, although expensive (~80% of propagation time), is necessary to avoid serious aliasing. As well as clipping, filtering is used during propagation to reduce aliasing effects.

The propagation phase is essential, calculating the solution to a recursive problem by *shooting* light energy from each surface to all other surfaces visible from the current surface. This is discussed in more detail in Section 3.6, when we consider *gathering* light energy from the VLF onto dynamic surfaces. This process of *gathering* energy for dynamic surfaces is a key difference to the existing *shooting* mechanism, used iteratively in the VLF for static surfaces during propagation.

2.4.1.3 Walkthrough

Given this data structure and supposing that all radiance maps in all tiles in all PSF had been populated with their correct outgoing radiances for some globally illuminated scene, then we have a method for finding an approximation of the radiance traveling in any direction ω from any point P on any surface in the scene. We find the nearest PSF direction to ω which we can call ω'. This ω' will have a corresponding rotation matrix which will rotate point P into the canonical PSF direction (0,0,1). This same rotation matrix will rotate the point P into the canonical frame of reference. Call this point P'. The projection of this P' onto the XZ plane of the canonical PSF (i,j in fig 1) will give a tile and a cell within that tile which emits radiance in the desired direction.

We only have to search through the surface identifiers for that tile until we find the surface identifier for the point P, and then look at the corresponding pixel of the outgoing radiance map.

In a walkthrough we will need to determine the nearest PSF direction frequently – so the method used needs to be quick – and we would like to maintain a constant frame rate – so we would like it to be a constant time operation. This is achieved using a triangle based subdivision of the hemisphere (See 2.4.1.1).

A better approximation of the actual radiance required can be achieved by interpolating the radiance values from the 8-neighbourhood of pixels around the projection of P'.

A better approximation still can be achieved by picking the three closest PSF directions to the desired direction and interpolating the 8-neighbour interpolated radiances, using spherical interpolation weights. This method is used in the RENDER_RAYTRACE mode of the existing walkthrough code. See section 2.4.2.2 for a description of how the walkthrough modes are implemented.

2.4.2 Overview of the existing VLF code

The existing VLF code base was ~90,000 lines of C++ code. The code was object-oriented and made heavy use of templates. These features make the code flexible but imposed a steep learning curve when modifying the code. The code was supplied as a Microsoft Visual Studio solution (vlfApps.sln) broken down into projects along conceptual lines. For the purpose of this report, the main areas of interest are the vlfPropagator and vlfGlutWalkthrough projects, which create and display the VLF data respectively. Support classes are defined in the vlf, glib, scenegraph and utils projects. For the purposes of reporting clarity, a new namespace 'dynamic::' and the files dynamic.cpp and dynamic.h have been created to contain the majority of changes. Existing classes were only modified where it was more sensible to do so than creating new code in these files. The whole C++ solution is included on a CD with this report. Relevant code snippets are included in the body of the report. Most changes to the original code are included as appendices. The following sections outline the existing functionality in the two main code projects in more detail.

2.4.2.1 vlfPropagator - Propagation of light to create the VLF file

An outline of the process is given in 2.4.1.2. The input to the process is a scene file in wavefront **.obj* format containing geometry information and a material file (**.mtl*) describing surface properties. The name of this file and the parameters controlling number of VLF directions, tile and cell resolution and many other parameters are defined in the header file *vlfGlobalDeclarations.h*.

The propagator process proceeds through a number of iterations of propagation. At the end of each iteration it outputs a VLF **.bin* file which contains the state of the VLF at that point. This file can be used as a starting point to continue propagating light in the future. It is also useful in checking early progress, in case mistakes have been made in the scene definition. These **.bin* files are the input to the walkthrough process.

2.4.2.2 vlfWalkthrough (viewer)

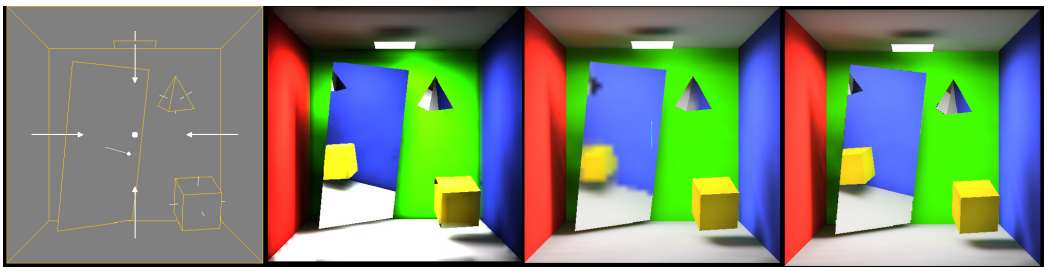

Figure 16 - Rendering modes (Geometry, raytrace, progressive (low quality), coherent)

A case statement in the display() routine of vlfGlutWalkthrough.cpp selects the current viewing mode. There are four main ones shown in Figure 16. Their descriptions follow.

2.4.2.2.1 RENDER_GEOMETRY

This mode is the simplest and thus has the highest frame rate. It displays only line geometry of the scene and is useful for positioning the camera before switching to a more computationally expensive mode.

2.4.2.2.2 RENDER_RAYTRACE

From the camera position, a false colour OpenGL rendering of the scene geometry is created and then read back into main memory. Each false colour corresponds to a surface identifier S and so, for each pixel and corresponding primary ray in the image, a ray intersection point P can be calculated. The ray direction will intersect one triangle in the hemispherical subdivision (see 2.4.1.1), and thus provide three vertices corresponding to three PSF directions. Each direction provides a rotation matrix which is used to rotate the point P into canonical PSF co-ordinates. This rotated point is projected onto the XZ plane of the PSF to yield a pixel on the radiance map corresponding to S (which is found in the list of surface IDs for the intersected tile). The pixel radiance returned for each PSF is interpolated from the surrounding 8-neighbourhood, and the three PSF radiances are interpolated using spherical interpolation weights.

If the primary ray strikes a specular surface, the ray is reflected into the scene until a diffuse surface is hit and radiance is determined as above. This approach provides near constant time lookups for direction and radiance for diffuse surfaces, but for specular surfaces the time per pixel depends on the depth of ray reflection into the scene. The time taken to lookup a surface ID in a tile is logarithmic in the average number of surfaces intersected by a tile.

Visually, the shading on surfaces exhibits sampling artifacts due to the approximations in the light field. The diffuse visual quality can be improved upon as described in the progressive mode and the speed of ray tracing improved as described in the coherent mode.

2.4.2.2.3 RENDER_PROGRESSIVE

All diffuse surfaces are assigned a total texture map which accumulates light energy during the propagation stage. This texture map can be used directly to texture the diffuse surfaces, thus giving sharp visible edges. This technique is used in both progressive and coherent modes. In addition, the reflected specular rays can return the

bilinearly interpolated radiance from the intersection point on the eventually struck diffuse total texture map, rather than the spherically interpolated light field. Because tracing rays for the specular surfaces is slower than reading the reflection information directly from the light field, the progressive mode will use the light field information (with lower directional resolution) while the camera is in motion, and then render the higher quality version using ray tracing through the reflections when the camera is at rest. The high quality rendering progresses in a different thread of execution, so this runs well on a dual processor PC. However the frame rate for the progressive mode is surpassed by the coherent ray tracing mode outlined below.

2.4.2.2.4 RENDER_COHERENTRT

The coherent mode has the same visual quality as the progressive mode but it exploits the coherence in adjacent rays to speed up raytracing for specular surfaces. The approach is similar to Wald et al [32] and uses SIMD instructions to test bundles of 4 rays at once and a cache aligned acceleration structure (an axis aligned BSP tree) to speed up scene traversal during ray tracing. This carefully optimized code is difficult to modify but, due to its low frame rate, is be the natural choice of rendering method to add dynamic objects to.

2.4.2.3 Object oriented design

Given the object oriented design of the existing code base, it makes sense to reuse objects and design conventions where possible. This section describes some key classes and methods used in the code. It is not an exhaustive list of the available code objects. Descriptions are brief and expanded later in the report where required.

Figure 17 is a class diagram outlining the classes, their methods and their inheritance. The descriptions below refer to classes in the diagram. This diagram is a useful reference to use while reading code examples in the implementation section. Methods in blue were added as part of this project. These are few as most code implementation was kept in a separate namespace for clarity, however those not in blue may have modified slightly as part of the project where required.

2.4.2.3.1 vlf::glTiledVLF and ancestors

The main VLF class. This provides access to the scene description object and the directions of the vlf. Methods in this class allow various high level interactions with the information within the VLF.

2.4.2.3.2 vlf::direction

Represents a unique direction within the VLF. Point() returns the direction as a point on a unit sphere and areaWeight() returns a normalisation term which compensates for the variation in solid angle area of that direction.

2.4.2.3.3 vlf::facesetand ancestors

Represents a group of related faces. This class is used to hold the geometry of the scene and was reused in the dynamic VLF implementation to hold the geometry of the dynamic object.

2.4.2.3.4 vlf::face

Represents a single face within a faceset. An texture is assigned to a hold diffuse surface representation if required.

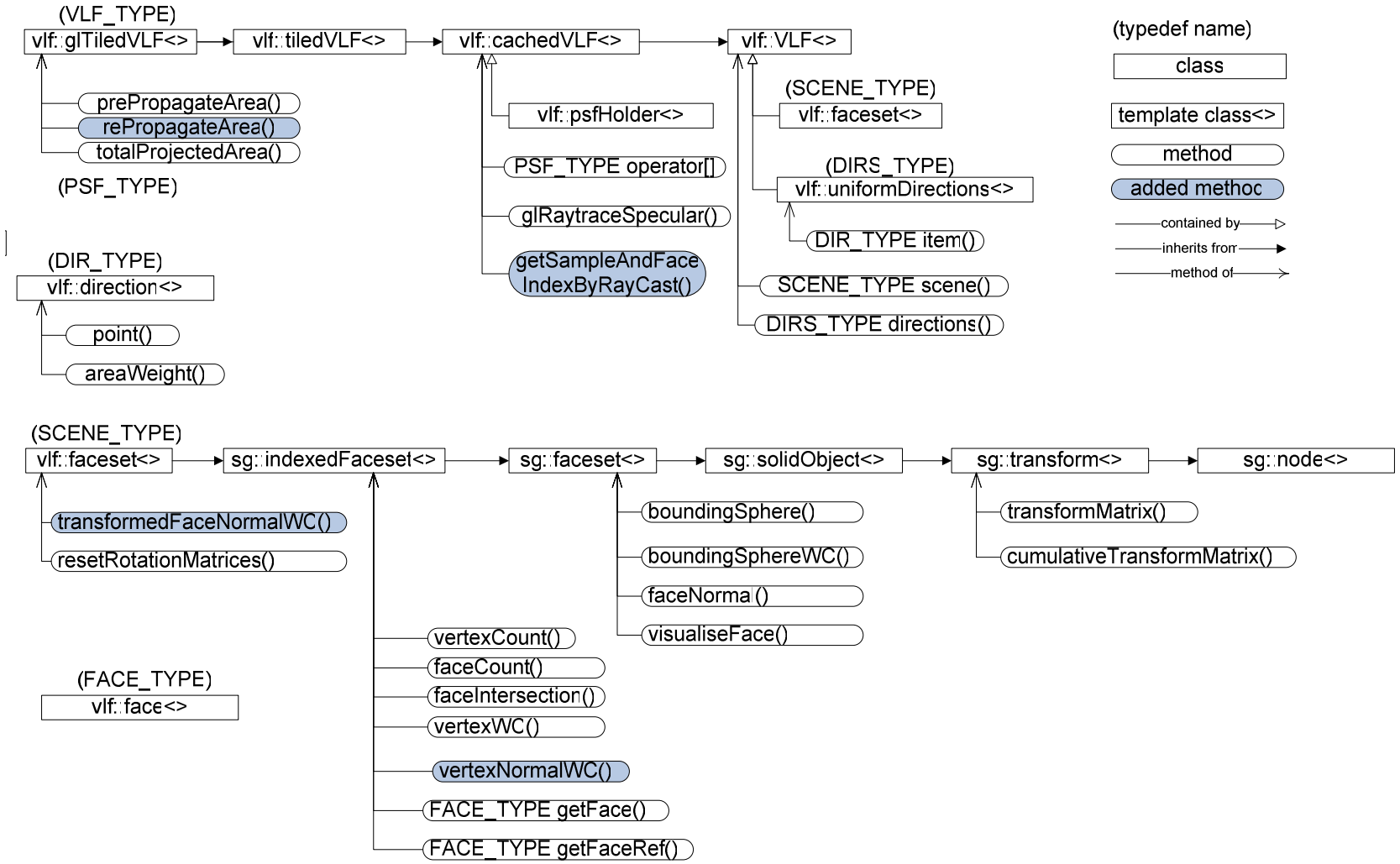

Figure 17 Overview of main classes and methods

3 Implementation

In Section 1, the topic of global illumination was introduced and the motivation and aims of this project outlined. In Section 2, relevant prior work was considered and the concept of Virtual Light Fields explored in more detail. In particular a high level overview of the existing VLF implementation was given.

In this section, the extensions to the existing VLF code are covered in detail. The high level approach was outlined in section 1.6. Each subsection in this section proceeds as a discussion of each milestone - from the high level aim for that section to the detailed implementation - including screenshots of results, and code snippets or listings, where they enhance understanding. This layout has been chosen so that concepts and their relevant implementations sit close together in the text to avoid a disjointed reading experience. The overall results of the implementation stages are drawn together in Section 4.

3.1 Adding a simple object to the existing scene

3.1.1 Definition

Geometry in the VLF is loaded from wavefront **.obj* files into an object of class vlf::faceset. This class is an efficient representation of the geometry, using indexing to avoid redundancy. The classes it derives from supply methods for intersecting rays with the geometry, and a matrix positioning the object within world co-ordinates (sg::transform). This makes it an ideal choice for reuse to ultimately represent the dynamic object.

The simplest rendering mode for the VLF scene is the geometry mode (see 2.4.2.2.1). As a first stage the aim was to add a new, simple, static object to the scene as a vlf::faceset and to render it in the geometry mode. A cube was chosen as the dynamic object as being simple and easy to perceive correctly in the scene. The cube definition was hard coded in the dynamic.h header file. The vlf::faceset object, representing this dynamic cube, is initialised in dynamic::initBox(), which is called from main() in vlfGlutWalkthrough.cpp. See Listing 1 (end of document). The cube is defined as 6 faces consisting of 2 triangles per face. Thus 12 vlf::face objects in total.

3.1.2 Wireframe Viewing

With our object defined and accessable via the global scope dynamic::myIndexedFaceSet we now need to render it. The main display() loop in vlfGlutWalkthrough.cpp consists of a case statement which switches depending on the current rendering mode. This display routine is assigned to the glut display callback during initialisation.

```
void display(void)
{
   …
   if(renderType==RENDER_DIAGRAM)
   {
      …
   }
   else if (renderType==RENDER_GEOMETRY)
   {
      …
         // existing scene rendering loop
         for (unsigned int i=0;i<myVLF.scene().faceCount();i++) {
```

```
            myVLF.scene().visualiseFace(i,false,true,true,fc,true);
         }

#ifdef DO_DYNAMIC_OBJECT
         // visualise dynamic object faces
         if(showDynamicObject) {
            for (unsigned int i=0;i<dynamic::myIndexedFaceSet.faceCount();i++) {
            dynamic::myIndexedFaceSet.visualiseFace(i,false,true,true,COLOUR_WHITE,true);
            }
         }
#endif
      …
   }
   else if (renderType==RENDER_PROGRESSIVE)
   {
      … // other cases omitted
   }

   glutSwapBuffers();
   …
}
```

The code shaded in light blue has been added as part of this project (this convention applied to the whole document). In this case the code uses the existing visualiseFace() method to draw the new geometry as a wireframe into the existing OpenGL back buffer. This utilises the existing z-buffer information to ensure correct occlusion. When the buffers are swapped at the end of the display() call the cube is rendered as expected in the centre in of the existing scene. See DIAGRAM. The ability to reuse the visualiseFace method is one of the advantages of using the vlf::faceset object to store the geometry.

3.2 Moving the object

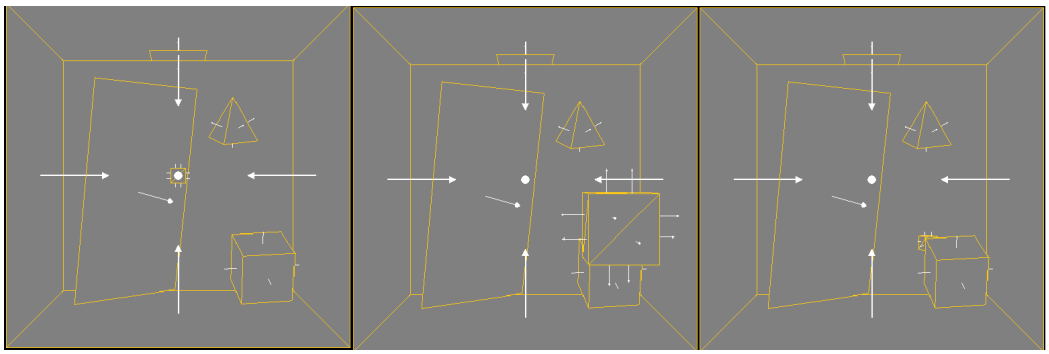

Figure 18 Three screenshots in geometry mode with normals displayed. Cube added but static (left). Cube moved toward camera demonstrating occlusion of static scene (middle). Cube moved behind static cube demonstrating occlusion of dynamic object (right).

With the object visible in the scene, the next step is to move it interactively. The existing VLF implementation allows for movement of the camera position using the keyboard. This is achieved via the glut keyboard callback calling methods on the worldCam object. A similar arrangement was implemented for the dynamic object. A user would either want to move the camera or the object, but not both at the same time, so a mode switch was introduced that used the same movements for either object or camera depending on the movement mode. The vlf::faceset class inherits from sg::transform, which contains a transform matrix placing the object relative to the WC system. Motion is achieved by creating a new matrix, by multiplying the existing transform matrix by the relevant rotations and translations (supplied by the keyboard input) and then overwriting the existing transform matrix in the faceset.

```
// Keyboard callback
void keyboard(unsigned char key,int x,int y)
```

```
{
   switch(key)
   {
   ...
   case 'o':
      // Toggle movement mode, do we want to move object or camera?
      if(dynamic::dObjMovement)
         dynamic::dObjMovement = false;
      else
         dynamic::dObjMovement = true;
      std::cout       <<       "Movement       mode       is       now:       "       <<
((dynamic::dObjMovement)?("object"):("camera")) << std::endl;
      break;
   ...
      case 'w':
      if(!dynamic::dObjMovement)
         worldCam.moveForward(moveStep);
      else // Translate the dynamic object along Z axis
         dynamic::translate(glib::vector3D<MY_FLOAT_TYPE>(0.0f,0.0f,-moveStep));
      moveDetected();
      break;
   ...
   case 4: // 'CTRL+d'
      if(!dynamic::dObjMovement)
         worldCam.strafeRight(moveStep);
      else
         dynamic::rotateY(rotateStep);
      moveDetected();
      break;
   ... // other movement cases omitted for brevity
}
```

```
void dynamic::translate(glib::vector3D<MY_FLOAT_TYPE> &v)
{
   // pop, translate and push the transform matrix for the dynamic faceset
   glib::taggedTransformationMatrixf cumMatrix =
      myIndexedFaceSet.cumulativeTransformMatrix();
   cumMatrix.postTranslate(v);
   if(dynamic::recordAnimation) {
      dynamic::animation.push_back(cumMatrix);
   }
   myIndexedFaceSet.transformMatrix(cumMatrix); // clears stack and replaces with supplied
matrix
}
void dynamic::rotateY(MY_FLOAT_TYPE angle)
{
   // pop, translate and push the transform matrix for the dynamic faceset
   glib::taggedTransformationMatrixf cumMatrix =
      myIndexedFaceSet.cumulativeTransformMatrix();
   cumMatrix.postRotateY(angle);
   myIndexedFaceSet.transformMatrix(cumMatrix);
}
... // other rotates left out
```

The keyboard controls for moving the object are:

Key	**Action**
o	Toggle camera or dynamic object motion mode
a	Move object along –X axis
d	Move object along +X axis
x	Move object along +Z axis
w	Move object along -Z axis
CTRL+w	Move object along +Y axis
CTRL+x	Move object along -Y axis
CTRL+d	Clockwise rotation around Y axis
CTRL+a	Counter clockwise rotation around Y axis
Z	Clockwise rotation around X axis
Q	Counter clockwise rotation around X axis

All motions are in the object's local co-ordinate system. We now have a simple, wireframe object which can be moved within the scene in geometry mode. The next step is to consider shading the object in one of the VLF shading modes.

3.3 Adding the dynamic object to VLF shading modes

At this point we have added a new object to the existing VLF scene, rendered as a flat shaded wire-frame, which can move amongst the static objects in the scene with correct occlusion. This is implemented using existing code wherever possible. However the geometry mode doesn't express any of the features of a VLF. It is purely a convenience mode rendered entirely in OpenGL and uses none of the light distribution information from the VLF.

The simplest rendering mode to actually display shading from the VLF is the RENDER_RAYTRACE mode. However this mode doesn't use the total diffuse texture maps from the VLF and so is visually inferior to the RENDER_PROGRESSIVE mode. It also doesn't backward ray-trace the specular surfaces and so reflections in specular surfaces are rougher approximations. The RENDER_COHERENT mode has all the benefits of the RENDER_PROGRESSIVE mode but the ray-tracing code is much faster but also more complex. Given this the next step was to display a flat shaded version of the dynamic object in the RENDER_PROGRESSIVE mode. The ultimate aim is to add the dynamic object to the much quicker RENDER_COHERENT mode (3.3.2).

3.3.1 Flat shading and occlusion in progressive mode

In all VLF shading modes the diffuse and specular surfaces are calculated separately and then combined at the last stage of rendering. Different methods are used to ensure correct occlusion between specular and diffuse surfaces. When adding a movable object to the scene, the relevant occlusion code must be altered to ensure the object occludes and is occluded correctly.

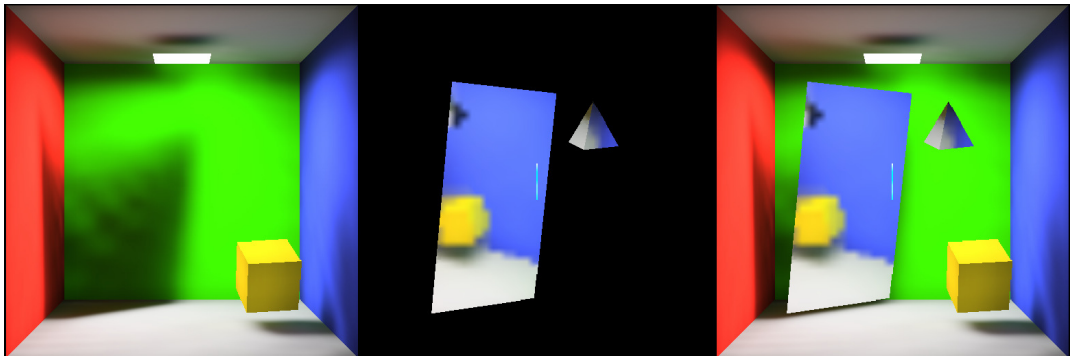

Figure 19 The rendering stages for the low quality progressive mode before changes are made for a dynamic object. Diffuse surfaces only (left). Low quality specular viewport texture (middle). Final composite image (Right).

In the progressive mode the diffuse surfaces are all handled the same way. The total radiance texture maps from the VLF are used to texture the diffuse polygons using OpenGL. The specular surfaces are rendered using two different techniques. When the camera is in motion we need to maintain a high frame rate. In order to do this the incident light per visible pixel on the specular surfaces is calculated by interpolating the irradiance from the 3 nearest incident rays to the reflected direction using the data in the VLF. This is a near constant time lookup in the VLF per specular

pixel and is the quickest method for providing an approximation of the reflections of static geometry. However, the quality of the image generated depends on the directional resolution in the VLF. Figure 19 shows specular surfaces rendered from a VLF containing 512 directions and blurring in the reflection is apparent.

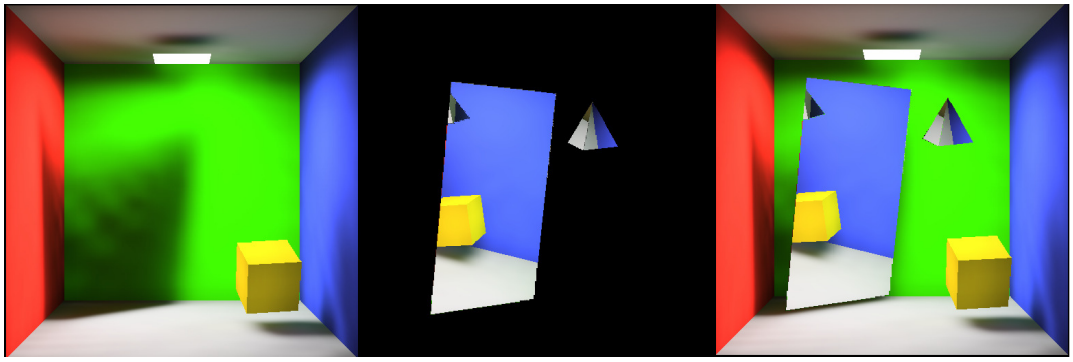

Figure 20 The rendering stages for the high quality progressive mode before dynamic object is added. Diffuse surfaces only (left). High quality specular viewport texture (middle). Final composite image (Right).

A higher quality specular surface is achieved once the camera is at rest, through tracing rays into the scene through the reflections, until a diffuse surface is struck. This intersection point can then be transformed into a point on the relevant diffuse surface's texture map, and the pixel value used to colour the specular pixel in the final image. This higher quality rendering is approximately 8 times slower[17] than the VLF method for the camera position shown in the figures. The rendering time depends upon the number of specular pixels visible and the level of recursion reached per specular pixel and so the camera at rest rendering time will increase as the proportion of visible specular surfaces increases.

In both high and low quality progressive rendering modes, the diffuse and specular surfaces are rendered separately and then combined to form the final image. In combining the two images the correct occlusion must be maintained.

3.3.1.1 Dynamic object occlusion in low quality progressive mode

In low quality mode this is achieved by rendering the scene to a false colour index buffer. Specular surfaces are given a unique index, while diffuse surfaces are assigned a default 'background' index. This index buffer is then read back to the CPU and reflected rays are only calculated for visible specular pixels. Thus the z-buffer is used to determine correct occlusion between specular and diffuse surfaces. The specular texture is created with an alpha channel which is used to correctly overlay the existing diffuse framebuffer with the specular viewport.

To account for the occlusion of the added dynamic object this code was changed. The dynamic object was rendered to the index buffer using the background index in addition to all the existing static geometry. This meant that where the dynamic object occluded a specular surface it would be correctly 'cut-out' during the specular rendering stage and thus not overlaid during composition of the final image (Figure 21, middle).

[17] Single CPU AMD Athalon 2500+, 1024MB.

The dynamic object can then be rendered as usual to the frame-buffer using flat shading. The OpenGL depth buffer will account for the occlusion between dynamic and static diffuse surfaces.

Figure 21 Changing the low quality progressive mode to account for dynamic object occlusion. Render the dynamic object to the frame-buffer (left, before rendering static diffuse surfaces). Account for the dynamic object when rendering the specular index buffer (middle). Final composite (right).

This process can summarised by the following pseudo-code. Changes to the existing method are *in italic*.

- Render static diffuse textures to frame-buffer using OpenGL
- *Render dynamic object to frame-buffer using OpenGL*
- Create view-port sized specular texture
    - Render specular surfaces with false index colour using OpenGL, *render dynamic object with background index colour to account for specular occlusion using the depth buffer*
    - Lookup specular pixel values using interpolation from VLF
- Write specular view-port texture to the frame-buffer using alpha channel to overwrite existing pixels where required.

The actual code can be found in Listing 2.

3.3.1.2 Dynamic object occlusion in high quality progressive mode

In high quality mode the specular occlusion is calculated differently but otherwise the method is similar. The execution proceeds in a separate thread to improve performance on multi-processor computers. For each pixel in the image a ray is traced into the scene. If it hits a diffuse surface the background index is assigned to an indexMap. If it hits a specular surface the ray is reflected and tracing continues. When a diffuse surface is finally intersected the intersection point is transformed into the relevant texture map to determine a colour value. This is assigned to a specular map. When the specular view-port is created the index map is used to determine whether to render a specular pixel. If so the pixel is set to the value from the specular map and the alpha channel value is set to 1.0. Otherwise the pixel and its alpha are set to 0.0. The specular texture can now be alpha blended with the diffuse frame-buffer as before.

Figure 22 Changing the high quality progressive mode to account for dynamic object occlusion. Render the dynamic object to the frame-buffer (left, before rendering static diffuse surfaces). Account for the dynamic object when rendering the specular view-port (middle). Final composite (right).

The pseudo code for this plus the change required for correct dynamic object occlusion (Figure 22) is:

- Render static diffuse textures to frame-buffer using OpenGL
- *Render dynamic object to frame-buffer using OpenGL*
- In a separate thread
    - Create a hiQualityIndexMap and hiQualitySpecularMap by recursively tracing a ray per image pixel
    - *Per pixel, if a specular surface is hit by a primary ray, then also intersect the ray with the dynamic object. Compare the intersection depth and if the dynamic surface occludes the specular surface then set the hiQualityIndexMap for this pixel to the background index.*
- Create a specular view-port texture by writing an (alpha =1.0) pixel from the hiQualitySpecularMap when the corresponding hiQualityIndexMap indicates.
- Write specular view-port texture to the frame-buffer using alpha channel to overwrite existing pixels where required.

The actual code can be found in Listing 3.

3.3.2 Flat shading and occlusion in coherent mode

The coherent mode outlined in Section 2.4.2.2.4 is the fastest, highest quality rendering mode for the VLF solution. Diffuse surfaces are handled the same way as for the progressive mode. But coherence between adjacent specular rays is exploited by calculating the intersections of bundles of 4 rays in parallel using the SIMD instruction set available on modern CPUs. To maintain the data rate required by such parallel processing carefully optimized cache aligned binary BSP trees are constructed from the scene description. These trees are represented in the sg::fastTriangleIntersector class.

The primary focus of this project is on shading a diffuse dynamic object at interactive rates. Reflections of the dynamic object in the static specular surfaces is an important visual effect – but to implement reflections at interactive rates it would be necessary to represent the dynamic object geometry in some form of the fast triangle intersector class. It was decided this would be left as future work as optimised ray-tracing data structures are not a core Virtual Light Field concept, and the whole of the allotted time could be absorbed by implementing optimized C++ code for ray-tracing rather than concentrating on rendering surfaces from the information in the VLF.

Because of this decision ray intersections performed on the dynamic object were executed by existing - non-parallel - ray-intersection code (sg::faceset:: faceIntersection(…)). To determine the correct occlusion of the dynamic object in front of coherent specular surfaces, it was necessary to intersect each primary specular ray with the dynamic object. This means for each ray in a 4 bundle of coherent rays, the fast parallel intersection tests are being interrupted to switch back to a non-parallel, non-optimized intersection with the dynamic object. This causes a serious loss of performance, but still returns higher frame rates than the progressive mode for equivalent scenes. The performance degradation per frame is greater as the proportion of visible static specular pixels covering the image plane increases. This performance impact only occurs for primary specular rays. To account for dynamic object reflections each bounce of each ray would have to be interrupted. Were this to happen, the frame rate would likely fall to non-interactive rates. Hence the decision to leave out dynamic object reflections at this stage.

This decision has another efficiency implication. Dynamic objects would not need to be shaded on surfaces facing away from the camera. This is explained in Section 3.6.5.

The code which performs the coherent specular rendering is in rayTraceSpecularDiffuseCoherentRT(…) in vlfGlutWalkThrough.cpp. The basic premise is the same as for the progressive mode. When rendering the specular viewport map, don't render pixels which would be occluded by the dynamic object. This is determined through ray intersection depth comparison between the static specular surface and the dynamic surface.

Here are the code additions required for dynamic object occlusion. The existing code is edited for brevity.

```
bool rayTraceSpecularDiffuseCoherentRT(…)
{
   … // cut
   for( // 2x2 ray bundles)
   {
         … // cut setup code
         // ray structures for additional dynamic object test, origin will not change
         glib::ray3D<MY_FLOAT_TYPE> ray=glib::ray3D<MY_FLOAT_TYPE>();
         ray.Origin(eye);
         glib::intersectionInfo<float> dynIx;
         unsigned dynIxFace=0;
         … // cut
         // loop over bundles, performing intersection tests in parallel
         // here the additional code for interrupting one ray in the bundle is given
         // in the actual code this occurs four times in total (once per ray per bundle)
         … // cut
         if (RAY01_ENABLED(currentRayBundle.m_flag) && RAY01_HIT(currentRayBundle.m_flag))
            {
               … // cut

               if( // surface hit is non-reflective (~diffuse) )
               {
                  DISABLE_RAY01(currentRayBundle.m_flag);
                  … // cut
               }
               else
               {
                  // Hit specular - but first check if dyn object occludes hit point
                  // this ray test is much slower than the fast triangle intersector
                  // which is a potential speed-up
                  // if we hit the dynobject first then take same action as for diffuse
above

                  if(showDynamicObject)
                  {
```

```
                        // assign ray an orientation from eye through pixel

   ray.Orientation(glib::vector3D<MY_FLOAT_TYPE>(currentRayBundle.m_dirX[0],
                                                 currentRayBundle.m_dirY[0],
                                                 currentRayBundle.m_dirZ[0]));

                        // intersect with faceset - allow RT to continue as before if there is
no hit

                        //  if  less-than  dyn  obj  occludes  so  take  same  action  as  above,
otherwise continue
                        if ( (depth==0)
                           &&
(dynamic::myIndexedFaceSet.faceIntersection(dynIxFace,ray,&dynIx,0.00001f,true))
                           && (dynIx.t() <= currentRayBundle.m_tIx[0])
                           )
                        {
                           DISABLE_RAY01(currentRayBundle.m_flag);
                           RAY01_CLEAR_HIT(currentRayBundle.m_flag);
                        }
                        else
                        {

                           // previously the only behaviour in this loop...
                           // Hit specular - ray continues - don't mark ray as hit yet, wait
till diffuse is found

                           RAY01_CLEAR_HIT(currentRayBundle.m_flag);

                        }
                     }
                     else {
                        RAY01_CLEAR_HIT(currentRayBundle.m_flag);
                     }
               }
      }
}
```

3.4 Simple shading for the dynamic object

At this point we have a cube rendered in a uniform colour, with correct occlusion in both the progressive and coherent modes of the VLF walkthrough. From this point onwards, all changes are applied to the coherent rendering mode, as it is the quickest, and we have decided to ignore dynamic object specular reflections. The flat coloured surfaces don't look realistic when compared with the surrounding globally illuminated static geometry. The object retains the same illumination wherever it is in the scene. A good first improvement would be to shade the object as though it was reacting to the primary light emitters in the scene. The simplest approach is to approximate the light sources in the scene with OpenGL lights, and use the Gouraud shading built into OpenGL to shade the cube. This provides a fast rendering of the cube.

Gouraud shading is a simple method for calculating local diffuse reflections. Perfect diffuse surfaces obey Lambert's cosine law, which states that the light intensity reflected from a point is proportional to the cosine of the angle between the surface normal and the direction to the light. The observer position doesn't have to be accounted for because perfectly diffuse surfaces scatter light equally in all directions. This is expressed by the following formula:

$$I = k_a I_a + k_d \sum_{i=1}^{N} I_{pi} \bullet (n \bullet l_i)$$

Where

I is amount of light reflected by the point on the diffuse surface

k_a is the co-efficient of ambient reflection for the surface

I_a is the total amount of ambient light

k_d is the co-efficient of diffuse reflection for the surface

n is the surface normal (a normalized vector)

l_i is the vector to the ith light source

I_{pi} is the normalized intensity of the light energy from the ith light source

N is the number of light sources

By picking suitable diffuse (Id) and ambient (Ia) light values for an omni-directional OpenGL light source and placing it at the centre of the emitter polygon, we achieve the results in Figure 23. where the surfaces facing the primary light source appear brighter than those facing away. The ambient light value stops the cube from appearing black where the face normals point away from the light sources.

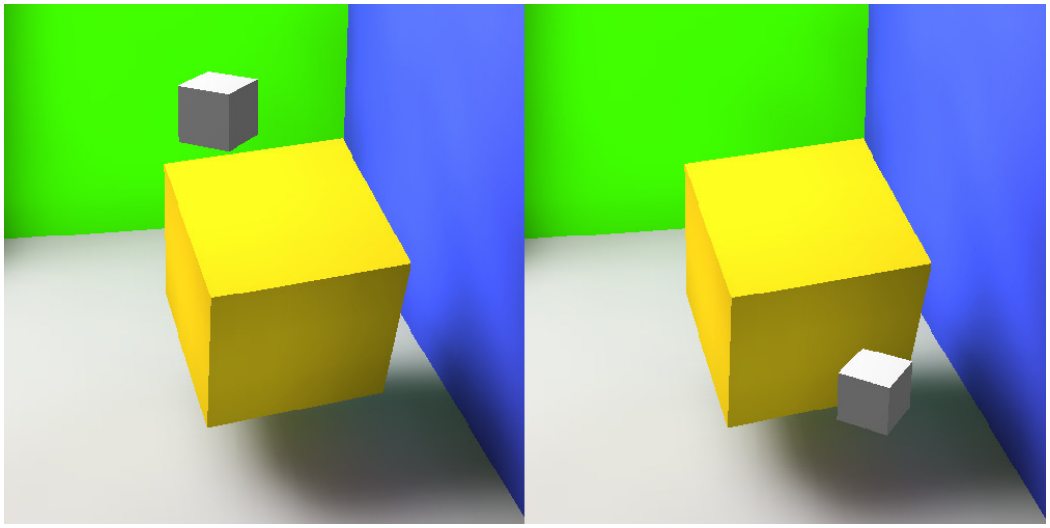

Figure 23 Object shaded using an approximate OpenGL light. Note however that the object doesn't repond to shadows cast by scene geometry.

This approach has the advantage of being fast, but the object doesn't react to the static scene geometry. For example in the right hand screenshot of Figure 23, the dynamic white cube sits in the shadow of the static yellow cube. However the illumination looks almost the same as when it is directly in the path of the light rays from the primary light source. Also this simple shading takes no account of indirect light and so dynamic surfaces don't exhibit colour bleeding.

The ambient and diffuse light values for the light source in this approach were picked manually. One area for further work would be to estimate suitable values for the diffuse and ambient light settings from the data in the VLF.

When implementing this method it was important to calculate the correct surface normals for the dynamic object. This is because the surface normal (n) is used in Gouraud shading in OpenGL to determine the appropriate colour value per polygon vertex, as shown in the formula above. Because the dynamic object can move we have to apply the same transformation matrix used on the dynamic vertex positions to the dynamic surface normals. However, normals become skewed when operated on by

translating or scaling matrices. To avoid this, the surface normals need to be operated on by the rotational part of the transformation matrix only. The code to achieve this existed partly in the VLF code-base already, but was unused, and so a new method (transformedFaceNormalWC) had to be created to access it. The relevant code for this approach is in Listing 4.

3.5 Casting shadows

Aside from the shading, something else sticks out from viewing Figure 23. The dynamic object in the left hand screenshot looks as though it could be directly above the static yellow cube. Or it could be further back in the scene and not vertically above the yellow cube at all. The text states that the dynamic object in the right hand screenshot is in the shadow of the yellow cube, but you can't tell this just from looking. The object could easily be much smaller and closer to the observer's eye point and not under the yellow cube at all. One way to remove the ambiguity is to have the dynamic object cast a shadow onto the static scene geometry, to provide a valuable positional cue.

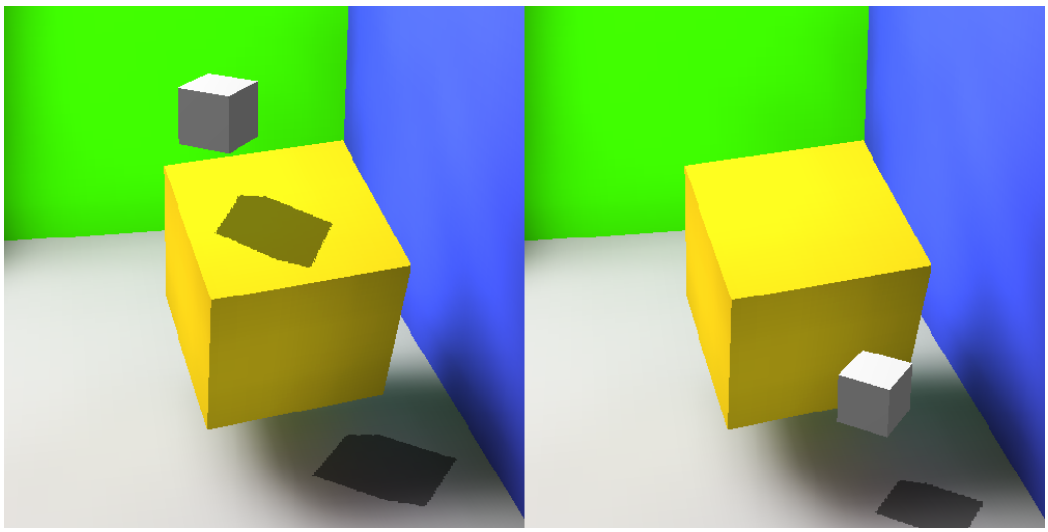

Figure 24 The use of shadow mapping to produce hard shadows gives a good positional cue. Note however the shadow 'piercing' effect in the left screenshot discussed in Section 3.5.2

Two methods of approximating shadows are discussed in Section 2.3: Shadow mapping; and shadow volumes. I chose to implement shadow mapping. This seemed a natural choice as the routines required are implemented efficiently on modern graphics processing units (GPUs) and GPUs provide the fast rasterization already required for calculating occlusion along rays in the VLF. Additionally the processing required for shadow volumes scales with the square of the number of surfaces, which would make it slow per frame for large number of dynamic surfaces. Shadow mapping works in the image plane rather than object space so clipping and visibility calculations reduce the amount of work required per frame. However shadow volumes have the advantage of simplifying the calculation of shadows reflected in specular surfaces (which is not implemented as part of this project). The reflection of dynamic objects and their shadows is expanded on in the further work section at the end.

3.5.1 Shadow mapping implementation

The technique is outlined in Section 2.3.1 and explained in detail in Everitt et al [18]. The detailed explanation here draws from Everitt, but differs from shadow mapping in general. Usually the technique is:

1. Render the scene to the frame buffer from the eye point of view in a dim light. This dim illumination will form the shadowed areas.
2. Render to the shadow map (depth buffer) from the light point of view.
3. Render the scene again, but render only those portions of the scene indicated to be unshadowed (using the transformed light depth buffer) using a normal level of illumination.

However, we already have the scene rendered at the normal level of illumination (step 3) in the form of diffuse total radiance maps from the VLF. So all we want to do is reduce the level of illumination in the areas of the existing static diffuse surfaces which fall into shadow due to the dynamic object. The approach I use is to overwrite the fragments in the scene with black where they are shadowed. This is then improved using alpha blending (See Sect 3.5.3).

To determine which areas fall in shadow we need a depth map rendered from the light view (stored as a texture) and a method of transforming vertex co-ordinates in the eye view into this depth (texture) map . An efficient method for doing this is to use the OpenGL texture co-ordinate generation facility (texgen). This facility generates texture co-ordinates from other vertex attributes. In particular the GL_EYE_LINEAR texgen mode transforms the eye space vertex position according to a user specified matrix (T_e also known as eye linear texgen planes). If we consider the various co-ordinate systems and transforms between them, as shown in Figure 25 below:

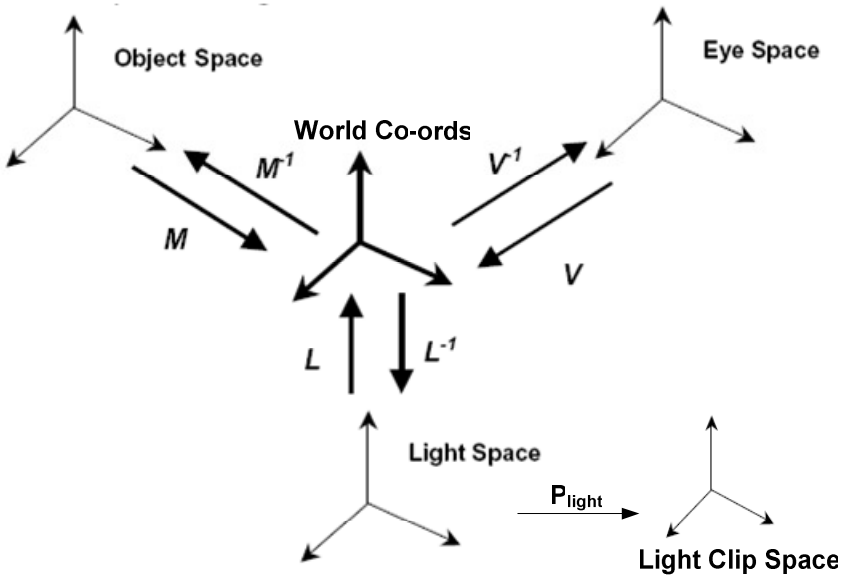

Figure 25 The co-ordinate systems and transforms involved in shadow mapping. The standard OpenGL modelview matrix would be $V^{-1}M$ using these conventions.

We can see the required combination of transforms to map from eye space to the light clip space is:

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = SP_{light}L^{-1}V \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Where P_{light} is the projection matrix for the light frustum. After division by perspective the co-ordinates in clip space are in the range [-1,1]. But the texture map is addressed by the range [0,1]. So we need to premultiply by a scale matrix S where:

$$S = \begin{pmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

OpenGL will apply two transformations to the eye co-ordinates during rendering. The *texture matrix T* and the *texgen planes (Te).*

OpenGL automatically multiplies the eye co-ordinates by the inverse of the modelview matrix in effect before multiplying by T_e and storing the result when T_e is specified. So if we set the modelview matrix to contain the camera's view matrix (V^{-1}) when specifying T_e we eliminate having to calculate V and we can simply leave the texture matrix T as the identify matrix and specify the eyelinear texgen planes in OpenGL as:

$$T_e = SP_{light}L^{-1}$$

The pseudo code for this implementation is as follows:

During scene initialisation call dynamic::setupShadowLight(…) . This will:

- Determine the position of the first light source in the scene
- Determine the centre point of the dynamic object
- Create a glib::camera<> object at the light position, pointing at the dynamic object. With a suitable shadowmap resolution for the image plane.
- Use OpenGL and the GL_MODELVIEW_MATRIX matrix mode to calculate the matrix L^{-1} referred to as glLightViewMatrix in the code
- Similarly calculate the light projection matrix P_{light} referred to as glLightProjectionMatrix in the code
- Generate a texture to hold the shadow map

Then once per frame recalculate these parameters by calling dynamic::recalculateShadowLight(…). This will:

- Re-position the light camera
- Recalculate glLightViewMatrix
- Use OpenGL calls to calculate T_e, referred to as glTextureMatrix in the code.

- Determine the front clipping plane for the current light view to avoid rendering back projected shadows (see Section 3.5.4 below).
- Render the view from the light to obtain the depth buffer
- Copy the light view buffer into the shadowMapTexture for later use

Finally to render the shadow call dynamic::renderShadow(…) with the current modelview matrix set to the camera view. This will:

- Use glTexGen* calls in GL_EYE_LINEAR mode to generate the texture co-ordinates which translate the eye view depth value into the shadow map depth buffer for comparison.
- Setup the shadow map comparison using glTexParameteri calls.
- Apply the front clipping plane to avoid backward shadow projection
- Enable shadow blending using alpha channel.
- Render areas on static diffuse surfaces which should receive shadow.

The code for the shadow mapping implementation is in Listing 5.

3.5.2 Shadow ‘piercing'

Usually in shadow mapped scenes all shadow illumination is calculated in one pass per light source. So if two objects project overlapping shadows the overlap area has the same shadow intensity. However, as soft shadows are already incorporated in the diffuse texture maps generated from the light propagation, we can easily see where the shadow 'pierces' a static object and overlays the existing soft shadow behind a static object (See Figure 24 above). This effect is not physically correct. Only the first surface ‘hit’ by a shadow projection should be hard shadowed, the current implementation does not take account of this. This is because if the projected hard shadow was to cease as it enters the penumbra of the already present soft shadow on the static texture map the perceptual effect would be even more jarring than allowing this ‘shadow piercing’ to occur.

The contrast between existing soft shadows and hard shadows is reduced by introducing shadow blending (see next section).

3.5.3 Shadow blending

In real scenes a shadow isn’t completely black. Some of the surface properties of the shadowed surface contribute to the light received by the eye. So a red surface doesn’t appear completely black once it’s in shadow, it just appears a much darker red. We can preserve the existing shadowed surfaces properties in OpenGL by using alpha blending when writing the hard shadows to the image buffer.Note that some of the existing yellow colour is preserved inside the hard shadow cast on the static yellow cube (Figure 26, right).

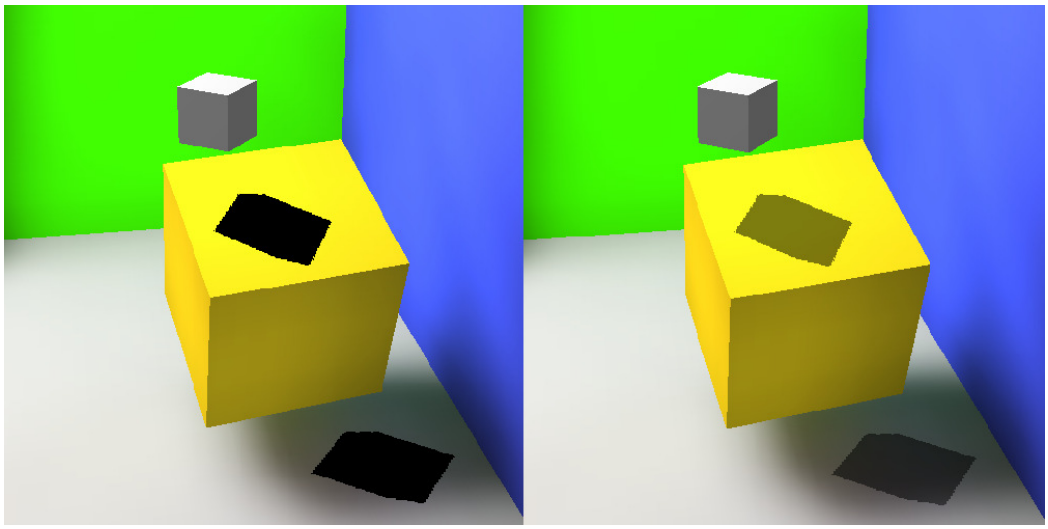

Figure 26 Hard shadows with (right) and without (left) alpha blending.

The following snippet demonstrates setting OpenGL blending function to retain some of the destination texture (the pre-rendered static surface).

```
   glEnable(GL_BLEND); // blend the shadow rather than overwriting the existing colour.
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   for(...) { // loop over diffuse surfaces in scene
      ...
      glColor4f(0.0, 0.0, 0.0, 0.5); // use alpha to preserve some destination colour
(blending)
      // render diffuse surfaces
   }
   glDisable(GL_BLEND);
```

3.5.4 Removing the reverse projection of the shadow

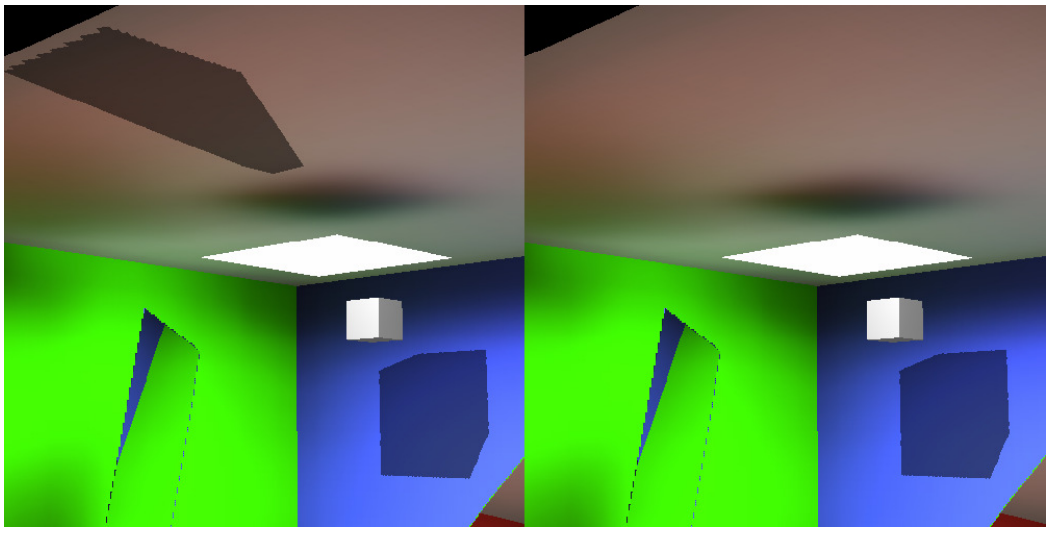

Figure 27 The back projection of the shadow (left). Back projection removed (right).

Due to the projection technique used the reverse projection of the shadow will be formed in the scene unless some measure to remove it is taken. In this implementation a clipping plane is placed at the cameras view plane. However only surfaces facing away from the light direction will cause a reverse projection. So a better method is to cull all back facing surfaces with respect to the light direction. This can be considered to be a special case of the shadow piercing which can be

removed because the effect is physically incorrect and it doesn't add any perceptual benefits (unlike for surfaces with normals facing the light source, where we don't want the hard shadow to stop suddenly where the slow gradient of the shadow penumbra starts).

3.5.5 Shadow map resolution

As mentioned in 2.3.1 shadow maps can be under-sampled which leads to jagged edges. This is illustrated in Figure 28. Increasing the shadow map resolution addresses the problem.

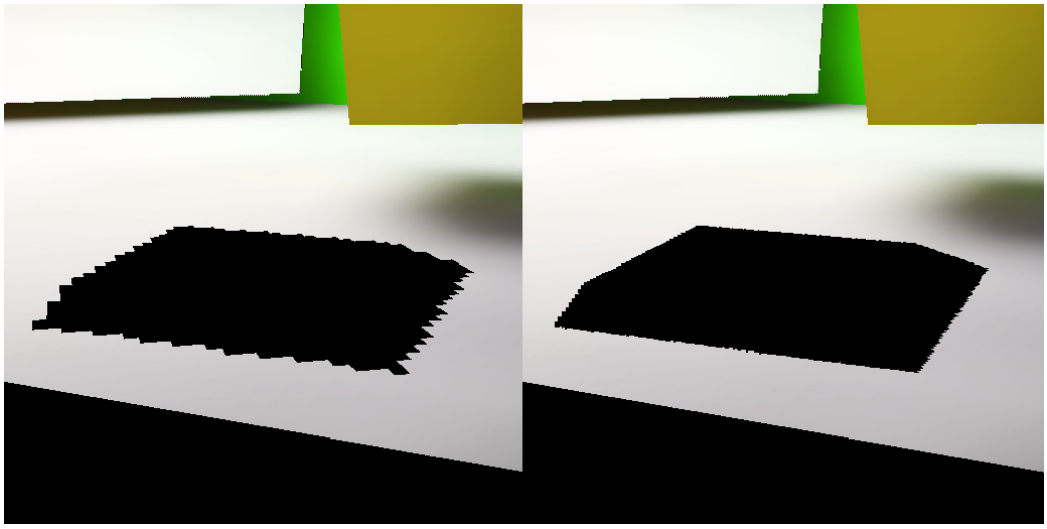

Figure 28 Increasing shadow map resolutions from 128x128 to 512x512 addresses the problem of under-sampling.

3.6 Advanced diffuse shading using the VLF

Our static object now casts hard, blended shadows onto static geometry. It is shaded using Gouraud shading due to an OpenGL approximation of the scene light source. This combination of techniques is fast so we can render the object at an interactive rate (see Section 4.1 for timing results). These techniques also have limitations. The dynamic object doesn't respond to occlusion from the static scene (it doesn't receive shadows) and it doesn't display the effects of diffuse inter-reflection. This section describes an improvement to the shading implementation which addresses these two drawbacks – at the expense of rendering time. However the overall frame time with the new technique can still be considered interactive for some situations (see 4.1.2). We also consider potential optimizations.

3.6.1 Theory

From the radiance equation in section 1.2 we saw that the radiance leaving a point on a surface in a particular direction was a sum of the radiance emitted at that point and the integral of all the incoming radiance over the hemi-sphere above that point. The integral includes a Bi-Directional Reflectance Distribution Function (BRDF) which relates the radiance along the incoming direction to the radiance in the outgoing direction. For a non-emissive surface (any surface other than a light source in our scene description) the first term is zero, leaving us with the integral only. Also from section 3.4 we know that perfectly diffuse surfaces scatter light equally in all directions. Thus the BRDF term is a constant value independent of incoming light

direction for the perfect diffuse surfaces that the VLF models. This leaves us with the following formula for the outgoing radiance in any direction at point p:

$$L(p) = \frac{\rho_d}{\pi} \int_{\Omega} L(p, \omega_i) \cos\theta_i d\omega_i$$

Where $\frac{\rho_d}{\pi}$ is the term the BRDF reduces to for a perfect diffuse surface and ρ_d is the diffuse reflectance of the surface, which is a user specified value.

The radiance equation is recursive. For example we desire the radiance L(p) in any direction at a point p on a particular diffuse surface. This depends on the integral on the right hand side (RHS) of the equation, and this integral depends on the incident radiance over the hemisphere above p. If we trace back any one of these incident rays to its originating surface (say this is also a diffuse surface) then how do we determine the correct outgoing radiance? We have to solve the integral again for this new set of incident rays. The VLF has already solved this recursive problem for us. To shade the dynamic object we wish to know the correct radiance values for the hemisphere of incident directions over each sample point on the dynamic object surface. This is the value returned via a lookup along an existing direction in the VLF – we don't need to solve more than one integral per sample point and each integral consists of the same number of terms as there are directions in the VLF hemisphere. The integral becomes a simple addition of these lookup values divided by the domain of the integral.

Therefore a fast method of approximating the integral for a specific point on the dynamic surface using the VLF data structure would be to determine which incident directions from the VLF exist in the hemi-sphere above the surface normal. The incident radiance at that point from the nearest intersected surface cell along each direction can then be looked up from the VLF and the sum total incident radiance calculated.

This method only works one way. We are *gathering* the incident radiance on the dynamic surfaces, but we are not then *shooting* this back into the scene. Nor are we altering the textures in the scene to account for the fact that the gathered radiance will now not reach the static surfaces beyond the dynamic surface for each gathered ray. However the effect of shadow mapping is a rough approximation of the effect the dynamic object would have on the scene.

The vlf::face structure can store a total diffuse radiance texture map for each face in the dynamic object. Ideally we would calculate the incident radiance for each texel in this texture map. However given the large number of texels in a reasonable resolution texture map this would be expensive. A quicker approximation is to calculate the integral for the vertices of the face polygons only. This value can then be used for Gouraud shading as before. This is the approach implemented here. Enhancement by increasing the number of sample points is discussed later.

When calculating the integral we need to account for the discrete nature of the VLF data structure. Specifically two types of normalisation need to be considered.

3.6.2 Normalising for variation in solid angle in VLF directions.

The PSF directions used in the VLF are calculated based on a recursive subdivision of a regular tetrahedron [33]. As discussed in 2.4.1.1 this is to allow constant time lookups of the nearest direction to an arbitrary vector. However this

subdivision scheme leads to as much as 60% variation in the size of solid-angle formed by the directions in the VLF (Figure 15). This leads to an unequal distribution of radiance from a cell over the hemi-sphere of directions during VLF propagation.

In practice this means that the sampling of the radiance in the scene is biased more towards some directions than others. This will affect our dynamic object shading because in determining the colours to shade the surfaces we are attempting to reconstruct the correct incident illumination from the samples of radiance available in the VLF. To counter this biased sampling each PSF direction is given a normalised weight based on its solid angle. This value is used to weight radiance arriving along this direction during the calculation of the integral. This is the relevant code snippet from vlfGlutWalkthrough.cpp

```
// weight by area of PSF triangle on hemi-sphere
tmpcol = (tmpcol*pd->areaWeight());
```

3.6.3 Normalising for the total projected unit cell area over hemisphere

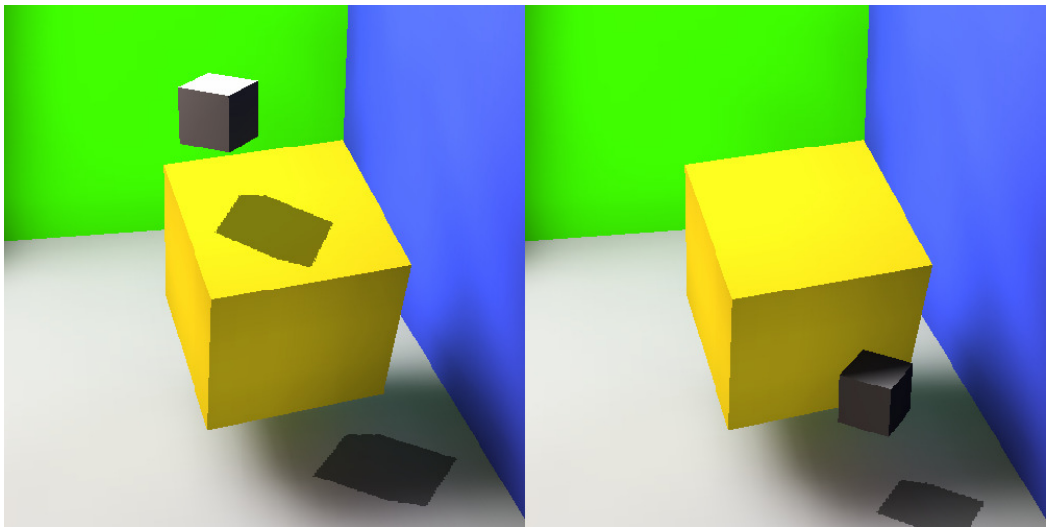

Figure 29 The result of itegrating the irradiance over the hemisphere above the surface normal at each polygon vertex. Diffuse reflectivity = 1.0. Note the dynamic object now receives shadow from the static scene (right).

Traditionally when integrating the irradiance at a point over the hemisphere above that point we consider the incident rays to be infinitely thin. However for the VLF each radiance sending cell has a discrete area. If we project the discrete area for each sending cell onto the unit hemisphere above the receiving surface point then the areas overlap and the domain of the integral (the sum of these areas) is much greater than the area of a unit hemisphere. To account for this we calculate a 'total projected unit area' which is the sum of the projected area of a unit cell projected into each PSF. This value multiplied by the actual area of a sender cell gives the domain for the integral.

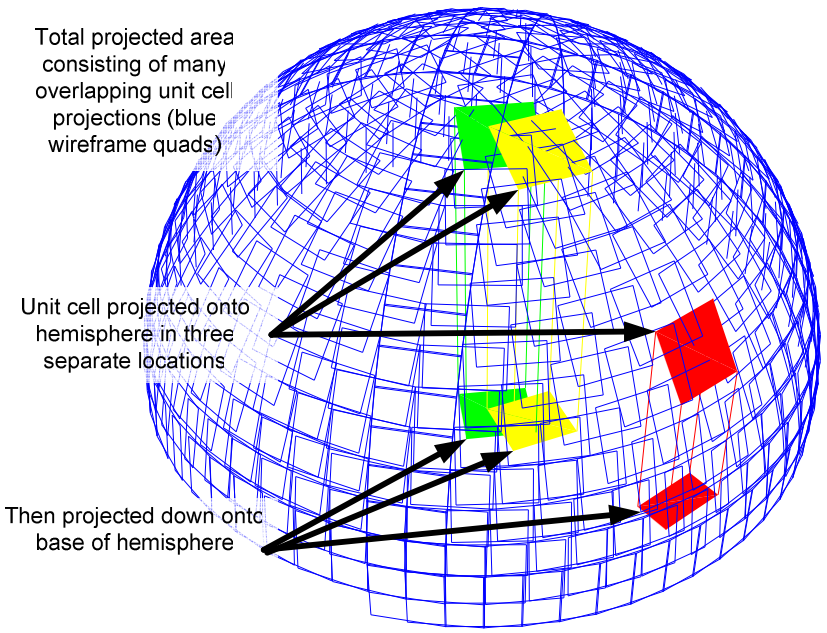

Figure 30 Visualisation of the total projected unit cell area over the hemisphere. Note the contribution (area of unit cell projection onto 2D base plane) of directions forming a greater angle with the surface normal falls as the angle increases. Here the direction corresponding to the red cell projection contributes less energy than the green or yellow directions.

We also need to account for the $\cos\theta$ term in the integral. This term accounts for the fact that sender surfaces which are at greater angles from the receiver surface normal contribute less energy to the receiver cells than those directly overhead.

When calculating the total projected unit area the unit cell is projected onto the hemisphere and then back down to the plane of the surface where the area is measured (Figure 30). This value is a better approximation of $\cos\theta$ than using just the angle between the sender and receiver surface normals because it accounts for the finite area of the sender cell. The code was changed to record this value for each PSF direction. This value can then be used as a quick lookup for the proportion of energy received by the receiver surface as long the correct corresponding PSF index can be found. This is achieved by considering the receiving surface normal as being co-incident with the Z-axis of the canonical PSF frame of reference and looking up the PSF index which corresponds to the direction to the sender surface in this frame of reference via the following process.

1. Determine the vector pointing from the receiving point to the sender surface in WC (vdir in code).
2. Transform this vector into the canonical *face* frame of reference UVN. Where the receivers face normal is parallel to the +tive Y-Axis.
3. Rotate this vector into the canonical *PSF* frame of reference. This is a 90 degree rotation around the X-axis of the canonical face frame of reference, as the canonical PSF frame places the hemisphere in the +tive Z space above the X-Y plane.
4. Use this transformed vector to lookup the corresponding PSF index.

5. Use this index to lookup the pre-calculated energy proportion required.

Figure 31 visualises this set of co-ordinate systems. The transformed vector pointing to the sender cell is coloured in red in each frame. The cells projected onto the hemisphere have been made smaller to avoid overlapping for the sake of clarity. This also highlights the uneven sampling density over the hemisphere which is address by the normalisation covered in Section 3.6.2. The projected cells are shaded from green to black. This shading corresponds to relative contribution each direction makes to the overall integral (in effect this is $\cos\theta$ varying from 1 (strong green) directly above the receiving point to 0 (black) as the angle increases).

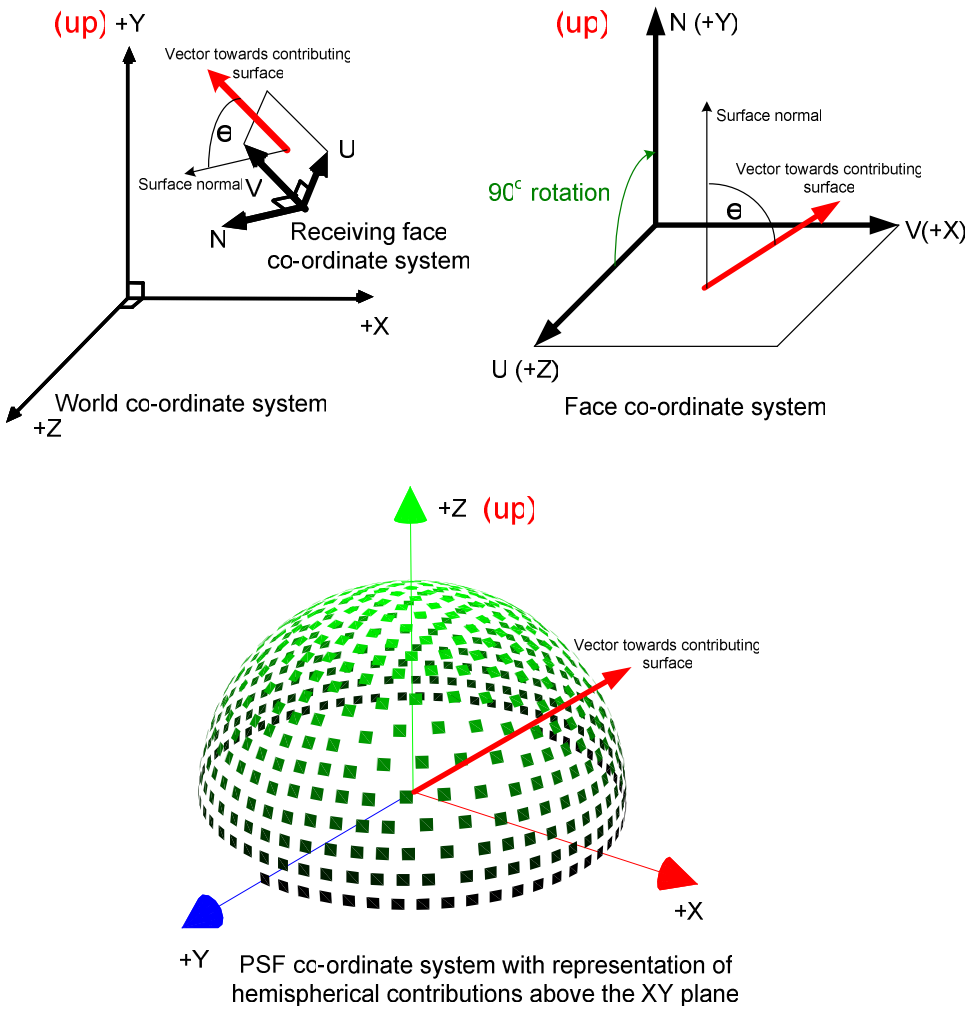

Figure 31 Visualising the co-ordinate systems and transforms involved in accounting for the cosine term in the integral (see text).

3.6.4 Implementation of advanced shading

To summarise each of the steps in the method outlined above here is the pseudo code for shading the object from the VLF.

- For each face
  - Determine the correct surface normal

- If the face isn't visible to the camera, next face. (*this has implications for reflections in specular surfaces – see 5.1.1).*
- For each pre-calculated subdivided triangle within the face
    - For each vertex in the triangle
        - Transform to correct location for dynamic object
        - If the colour value for this vertex has been cached then use the cached version, else gather the incident light…
        - For each direction in the VLF
            - Check direction is incident on surface, invert if not
            - Take vector from receiving point to sending cell and transform into the PSF co-ordinate system, to lookup the cosine weighting.
            - Sample the radiance from the sender cell
            - Normalise by the PSF area weight (account for variation in solid angle
            - Weight by cosine term
            - Divide by sender cell area (to account for actual cell projected area rather than unit cell projected area)
            - Accumulate result into total radiance sum (col in code).
        - Multiply the sum total by rho_d/PI and normalise by total unit cell projected area
        - Cache this value because it will be shared by other subdivided triangles.
    - Draw the triangle with Gouraud shading.

This cost of this algorithm is O(number of directions in VLF * number of visible sub-divided vertices * log(average number of faces per tile in VLF))

The relevant parts of the code edited for brevity are included here:

```
for( //each face in dynamic object)
{

glib::colourRGB<MY_FLOAT_TYPE>
rho_d=((VLF_MATERIAL_TYPE*)dynamic::myIndexedFaceSet.faceMaterial(f))-
>kd()*((VLF_MATERIAL_TYPE*)dynamic::myIndexedFaceSet.faceMaterial(f))->diffuse();

...

for( // each vertex for each polygon for each face ) // full code on CD too long for here
{
for( // each direction in the VLF )
{

MY_DIR_TYPE* pd = myVLF.directions().item(di);

...

glib::vector3D<MY_FLOAT_TYPE> vdir=glib::vector3D<MY_FLOAT_TYPE>(pd->point());
glib::ray3D<MY_FLOAT_TYPE> ray=glib::ray3D<MY_FLOAT_TYPE>(vertex,vdir);

...

sg::plane<float> plane;

// determine the matrix which rotates a vector in WC into the canonical face frame (Y-axis
parallel to face normal)
```

```
plane.transformMatrix( dynamic::myIndexedFaceSet.getFace(f)->toCanonical() );

// use the transformNormal method to apply only the rotation portion of this matrix to the
vdir vector

glib::vector3D<MY_FLOAT_TYPE> hdir= plane.transformNormal( vdir );

// the hdir vector now points in the direction to the sender surface in the canonical face
frame.
// however we need to rotate this to be in the canonical PSF frame

// rotate this vector 90 degrees around the X axis so it's now in the PSF canonical frame
(+tive Z-axis is top of hemisphere)

hdir = (hdir.homogenize() * rot90aroundX).dehomogenize();

// hdir now points to the PSF direction corresponding to the cosine scaling factor we need.

hdir.normalize();

// find VLF direction corresponding to vector from receiver to sender
myVLF.directions().item(pHemiDir,hdir);
hdir = pHemiDir->point();

...

result= myVLF.getSampleAndFaceIndexByRaycast(tmpcol,ray,faceInd);
...
// weight by area of PSF triangle on hemi-sphere
tmpcol = (tmpcol*pd->areaWeight());
// also weight by fraction of projected unit cell on hemisphere (roughly a cosine term over
the hemisphere)

projArea = dynamic::PSFUnitProjectedArea.at(pHemiDir->m_linearIndex);
...

float areaDMactual=(actualV*actualU)/static_cast<float>(NsU*NsV);

tmpcol *= projArea / areaDMactual;
col+=tmpcol;

...
} // end loop over all directions for this vertex

col=col/static_cast<float>(myVLF.totalProjectedArea() );

//we want L = rho_d/pi * integral
// sum terms of integral.
col = (rho_d / static_cast<float>(PI)) * col;
} // finished gathering irradiance for this vertex
} // finish face from dynamic object
```

3.6.5 Reflections of the dynamic object in the static scene

In section 3.3.2 the modifications implemented for correct occlusion of static specular geometry by a dynamic object are described. The dynamic object is intersected with each primary specular ray that hits a specular surface. However the dynamic object is not intersected with subsequent reflections of the specular ray, and so does not appear as a reflection in specular surfaces. As mentioned it would be simple enough to intersect the object at each level of recursion and so account for the reflection, but because the intersection code for the dynamic object isn't optimized for coherent ray-tracing this would badly affect frame rendering times. Given that we are not rendering reflections then we don't need to gather light for vertices facing away from the camera, which speeds up the light gathering process. If the dynamic object intersection code is optimized in the future and interactive reflections are possible then the correct reflection may require gathering of light on surfaces facing away from the camera. At this point level of detail optimizations (see 3.6.6) would be useful as reflections are often much smaller than the rendered object.

3.6.6 Increasing the number of sample points

As we are gathering irradiance at the vertices of face polygons a simple way to increase the resolution is to use triangular subdivision to form more sample points at the mid-point of each existing triangle edge (Listing 6). The subdivision vertices are pre-calculated and stored in memory, so during run-time the user can select the level of subdivision to use for shading. This approach gives us the flexibility to implement some form of level of detail shading, where if the polygons project onto a large enough area of the camera image plane a higher level of subdivision can be selected for shading. This could be implemented as future work.

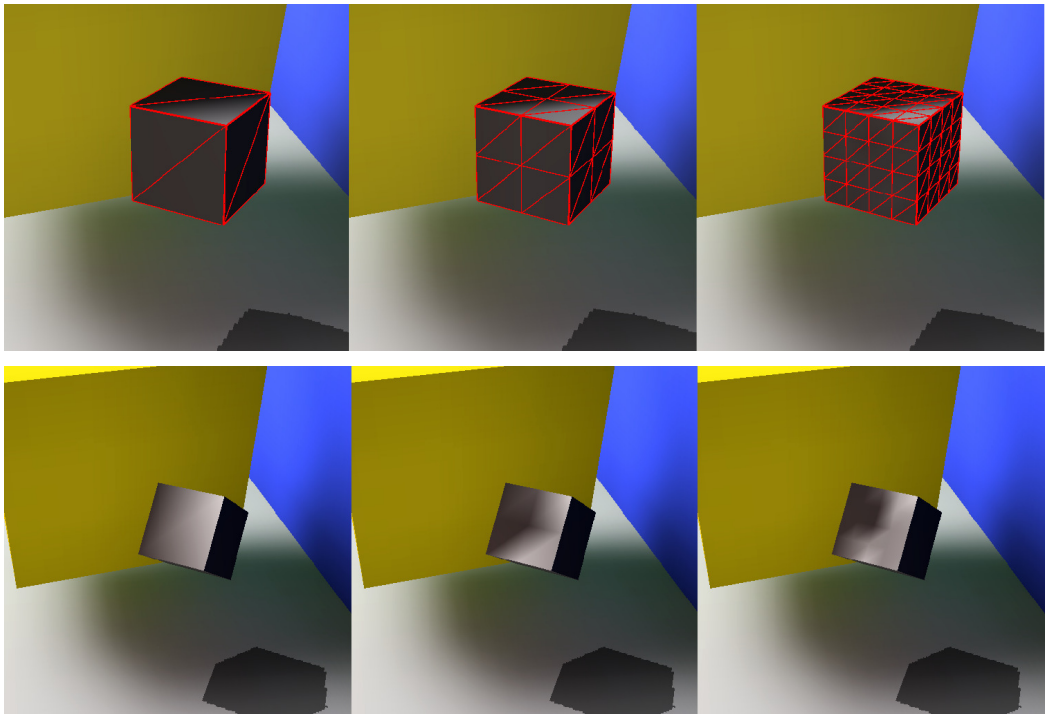

Figure 32 Upper row: triangle subdivision at levels 0,1,2. Lower row: the effects on shading

Triangle subdivision is impractical after 3 levels of subdivision as the vertex count increases exponentially[33]. Specifically the vertex count V at subdivision level l (where l=0 is two triangles per cube surface) is:

$$V = \frac{(2^l + 1)(2^l + 2)}{2}$$

Additionally it could be argued that perceptual advantages due to the higher resolution lighting effects received on the cube surface are not worth the expense of the (much) lower frame rate. This is discussed further in the results section.

When considering sampling the irradiance at every texel in a diffuse radiance texture for the dynamic object it is worth noting that if each PSF is considered for all texels at once we could perhaps project each visible intersected sender surfaces total radiance map for that PSF onto the receiving surface. This may be possible to implement efficiently in hardware using multiple passes to build a texture. This is mentioned in the further work section.

3.6.7 Decreasing the number of sampled directions

The innermost loop in the algorithm outlined in 3.6.4 is over each of the directions of the VLF. A crude method of reducing the cost of this algorithm is to sample only every other direction. Because the indexing scheme is based on a tetrahedral subdivision of the hemisphere this biases the result because it will skew the sampling density over the hemisphere, but it may be useful in some cases. Useful further work could be done to apply some form of importance sampling at the stage to reduce the number of directions contributing and thus speeding up the algorithm.

3.7 Scene energy balance considerations

During the integration of light from the VLF we have ignored the fact that the irradiance receiving point should also be considered to possess a finite area. The received irradiance per incoming direction should be multiplied by the area of the intersection between the projection of the senders cell onto the receivers cell area. This is how the propagation of diffuse energy proceeds in the full VLF solution – however to achieve this requires polygon clipping, which is expensive. Roughly 80% of the propagation phase for the VLF consists of clipping operations. This is a necessity for the full VLF solution because light is propagated through many iterations and so errors in propagation are accordingly amplified.

For the dynamic object we consider only one iteration of gathering irradiance onto a diffuse surface. Additionally we only wish to give the impression of correct irradiance to a casual human observer. For one iteration only the errors accumulated are small compared to the speed increase gained by avoiding clipping. For these reasons the intersection area of projected cells was ignored. It is assumed that the area of intersection equals the area of the projected cell.

Additionally the dynamic object is intercepting light in the scene. The rays whose paths are truncated by the presence of the dynamic object should be followed to the next surface intersection and the corresponding amount of radiance removed in order to maintain the existing energy balance in the scene. Such an operation would be expensive to implement as outlined here, although perhaps it could be achieved via a technique similar to incremental progressive radiosity [23]. For our purposes it was reasonable to ignore this effect, as it is assumed the surface area of dynamic objects would be small compared to the scene. The use of shadow mapping approximates this effect for direct light rays, but not for indirect rays.

3.8 Summary

This chapter described in detail the steps taken to add new movable geometry to existing VLF solutions. The approach proceeded via a number of milestones beginning with the addition of a simple static wire-frame cube to the existing geometry rendering mode. The ability to move this object interactively was then added. This object was integrated into the more complex progressive rendering mode which rendered the static geometry using the GI solution but the dynamic object with flat shading. Occlusion had to be handled separately for diffuse and specular surfaces and separately for the low and high quality specular rendering modes. The complex coherent rendering mode offers further speed gains over the progressive rendering mode and so the object was integrated with correct occlusion into this mode too.

Once using the fastest rendering mode (coherent) we could address the illumination effects caused by the dynamic object in the scene. Shadows provide an

important depth cue and so the ability for the dynamic object to cast shadows was added. This was implemented efficiently in graphics hardware using the shadow mapping technique. The dynamic object also needed to be shaded realistically. An efficient first approximation using an OpenGL light was implemented. This didn't account for shadows cast onto the dynamic object or diffuse inter-reflection between static and dynamic faces. A better but slower technique was to integrate the incident light on the dynamic object using the data from the VLF. This allowed the dynamic object to receive shadow and diffuse inter-reflection, but not to send diffuse inter-reflections into the scene. The sampling resolution of this technique was improved through triangular subdivision – at the expense of rendering time.

4 Results

The implementation section described in detail the enhancements made to the existing VLF walk-though code base in order to enable a moveable object at interactive rates with varying degrees of integration in the scene. These degrees of integration are summarised in the following tables.

Dyn Obj shading	Geometry	Progressive	Coherent
Wireframe	✓		
Flat shading		✓	
OpenGL light			✓
VLF light			✓

Shadow mapping plus:	OpenGL light	VLF light
Dyn Obj appearance		
Receives shadow		✓
Receives diffuse interreflection		✓
Static scene appearance		
Receives shadow	✓	✓
Receives diffuse interreflection		

In this section we present the results of these enhancements and attempt to measure how successfully the aims were achieved.

As described in Section 1.5 the aim of this project is to achieve realistic rendering of a scene containing a dynamic object which can be moved at an interactive rate. These aims can be evaluated through both quantitative and qualitative tests. Quantitatively we can measure the time taken to render frames of different types of scene in different modes and determine the time contribution of each stage of rendering. Qualitatively we can judge how 'realistic' the desired effect (shadows, diffuse inter-reflection) looks. Such qualitative judgements can be subjective, but for a given dynamic object position we can render a similar static object using the full VLF method and judge how close the dynamic object shading mimics the full 'correct' solution.

When discussing interaction we need to define what constitutes an interactive frame rate. As frame rates vary depending on hardware platform and we have only one platform to test on we will define the tests relative to the performance of the original code without the dynamic object extensions. If we assume that in coherent mode the original VLF team could achieve 20 frames per second for a simple scene on their target hardware (as is claimed in [2]), then if we achieve 2 frames per second in coherent mode without enabling any of the dynamic object enhancements for a similar scene on the available test hardware we can take 2 frames per second as being an interactive rate. This relies on the assumption that there is a constant scale factor for performance between these test platforms, which is unlikely given the nature of PC hardware (different chipsets, cache sizes, memory, GPUs etc) however it is adequate for our purposes. In fact 15FPS is considered an acceptable interactive frame rate [12] so any adjusted rate consistently above 15FPS is accepted as interactive.

All tests were executed on a single processor home PC with the following spec. CPU: AMD Athalon 2500+ Main RAM: 1024MB. GPU, Nvidia Geforce 6600GT RAM: 128MB. It should be noted that the progressive mode was designed to render the hi-quality specular surfaces in parallel on a dual processor PC so timings could be improved on a more suitable test system.

4.1 Frame rendering times

The original VLF walkthrough implementation achieved 20 frames per second in coherent mode on a dual Xeon 1.7Ghz workstation for a simple office scene. With 128x128 rays per PSF, 16x16 tiles per PSF and 2049 directions. To keep propagation time and memory size requirements manageable on the our lower spec test

workstation all test scenes used here were rendered with 513 directions with 8x8 cells (rays) per tile and 8x8 tiles per PSF.

To record video a simple animation recording and playback system was implemented. Keyboard control of motion was recorded from the user and then played back with one dynamic object transformation per frame rendering pass.

Timer code was used to record the time taken for each render stage per frame. Each frame was rendered at least three times (and some times as many as 55 times) and then times were averaged to produce the results below. Animations were designed to exhibit the strength and weaknesses of the algorithms used. Video of the animations is available on the CD.

4.1.1 Progressive rendering frame times[18]

Flat shading of the dynamic object, correct occlusion and shadow mapping were implemented in the progressive mode before development moved onto the coherent mode.

The overall time taken per frame when producing a rendering in progressive mode consists of:

1. Time taken to render static diffuse surfaces (texture map)
2. Time taken to render dynamic object flat shaded surfaces
3. Time taken to prepare and render dynamic shadows from the shadow map
4. Time taken to calculate and render the specular viewport texture

The average taken for step 1 & 2 combined for any test scene (of roughly 20-40 polygons) with a 512x512 pixel viewport was approximately 1 millisecond. The time take for step 3 was roughly one tenth of a millisecond. These low times are due to the accelerated OpenGL hardware used and are clearly within interactive frame rate requirements and so will not be explored further.

Step 4 is by far the most time consuming as one would expect as it involves determination of and lookups along a PSF directions in the low quality mode and recursive ray-tracing in the high quality mode. It increases as the number of specular pixels covering the viewport increases. It is this step we shall examine in more detail. We wish to test how the addition of a dynamic object to the scene impacts the specular rendering time. Three tests were performed on the same scene (Figure 33).

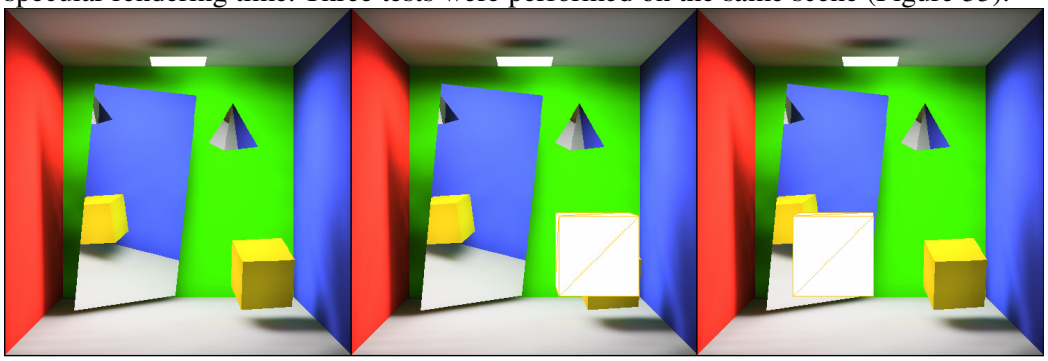

Figure 33 The progressive mode test scene for specular timings. No object (left), specular surface uncovered (middle) and covered (right)

[18] Results here will not be optimum because the test hardware does not have dual CPUs and therefore doesn't exploit the parallel code for rendering the high quality specular mode.

Each test was performed three times and the results averaged to reduce the impact of other background processing due to the operating system etc. Error bars on the graph indicate the width of one standard deviation in the measurements. The tests were:

1. Time taken to render specular pixels with no dynamic object code used. (this is equivalent to the original code base and so is our control case).
2. Time taken to render specular pixels with dynamic object present but not covering the specular surface.
3. Time taken to render specular pixels with dynamic object present and covering the specular surface.

Tests 2 & 3 will help determine the effect the occlusion calculation method has on the render time. We consider the high quality and low quality render times separately. As high quality mode is only used when the object is at rest the timings were taken without object motion with and without the dynamic object code included.

4.1.1.1 Low quality progressive mode frame time

We expect that the addition of a dynamic object has negligible effect on low quality specular frame time when the specular surface is not covered because the same number of specular pixels have to be looked up in the VLF in both cases.

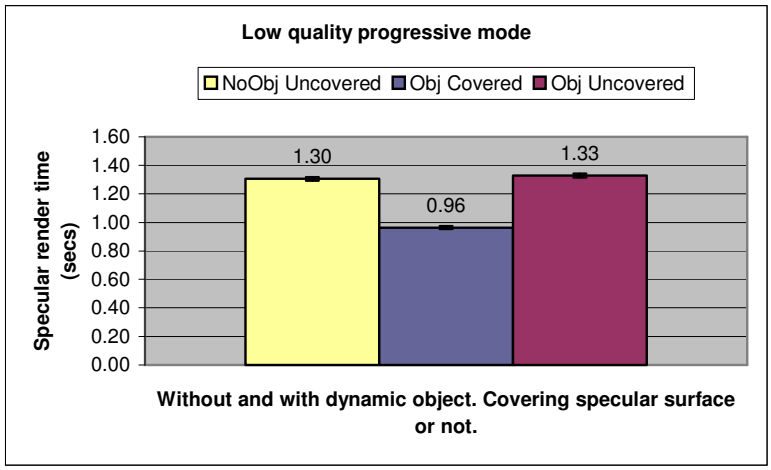

Figure 34 Specular render time results low quality progressive mode

We expect the specular frame time to reduce if the dynamic object occludes the specular surface, because the index map created by rasterising the scene using OpenGL will contain less specular pixels, and so less specular lookups are required. These expectations are confirmed by the results of the test illustrated in Figure 34.

4.1.1.2 High quality progressive mode frame time

We expect that the addition of a dynamic object will increase high quality frame time whether the specular surface is occluded or not. This is because every primary specular ray needs to be additionally

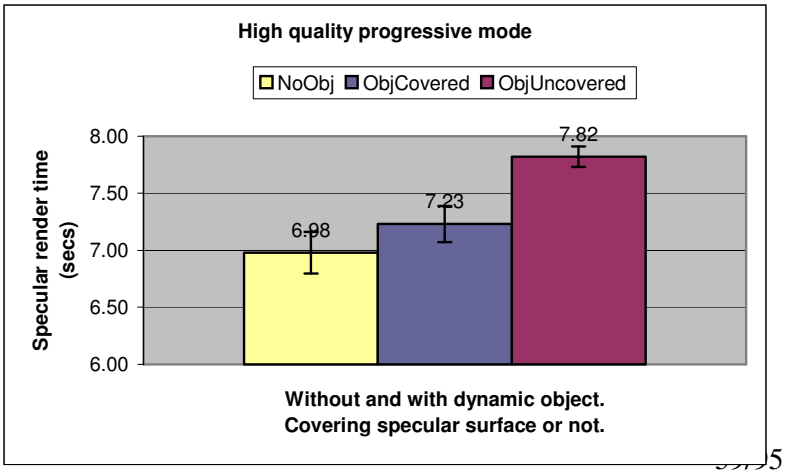

Figure 35 Specular render time results high quality progressive mode.

intersected with the dynamic object to determine occlusion. When the surface is occluded no further recursion of the specular ray is necessary and so we expect a faster render time than when the specular surface is occluded.

These expectations are confirmed by the results in Figure 35. A better method of determining specular surface occlusion for primary specular rays (such as the rasterised index map used in the low quality mode) would improve this method and reduce the frame time overall. However as specular render time in coherent mode is a great improvement on the progressive mode these times are no longer a concern.

4.1.2 Coherent rendering frame times[19]

We have established that shadow mapping and diffuse texture mapping in OpenGL take around 1 millisecond and so are well within interactive frame rate requirements. Similarly measuring the time taken for the approximation of the light source using an OpenGL light to produce approximate diffuse shading in coherent mode has a negligible effect (tenth of a millisecond). So we are only concerned with the effect the dynamic object has on specular surface render time and the amount of time it takes to integrate the light from the VLF in the advanced shading mode.

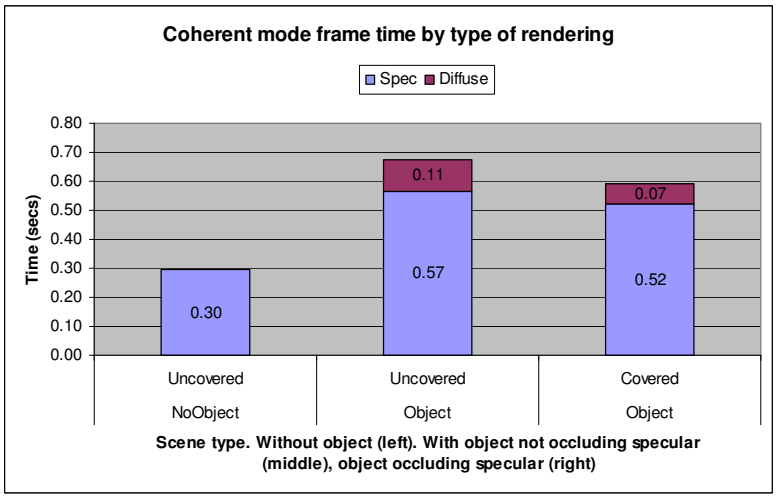

Figure 36 Coherent frame render time results

The specular occlusion for the dynamic object in coherent mode is calculated using a similar technique to the progressive mode and so we expect similar results. I.e. adding the dynamic object will have an adverse effect on specular pixel rendering times, and the effect will be worse when the object doesn't occlude the specular surfaces. The same test scenes were used as for the progressive mode, but 55 frame times were averaged to get the results (the resulting standard deviations ranged from 2% to 0.8% which are too small to show on the resulting graph). The dynamic object is rendered using light integrated from the VLF and shadow mapping.

The results (Figure 36) confirm the expectation. However the degradation of specular rendering time with the object as opposed to without is much worse than in the progressive case (in the non-occluding case coherent specular rendering takes 200% of the time taken for the original code without the dynamic object, as opposed to 112% in the similar progressive case). This is because the highly optimized SIMD intersection code used in the coherent mode is being interrupted once per primary specular ray to perform a slower (non-optimized) intersection with the dynamic

[19] Results here will not be optimum because the test hardware does not have dual CPUs and therefore doesn't exploit the parallel code for rendering the coherent specular map.

object. Thus much of the advantage in parallelising the task is lost. Either a better method of occlusion determination for specular surfaces (similar to the low quality progressive) or implementing the dynamic object intersection routines using the coherent method would improve the result. The reasoning for the approach taken here was discussed in section 3.3.2.

The time taken for the diffuse gathering in the un-occluded case is roughly 30% of the time required for the specular rendering in the case of the original code. If we equate the original 0.3 seconds per frame to being 20FPS on the test hardware from the original paper and ignore for now the slowdown caused by the non-optimized specular occlusion then the this would provide 0.41 second per frame and an adjusted rate of just under 15FPS. This is just outside the definition of interactive we set out at the beginning of the section but it is an encouraging result because no particular optimisations have been applied to the diffuse result so far, and the visual results are perceptually acceptable (as shown in the next section).

The time taken for the diffuse gathering in the occluded case is roughly 2/3 of the time for the un-occluded case. This is due to the face that only 2 faces of the cube are visible in this case instead of 3 as in the un-occluded case. This is in-line with the linear scaling in the number of visible vertices predicted for the diffuse gathering method (see section 3.6.4).

In section 3.6.6 triangular subdivision is used to increase the number of sample points. We expect the diffuse gathering time to scale linearly in the number of visible vertices. To test this the same scene was used with increasing depth of triangle subdivision. The ratio of number of samples to time remains constant as expected at roughly 10ms per sample (Figure 37).

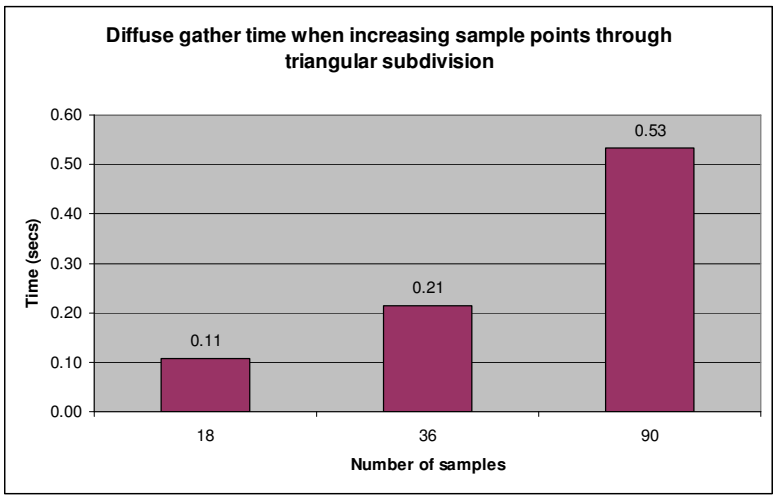

Figure 37 Increasing number of diffuse sample points

In section 3.6.7 the simple optimization of reducing the number of directions sampled by only considering every other direction was implemented. This does halve the diffuse gather time as expected. The visual effect can be seen in section (4.2.5).

4.1.3 Quantitative Summary

There are a large number of factors affecting the rendering time. In particular scenes with larger numbers of visible specular pixels will take much longer than those without when ray-tracing is used (high quality progressive or coherent) rather than lookup from the VLF. This scales exponentially depending on the depth of recursion required and is the case whether dynamic objects are used or not. The test scenes used contain relatively large specular reflectors and are probably not typical of many

scenes used in VR experiments. Even so the method used for determining the correct occlusion of the dynamic object with regard to specular surfaces should be improved to scale more predictably. A primary ray rasterization method such as that used for the low quality progressive mode could be implemented with little further effort.

However for the reasons given in section 3.3.2 specular intersections were not the focus of this work. The results for the diffuse shading methods suggest if colour bleeding and receiving shadow on the dynamic object are not important then the OpenGL approximation of the light sources plus shadow mapping is easily fast enough to provide interactive frame rates on our test hardware. If advanced shading is required then the integration of light from the VLF is a suitable method for simple objects (with low visible vertex counts), but as the gathering time scales linearly in number of directions and vertices then for higher resolution VLF scenes or more complex dynamic objects the method will eventually no longer provide an interactive rate. The failure point can be postponed simply by halving the number of sampled directions, at a cost in visual accuracy due to the sampling bias this introduces. There is a large scope for further optimization discussed in section 5.1.

4.2 Qualitative Tests

Having discussed the cost for each method we can now look at the visual effect produced. The screenshots throughout the project give a good impression of each effect in isolation. In this section we concentrate on judging the accuracy of and integrated impression given by the advanced dynamic shading in coherent mode including shadow mapping. Where relevant we will compare the dynamic object to a pre-rendered scene using a similar static object in the same position to see how closely the dynamic shading matches the ‘correct’ shading solution. As the project is interactive there are some videos demonstrating the object moving on the attached CD.

4.2.1 Luminance of dynamic diffuse cube

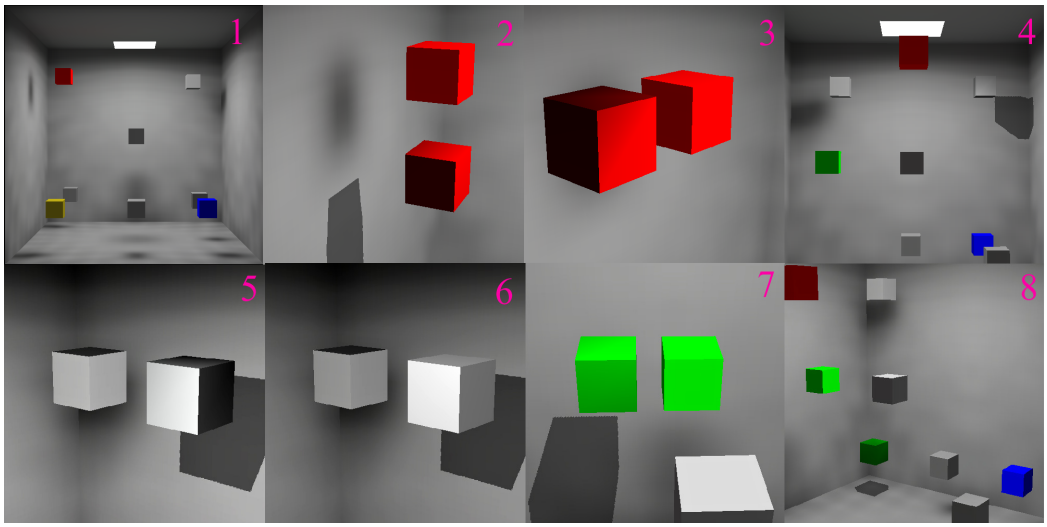

Figure 38 The luminance test scene. The dynamic object can be determined by it's hard shadow. See text for descriptions.

A diffuse scene containing cubes with different diffuse material properties in different positions was pre-rendered using the VLF. The same properties were then assigned to the dynamic cube and it was placed in similar positions in the scene and a

subjective judgement was taken by the author on how well the object matched the 'correct' solution. A more rigorous approach using ROC (receiver operator characteristic) curves and a selection of subjects willing to judge the results could have been used had more time been available.

Figure 38 is a selection of comparisons showing the dynamic cube in various positions and materials similar to nearby pre-rendered static cubes. 1. Is the whole scene without the dynamic cube. 2 and 3 show the dynamic cube in red near a static cube of the same colour and reflectivity. Note the hard shadow contrasts strongly with the soft shadow but gives a useful positional cue. In 3 the dynamic cube is on the left. The top surface is darker than the equivalent static object but the match is otherwise good and certainly the object looks to 'belong' to the scene. Differences are due to the rougher approximations being made during the integration. Specifically the projected intersection area between sender and receiver cell is not being calculated. 4 places a white dynamic cube at the top right corner for comparison with a similar static cube at the top left corner. The hard shadow shade is a good approximation of the soft shadow shape and the luminance is well matched. 5 and 6 place the dynamic white cube next to its static counterpart at the top corner of the scene. The shading in 6 uses the OpenGL light approximation for comparison with the advanced shading in 5. You can see the sides pointing away from the light are more accurately shaded for the advanced method. 7 and 8 place a green dynamic cube next to its counterpart, and then down in the bottom corner. Note how the dynamic object becomes darker the further it is from the light.as you would expect. Again the colour match in 7 is accurate.

4.2.2 Colour bleeding for dynamic diffuse surfaces

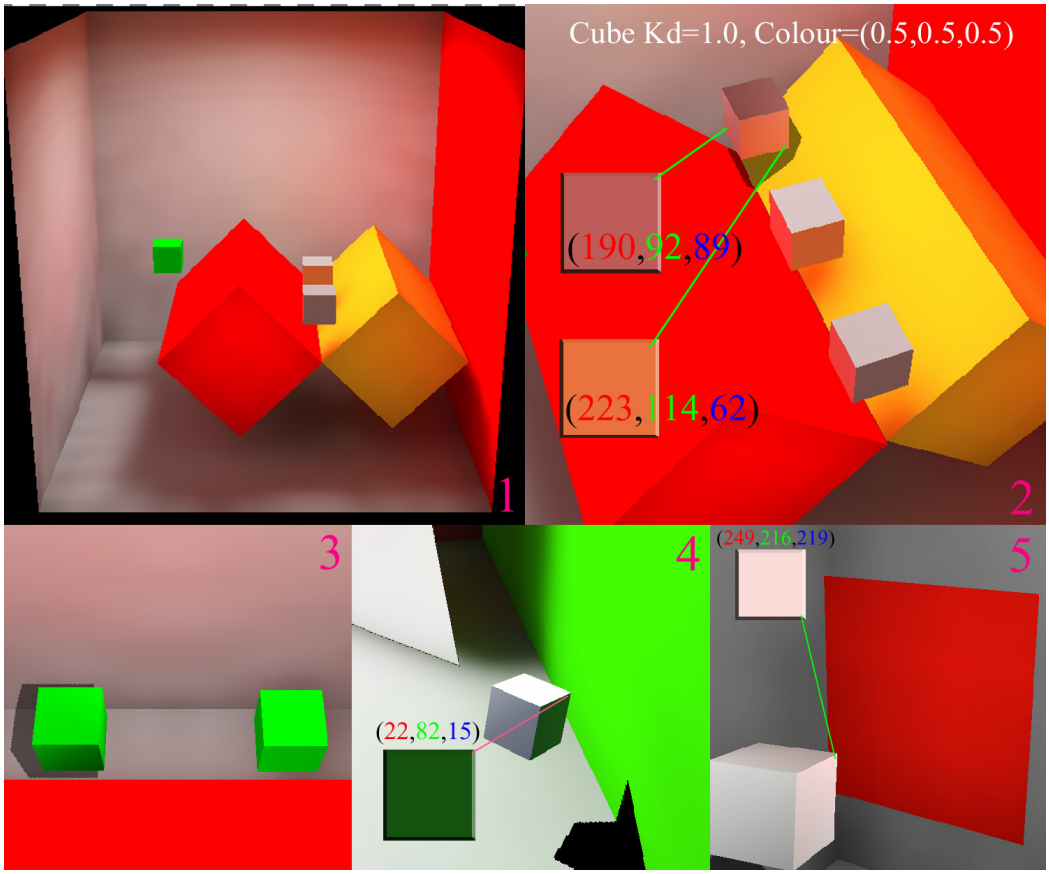

Figure 39 Demonstrations of diffuse inter-reflections. Pixel colours extracted and illustrated on a 2D slab where appropriate.

The advanced shading by integrating light from the VLF is the only implemented shading method to account for diffuse inter-reflections.

Figure 39 (1) is the scene rendered to test diffuse inter-reflection. (2) shows a dynamic white cube near to static white cubes with the same material parameters. The overall appearance is very similar to the static 'correct' solution. The RGB values for the reflected colours are shown in (2). As expected there is a strong red diffuse reflection from the nearby red surface and a strong yellow/red reflection on the surface close to both red and yellow surfaces. The top surface of the cube is darker than for the static cubes and demonstrates that the approximations in gathering won't always provide an exact match. But perceptually the effect is acceptable. (3) shows a green static cube not picking up any diffuse red as you would expect (and similar to the static green cube nearby). 4 and 5 are from different scenes with the diffuse colour of the dynamic cube at RGB = (1,1,1). They also demonstrate believable colour bleeding.

4.2.3 Shadows cast by the dynamic object

The hard shadows cast by shadow mapping are a strong contrast to the already present soft shadows in the scene. From Figure 38 and Figure 39 it is easy to identify the dynamic object by the shadow it casts. This could distract from the realism of a dynamic scene, however techniques such as percentage closer filtering or multiple shadow maps blurred together could be implemented to create soft shadows at

interactive rates. From certain viewing positions the hard shadows appear jagged. This is a weakness of the shadow mapping method which can be addressed through increasing the resolution of the shadow map. This is illustrated in Figure 24.

4.2.4 Shadows received by the dynamic object

The advanced shading by integrating light from the VLF is the only implemented shading method to account for shadows received on the dynamic object. This can be seen by comparing the right hand screen shot of Figure 24 to the frames of the animation in Figure 40 below.

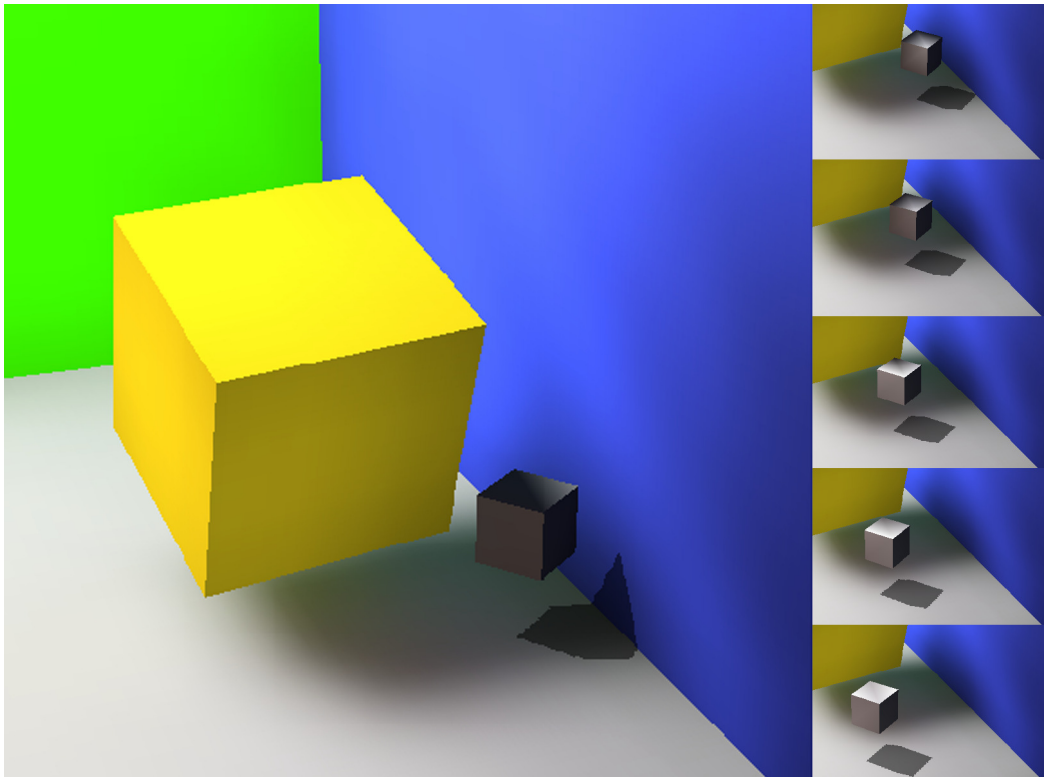

Figure 40 The advanced shading accounts for static object occlusion in the form of received shadow. Here dynamic cube moves from the shade into the light

Even at the lowest sampling rate (once per vertex per face, so the corners only in Figure 40) the impression of received shadow is strong. By using triangular subdivision we can increase the resolution of the received shadow . However in my opinion the visual benefit is often not worth the increase in render time (Figure 41). However the benefits of increasing the sampling resolution can be seen as the object passes through a shadow on the video from the CD.

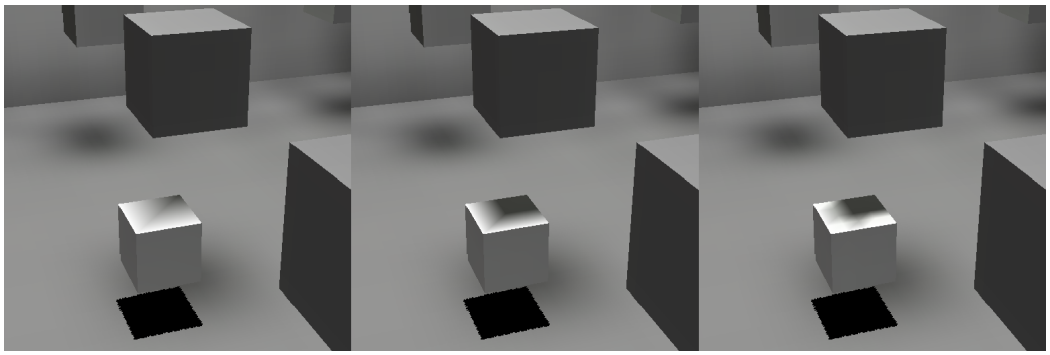

Figure 41 Shadows at 0,1,2 levels of triangular subdivision. For many applications the lowest level may be sufficient.

4.2.5 Sampling half the VLF directions

By simply sampling every other direction in the VLF (i.e. skipping every other linear index into the VLF directions) we can halve the time taken for diffuse gathering. This technique biases the result and a more intelligent scheme should be devised. However the effects are still acceptable (Figure 42) and this technique may be useful in some applications. A video demonstrating the technique is available on the CD.

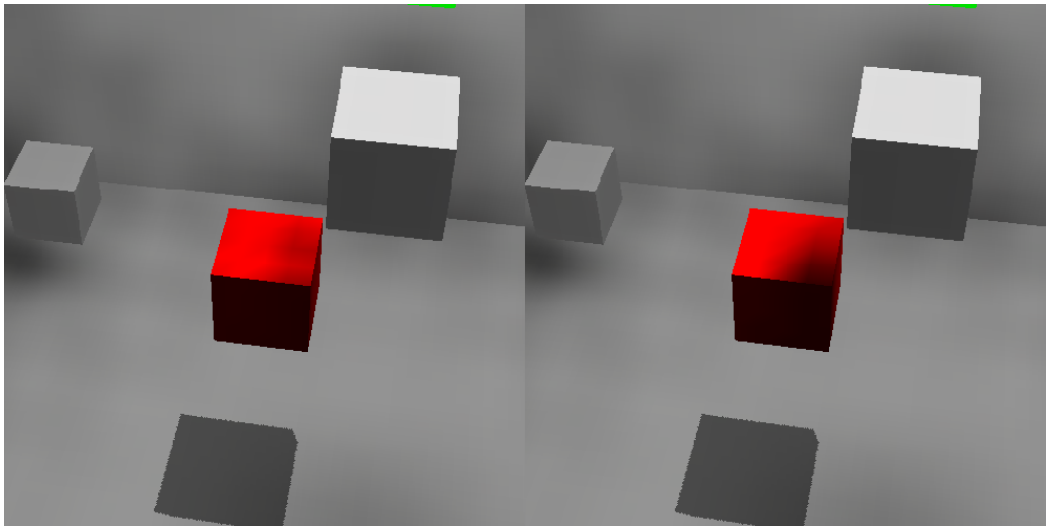

Figure 42 A red dynamic cube in shadow sampled along all directions at subdivision level 2 (left). Sampled along every other direction (right). Note the similar result.

4.2.6 Video

The accompanying CDROM disc contains some video demonstrating the dynamic object motion in real-time. These were recorded on the available test system and show the relative interactivity with different settings such as higher triangle subdivision or only sampling half the number of directions. The specular rendering loop was disabled for these videos as we wish to compare relative diffuse gathering performance.

4.2.7 Qualitative Summary

Two realistic modes of shading of shading have been implemented. The advanced VLF shading mode is superior to the OpenGL approximation mode because

it accounts for both diffuse inter-reflections and shadows (received on the dynamic object only in both cases). The only effect not approximated is the effect diffuse reflection from the dynamic cube would have on the scene itself. This would be expensive to implement and is left as a consideration in further work. Through comparison of the VLF dynamic shading mode with the full static solution it can be seen that the VLF shading mode produces results close to what would have been obtained for a full solution but at an interactive rate. Differences are due to the nature of the approximations used, but the realism of the overall visual result is more than acceptable.

5 Conclusions

In Chapter 1 we set out our aim for this project, which was to add movable geometry to the existing VLF solution which would be shaded realistically at interactive frame rates.

In Chapter 2 relevant existing literature was reviewed. The most relevant existing approach is incremental progressive radiosity, which depends on the hierarchical nature of existing efficient radiosity solutions and aims to provide a physically accurate result by changing the parts of the existing GI solution affected by the re-positioning of existing geometry. This project approaches the problem from the opposite starting point. Approximations of increasing complexity were used to integrate additional simple, movable geometry to the VLF solution.

In Chapter 3 the implementation of these approximations was described. The accuracy of these approximations increase with each milestone, culminating in an advanced VLF shading mode which integrates the light incident on the dynamic object from the VLF in order to shade the object in a manner that accounts for received diffuse inter-reflection and received shadow from the scene.

In Chapter 4 the results of these approximations were presented and discussed.

This final chapter draws together the conclusions from the work implemented in the project. Two main methods of integrating additional dynamic geometry to an existing VLF solution were developed.

The first (basic) mode shades the dynamic object by approximating the existing light source using an OpenGL light and allows the dynamic object to cast hard shadows into the scene using shadow mapping. This method would allow interactive rates on a high spec consumer PC such as that used in the original VLF paper[20]. However it doesn't account for shadows cast by existing geometry and it doesn't account for diffuse inter-reflection. Frame render times for scenes containing large or numerous specular reflectors would be badly affected in coherent mode due to the simple intersection routines used to determine occlusion. These drawbacks can be easily addressed with some further work. (See next section).

The second (advanced) shading mode uses data from the VLF to integrate the incident radiance on the dynamic object surfaces. This method is visually superior to the light approximation method because it accounts for shadows cast by existing geometry and accounts for diffuse inter-reflection received on the dynamic object. Shadow mapping is used again to cast shadows from the dynamic object into the scene. Frame rates in specular scenes suffer from the same specular occlusion determination issue as for the first method. The time taken by this method scales linearly in number of VLF directions, number of visible dynamic surface vertices and the log of the average number of polygons per VLF tile. This means that render time is dependent on the resolution of the VLF and the complexity of the dynamic object. Visual quality of the dynamic object shading can be altered by two parameters: The number of sample points (fewer is faster but less accurate), and the number of directions sampled (sampling only half the directions takes half the time, but is also less accurate). Interactive rates would be possible on the original VLF hardware[20] but

[20] Dual Xeon 1.7GHz with GPU

only for simple (low vertex count) dynamic objects and low VLF resolution (513 directions). However further work (next section) can improve frame rates.

The two methods provide two levels of visual quality. Either of which could be acceptable depending on the application. The simple mode sacrifices accuracy of shading for faster frame rates and scales well in the number of objects. The advanced mode provides dynamic objects which react more realistically to the existing light in a scene – but take far longer to render. The method doesn't scale well in number of dynamic objects given the relatively large amount of time taken to gather diffuse light on the object.

5.1 Further work

5.1.1 Specular surfaces

The VLF implementation this work was based on models only perfect diffuse or perfect specular surfaces. This project has focused mainly on perfect diffuse materials for two reasons: because diffuse surfaces are more common in natural scenes and therefore more likely to be of use in VR experiments; because the work required to implement diffuse shading involves a deeper understanding of the VLF techniques.

This focus means that two important areas of a global illumination solution have been put to one side. Neither method implemented here displays the reflection of the dynamic object or its shadow in the specular reflectors in the scene. Also neither method allows for dynamic objects with specular surfaces.

To achieve good quality specular reflections at interactive frame rates the existing VLF work draws on the coherent ray-tracing techniques of Wald et al [32]. Implementing specular dynamic objects and the allowing for the reflections of dynamic objects in existing specular surfaces at a similar frame rate and visual quality as the existing static surfaces would require defining the dynamic object in terms of an efficient axis aligned BSP tree and allowing intersections to proceed in parallel via SIMD. The fast triangle intersector class in the existing VLF code would be a good starting point for this further work.

When considering reflections using ray-tracing you need to be able to determine the colour at the intersection point on the dynamic object. For this reason it may be worth using the diffuse texture map assigned to each vlf::face object to store the results of shading the dynamic object. This has implications for the performance of the advanced diffuse shading mode because we could no longer avoid gathering diffuse light for vertices on surfaces facing away from the camera. As reflections are often small perhaps reflected colour could be approximated by gathering the light at only one point at the centre of each reflected vlf::face which doesn't face the camera to save time.

5.1.2 Static specular occlusion

Even without a full coherent intersection implementation for the dynamic object it is possible to address the slow-down caused by interrupting the existing coherent ray intersection implementation when determining occlusion of specular objects.

Instead of using ray-tracing to determine which primary rays hit a specular surface a rasterisation method similar to that used for the low quality progressive mode can be used to work out which pixels need ray intersection calculations with the static scene. This would avoid interrupting the coherent code with non-optimized

dynamic object ray intersection operations. However this wouldn't address the issue of reflecting the dynamic object in existing specular surfaces.

5.1.3 Dynamic diffuse occlusion

Currently there is no occlusion testing done before gathering the diffuse radiance for a vertex in the advanced shading mode. If this vertex is occluded from the camera view and doesn't contribute to the interpolation of a neighbouring vertex in the same face then it doesn't need a colour assigning.

5.1.4 Hard shadow reflections

Ray-tracing determines the reflections in specular surfaces and ray-tracing is an object space method. Shadow mapping is an image space method. As a result it is not trivial to reflect the shadows formed via shadow mapping in the specular surfaces of the scene. Shadow volumes would be a preferable approach from this point of view. However reflections of shadows from shadow maps could perhaps be rendered by rendering the shadow map from the reflected point of view for each mirror in the scene.

5.1.5 Soft shadows

The shadow mapping technique could be enhanced to blend together shadow projections from multiple points on a light source. Or a filtering technique such as percentage closer filtering could be used to blur the edges of the hard shadow. Either of these techniques could be implemented to take only a constant multiple of the time taken for hard shadows, which is still well within interactive frame rates.

5.1.6 Importance sampling

In this implementation the time taken for diffuse gathering in the advanced shading mode can be halved by sampling only half the incident directions. This method biases the result in favour of the directions lower in the hemisphere above a surface normal because of the design of the linear indexing into the hemisphere. More intelligent methods of importance sampling could be devised.

5.1.7 Caching

Once a point on a diffuse surface has had the light gathered the result could be cached along with the direction of the surface normal. When gathering light this cache could be visited first. The values could be stored in some form of space subdivision structure like a kd-tree. If a value is available within some threshold distance of the required point it could be reused. The angle of the required surface normal would also need to be considered. Perhaps radiance values over large solid angles over the sphere of directions at that point could grouped together. Such a solution would benefit applications where objects often passed the same set of points in space.

5.1.8 Exploit coherence in diffuse gathering

The current implementation gathers the diffuse radiance at the vertices of a polygon and interpolates between vertices. A higher resolution method would be to gather incident radiance for every texel in the diffuse texture map assigned to a vlf::face. However to loop over every VLF direction for every texel is much too expensive. Each PSF direction stores a radiance map of radiance travelling in that direction. Combined with the visibility calculation this map could be projected onto

the dynamic diffuse texture map for each direction in turn using the GPU. This may provide a crude approximation of the diffuse irradiance for all texels in a dynamic surface texture map at an interactive rate.

5.1.9 Parallel processing

Diffuse gathering and specular rendering could be performed in parallel. Or diffuse gathering could split between processors on a face by face basis.

5.1.10 OpenGL light parameter estimation

The basic shading mode uses an OpenGL light to approximate the first light source in the scene. This assumes only one light emitter. It is trivial to change the implementation to approximate more than one light source, but currently the lighting parameters are chosen manually. A method of approximating suitable values for the diffuse and ambient parameters of the light should be easy to implement.

5.1.11 Receiving hard shadows during basic shading

One drawback of the basic shading using an OpenGL light is that the dynamic object doesn't receive shadows from static geometry in the scene. It would be simple to extend the shadow mapping implementation to also render shadows from the existing scene geometry onto the dynamic object.

6 Bibliography

[1] Heckbert, P. **Adaptive Radiosity Textures for Bidirection Ray Tracing.** SIGGRAPH 24, 145-154. 1990.

[2] Khanna, P., Slater, M., Mortenson, J., Yu, I. **A Virtual Light Field for Propagation and Walkthrough of Globally Illuminated Scenes**. Proceedings of Computer Graphics International, 2004

[3] Slater, M., Steed, A., Chrysanthou, Y. **Computer Graphics and Virtual Environments, from realism to real-time.** Addison Wesley. ISBN: 0-201-62420-6

[4] Khanna, M., Mortenson, J., Yu, I. P., Slater, M. **A Visibility Field for Dynamic Ray Tracing,**, Interim Technical Report, http://www.cs.ucl.ac.uk/research/vr/Projects/VLF/Media/vlfDynamicRTpaper/index.htm, 2004

[5] Goral, C. Torrance, K., Greenberg, D. Battaile, B. **Modelling the interaction of light between diffuse surfaces**. Computer Graphics (ACM SIGGRAPH '84 Proceedings), vol. 18, pp. 212–222. 1984.

[6] Damez, C, Dmitriev, K and Myszkowski, K. **State of the Art in Global Illumination for Interactive Applications and High-quality Animations**. Computer Graphics Forum, 21(4):55-77, 2003

[7] Cohen, M. F., Chen, S. Wallace, J., Greenberg, D. **A Progressive Refinement Approach to Fast Radiosity Image Generation.** Computer Graphics, Volume 22, Number 4, August 1988

[8] Cohen, M. F., Greenberg, D. P., Immel, D. S. **An Efficient Radiosity Approach for Realistic Image Synthesis.** IEEE Computer Graphics and Applications, March 1986 pp 26-35.

[9] Lischinski, D. Tampieri, F. Greenberg, D.P. **A Discontinuity Meshing Algorithm for Accurate Radiosity.** IEEE CG&A, 1992

[10] Sillion, F., Puech, C. **Radiosity & Global Illumination**. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-277-1

[11] Lombard, M., Ditton, T. B. **At the heart of it all: The concept of presence**. 1997. http://jcmc.indiana.edu/vol3/issue2/lombard.html

[12] Slater, M., Linakis, V., Usoh, M., Kooper, R. **Immersion, Presence, and Performance in Virtual Environments: An Experiment using Tri-Dimensional Chess.** http://www.cs.ucl.ac.uk/external/M.Usoh/Papers/Chess/index.html

[13] Welch, R. B., Blackmon, T. T., Liu, A., Mellers, B. A. and Stark, L. W. **The effects of pictorial realism, delay of visual feedback, and observer interactivity on the subjective sense of presence**. Presence-Teleoperators and Virtual Environments 5, 263-273 (1996).

[14] Thompson, W.B., Shirley, P., Smits, B., Kersten, D.J. and Madison, C. **Visual Glue**, http://www2.cs.utah.edu/vissim/papers/glue/glue.html, 1998.

[15] Slater, M., Usoh, M., Chrysanthou, Y. **The Influence of Dynamic Shadows on Presence in Immersive Virtual Environments,** http://www2.cs.ucy.ac.cy/~yiorgos/publications/influence_of_shadows95.pdf

[16] Hasenfratz, J.-M., Lapierre, M., Holzschuch, M. and Sillion, F.X. **A Survey of Real-time Soft Shadows Algorithms**. State of the art report, Eurographics 2003.

[17] Crow, F.C., **Shadow algorithms for computer graphics**. Computer Graphics (SIGGRAPH 1977)

[18] Everitt, C., Rege, A., Cebenoyan, C. **Hardware shadow mapping.** http://developer.nvidia.com/object/hwshadowmap_paper.html

[19] Williams, L. **Casting Curved Shadows on Curved Surfaces**. Computer Graphics (SIGGRAPH 1978)

[20] Stamminger, M. and Drettakis, G. **Perspective Shadow Maps**. (SIGGRAPH 2003)

[21] Reeves, W.T., Salesin, D. H. and Cook, R. L. **Rendering antialiased shadows with depth maps**. Computer Graphics (SIGGRAPH 1987)

[22] McGuire, M., Hughes, J.F., Egan, K.T., Kilgard, M. J. and Everitt, C.. **Fast, Practical, and Robust Shadows.** http://developer.nvidia.com/object/fast_shadow_volumes.html

[23] Chen, S. **Incremental radiosity: an extension of progressive radiosity to an interactive image synthesis system**. Computer Graphics, Volume 24, Number 4, August 1990.

[24] George, D., Sillion, F. and Greenberg, D. **Radiosity redistribution for dynamic environments**. IEEE Computer Graphics and Applications, 10(4):26–34, 1990.

[25] Hanrahan P,.Salzman, D, Aupperle, L. **A rapid hierarchical radiosity algorithm**. SIGGRAPH 1991 proceedings.

[26] Forsyth, D., Yang, C. Teo, K. **Efficient Radiosity in dynamic environments.** Eurographics 1994.

[27] Drettakis, G., Sillion, F. **Interactive update of global illumination using a line-space hierarchy**. SIGGRAPH 1997 proceedings.

[28] Granier, X., Drettakis, G. **Incremental updates for rapid glossy global illumination.** Eurographics 2001.

[29] Glassner, A. S., **Space Subdivision for Fast Ray Tracing**, IEEE Computer Graphics and Applications, 4(10), October 1984, pp. 15-22.

[30] Arvo, J., Kirk, D., **Fast Ray Tracing by Ray Classification.** Computer Graphics, Volume 21, Number 4, 1987.

[31] Rubin, S., Whitted, T. **A 3-Dimensional Representation for Fast Rendering of Complex Scenes**. ACM 1980.

[32] Wald, I., Slusallek, P., Benthin, C. and Wagner, M. **Interactive Rendering with Coherent Ray Tracing**. Eurographics 2001, Volume 20, Number 3.

[33] Slater, M. **Constant time queries on Uniformly Distributed Points on a Hemisphere**, Journal of Graphics Tools, 7(1):33-33. (2002)

[34] Udeshi, T., Hansen, C. **Towards interactive rendering of indoor scenes: ahybrid approach.** Eurographics 1999.

[35] Keller, A. **Instant Radiosity**. SIGGRAPH 1997.

[36] Gautron, P., Krivánek, J., Bouatouch, K., Pattanaik, S. **Radiance Cache Splatting: A GPU-Friendly Global Illumination Algorithm.** Eurographics 2005

[37] Wald, I., Kollig, T., Benthin, C., Keller, A., Slusallek, P. **Interactive Global Illumination.** Technical Report TR-2002-02

[38] **Jaroslav K., Gautron, P., Pattanaik, S. Bouatouch, K. Radiance caching for efficient global illumination computation.** http://www.irisa.fr/bibli/publi/pi/2004/1623/1623.html. 2004

[39] Ward, G., Simmons, L.M., **The Holodeck Interactive Ray Cache.** (1999) ACM Transactions on Graphics, 18(4):361-98.

[40] Jensen, H. **Global Illumination using Photon Maps.** Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering), pages 21-30, 1996.

[41] Larsen, B., Christensen, N. **Simulating Photon Mapping for Real-time Applications**. Eurographics symposium on rendering 2004.

Listing 1 Cube definition

For all listings a `shaded background` indicates new code written as part of this implementation. For brevity large chunks of code not essential to understanding are cut out. This is indicated with '…'

```
namespace dynamic {
...
GLfloat n[6][3] = {  /* Normals for the 6 faces of a cube. */
   {0.0, 0.0, 1.0}, {1.0, 0.0, 0.0}, {0.0, 0.0, -1.0},
   {-1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, -1.0, 0.0} };

GLint faces[6][4] = {  /* Vertex indices for the 6 faces of a cube. */
   {0, 1, 2, 3}, {1, 4, 5, 2}, {4, 7, 6, 5},
   {7, 0, 3, 6}, {3, 2, 5, 6}, {7, 4, 1, 0} };

GLfloat v[8][3];  /* Will be filled in with X,Y,Z vertexes. */
   ...
   vlf::faceset<MY_FACE_TYPE,    MY_INDEX_TYPE,    MY_FLOAT_TYPE,    4,    MY_TEXTURE_TYPE>
myIndexedFaceSet;
}

void dynamic::initBox()
{
   ...

   // setup vertex data structures
   glib::point3D<MY_FLOAT_TYPE> vertex[8];
   // Choose cube extents to fit world co-ordinate system. This provides a small cube
compared to the room dimensions
   float minimumExtent = -0.05f;
   float maximumExtent =  0.05f;

   int vertexIndWithSameComponentMin[] = {0,3,7,6, 0,1,4,7, 4,5,6,7}; // groups of 4 for the
same component
   int vertexIndWithSameComponentMax[] = {1,2,4,5, 3,2,5,6, 0,1,2,3}; // groups of 4 for the
same component

   // setup material structures - not actually used but required for class initialisation
   typedef glib::colourRGB<MY_FLOAT_TYPE> colorType;
   glib::standardMaterial<MY_FLOAT_TYPE,colorType>*                                mRed=new
glib::standardMaterial<MY_FLOAT_TYPE,colorType>;
   mRed->diffuse(COLOUR_CORNELL_RED);
   mRed->emission(COLOUR_BLACK);
   mRed->ks(0.0f);
   mRed->kd(1.0f);

   /* Assign cube vertex data. */
   for(int i=0;i<12;i++) {
      glib::point3D<MY_FLOAT_TYPE>& vMin = vertex[ vertexIndWithSameComponentMin[i] ];
      glib::point3D<MY_FLOAT_TYPE>& vMax = vertex[ vertexIndWithSameComponentMax[i] ];
      switch( i/4 ) // integer division to determine component to set
      {
         case 0:
            vMin.X(minimumExtent);
            vMax.X(maximumExtent);
         break;
         case 1:
            vMin.Y(minimumExtent);
            vMax.Y(maximumExtent);
            break;
         case 2:
            vMin.Z(minimumExtent);
            vMax.Z(maximumExtent);
            break;
         }
   }

   // Add vertices to faceset, assume first returned index is 0
   for (int i =0; i<8; i++) {
      myIndexedFaceSet.addVertex(vertex[i]);
      ...
   }
```

```
   for (int i =0; i<8; i++) { // cube is centred on origin so normals are same as vertices
      std::cout << "Normal: " << myIndexedFaceSet.addNormal(
         vertex[i].asVector().normalized()
         ) << std::endl;
   }

   unsigned int mInd = myIndexedFaceSet.addFaceMaterial(mRed);

   // add faces to faceset

   for(int i=0;i<6;i++) {

      // add each cube face as two triangles
      myIndexedFaceSet.addFace( MY_FACE_TYPE( faces[i][0],faces[i][1],faces[i][2],
                                                          faces[i][0],faces[i][1],faces[i][2],
mInd)
                           );
      myIndexedFaceSet.addFace( MY_FACE_TYPE( faces[i][2],faces[i][3],faces[i][0],
                                                          faces[i][2],faces[i][3],faces[i][0],
mInd)
                           );
   ...
}
```

Listing 2 Low quality progressive mode including dynamic object

```
void display(void) {
   ...
   else if (renderType==RENDER_PROGRESSIVE)
   {
      // SORT OUT lo-quality specular map if necessary
      if (!highQualitySpecularReady)
      {
         util::timer<__MHZ> glCalcSpecularTimer;
         double glCalcSpecularSecs;
         glCalcSpecularTimer.start();
         if (reSampledTRM)

   myVLF.glRaytraceSpecular(specularCam,loQualitySpecularMap,loQualityIndexMap,thebackground
Index,NULL,backBuf,1.0);
         else

   myVLF.glRaytraceSpecular(specularCam,loQualitySpecularMap,loQualityIndexMap,thebackground
Index,NULL,backBuf,radianceScaling);

         glCalcSpecularSecs = glCalcSpecularTimer.stop();
         std::cout << frameNo << " CalcLoSpec: " << glCalcSpecularSecs << std::endl;
      }


      // cut out code setting up GL state etc
      ...


#ifdef DO_DYNAMIC_OBJECT
      // begin JDW
      // visualise all faces
      // this places the box correctly in the depth buffer for diffuse surfaces
      if(showDynamicObject) {
         glDisable(GL_TEXTURE_2D);
         for (unsigned int i=0;i<dynamic::myIndexedFaceSet.faceCount();i++)    {
            dynamic::myIndexedFaceSet.visualiseFace(i,false,false,true,COLOUR_WHITE,true);
         }
         glEnable(GL_TEXTURE_2D);
      }
      // end JDW
#endif

      const bool showDiffuse = true;
      if(showDiffuse) {
         for (unsigned int f=0;f<myVLF.scene().faceCount();f++)
         {
            glBindTexture(GL_TEXTURE_2D, diffuseTextures[f]);
            glBegin(GL_POLYGON);
            for (unsigned int v=0;v<myVLF.scene().vertexCount(f);v++)
            {
               // map a vertex into its diffuse map
               // JDW Note this homogenize/dehomogenize technique is used often.
               // The transform matrices have 4 dimensions and so to multiply by a vector
               // we need to transform to a vector4D with homogenize and then convert the
               // result back to vector3D with dehomogenize()
               glib::point3Df tc=
                  (myVLF.scene().getFace(f)->toCanonical()
                  * myVLF.scene().vertexWC(f,v).homogenize()).dehomogenize();
               // convert value to a texcoord between 0-1
               float s=tc.X()/myVLF.scene().getFace(f)->faceExtents.xSize();
               float t=tc.Z()/myVLF.scene().getFace(f)->faceExtents.ySize();
               if (doTexCoordShift)
               {
                  // we want to shift half a pixel in texture space:
                  ... // cut

               }
               // apply texture coordinate scaling it to account for a differently
               // size actual opengl texture [power of two texture]

   glTexCoord2f(MAX2(0,s*diffuseScaleFactors[f*2]),MAX(0,t*diffuseScaleFactors[f*2+1]));
```

```
               myVLF.scene().vertexWC(f,v).glApply();
            }
            glEnd();
         }
      } // showDiffuse

      const bool showSpecular=true;

      if(showSpecular) {
         if (highQualitySpecularReady)
         {
            // fillin texture
            ... // cut
         }

         else
         {
            // fillin texture
            ... // cut
         }

         // render a polygon onto the screen textured with the specular texture
         // set GL state
         ... //cut

         // setup the texture
         ... // cut

         // Generate The Texture

   glTexImage2D(GL_TEXTURE_2D,0,4,specularTexures,specularTexures,0,GL_RGBA,GL_FLOAT,specula
rBuf);

         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

         //
         // draw polygon
         glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
         glBegin(GL_QUADS);
         ... // cut
         glEnd();
      } // showSpecular

   } // end else
   ...
} // end display()


vlf::cachedVLF<> { // from vlfCachedVLF.h

   ...

      // false colour raycasts specular surfaces only,
      // returns an image of the colours and an index map, which has the backgroundIndex
      // in pixels where a specular surfaces wasn't to be found
      bool glRaytraceSpecular( ... )
      {

         ... // cut

         util::glContextHolder ctx;
         if (pbuf)
            ctx.saveContext();

         // save GL state
         glPushAttrib(GL_ALL_ATTRIB_BITS);
         glPushMatrix();

         // make the backbuffer current
         if (pbuf)
         if (!pbuf->makeCurrent())
         {
            std::cerr << "ERROR: vlf::cachedVLF::glRaytraceSpecular, could not make create
off-screen GL buffer current" << std::endl;
```

```
            return false;
         }

         util::colourIndex colI;
         colI.getBitCounts();
         unsigned char biR,biG,biB,biA;
         colI.index2RGBA(&biR,&biG,&biB,&biA,backgroundIndex);

         // set clear colour, we clear buffer with the non mask index!

   glClearColor((float)biR/255.0f,(float)biG/255.0f,(float)biB/255.0f,(float)biA/255.0f);
         glClearDepth(1.0);

         // clear buffers
         glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

         // set up viewing
         ... // cut

         // render the faces in false colour
         // specular faces are rendered with their index and diffuse faces
         // with the background colour
         unsigned char R,G,B,A;
         for (unsigned f=0;f<m_scene.faceCount();f++)
         {
            // set the indexed colour or background
           if (((VLF_MATERIAL_TYPE*)m_scene.faceMaterial(f))->ks()>0)
           {
            //pbuf.colourIndex.index2RGBA(&R,&G,&B,&A,f);
            colI.index2RGBA(&R,&G,&B,&A,f);
            glColor4ub(R,G,B,A);
         }
         else
            glColor4ub(biR,biG,biB,biA);


         // render face
         glBegin(GL_POLYGON);
         for (unsigned int i=0;i<m_scene.vertexCount(f);i++)
            m_scene.vertexWC(f,i).glApply();
            glEnd();
         } // for f
#ifdef DO_DYNAMIC_OBJECT
         if(accountForDynamicObjectInProgSpec) {
            // begin JDW
            // account for our dynamic object in the specular map - make it appear
            // like a regular diffuse object so it can show through the specular 'mask'

            // make object diffuse by setting colour corresponding to background index
            glColor4ub(biR,biG,biB,biA);
            for (unsigned int i=0;i<dynamic::myIndexedFaceSet.faceCount();i++) {
               dynamic::myIndexedFaceSet.visualiseFaceInheritColour(i,true);  //  draw  the
outline as in normal rendering
            }
         }
            // end JDW
#endif // DO_DYNAMIC_OBJECT

...

         // read back the back-buffer to main memory
         glReadPixels(0,0,cam.uRes(),cam.vRes(),GL_RGBA,GL_UNSIGNED_BYTE,backBuf);

...

         unsigned long int colorIndex;
         // build index buffer from back-buffer in memory
         ... // cut
...

         // vars to work out pixel centre incrementally
         glib::point3D<FLOAT_TYPE> pixelCentre;
         glib::point3D<FLOAT_TYPE> eye=cam.Eye();
         ... // cut
```

```
         glib::ray3D<FLOAT_TYPE> rayFromFace;
         FLOAT_TYPE depth=0;
         // now build image
         for (unsigned int y=0;y<cam.vRes();y++)
         {
            pixelCentre=pixelCentreOrigin+((FLOAT_TYPE)y)*vVec;
            for (unsigned int x=0;x<cam.uRes();x++)
            {
               // update pixel centre
               pixelCentre=pixelCentre+uVec;

               glib::index2Dui pixelIndex(x,y);
               glib::index2Dui pixelIndexRev(x,specularMap.vRes()-1-y);
               glib::colourRGB<FLOAT_TYPE> c1,c2,c3,color(0,0,0);

               // check if we have a background pixel
               unsigned int faceIndex=indexMap(pixelIndex).R();
               if (faceIndex==backgroundIndex)
                  continue;

               // retrieve the pixel
               glib::ray3D<FLOAT_TYPE> ray=
               glib::ray3D<FLOAT_TYPE>(eye,(pixelCentre-eye).normalized());

...
               // find depth of face
               m_scene.intersectionDepth(depth,faceIndex,ray);
...
               // find the three nearest PSFs
               DIR_TYPE n1,n2,n3;
               FLOAT_TYPE kn1,kn2,kn3;
               if
(m_directions.triangleItem(&n1,&n2,&n3,&kn1,&kn2,&kn3,ray.Orientation().normalized()))
               {
#ifdef DIRECTION_AREA_EFFECTS_TRILINEAR_VIEWING
#ifdef PROPAGATE_USING_SOLID_ANGLE_AREAS
                  // Scale up radiance value as viewing is not a function of solid angles,
rather that of rays...
                  kn1 /=n1.areaWeight();
                  kn2 /=n2.areaWeight();
                  kn3 /=n3.areaWeight();
#endif
#endif

                  rayFromFace.Origin(ray.Origin()+(ray.Orientation()*depth));
                  rayFromFace.Orientation(ray.Orientation().inverted());
               }
               else
               {
                  specularMap(pixelIndex)=COLOUR_GREEN;
                  continue;
               }

               // check that we have a valid face here
               //unsigned faceIndex=indexMap(pixelIndex).R();
               if (faceIndex<m_scene.faceCount())
               {
                  bool r1=false,r2=false,r3=false;

...

                  r1=getSample(n1.m_linearIndex,c1,rayFromFace,faceIndex);
                  r2=getSample(n2.m_linearIndex,c2,rayFromFace,faceIndex);
                  r3=getSample(n3.m_linearIndex,c3,rayFromFace,faceIndex);

...

                  if (r1 && r2 && r3)
                  {
                     color=scaling*((kn1*c1)+(kn2*c2)+(kn3*c3));
                     specularMap(pixelIndex)=color;
                     indexMap(pixelIndex).R()=faceIndex;
                     continue;

                  }
                  else if (r1 || r2 || r3)
                  {
```

```
                        ... // cut
                        specularMap(pixelIndex)=color;
                        indexMap(pixelIndex).R()=faceIndex;
                        continue;
                     }
                    else
                       specularMap(pixelIndex)=COLOUR_RED;
                }

                // if we fell through to here we couldn't get a specular value
                indexMap(pixelIndex).R()=backgroundIndex;
             }
          }

...
          return true;
       }; // glRaytraceSpecular

} // end of object definition vlf::cachedVLF<>
```

```
//This method was added to sg::faceset (file sgFaceset.h)
// Begin JDW
       // We want to render a face in the existing colour only - for use when rendering false
colour indexmap for dynamic object
       void visualiseFaceInheritColour(FACE_INDEX_TYPE f,bool outline=false)
       {
          if (vertexCount(f)<3) return;
          #ifdef _GL_AVAILABLE
             glPushAttrib(GL_ALL_ATTRIB_BITS);
             glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
             glEnable(GL_CULL_FACE);
             glCullFace(GL_BACK);

             glDisable(GL_LIGHTING);

             glMatrixMode(GL_MODELVIEW);

             if (outline)
             {
                glEnable(GL_POLYGON_OFFSET_FILL);
                glPolygonOffset(1.0,1.0);
             }

             // now render the face
             glBegin(GL_POLYGON);
             for (unsigned int i=0;i<vertexCount(f);i++)
             {
                glib::point3D<FLOAT_TYPE> v=vertexWC(f,i);
                glVertex3f(v.X(),v.Y(),v.Z());
             }
             glEnd();

             // render the outline if required
             if (outline)
             {
                glDisable(GL_POLYGON_OFFSET_FILL);
                glPolygonMode(GL_FRONT,GL_LINE);

                // now render the covered pixel
                glBegin(GL_POLYGON);
                for (unsigned int i=0;i<vertexCount(f);i++)
                {
                   glib::point3D<FLOAT_TYPE> v=vertexWC(f,i);
                   glVertex3f(v.X(),v.Y(),v.Z());
                }
                glEnd();
             }

             glPopAttrib();
          #else
             std::cerr  <<  "ERROR:  sg::faceset::visualiseFace,  gl  not  available"  <<
std::endl;
          #endif
       };
// End JDW
```

Listing 3 High quality progressive mode including dynamic object

See Listing 2 for the relevant part of the display() method.

```
void progressiveRenderer(void *dummy)
{
   std::cout << "render thread started!" << std::endl;
   finishedProgressiveRenderThread.ResetEvent();

   if
(hiQualitySpecularMap.uRes()!=specularCam.uRes()||hiQualitySpecularMap.vRes()!=specularCam.v
Res()||

   hiQualityIndexMap.uRes()!=hiQualitySpecularMap.uRes()||hiQualityIndexMap.vRes()!=hiQualit
ySpecularMap.vRes())
   {
      std::cerr  <<  "ERROR:  progressiveRenderer,  hiQualitySpecularMap  wrong  size"  <<
std::endl;
      exit(1);
   }

   while (runProgressive)
   {
      //interrupted:
      if (!highQualitySpecularReady)
      {
         //util::timer<__MHZ> comptime;
         //comptime.start();

         interruptRendering=false;
         // redo specular image
         for (unsigned int y=0;y<specularCam.vRes();y++)
         {
            for (unsigned int x=0;x<specularCam.uRes();x++)
            {
               if (interruptRendering)
               {
                  //comptime.stop();
                  goto interrupted;
               }

               glib::index2Dui pixelIndex(x,y);


               // this is the pixel through which the ray travels
               glib::window3D<MY_FLOAT_TYPE> pixel=specularCam.pixelWC(pixelIndex);

               // now construct a normalized ray going from eye through pixel
               glib::ray3D<MY_FLOAT_TYPE>
ray=glib::ray3D<MY_FLOAT_TYPE>(specularCam.Eye(),(pixel.center()-
specularCam.Eye()).normalized());

               // check if we have a specular reflection in this pixel
               glib::intersectionInfo<float> ix;
               unsigned ixFace=0;


#ifdef DO_DYNAMIC_OBJECT
               // begin JDW
               /*
               Original test is:
               If this ray doesn't intersect the scene
               OR
               It does intersect the scene but intersected face is diffuse
               THEN
               place backgroundIndex pixel in hiQualityIndexMap

               Add some variables for a second intersection test - to be applied only
               if we find a specular surface - this way we override what would have
               been incorrect depth rendering only when needed
               */
               glib::intersectionInfo<float> dynIx;
               unsigned dynIxFace=0;
#endif
```

```
                // end JDW
                if ( (!myVLF.scene().faceIntersection(ixFace,ray,&ix,0.00001f,true)) ||
                   (((VLF_MATERIAL_TYPE*)myVLF.scene().faceMaterial(ixFace))->ks()==0) )
                {
                   hiQualityIndexMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-1-
y)).R()=thebackgroundIndex;

                   continue;
                }
#ifdef DO_DYNAMIC_OBJECT
                if(accountForDynamicObjectInProgSpec) {
                // begin JDW
                // we have hit a specular surface - BUT it could be occluded by the dynamic
object
                //  so  check  for  an  earlier  intersection  point  and  if  found  place  the
background
                // index in this pixel

                   if (

   (dynamic::myIndexedFaceSet.faceIntersection(dynIxFace,ray,&dynIx,0.00001f,true))
                      &&(dynIx.t() <= ix.t())
                      )
                   {
                      hiQualityIndexMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-1-
y)).R()=thebackgroundIndex;
                      continue;
                   }
                }
                // end JDW
#endif


                // we have a specular reflection trace until diffuse surface is struck
                bool foundDiffuse=false;
                int recursiveDepth=0;
                do
                {
                   // sort out new ray
                   ray.Origin()=ray.Origin()+(ray.Orientation()*ix.t());

   ray.Orientation(ray.Orientation().inverted().reflected(myVLF.scene().faceNormalWC(ixFace)
));


                   if (!myVLF.scene().faceIntersection(ixFace,ray,&ix,0.00001f,true))
                      break;
                   else
                      foundDiffuse=
(((VLF_MATERIAL_TYPE*)myVLF.scene().faceMaterial(ixFace))->kd() >1E-9);

                   if (interruptRendering)
                   {
                      //comptime.stop();
                      goto interrupted;
                   }
                }
                while ((!foundDiffuse) && (++recursiveDepth<MAX_SPECULAR_RAYTRACE_DEPTH));


                // handle traced ray
                if (foundDiffuse)
                {
                   //hiQualityIndexMap(pixelIndex).R()=ixFace;
                   hiQualityIndexMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-1-
y)).R()=ixFace;

                   glib::point3D<MY_FLOAT_TYPE> ixp=ray.Origin()+(ray.Orientation()*ix.t());

                   // map this to a diffuse pixel on its total diffuse map
                   glib::point3Df tdmp=
                      ((myVLF.scene().getFace(ixFace)-
>toCanonical())*glib::point4Df(ixp.X(),ixp.Y(),ixp.Z(),1)).dehomogenize();

                   // now the X and Z coordinates of this point indicate a point on
                   // the diffuse map for ixFace, map this to an index + a fractional part
```

```
                    float                                   tWR=myVLF.scene().getFace(ixFace)-
>faceExtents.xSize()/myVLF.scene().getFace(ixFace)->m_totalDiffuse.uRes();
                    float                                   tHR=myVLF.scene().getFace(ixFace)-
>faceExtents.ySize()/myVLF.scene().getFace(ixFace)->m_totalDiffuse.vRes();

                    // make sure point is inside diffuse map
                    float   s   =   MAX2(0,   MIN2(tdmp.X(),   myVLF.scene().getFace(ixFace)-
>faceExtents.xSize()) );
                    float   t   =   MAX2(0,   MIN2(tdmp.Z(),   myVLF.scene().getFace(ixFace)-
>faceExtents.ySize()) );

                    // extract integer index and fractional part
                    glib::index2Dui index;
                    glib::index2D<float> indexFractional;
                    double n;
                    indexFractional.U() = modf((s/tWR),&n);
                    index.U()=n;
                    indexFractional.V() = modf((t/tHR),&n);
                    index.V()=n;

                    if (index.U()<myVLF.scene().getFace(ixFace)->m_totalDiffuse.uRes()&&
                       index.V()<myVLF.scene().getFace(ixFace)->m_totalDiffuse.vRes())
                    {
                       glib::colourRGB<MY_FLOAT_TYPE>
color=diffuseMaps[ixFace].bilinear(index,indexFractional);
                       if (doToneMap)
                          hiQualitySpecularMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-
1-y))=color;
                       else
                          hiQualitySpecularMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-
1-y))=color*radianceScaling;
                    }
                    else
                       std::cerr  <<  "ERROR:  progressiveRenderer,  missed  diffuse  map"  <<
std::endl;
                }
                else
                {
                   hiQualityIndexMap(glib::index2Dui(x,hiQualitySpecularMap.vRes()-1-
y)).R()=thebackgroundIndex;
                   continue;
                }

             } //for {x}
          } //for {y}

          highQualitySpecularReady=true;

       }

interrupted:
       Sleep(progressiveRendererSleepMS);
    }
    finishedProgressiveRenderThread.SetEvent();
    std::cout << "render thread terminated!" << std::endl;
    _endthread();
} // progressiveRenderer
```

Listing 4 Preserving correct surface normals after object motion

From vlfGlutWalkThrough.cpp.

```
// inside display() for the coherent shading mode
...
for (unsigned int f=0;f<dynamic::myIndexedFaceSet.faceCount();f++)
{

   GLfloat glFaceNormal[3];
   dynamic::myIndexedFaceSet.transformedFaceNormalWC(f).get3fv(glFaceNormal);

   glib::point3D<MY_FLOAT_TYPE> v;
   glBegin(GL_POLYGON);
   for (unsigned int i=0;i<dynamic::myIndexedFaceSet.vertexCount(f);i++)
   {

      v=dynamic::myIndexedFaceSet.vertexWC(f,i);

      glNormal3fv(glFaceNormal);
      glVertex3f(v.X(),v.Y(),v.Z());
   }
   glEnd();

}
// display continues
```

From vlfFaceset.h

```
// added to get transformed face normals because now objects can move
glib::vector3D<FLOAT_TYPE> transformedFaceNormalWC(FACE_INDEX_TYPE f)
{
   // operate on this normal using only rotation matrices.
   return ( normalTransformMatrix()* getFace(f)->n().homogenize()).dehomogenize() ;
}
```

Listing 5 Shadow mapping implementation

From dynamic.h

```
   GLdouble glLightViewMatrix[16];
   GLdouble glLightProjectionMatrix[16];
   GLdouble glTextureMatrix[16];
   GLdouble glLightClipPlane[4];
   GLdouble glBiasMatrix[16] = { 0.5, 0.0, 0.0, 0.0,
                         0.0, 0.5, 0.0, 0.0,
                         0.0, 0.0, 0.5, 0.0,
                         0.5, 0.5, 0.5, 1.0 };


   glib::transformationMatrixf textureMatrix;

   // will reference the shadow map texture
   GLuint shadowMapTexture;
   int shadowMapURes = 512;
   int shadowMapVRes = 512;
   bool removeReverseShadowProjection = true;
   bool noShadowOnBackFaces = true;
```

From dynamic.cpp

```
void dynamic::setupShadowLight(MY_SCENE_TYPE& scene, glib::bufferCameraf& worldCam) {
   int emitterInd=-1;

   for (unsigned int f=0;f<scene.faceCount();f++) {
      if(scene.getFace(f)->isEmitter()) {
         emitterInd = f;
         break;
      }
   }

   if(emitterInd<0) {
      std::cout << "No emitter in scene! Can't simulate light." << std::endl;
      return;
   }


   // find the centre of the emitter face - assume it's a rectangle
   // create a virtual light at the centre
   glib::point3D<MY_FLOAT_TYPE> p0 = scene.vertexWC(emitterInd,0);
   glib::point3D<MY_FLOAT_TYPE> p1 = scene.vertexWC(emitterInd,2);
   glib::vector3D<MY_FLOAT_TYPE> v = p1-p0;
   lightPos = p0 + v*0.5f;


   // look towards the centre of the dynamic object
   dynamic::lookAt = (myIndexedFaceSet.cumulativeTransformMatrix()
    * myIndexedFaceSet.boundingSphereWC().centre().homogenize()).dehomogenize();
   lightCam = glib::camera<MY_FLOAT_TYPE>(lightPos,
                         lookAt,
                        28,
                        dynamic::shadowMapURes,
                        dynamic::shadowMapVRes);

   glPushAttrib(GL_ALL_ATTRIB_BITS);
   glMatrixMode(GL_MODELVIEW_MATRIX);
   glPushMatrix();
   glLoadIdentity();


   lightCam.setGlu();
   glGetDoublev(GL_MODELVIEW_MATRIX, glLightViewMatrix);

   glLoadIdentity();

   gluPerspective(
      45.0f,
      1,    // aspect ratio
      lightCam.Front(),        // near clipping plane
      lightCam.Back()          // far clipping plane
   );

   glGetDoublev(GL_MODELVIEW_MATRIX, glLightProjectionMatrix);
```

```
   // create texture for shadow map
   glGenTextures(1, &shadowMapTexture);
   glBindTexture(GL_TEXTURE_2D, shadowMapTexture);
   glTexImage2D(     GL_TEXTURE_2D,     0,     GL_DEPTH_COMPONENT,     dynamic::shadowMapURes,
dynamic::shadowMapVRes, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

   glPopAttrib();
   glPopMatrix();

}
```

```
void dynamic::recalculateShadowLight(MY_SCENE_TYPE& scene, glib::bufferCameraf& worldCam) {

   // object moves - so adjust light direction
   // however the bounding sphere is calculated once and cached, so it
   // doesn't move from it's initial position - so take the cached initial centre
   // point and transform via the transform matrix in the faceset to
   // get current centre position

   dynamic::lookAt = (myIndexedFaceSet.cumulativeTransformMatrix()
       * myIndexedFaceSet.boundingSphereWC().centre().homogenize()).dehomogenize();

   dynamic::lightCam.lookAt(
      lightPos,
      lookAt,
      28,
      dynamic::shadowMapURes,
      dynamic::shadowMapVRes
   );

   // setup glLightClipPlane plane equation from camera VRP and VPN
   // This is required to remove back projected shadow if we allow shadow piercing onto
   // surfaces with normals pointing away from light (we don't any more)

   glLightClipPlane[0] = dynamic::lightCam.DOP().X();
   glLightClipPlane[1] = dynamic::lightCam.DOP().Y();
   glLightClipPlane[2] = dynamic::lightCam.DOP().Z();

   glLightClipPlane[3] = -1.0 * (lightCam.DOP().X() * lightPos.X() +
      lightCam.DOP().Y() * lightPos.Y() +
      lightCam.DOP().Z() * lightPos.Z());

   // lightPlance is used if we want to render this plane during debugging

   lightPlane.setNormal( dynamic::lightCam.DOP() );
   lightPlane.setPosition( dynamic::lightPos );

   glMatrixMode(GL_MODELVIEW_MATRIX);
   glPushMatrix();
   glPushAttrib(GL_ALL_ATTRIB_BITS);

   glLoadIdentity();

   lightCam.setGlu();

   glGetDoublev(GL_MODELVIEW_MATRIX, glLightViewMatrix);

// perform calculation of glTextureMatrix using OpenGL
   glLoadMatrixd(glBiasMatrix);
   glMultMatrixd(glLightProjectionMatrix);
   glMultMatrixd(glLightViewMatrix);

   glGetDoublev(GL_MODELVIEW_MATRIX, glTextureMatrix);

   glPopMatrix();
   glPopAttrib();

   //Render shadow buffer from lights POV
   glPushAttrib(GL_ALL_ATTRIB_BITS);
   glPushMatrix();
```

```
   //Write to Z-Buffer if drawn Z value is less than or equal to that in buffer
   glClearDepth(1.0f);
   glDepthFunc(GL_LEQUAL);
   glEnable(GL_DEPTH_TEST);

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glMatrixMode(GL_PROJECTION);
   glLoadMatrixd(glLightProjectionMatrix);

   glMatrixMode(GL_MODELVIEW);
   glLoadMatrixd(glLightViewMatrix);

   //Set viewport size to match shadow map
   glViewport(0, 0, dynamic::shadowMapURes, dynamic::shadowMapVRes);

   //Cull the front faces to use the back for depth comparison - eliminates shadow acne
   glCullFace(GL_FRONT);

   //No color reaches buffer so render flat for speed
   glShadeModel(GL_FLAT);
   glColor3f(1.0,1.0,1.0);
   glColorMask(1, 1, 1, 1);
   //glColorMask(0, 0, 0, 0);

   // draw object into depth buffer
   for (unsigned int i=0;i<dynamic::myIndexedFaceSet.faceCount();i++)   {
      dynamic::myIndexedFaceSet.visualiseFace(i,false,false,false,COLOUR_WHITE,false);
   }

   //Copy depth buffer into shadow map
   glBindTexture(GL_TEXTURE_2D, shadowMapTexture);
   glCopyTexSubImage2D(GL_TEXTURE_2D,    0,    0,    0,    0,    0,    dynamic::shadowMapURes,
dynamic::shadowMapVRes);

   //restore states
   glCullFace(GL_BACK);
   glShadeModel(GL_SMOOTH);
   glColorMask(1, 1, 1, 1);

   glPopAttrib();
   glPopMatrix();
}
```

```
// Remember that GL_EYE_LINEAR post multiplies by the current modelview matrix
// so this has to be set to the camera view matrix before calling this function.
void dynamic::renderShadow(MY_SCENE_TYPE& scene) {

   glPushAttrib(GL_ALL_ATTRIB_BITS);

   GLfloat rowvals[4];

   // convert column major to a row at a time
   for(int i=0;i<4;i++) {
      rowvals[i] = glTextureMatrix[(i*4)];
   }

   //texture coordinate generation.
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
   glTexGenfv(GL_S, GL_EYE_PLANE, rowvals);
   glEnable(GL_TEXTURE_GEN_S);

   for(int i=0;i<4;i++) {
      rowvals[i] = glTextureMatrix[(i*4)+1];
   }
   glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
   glTexGenfv(GL_T, GL_EYE_PLANE, rowvals);
   glEnable(GL_TEXTURE_GEN_T);

   for(int i=0;i<4;i++) {
      rowvals[i] = glTextureMatrix[(i*4)+2];
   }
   glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
   glTexGenfv(GL_R, GL_EYE_PLANE, rowvals);
   glEnable(GL_TEXTURE_GEN_R);
```

```
   for(int i=0;i<4;i++) {
      rowvals[i] = glTextureMatrix[(i*4)+3];
   }


   glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
   glTexGenfv(GL_Q, GL_EYE_PLANE, rowvals);
   glEnable(GL_TEXTURE_GEN_Q);


   glBindTexture(GL_TEXTURE_2D, shadowMapTexture);
   glEnable(GL_TEXTURE_2D);

// wrapped in a define because the extension variable below comes from glew.h which could be
excluded
// not the neatest way but didn't want to remove code I already knew to be 'safe'
#ifdef DO_DYNAMIC_SHADOWS
   //Enable shadow comparison
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_COMPARE_R_TO_TEXTURE);
#endif // DO_DYNAMIC_SHADOWS
   //Shadow comparison should be true if in shadow, i.e. if r>texture
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_GREATER);

   //generate an intensity result (1 in all channels), alpha value will be 1
   glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_INTENSITY);

   // draw scene in black where it falls in shadow from the object

   // clip scene behind light because otherwise will get reverse shadow projection
   if(dynamic::removeReverseShadowProjection) {
      glClipPlane(GL_CLIP_PLANE0,glLightClipPlane);
      glEnable(GL_CLIP_PLANE0);
   }


   // Use these two for overwriting the destination colour and producing black shadows
   //glEnable(GL_ALPHA_TEST);
   //glAlphaFunc(GL_GEQUAL,0.99f);

   glEnable(GL_BLEND); // blend the shadow rather than overwriting the existing colour.
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);


      for (unsigned int f=0;f<scene.faceCount();f++)
      {
         if (((VLF_MATERIAL_TYPE*)scene.faceMaterial(f))->ks()>0) {
            continue;
         }

         // don't allow shadow to 'pierce' object on back faces, cull where
         // face normal points in same direction as VPN

         if(dynamic::noShadowOnBackFaces) {
            if ( (dynamic::lightCam.VPN() dotProduct scene.faceNormal(f)) <= 0) {
               continue;
            }
         }

         //float fc = static_cast<float>(f)/static_cast<float>(scene.faceCount());
         //COLOUR_RED.glApply();
         //glColor3f(fc,1.0,1.0);
         // black shadows when used with alpha test
         //glColor3f(0.0,0.0,0.0);
         glColor4f(0.0, 0.0, 0.0, 0.5); // use alpha to preserve some destination colour
(blending)
         glBegin(GL_POLYGON);
         for (unsigned int v=0;v<scene.vertexCount(f);v++)
         {
            scene.vertexWC(f,v).glApply();
         }
         glEnd();
         // uncomment to render wireframe and thus see shadow edges
         // scene.visualiseFace(f,true,false,false,COLOUR_RED,false);
      }
   ...
   glPopAttrib();
}
```

Listing 6 Triangular subdivision code

From vlfDynamic.h

```
namespace dynamic
{

   /*
   Defines a triangle through three indexes into the triangleVertices index.
   Stored in anti-clockwise order
   */
   class indexedTriangle
   {
   public:
      int p,q,r;
      indexedTriangle(int pi, int qi, int ri) {
         p=pi; q=qi; r=ri;
      };
      int at(int pos) {
         switch(pos) {
            case 0:
               return p;
               break;
            case 1:
               return q;
               break;
            case 2:
               return r;
               break;
         }
      }

   };

   /*
   Holds information about faces decomposed into subtriangles for rendering
   */
   class triangleSubdivision
   {
   private:
      // Use this to track cuts when initially decomposing faces into subtriangles.
      // The key is the starting point index * 1000 + end point index;
      // This assumes less than 1000 triangle index points so be careful!
      std::map<int,int> triEdgeCutTracker;
      bool _initialised; // call subdivideAllFaces to initialise

      // Holds the indexed vertices of the triangles produced through subdivision
      // Added as they are created so we can assume we will never ask for an
      // out of range index during the creation process
      std::vector<glib::point3Df> triangleVertices;

      //// Holds the triangle list, per depth, per face in the dynamic scene (vlf::faces)
      //std::vector<       std::vector<       std::vector<indexedTriangle*>*       >*       >
vectorOfTrianglesPerDepthPerFace;

   public:
      // Holds the triangle list, per depth, per face in the dynamic scene (vlf::faces)
      // public so we can initialise in initBox() outside of this class. Not the cleanest
      // way but OK for now.
      std::vector<       std::vector<       std::vector<indexedTriangle*>*       >*       >
vectorOfTrianglesPerDepthPerFace;

      triangleSubdivision(){_initialised=false;};
      void addCut(int startPointIndex, int endPointIndex, int newPointIndex);
      /* Returns 0 or index */
      int getCut(int startPointIndex, int endPointIndex);
      /* Returns triangle vertex index */
      int addTriangleVertex(glib::point3Df &p);
      glib::point3Df& getTriangleVertex(int index) { return triangleVertices.at(index); };
      std::vector<indexedTriangle*>& getTriangleList(int faceIndex, int depth) {
         return *vectorOfTrianglesPerDepthPerFace[faceIndex]->at(depth);
      }
      int vertexCountAtDepth(int depth) {
         // assume same tri count per face
         return 3 * vectorOfTrianglesPerDepthPerFace[0]->at(depth)->size();
      }
```

```
      int totalVertexCount() { return triangleVertices.size(); };

      void subdivideAllFaces();
      std::vector<indexedTriangle*>* subdivideIndexedTriangle(indexedTriangle& tri);
      // Write vertices out to console
      void listVertices() {
         for(int i=0;i<triangleVertices.size();i++) {
            std::cout << "V[" << i << "]: " << triangleVertices[i] << std::endl;
         }
      };
      // Check the lists of faces per depth are correct
      void listHeirarchies() {


         for(int iFace=0;
            iFace<vectorOfTrianglesPerDepthPerFace.size();
            ++iFace)
         {
            for(int iDepth=0;
               iDepth<vectorOfTrianglesPerDepthPerFace[iFace]->size();
               iDepth++)
            {

               std::vector<indexedTriangle*>*
triList=vectorOfTrianglesPerDepthPerFace[iFace]->at(iDepth);
               typedef std::vector<indexedTriangle*>::iterator TI;
               for(TI ti=triList->begin();
                  ti!=triList->end();
                  ++ti)
               {
                  indexedTriangle* tri = *ti;
                  std::cout << "Face[" << iFace <<"], "
                     <<"Depth[" <<iDepth <<"], "
                     << tri->p << ","
                     << tri->q << ","
                     << tri->r << "." << std::endl;
               }
            };

         };
      };
   };

   ... // cut
   // holds triangle subdivision information
   dynamic::triangleSubdivision ts;
   int subdivisionLevel=0; // level of triangular subdivision to use on dyn faces
   ... // cut

}
```

From vlfDynamic.cpp

```
void dynamic::initBox() // only the relevant code shown
{
   // initialise triangle subdivision scheme for VLF light rendering
   ts = dynamic::triangleSubdivision();

   ... // cut

      // additionally define these faces in the triangle subdivision routines to
      // store the further subdivided faces for later OpenGL rendering.
      // This avoids the overhead of more MY_FACE_TYPE objects which aren't strictly needed
      // note these triangles index points in a seperate structure, but correspond to
      // same co-ordinates.
      indexedTriangle* t1 = new indexedTriangle( faces[i][0],faces[i][1],faces[i][2] );
      indexedTriangle* t2 = new indexedTriangle( faces[i][2],faces[i][3],faces[i][0] );

      std::vector<   indexedTriangle*   >*   initialTriangleList1   =   new   std::vector<
indexedTriangle* >;
      std::vector<   indexedTriangle*   >*   initialTriangleList2   =   new   std::vector<
indexedTriangle* >;
      initialTriangleList1->push_back(t1);
      initialTriangleList2->push_back(t2);

      // add depth zero list to face zero
```

```
      std::vector<  std::vector<indexedTriangle*>*  >*  listOfTriListsAtDepth0_t1  =  new
std::vector< std::vector<indexedTriangle*>* >;
      std::vector<  std::vector<indexedTriangle*>*  >*  listOfTriListsAtDepth0_t2  =  new
std::vector< std::vector<indexedTriangle*>* >;

      listOfTriListsAtDepth0_t1->push_back(initialTriangleList1);
      listOfTriListsAtDepth0_t2->push_back(initialTriangleList2);

      // so ultimately we have 12 faces in the subdivision scheme.
      ts.vectorOfTrianglesPerDepthPerFace.push_back(listOfTriListsAtDepth0_t1);
      ts.vectorOfTrianglesPerDepthPerFace.push_back(listOfTriListsAtDepth0_t2);


   }
   ts.subdivideAllFaces();
   //ts.listVertices();
   //ts.listHeirarchies();

... // cut
}



void dynamic::triangleSubdivision::subdivideAllFaces() {
   if (_initialised) return; // only need to do this once.

   _initialised = true;

   int depth=0;
   int MAX_SUBDIVISION_DEPTH=2;
   while(depth < MAX_SUBDIVISION_DEPTH) {
      for(int faceIndex=0;faceIndex<dynamic::myIndexedFaceSet.faceCount();faceIndex++) {

         // Get triangle list at depth depth for face faceIndex
         // Assumes depth 0 for each face is already initialised

         std::vector<indexedTriangle*>*  newList  =  new  std::vector<indexedTriangle*>;  //
initialises an empty list for next lower depth

         vectorOfTrianglesPerDepthPerFace[faceIndex]->push_back(newList); // add new depth
to back (index is effectively depth + 1)
         //foreach triangle in list

         std::vector<indexedTriangle*>*                    currentList                    =
vectorOfTrianglesPerDepthPerFace[faceIndex]->at(depth);

         typedef std::vector<indexedTriangle*>::iterator TI;
         for(TI i=(*currentList).begin();i!=(*currentList).end();++i) {
            indexedTriangle* tri = *i;

            std::cout << "tri.r: " << tri->r << std::endl;
            std::vector<indexedTriangle*>* tmplist = subdivideIndexedTriangle(*tri);
            //foreach(tri in tmplist) {
            for(TI j=(*tmplist).begin();j!=(*tmplist).end();++j) {
               newList->push_back(*j);
            }
         }
      }
      depth++;
   }
}

void  dynamic::triangleSubdivision::addCut(int  startPointIndex,  int  endPointIndex,  int
newPointIndex)
{
   // sanity
   if(startPointIndex==endPointIndex) return;

   // Always use lower index first as we could be traversing vertices in either order and
there is only one cut between vertices
   if(startPointIndex>endPointIndex) {
      int tmp=startPointIndex;
      startPointIndex=endPointIndex;
      endPointIndex=tmp;
   }
   // This will overwrite any existing value without checking.
   triEdgeCutTracker[(startPointIndex * 1000) + endPointIndex] = newPointIndex;
```

```
}

/* return -1 if none otherwise index of existing cut vertex */
int dynamic::triangleSubdivision::getCut(int startPointIndex, int endPointIndex)
{
   // A map assigns the default value of the mapped type if the key
   // doesn't already exist. In this case the default of an int is 0 so
   // treat a return value of 0 as the index not yet existing.
   // 0 will never be a valid cut index as is has to be some form
   // of sum between two triangle indices

   // Always use lower index first as we could be traversing vertices in either order and
there is only one cut between vertices
   if(startPointIndex>endPointIndex) {
      int tmp=startPointIndex;
      startPointIndex=endPointIndex;
      endPointIndex=tmp;
   }
   return triEdgeCutTracker[(startPointIndex * 1000) + endPointIndex];
}

int dynamic::triangleSubdivision::addTriangleVertex(glib::point3Df &point) {
   *back_inserter(triangleVertices)=point;
   std::cout << "Size:" << triangleVertices.size()-1;
   return triangleVertices.size()-1;
}

std::vector<dynamic::indexedTriangle*>*
dynamic::triangleSubdivision::subdivideIndexedTriangle(dynamic::indexedTriangle& tri) {

   // new empty triangle list to fill and return
   std::vector<dynamic::indexedTriangle*>* triList =
       new std::vector<dynamic::indexedTriangle*>;

   glib::point3Df p=triangleVertices[tri.p];
   glib::point3Df q=triangleVertices[tri.q];
   glib::point3Df r=triangleVertices[tri.r];

   int pqi; // halfway vertex indices
   int qri;
   int rpi;

   pqi=getCut(tri.p,tri.q);
   if(!pqi) { // there is no existing vertex cutting these two points
      glib::point3Df pq = (p+q); // new halfway point
      pq.X( pq.X()/2.0f );
      pq.Y( pq.Y()/2.0f );
      pq.Z( pq.Z()/2.0f );

      pqi = addTriangleVertex(pq);
      addCut(tri.p,tri.q,pqi);
   }
   // repeat for qr and rp

   qri=getCut(tri.q,tri.r);
   if(!qri) { // there is no existing vertex cutting these two points
      glib::point3Df qr = (q+r); // new halfway point
      qr.X( qr.X()/2.0f );
      qr.Y( qr.Y()/2.0f );
      qr.Z( qr.Z()/2.0f );

      qri = addTriangleVertex(qr);
      addCut(tri.q,tri.r,qri);
   }

   rpi=getCut(tri.r,tri.p);
   if(!rpi) { // there is no existing vertex cutting these two points
      glib::point3Df rp = (r+p); // new halfway point
      rp.X( rp.X()/2.0f );
      rp.Y( rp.Y()/2.0f );
      rp.Z( rp.Z()/2.0f );

      rpi = addTriangleVertex(rp);
      addCut(tri.r,tri.p,rpi);
   }

   triList->push_back( new indexedTriangle(tri.p,pqi,rpi) );
```

```
    triList->push_back( new indexedTriangle(pqi,tri.q,qri) );
    triList->push_back( new indexedTriangle(qri,tri.r,rpi) );
    triList->push_back( new indexedTriangle(rpi, pqi, qri) );
    return triList;
};
```

7 Accompanying CD-ROM disc

7.1 /source

This directory contains the full source code for the dynamic object VLF code. It is in the form of a Microsoft visual studio solution. \source\MS_VC_NET2003\vlfApps.sln is the main solution file. To compile and run the code the following extra open source code may be required:

The image debugger (http://www.billbaxter.com/projects/imdebug/) - used to produce some screenshots.

The OpenGL Extension Wrangler Library (http://glew.sourceforge.net/) – used to check necessary OpenGL extensions are available.

7.2 /video

Contains video files in Microsoft WMV9 format. These files demonstrate the application running in real time.

7.3 /misc

Contains AC3D scene files and the exported *.obj versions required to render the test scenes used. Also contains the VLF *.bin files for these scenes and the raw spreadsheets used to create the results section.