Point Based
Modelling and Rendering

Dissertation

Jeremy Campbell
ma2jpc@bath.ac.uk
BSc (Hons) Mathematics and Computing

May 2006
Point Based Modelling and Rendering

Submitted by Jeremy Campbell

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.................................................................
Abstract

As the resolution of 3D scanners increases, scans return an increasing number of sample points from an object. With this amount of data, it is sometimes less efficient to model these points as polygon meshes or interpolate mathematical formulae for them then just storing the points themselves as primitives and rendering them directly to the screen.

This project investigates the use of points as rendering primitives through the development of a point renderer. It focuses on realistic shading, while maintaining real-time operation with models of arbitrary complexity.

Previous research from other people into efficient data structures and rendering of primitives has been used as a starting point. This project then extends the work of previous point-based renderers by introducing several shading techniques, resulting in a versatile real-time software renderer. The project explores how the point model can be exploited to provide shading techniques such as shadows (on object and background), texture mapping and environment mapping.

This document guides the reader through the design, specification and implementation of the project and then tests the product against the specification to evaluate its successes and failures in what it set out to do.
Acknowledgements

I would like to thank my supervisor Professor Phil Willis who suggested this project and who has always taken the time to give any help or advice when I needed it.

I am also grateful to the Stanford 3D Scanning Repository (see [Stanford 2006]) for supplying the models used in this project and the Hotel U Důmu website: (http://www.udomu.360-panorama.net/) for the images used as backgrounds in this project.
# Contents

1 INTRODUCTION ........................................................................................................ 1

2 LITERATURE REVIEW ............................................................................................... 2

2.1 Data Structure ............................................................................................................. 2

2.2 Rendering ..................................................................................................................... 6

2.3 Shading ........................................................................................................................ 8

2.4 Optimisation .............................................................................................................. 12

2.5 Summary .................................................................................................................... 13

3 REQUIREMENTS SPECIFICATION ..................................................................... 15

3.1 Functional Requirements ......................................................................................... 15

3.2 Performance Requirements ..................................................................................... 15

3.3 Interface Requirements ............................................................................................ 15

3.4 Error Handling .......................................................................................................... 16

3.5 Portability .................................................................................................................. 16

4 DESIGN ........................................................................................................................ 17

4.1 Data Structures .......................................................................................................... 17

4.2 Point Attributes & Lookup Tables .......................................................................... 19

4.3 File Types ................................................................................................................... 23

4.4 Rendering ................................................................................................................... 23

4.5 Background ................................................................................................................ 25

4.6 Shading ....................................................................................................................... 26

5 IMPLEMENTATION ................................................................................................. 31

5.1 Language and Platform ........................................................................................... 31

5.2 Virtual Machine ........................................................................................................ 31

5.3 Maths Engine ............................................................................................................. 31

5.4 Error Handling .......................................................................................................... 32

5.5 Data Structures .......................................................................................................... 32

5.6 File Types ................................................................................................................... 33

5.7 User Interface ............................................................................................................ 33

5.8 Rendering ................................................................................................................... 34

6 TESTING ...................................................................................................................... 36

6.1 Requirements Testing ............................................................................................... 36

6.2 Software Testing ....................................................................................................... 38

7 CONCLUSION ............................................................................................................ 43

7.1 Improvements and Future Work ............................................................................ 44

8 BIBLIOGRAPHY ........................................................................................................ 45

8.1 References .................................................................................................................. 45

APPENDIX ....................................................................................................................... 47

A Accompanying CD ................................................................................................... 47

B Screen Shots ............................................................................................................... 47

C Help Documentation ................................................................................................ 49

D Code Listing ............................................................................................................... 51
1 Introduction

Now that the resolution of 3D scanners is so high, scans are returning an enormous number of sample points from a 3D object. When modelling with this amount of data, it is sometimes less efficient to join these points in a polygon mesh or interpolate mathematical formulae for them then just storing the points themselves as primitives. This alternative method of modelling objects requires a different approach to rendering them; one that creates an image from a set of discrete points in space. A point renderer must process each point primitive as quickly as possible because it works with substantially more primitives than a polygon renderer. It must also fill in the gaps between the points.

Modelling objects with point primitives, has advantages and disadvantages which must be considered to realise the appropriate usage of this type of renderer. For example, this technique would not be suitable for animated objects; the memory requirements for storing the points of each frame would be too large and performing transformations on subsets of the points would be too restrictive. Moreover, objects of low detail and soft curves would be more efficiently modelled with polygon-meshes or spline surfaces. This method is ideally suited to static objects of high detail such as statues, organs, etc.

Up to now, work has been done to create renderers that display point models without holes and with basic lighting models (such as QSplat [Levoy 1985]). This project will extend that work by investigating how materials, textures and shadows may be applied to the models so that they may be rendered with a higher level of photo-realism. The large number of primitives on the model surface and the inherent repetition in the point attributes will be exploited to develop efficient shading techniques.

The first objective of this project is to produce a fast and simple point-renderer that operates in real-time with objects of arbitrary complexity. The second objective is to develop pre-calculation strategies that apply colours to each point to achieve realistic shading effects.

The next section: ‘2 Literature Review’ contains a detailed study and evaluation of previous work in this area. In the following section: ‘3 Requirements Specification’ the key requirements for the projected are laid out, considering the aims of the project and the experience drawn from the literature review. The main body of this document will be found in section: ‘4 Design’ where the development of a solution that meets the design requirements will be described. The implementation of this design will then be documented in section: ‘5 Implementation’ and tested in section: ‘6 Testing’. The final section: ‘7 Conclusion’ will evaluate how well the project fulfilled its aims and how it could be improved upon with further work.
2 Literature Review

Although originally proposed in [Levoy 1985], modelling and rendering objects as discrete sets of points has just recently become a topic of interest amongst the research community. This has come from necessity after developments in 3D scanning equipment have resulted in data sets that are too large for traditional polygon mesh techniques to render in real time. This section describes some of the techniques that have already been developed in this area and in other areas of 3D graphics that may be applicable to this project. These techniques will then be combined and hopefully extended in this project in order to produce a point based renderer with a focus on photo-realism.

The following sections break the problem down into modules. For each aspect of the problem, some of the work that has already been produced by others will be critically evaluated and compared to evaluate the capabilities and limitations of each one.

2.1 Data Structure

Working with point primitives requires sophisticated data structures so that the large volumes of data may be effectively traversed. These data structures must be memory efficient and have fast access.

2.1.1 Structure

The development of a successful and popular point renderer called QSplat is described in [Rusinkiewicz and Levoy 2000]. QSplat generates a breadth-first tree of bounding spheres that encompass the bounding spheres of all of their children. These spheres are used in visibility culling to discount whole branches of the tree. Each node also has a normal cone that encompasses all of the normals of its children. This is used for back-face removal at an early stage, giving the possibility to mark whole branches as facing the camera or facing away from the camera. Therefore, this data structure improves upon an unordered sequence of points in space by aiding effective culling algorithms. They note that it also would extend well to a ray tracing/casting strategy of rendering.

[Pajarola 2003] describes an octree structure that adaptively partitions space depending on the sample points similar to the data structure in QSplat ([Rusinkiewicz and Levoy 2000]). However, it also includes the spatial extent (in the form of an elliptical disc) of each point. The disc is centered at the sample point and is contained in the plane that is orthogonal to the point normal. The disc is defined by two orthogonal vectors (in the disc plane) in the major and minor axis directions and with magnitudes of the major and minor disc radii respectively. The discs are defined such that they overlap with each other and there are no holes in the surface.

A different approach is presented in [Botsch et al. 2002] in which space is partitioned uniformly into a very compact octree data structure (requiring less than two bits per point on average). They show that their uniformly partitioned octree is optimal with respect to the balance between quantization error and sampling density; increasing one and not the other from this balance will have a smaller effect and will increase redundancy. The octree data structure that they propose partitions a cube in 3D space uniformly with each child node
representing an octant of the parent node. Each node is stored as an 8-bit code with each bit representing an octant in that node. The bit is set to 1 if the octant contains at least one of the point samples and 0 otherwise. They note that the total number of voxels increases like $O(n^3)$ but the number of voxels that intersect with the 2D point-defined-surface increases like $O(n^2)$. Therefore, there will be many more 0 bits than 1 bits as $n$ becomes large. This idea motivates an effective compression scheme in which sub nodes of octants with 0 bite-codes are ignored. This memory saving then propagates down the tree, resulting in a data structure where each point requires less than 3 bits of storage, independent of quantisation precision. They go on to suggest that this data structure is a good candidate for further compression, using an entropy encoder, due to a higher likelihood that a byte code will consist of four 1’s and four 0’s. This step then achieves a compression of less then 2 bits per point sample. Extra attributes, such as normals and colour data are stored separately in a parallel data stream, and efficient methods of compressing these attributes are also suggested.

2.1.2 Creating the structure

[Pajarola 2003] describes the use of generic homogeneous covariance matrices in calculating necessary sizes and orientations of elliptical splat discs. The sophisticated construction of this data structure makes it more efficient to use at the rendering stage and results in more accurate splat shapes than the similar data structure proposed in [Rusinkiewicz and Levoy 2000].

Starting with $N$ sample points, the average of all of the points $p$ is calculated and used as the center of a bounding sphere with a radius of the maximum distance of the $N$ points from $p$. The points are then transformed to a new coordinate system with $p$ as the origin and partitioned into octants with respect to the new coordinate system. This procedure is then recursively executed to progressively partition the points into an octree hierarchy and calculate bounding spheres for each node. The bounding normal cone for each node is also calculated in a similar way to the bounding spheres. An average of the point normals is calculated as the central axis of the cone and the maximum deviation of the normals from the average normal is used to find a bounding semi-angle for the cone.

The elliptical discs of each node are then calculated using generic homogeneous covariance matrices using method described as follows.

The covariance matrix of a set of $N$ points $p$, with average $p$ is given by:

$$M = n^{-1} \sum_{i=1}^{n} (p_i - p)(p_i - p)^T$$

However, this definition requires the covariance matrix to be redefined for every node in the tree (because every node will have a different average point). Therefore, the following matrix is defined:

$$M_h = n^{-1}RTM_hT^TR^T$$

Where

$$M_h = \sum_{i=1}^{n} (p_i^T,1)(p_i^T,1)^T$$
is the generic homogeneous covariance matrix of the points in world space. T and R are used to transform each point to a new coordinate system, centered at p and rotated such that the average normal of the points is the vertical axis. The covariance matrix M can now be generated by finding $M_h$ and dropping the forth row and column. Notice that M no longer depends on p and the generic homogeneous covariance matrix of a parent node can be found by simply summing the generic homogeneous covariance matrices of its child nodes. This reduces the original calculation from an $O(N \log N)$ operation to $O(N)$.

Once the covariance matrix M for a node has been calculated, the dimensions of the elliptical disc can be found in the following manner. The axis ratio of the disc is given by the eigenvalues of the upper-left 2x2-sub-matrix of M and the major and minor axis directions are the eigenvectors of the matrix (2D vectors in the plane with normal equal to the average normal of the points of the node). The major and minor axis lengths of the disc must be scaled to bound all of the points in the node (and their respective discs) as Figure 2.1 shows.

![Figure 2.1: From Pajarola 2003: a) bounding ellipse of points and b) conservative bounding disc.](image)

The discs of leaf nodes are calculated in a similar fashion as described above but only scaling the ellipse to sufficiently bound the neighbouring points considered. This method of construction is a one-time pre-computation that results in an effective hierarchical data structure that models a discrete set of point samples as a surface composed of elliptical discs with no holes.

This data structure becomes very useful in several algorithms because it organises an unordered set of point samples into sets of points in the same spatial neighbourhood of each other. Furthermore, these sets can be reduced to any size necessary.

2.1.3 Tree Traversal

In QSplat ([Rusinkiewicz and Levoy 2000]), the breadth first ordering of the tree allows the data structure to be read from the disc on-the-fly so that a low-resolution rendering can be displayed while the rest of the data is still being read from the disc. The equivalent is also true if the data is being transmitted over a data connection. The use of pointers in the tree structure is minimised by having only one pointer for all children of a node that points to the beginning of the data for the children of these nodes. This results in about 8-10% of the storage space of the tree being devoted to pointers.

On the other hand, [Botsch et al. 2002] considers both depth first and breadth first tree traversal for the voxel grid data structure that they implement and concludes that depth first storage ensures minimal memory requirements and maximal rendering performance.
2.1.4 Interpolating extra points

In [Alexa et al. 2001] a method is developed that interpolates continuously differentiable surfaces from discrete point sets using a moving least squares approach. Methods of up/down sampling the surface are developed and a rendering system is proposed, in which holes are filled by interpolating sufficient points in object space to render each point as a pixel without leaving holes in the surface. The technique is not particularly suited to real-time rendering, but could be used during modelling to increase point density in order to adequately sample textures mapped to the surface of point modelled objects. The following describes how the method works, together with Figure 2.2 below.

A plane is defined close to a small collection of points by minimising the square distances from the points projected onto the plane. The points $p_i$ are projected onto the plane and the vertical distances to their projection points $f_i$ are used (with a weight depending on their distance to the point of interpolation $q$) to interpolate a polynomial surface, by minimising:

$$\sum_{i=1}^{N} (g(x_i, y_i) - f_i)^2 \exp\left(\frac{||p_i - q||^2}{h^2}\right)$$

where $g$ is the interpolated polynomial defined on the plane and the $\exp(\cdot)$ factor is a radial Gaussian weighting function. This is depicted in Figure 2.2.

Figure 2.2. From [Alexa et al. 2001]: “First a local reference domain $H$ for $r$ is generated. The projection of $r$ onto $H$ defines its origin $q$. Then a local polynomial approximation $g$ to the heights $f_i$ is computed. The projection of $r$ onto $g$ is the result of the moving least squares projection procedure.

This surface is conjectured to be infinitely smooth. The parameter $h$ in the weighting function can be used to tune out features/noise of size $< h$. Small $h$ values result in more local approximations and larger values start to smooth out sharp features of the surface.

Note that the data structure proposed in [Pajarola 2003] would facilitate this procedure because it arranges the points into spatial neighbourhoods of each other so it would be simple to collect small subsets of points that are close to one another.

2.1.5 Lookup tables and compression techniques

Due to the large number of point samples required in point based rendering strategies, it is vital to store the data about each point as compactly as possible. This is often achieved by quantising the data with lookup tables. This is also allows calculations to be done on the lookup table, instead of on each point.
moving the complexity of the calculation to the size of the lookup table instead of
the complexity of the model.

In QSplat ([Rusinkiewicz and Levoy 2000]), the position and radius of each
sphere is stored as a 13-bit index into a lookup table of point vectors and radii
values. The hierarchical structure is exploited by making this data a multiple of
the position and radius of the parent sphere. The radius and center coordinates
are multiples of 1/13 of the radius and center of the parent sphere. This
quantization process is rounded up so that no holes appear in the surface and all
child spheres are enclosed by their parent. Hence, only 7621 of the 134
combinations are valid to fulfil these criteria and 13 bits are sufficient to index
these values. The normal for the node is coded as a 14-bit index to a lookup table
(with a mean quantization error of 0.01 radians) and 2 bits are used for the width
of the normal cone (sin(cone-half-angle) = 1/16, 4/16, 9/16 or 16/16). The
remaining 3 bits are used to specify the number of children for this node in the
tree structure. These compression methods result in a mean 4% error in the
position co-ordinate and 15% error in the sphere radius (due to rounding to
ensure child spheres are enclosed by parents). Assuming an average branching
factor for the tree of 3.5 results in a memory size of around 6 bytes times the
number of point samples.

[Botsch et al. 2002] suggest that normals and colours are stored as indices to
lookup tables. Starting with a regular octahedron (eight triangles) and then
repeatedly splitting each triangle into four triangles to obtain a uniformly
distributed collection of normals (of the triangles) in 3D space. Using a 13-bit
lookup code allows up to 8192 different normal directions. This level of accuracy
was sufficient in all of their test cases. An obvious extension to the normal lookup
table is to then pre-calculate lighting data per normal instead of per point and
use the normal index to directly access the light intensity for each point. This
moves a lot of complexity of lighting calculations away from the complexity of
the model.

2.2 Rendering

2.2.1 Transformations

The memory efficient data structure described in [Botsch et al. 2002] is exploited
further to perform matrix transformations more efficiently, using the same
spatial coherency strategy that the data structure employed. Matrix projective
transformations (using 4x4 matrices) normally require 14 additions, 16
multiplications and 3 divisions per point;

\[ p_i' = Mp_i \]

However, in the uniformly partitioned octree the displacement from the center of
a node to one of its octants (call it q) is not dependent on the data contained in
the structure and can be pre-calculated and reused. Hence if \( p_i = p + q \) where p
is the center of the parent node of the leaf node that represents point \( p_i \) and q is
the displacement vector from the p to \( p_i \) then:

\[ Mp_i = M(p + q) = Mp + Mq = p' + q' \]

Where \( p' = Mp \) is known by the parent and \( q' = Mq \) may be pre-calculated.
Reusing results from parent nodes, the average number of operations being performed per point (leaf-node of voxel grid) is 4 additions and 2 divides.

2.2.2 Culling

In QSplat ([Rusinkiewicz and Levoy 2000]), view frustum culling is performed by testing each sphere against the planes of the view frustum. If the sphere is completely contained within the frustum, then that branch of the tree can be marked and no further tests need to be carried out on it. Similarly, if the sphere is outside the frustum, the branch can be pruned and no children of that node will have to be processed.

Similarly, in QSplat ([Rusinkiewicz and Levoy 2000]), back-face culling is performed by checking the normal cone of each node against the view direction. If the cone faces entirely away from the camera, that node, and all of its children need not be processed further. Similarly, if the cone faces entirely toward the camera, the node can be marked and no further back-face culling tests need to be performed on that branch.

[Zhang and Hoff 1997] suggest a novel method of back-face culling requiring just one logical operation and an extra two bytes per rendering primitive. By grouping normals into clusters and assigning each cluster a bit-mask, each rendering primitive can be assigned a mask for the cluster that contains its normal. Before rendering a frame, a back-mask is generated by ORing the bit-masks for all of the back-facing normals. Now, to test if a rendering primitive is back-facing, its bit-mask is checked against the back-mask with an AND operation to check if its normal is contained in any of the back-facing clusters.

In the paper, the normals are split into 1536 clusters. Instead of storing a 1536-bit mask for each primitive, the mask is treated as a byte sequence and is stored as two bytes; the first byte is an offset and the second is an 8-bit mask. The mask is then checked against the back-mask like so:

\[
\text{BackMask}[\text{byteOffset}] \& \text{bitMask}
\]

Consideration has to be made for perspective views because the viewing position can also determine whether a normal is culled or not. This is handled conservatively by only adding normals that are not viewable from any direction in the field of view to the back-mask. This means that further checks have to be made on normals that pass the initial culling with the normal masks. This method of back-face culling is significantly faster than the standard method using the dot product of the normal and the viewing direction.

This idea can be implemented very easily with the indexed normal system suggested in [Botsch et al. 2002] and other literature, by storing a boolean value together with the normal in the lookup table. The back-face-culling test for each primitive can then be performed by using the index for its normal to check if that normal is back-facing. This has the advantage that no extra data has to be stored for each primitive. This will require an extra pre-processing step per frame to set the boolean value of each normal depending on whether it is culled or not. However, the large number of points to number of normals ratio means that this method will still save a lot of computation time.
2.2.3 Splat Shape

[Rusinkiewicz and Levoy 2000] explore three different kernel shapes for rendering points to the screen; squares, circles and circles with an approximate Gaussian alpha fall off. They note that the respective speeds compared to the square splat for the circle and fuzzy circle splats are: 2 times and 4 times. They show that aliasing is the most severe for the square splat and least for the fuzzy circle splat. The fuzzy circle splat requires that the splats be drawn in back-to-front order for accurate alpha blending. They implement this with a multi-pass rendering strategy. They also experiment with elliptical splats (depending on the orientation of the splat normal) showing that they improve the quality of silhouette edges, compared to circular splats. However, this splatting technique may result in holes appearing in the surface.

Elliptical splats are implemented successfully in [Pajarola 2003] due to an effective method of data-structure generation (see ‘2.1.2 Creating the structure’). The elliptical discs that are defined for each point primitive (octree leaf node) are large enough to bound all neighbouring points and the discs of non-leaf nodes are large enough to bound all discs of their child nodes. As each splat is the optimum shape to avoid holes while having a minimum surface area, the splats are visually attractive and are less prone to artefacts.

2.2.4 Progressive Detail

QSplat ([Rusinkiewicz and Levoy 2000]) implements a progressive rendering strategy to provide real-time frame rates for arbitrarily large models. If the projection of a sphere on the screen is below a certain threshold size, it is rendered to the screen without processing the children of that node. This threshold is adjusted dynamically to provide interactive frame-rates and then progressively reduced when user input stops and the scene becomes static. This results in a more pleasant user experience and makes QSplat well suited for low-end systems with poor memory and processing capabilities. When a frame has already been rendered to the screen and no further changes have occurred to the scene, the frame is rendered again at progressively higher resolutions until the highest quality image has been produced. They note that the speed and subtlety of this refinement procedure does not result in noticeable effects.

2.3 Shading

2.3.1 Lighting

The most common lighting model in use today in real time rendering is still the Phong illumination model (originally proposed in [Phong 1975]) due to its simplicity and computational ease. This is a local reflection model so it only considers reflections of direct rays of light from each of the light sources to the viewpoint, which it models as diffuse (equal scattering) and specular (perfect mirror) reflections. All other global effects (light arriving from other areas of the scene) are approximated with a constant ambient term.

The Phong illumination model states that the intensity (for each colour component: red, green, blue) for a pixel is given by:

\[ I = k_d I_d + \sum_{\text{lights}} (k_d I_{d,d} + k_s I_{s,s}) \]
Where \((k_a,k_d,k_s)\) are coefficients that define the properties of the material such that \(k_a+k_d+k_s=1\), the ambient term; \(I_a\) is a constant light intensity defined by the renderer and diffuse and specular terms; \(I_{a,d}\) and \(I_{a,s}\) (for each light; \(\lambda\)) are given by:

\[
I_{\lambda,d} = I_{\lambda} \cos \theta = I_{\lambda} (L \cdot N) \quad \quad I_{\lambda,s} = I_{\lambda} \cos^n \phi = I_{\lambda} (R \cdot V)^n
\]

Where:
- \(I_{\lambda}\) = Intensity of light \(\lambda\)
- \(\theta\) = Angle between \(L\) and \(N\)
- \(n\) = affects size of specular highlight
- \(\phi\) = Angle between \(R\) and \(V\)
- \(L\) = Point-to-light source vector
- \(R\) = \(L\) reflected about \(N\)
- \(N\) = Point normal
- \(V\) = Point-to-viewpoint vector

The assumption is often made that light sources and the viewer are an infinite distance away so that \(L\) and \(V\) can be assumed to be constant, thereby simplifying the calculation. Another approximation that is often made (sometimes referred to as Blinn’s method) is to note that:

\[
R \cdot V \approx \frac{1}{2} N \cdot (L + V)
\]

This reduces the computation involved in the specular term.

The light sources modelled in the equations above are point light sources that emit light in all directions. Directional light sources can be modelled by replacing the constant (for each light) \(I_i\) term with:

\[
I_{\lambda} = I_i \cos^m \phi = I_i (-L \cdot L_s)^m
\]

Where:
- \(I_i\) = constant light source intensity
- \(L\) = Light source-to-point vector
- \(\phi\) = Angle between \(-L\) and \(L_s\)
- \(m\) = affects the spread of the light source
- \(L_s\) = vector direction of light source

This simulates a light source with direction, such that the intensity of the light falls away as the vector toward the light source deviates from the light source direction. Note that the vectors \(L\) and \(V\) cannot be considered constant when using this technique.

The Phong illumination model gives fairly realistic results given the computation time involved but all the materials that it models tend to look plastic-like. Further research has been done to extend this illumination model to model a more diverse range of materials. For example [Blinn 1977] and [Cook and Torrance 1981] extend the specular term to model shiny metallic-like surfaces more realistically. The differences are subtle though and this technique is still not as popular as the original Phong model, due to the added computational requirements. Lighting quickly becomes too expensive for real-time rendering, even with local reflection models and it soon becomes more appropriate to use global illumination techniques such as ray-tracing or radiosity, etc.

### 2.3.2 Texture Mapping

Mapping textures to surfaces can simulate many shading effects and can reduce the necessary complexity of object geometry. This approach can greatly enhance the quality of a rendered image at relatively small cost. This process (as described in [Watt 1999]) can be achieved by mapping a two-dimensional image to a 3D
This is usually accomplished using an intermediate surface, such as a cube, cylinder or sphere and projecting a point on the surface of an object along its normal to the intermediate surface. The main problem with this technique is aliasing that results from sampling of the discrete texture image or from the geometric differences between the object surface and the intermediate surface. The texture image sampling problem can be alleviated by using a number of techniques described below:

- **Bilinear interpolation** – by projecting the screen pixel onto the texture image, a weighted average of the texture pixels that it covers can be calculated.

- **Mip-mapping** – a texture image is stored as a sequence of progressively smaller (by half) images. Then the appropriate image (mip-level) can be selected depending on the detail required.

- **Trilinear interpolation** – involves bilinear interpolation on two mip-levels and then a further interpolation between those two values.

This method of using intermediate surfaces to map textures to object is independent of object representation and can easily be applied to point models. Furthermore, it can be applied as a pre-calculation strategy by assigning colours to each point sample. However, care must be taken to ensure that the texture is adequately sampled (maybe by interpolating new points with the method described earlier by [Alexa et al. 2001]) to avoid aliasing.

### 2.3.3 Solid Textures

The idea of solid textures was proposed around the same time by [Perlin 1985] and [Peachey 1985]. They introduce the idea of ‘solid texturing’ whereby a texture is defined in a three dimensional region of space. This allows a complex object to be textured by simply mapping a 3D point on its surface to a 3D point in texture space. This gives the visual effect that the object was sculpted from a block of material with that internal texture. This technique works very well for applying textures of solid materials, such as stone and wood. However, this method is not suitable for textures that are 2D in origin, such as photographs of the object that must be mapped to the surface because it would be very difficult to create a useful 3D texture map.

[Perlin 1985] goes on to explain a method of simulating the textures of materials such as wood and marble using procedural methods based on a random noise function (now more commonly known as the ‘Perlin noise function’). A Perlin noise function is a random number generator that uses one or more parameters such that the corresponding output is always the same for each set of parameter values in the domain. It must also satisfy the conditions that it is statistically invariant under rotation or translation of the domain and its output frequency is narrow band pass limited over the domain. A realistic procedural solid marble texture, for example, can then be defined using this function as follows:

\[
\text{texture colour at (x,y,z) = marble_colours[COS(x + perlin_noise(x,y,z))]}
\]

There are many applications for Perlin noise functions for modelling natural materials, effects and behaviours, etc.

During the planning of this project it was decided that point-based rendering was most suited to static scenes of high complexity, such as statues. Given that
statues are normally carved from stone and have high detail, this method of texturing will be ideal for application in this project.

2.3.4 Environment Mapping

This idea was first suggested by [Blinn 1976]. By rendering several different views from the center of an object into a texture map called an ‘environment map’, reflections in objects can be simulated by mapping the environment map to the surface of the object. The area of the environment map that a projected pixel occupies is proportional to the curvature of the surface. This can result in inadequate sampling of the environment map causing aliasing. Anti-aliasing techniques (such as those described in Texture Mapping earlier) are necessary but need not be very sophisticated because incorrect reflections are not very noticeable to the human eye. This method is not correct for concave objects that reflect themselves because this cannot be incorporated into the environment map. This method approximates the quality of a ray-tracer when rendering shiny objects that reflect the environment and can be implemented in real-time. It is also independent of object representation and can be applied to a point renderer easily. Although the environment map need only be generated once as a pre-calculation (for static scenes), the mapping must be done every time the viewpoint changes because it is dependent on the viewing direction. Therefore, this technique may be too computationally expensive for a point based rendering system, where there are a lot of primitives.

2.3.5 Shadow Mapping

The most visually significant weakness of local reflection models is that they do not generate shadows. Shadows are very important to the human eye; giving information about the relative positions of objects in the 3D space and information about the location of light sources. Rendered images that do not have correct shadows look artificial and objects appear to be floating above surfaces. As a global illumination model is too computationally expensive to implement in a real-time renderer, some algorithms that specifically generate shadows must be investigated.

[Williams 1978] describes the concept of a shadow buffer. For a single light source, this pre-calculation strategy involves passing a scene through the rendering pipeline, using the light source position as the viewpoint, to generate a z-buffer. This z-buffer is then called a ‘shadow buffer’ and defines a surface where points on the far side are in shadow (from the light source considered) and points on the near side are in light. This buffer is used during rendering to determine if a point is in shadow or not by comparing the distance of the point from the light source to the relevant value in the shadow buffer. If the point to light source distance is greater than the shadow buffer value then the point is deemed to be in shadow. This method works independently of object representation so can be easily implemented in a point-based rendering system.

This requires a shadow-buffer for each light source, which must be recalculated each time its corresponding light source moves. However, the major consideration when using this method is aliasing because the shadow buffer is both created and accessed using point sampling. Aliasing can occur in two ways when using this method:

- When indexing the discrete shadow buffer
• Each value in the shadow buffer is an average distance for the region of space it represents.

The main effect of this is normally that shadow edges have jagged edges where the resolution of the shadow map is not high enough. Another effect of aliasing in shadow maps is ‘self shadowing’, where a point is incorrectly decided to be in shadow when it is very close to the shadow surface because of a rounding error during the calculation. This can be prevented by taking away a small amount from the distance of a point to the light source in order to make it slightly closer to the light source.

This method can be used to apply realistic shadows to a scene in real time but has aliasing problems that need to be considered.

This concept is developed further by [Stamminger and Drettakis 2002] to reduce the amount of aliasing. The method is view dependent and requires a new shadow map to be generated for each frame. By transforming a scene and the light source into camera space (view dependent) in order to generate the shadow map, the resolution of the shadow map is proportional to the proximity to the viewpoint; i.e. resolution is higher near the viewpoint and decreases for objects that are further away. This does not incur any more overhead in the shadow map generation algorithm because it just adds one more transformation to the set of transformations that are already applied to each rendering primitive.

This method would work well with point based rendering and would improve the quality of the shadows. However, generating the shadow map per frame, instead of once as a pre-calculation, would almost double the rendering time. Therefore, this method may have to be implemented in a refinement rendering or as an optional extra.

2.4 Optimisation

A common rule that is stated about optimisation is that it should not be attempted until after the coding has finished and that only time-critical areas of the code should be optimised. This is because optimisation normally results in code that is less readable, maintainable, and more difficult to debug. With this in mind, it is useful to be aware of the usual pitfalls that result in slow code and have an idea of the speed implications that result in certain coding decisions. These concerns are particularly relevant in the field of computer graphics because rendering is normally a very processor intensive procedure.

The following list presents some rules and tips that have been gathered from [LaMothe 2003]:

• Consider using 4x3 matrices instead of 4x4, or even performing transformations manually (unwinding matrices completely) as appropriate

• When interpolating z in view space it is better to interpolate 1/z because of the perspective transformation. This value can be used for perspective-correct texture/light/etc mapping

• Use a bias in the z-buffer (i.e. add a constant that depends on the frame number to the z-value before insertion) so that if can be used in several frames before being cleared (then clear with ASM code)

• When casting a float to an int, use ASM { fld f; fistp i; }
• Accessing data with array notation is faster than incrementing the pointer and then accessing it. That is: \(a[0] = 1; a[1] = 2\); is faster than \(*a = 1; a++; *a = 2;\)

• Place all related code into one file to help compiler to optimise the code

• Make predicates in if statements likely to be true (“branching the pipeline”)

• Align data structures to word length because the processor reads data as words

• Implement alpha-blending with a lookup table indexed over 8 alpha values (0,1 are special cases) and 16-bit colours (lookup table = 512Kb)

[LaMothe 2003] also provides code samples and tuition in the basic use of DirectX for plotting pixels to the screen in a Windows environment.

### 2.5 Summary

Several forms of data structure have been investigated, each with slightly different approaches and goals. In QSplat [Rusinkiewicz and Levoy 2000] they partition space adaptively (in to a bounding sphere hierarchy) so that the data is spread evenly over the tree whereas [Botsch et al. 2002] chose to uniformly partition space into a voxel grid. The voxel grid is shown to be very efficient in terms of memory requirements and rendering time. Although the bounding sphere hierarchy has several useful features (such as effective hole filling) and is a more convenient structure to use to interpolate extra points (using method described in [Alexa et al. 2001]) or perform ray-tracing, etc. The first objective of this project is to produce a fast and simple renderer that will make use of pre-calculation strategies later in the project. Therefore, the simple and efficient voxel grid data structure proposed in [Botsch et al. 2002] will be more suitable.

After investigating both methods of tree traversal [Botsch et al. 2002] decides to use depth first octree ordering for efficiency and algorithmic simplicity. However, [Rusinkiewicz and Levoy 2000] chose to use a breadth first ordering to allow the data to be loaded dynamically, as the object is rendered, from the disc or a data connection. Since on-the-fly object loading is not an objective of this project, it will be more suitable to use a depth first ordering for the octree data structure implemented in this project.

The generation of the data structure in [Pajarola 2003] assigns bounding elliptical discs to each point primitive in a very elegant algorithm. These elliptical discs are conducive to an effective hole-filling algorithm and produce a visually attractive splat shape. Therefore, it would be advantageous to add these attributes to the voxel grid data structure if possible. These splats could also be implemented with the alpha-blended Gaussian splats used in [Rusinkiewicz and Levoy 2000].

Using lookup tables for the normal of a point sample is a common strategy in point modeling because there are many more point samples than the number of normals in the table, so many points will have the same (or very similar) normals, causing redundancy. [Botsch et al. 2002] extend the use of this idea to use the normal lookup table to do lighting pre-calculations. Also, the novel method of back-face culling proposed by [Zhang and Hoff 1997] can also be incorporated into this table. This is a powerful tool and may be the basis of several other pre-calculation strategies.
Octree data structures lend themselves very nicely to culling algorithms because if a node can be culled, then all of its children may be culled without further calculation. In the voxel grid proposed in [Botsch et al. 2002] view frustum culling is achieved by testing the voxel-cube that a node represents against the view frustum, and similarly in the bounding sphere hierarchy proposed in [Rusinkiewicz and Levoy 2000] the bounding sphere is tested. If these volumes are either completely contained or completely external to the viewing frustum then their children can be marked accordingly and no further calculations are necessary for them. A common strategy for back-face culling is to define a ‘normal cone’ for tree nodes that contain all the normals of all of its child nodes. Therefore, if a normal cone faces completely toward or completely away from the viewpoint, its children can be marked as such. Occlusion culling is not a common topic in point based rendering because z-buffers already work well and point data is normally obtained from 3D range scanners that cannot work with significantly self-occluding objects.

The realism and computational ease of the Phong illumination model (first proposed in [Phong 1975]) make it the obvious choice when creating any real-time renderer. As more advanced models result in more computation but yield relatively small/subtle effects, it will probably be more efficient to implement other shading strategies to achieve higher realism, such as texture mapping and environment mapping. It was noted that procedural solid textures ([developed by Perlin 1985] and [Peachey 1985]) would be eminently suited to this project as they work well with static scenes of high complexity, such as statues. Additionally, procedural solid textures for natural materials (such as stone and wood) can be realistically generated using Perlin noise functions ([Perlin 1985]), which will also be suitable for 3D models of statues.

As a further step to improve the photo-realism of the renderer, shadows could be added to a scene using the shadow buffer described in [Williams 1978]. However, it is likely that this method would produce severe aliasing effects, so the technique of perspective shadow mapping (proposed in [Stamminger and Drettakis 2002]) could be used as a refinement, after the frame has first been rendered with a regular shadow buffer. The perspective shadow buffer requires the buffer to be recalculated once for every frame, instead of just once each time the light source is moved. It is therefore more computationally expensive, but it reduces the amount of aliasing and results in more visually attractive shadows.
3 Requirements Specification

The following sub-sections outline the requirements of this project. The word “must” will indicate that the requirement is essential for the completion of the project and the word “should” will indicate a preferred but non-essential requirement.

3.1 Functional Requirements

- Must load model data from a suitable file format into memory as a set of 3D points
- Must render point data to the screen
- Must interpolate a surface between the points and fill in all holes
- Must cull occluding points so that hidden surfaces are not rendered
- Must apply a basic lighting model
- Colours must be able to be assigned to each point primitive
- Should allow different materials to be simulated in the lighting model
- Should allow textures to be mapped to the surface of the object
- Should model and render a simple background behind the object
- Should show shadows in the scene
- Should reflect environment in the object

3.2 Performance Requirements

- Must render in real-time, at reduced quality if necessary
- Should allow models of arbitrary complexity to be loaded and rendered in real-time
- Should render at a progressively higher detail when the viewpoint is stationary
- Must implement culling techniques to avoid unnecessary processing of primitives
- Must be stable and never freeze or terminate unexpectedly
- Must remain responsive and inform the user when intensive processing causes inactivity

3.3 Interface Requirements

- Must allow user to define the scene by specifying which model to load
- Must allow user to edit shading options of the renderer
- Must allow user to alter camera position/direction using the mouse and keyboard
- Must allow user to exit the application
- Must have some form of Help documentation
3.4 Error Handling

- Must free any resources and exit if an error occurs
- Must record the reason for the error

3.5 Portability

- Code should be written so that it can easily be ported to other operating systems
4 Design

The project has been broken down into modules to make the code more navigable and so that sections of the code can be developed and tested independently of each other. The following subsections describe the design of each module.

4.1 Data Structures

When working with large data sets, it is necessary to arrange the data into a structure that will:

- Reduce and bound memory requirements
- Reduce and bound rendering time (possibly to a minimum frame rate)
- Aid in culling so that subsets of the data need not be processed if not necessary

The goal of this module is to develop a data structure that can be generated with the data supplied in a standard ply file (vertex and face data), which will fulfil these criteria.

The binary voxel grid proposed by [Botsch et al. 2002] was decided to be a good starting point for a data structure because it has a small memory usage and offers optimisations to rendering speed. The binary voxel grid data structure is an octree where each node is a cube bounding a region of space. The eight child nodes of a parent node refer to the eight octants of that cube. In the following text, the word voxel and block will both refer to the cube represented by a node of the binary voxel grid. The leaf nodes of the tree define points in 3D-space as the center of the cube corresponding to that node.

The data structure will be traversed in a depth first order because the traversal algorithm is much simpler and faster. Speed was an important design requirement of the renderer so that extra shading features could be added.

The data structure will be created from an unordered array of points and normal vectors in 3D-space. It will consist of a data stream (a stream of byte codes that describe the locations of the points) and an attribute stream (a stream of attribute values in the same order that the leaf nodes appear in a depth first traversal of the data stream). In the data stream, a byte code is calculated for each node of the tree by ensuring that the bit referring to the octant that contains one of the points of the object is set to 1. Error! Reference source not found. shows the byte codes for the two-dimensional case. In this diagram, a byte code of 9 would imply that the top left quadrant and the bottom-right quadrant both contain points, whereas the other quadrants do not. In the three-dimensional case this pattern is extended to 8-bits (one for each octant) with 256 separate byte-codes.
Although the data stream can be compressed by removing all zeros, it will begin by having a byte for every potential tree node for algorithmic simplicity. Then, when all of the points have been considered, the zero bytes may be removed. This makes the algorithm very memory intensive (although the memory footprint is constant for each grid-depth size, regardless of model complexity). Therefore the maximum grid depth is will be set as 9, making the memory footprint of the method ~20Mb. Larger grid depths may be achieved with multiple passes of this algorithm for sub-regions of the object.

The algorithm to create the data structure is as follows:

- Transform all points such that the minimum of all the points is the origin
- For each point p
  - Set current_node = array index of root node in data stream
  - Set current_block = grid cube that encompasses all points (root block)
  - For i = 1 to grid depth
    - Find which octant, of current_block, point p is in and update byte code of current_node (ensure bit of corresponding octant is set to 1)
    - Set current_node = array index of the child of current_node which contains p
    - Set current_block = corresponding octant of current_block
  - Set attributes for point in attribute stream and store data stream index
- Reorder attribute stream using data stream indices stored in previous step
- For leaf blocks that contain more than one point, take average of the attributes (normals, colours, etc) and replace the attribute entries with one average attribute entry in the attribute stream
- Compress data stream by removing all zero bytes

This algorithm is of O(n) complexity. This is optimal complexity because every point must be considered and it is assumed that the points are unordered so there is no data coherency that can be exploited. However, the speed of this algorithm is not of large concern because it is only executed once as a precalculation.

This data structure achieves each of the aims set out for this module:

- The memory requirement (for the point data) is reduced from 96 bits (3 floats) to ~3 bits per point. In [Botsch et al. 2002], they note that the total number of blocks increases like O(n^3) (when n is the number of points) but the number of blocks that intersect with the 2D point-defined-surface increases like O(n^2). Therefore, there will be many more 0 bits than 1 bits as n becomes large. The structure of the tree allows the 0 byte-codes to be omitted from the data stream without loosing information about the tree structure. This memory saving then propagates down the tree, resulting in a data
Design

structure where each sample point requires less than 3 bits of storage on average, regardless of quantization precision. The other attributes (such as normals, etc) are also compressed. This is described later in the section ‘4.2 Point Attributes & Lookup Tables’.

- The spatial coherency of the point data in this data structure will also offer reductions in rendering time. This is described later in ‘4.4 Rendering’.
- Occlusion culling can be aided by a very basic form of the Painter’s Algorithm: Rendering of points that will later be obscured by other points (hidden surfaces) is reduced by creating 8 point grids for an object (one for each octant of its bounding cube). These point grids are then rendered in order of their proximity to the viewpoint (i.e. closest to the viewpoint first) so that nearer points are rendered first.

The memory requirements and rendering time of an object are bounded by the depth of the voxel grid. The grid depth relates to the finest level of spatial precision for the point data; i.e. the maximum resolution is: \(\text{grid-size} / (2^{\text{grid-depth}})\). However, [Botsch et al. 2002] showed that the balance between quantisation error and sampling density of the points is optimal – increasing one and not the other will increase redundancy in the data. Figure 4.2 illustrates this with a simple example of circles drawn from points with different levels of noise and different sampling densities. In the example, there is data-redundancy in the top-right circle and the bottom-left circle; the quantisation precision is unnecessarily high in the top-right circle because the error is dominated by the sampling frequency. Likewise, in the bottom-left circle, the high sampling frequency is wasted by the noise in each sample. The top-left and bottom right circles have the least redundancy because the noise and sampling error are of the same order.

![Figure 4.2: Point based rendering of a circle with different quantization levels (left:5-bit, right:10-bit) and different sampling densities (top:2\pi/32, bottom:2\pi/1024), from [Botsch et al. 2002].](image)

### 4.2 Point Attributes & Lookup Tables

Lookup tables are a very efficient method of performing CPU intensive tasks at a more convenient time or representing a value in less memory by storing it as an index code (requiring less memory) into an array of common values it may take. One of the advantages of using points as a rendering primitive is that the large number of primitives means that many will share the same or very similar properties. Therefore, these properties may be collected in a lookup table which is indexed by each point, allowing pre-calculations to be performed on the lookup table data and the results shared among the points.

The following subsections describe the design of three lookup tables that will increase the efficiency (in both memory requirements and rendering speed) of the renderer.
4.2.1 Normals

Each point normal is comprised of three 32-bit floating point numbers. For complex objects consisting of hundreds of thousands of points, this space requirement too large, so the normal is quantised and stored as an index into a lookup table. As suggested in [Botsch et al. 2002], a set of uniformly distributed normals could be generated by starting with a regular octahedron (eight triangles) and then repeatedly splitting each triangle into four triangles, taking the normal of each triangle (as shown in Figure 4.3). The normal index is constructed in the following way: The first three bits refer to one of the eight initial triangles of the octahedron and every two bits after that refer to one of the four triangles resulting from recursively splitting that triangle. This allows any precision of normal to be obtained by simply adding two more bits to the lookup table index. Using a 13-bit index allows up to 8192 unique normal directions. This level of accuracy was sufficient in all of their test cases.

![Uniformly divided octahedron, used to define and index normal vectors, from [Botsch et al. 2002].](image)

Quantising normal values in this manner has more significant advantages than just reducing memory usage. As there are many times more points than quantised normals, many points will share the same normal. Therefore, many normal-dependent calculations, such as lighting and back-face culling, can be made more efficient by performing the calculations on a per normal basis instead of per point.

In the Phong Illumination Model (for an infinite light source and infinite viewpoint), the light intensity at a point on the surface of an object depends on the normal of that point, the viewing direction and object/view independent data about lighting conditions. Therefore, the lighting calculations can be performed on the normals in the lookup table and the results stored alongside each normal. These values can then be directly indexed at render-time. Lighting is explained in more detail later in the section ‘4.6.1 Lighting & Materials’.

Back-face culling can be performed (in world space) by checking each normal in the lookup table with the camera direction: if their dot product is greater than zero then normal is flagged as back-facing.

Environment mapping is performed by reflecting a view ray (directed half-line in the viewing direction originating from the viewpoint) in the normal vector of a point on the surface of an object. The colour reflected from the environment at that point is then obtained by finding the intersection of the resulting ray with the closest surface and taking the colour at the intersection point. By ignoring the position of the surface point and viewpoint, the environment map colour is a function of only the point normal and the viewing direction. Therefore, environment mapping (with infinite viewpoint and infinite background) can be pre-calculated on a per-normal basis by reflecting the view direction in the normal vector. The environment colour is obtained by intersecting the resulting ray (originating at the object origin) with the background and is stored in the normal lookup table so that it can be directly indexed. This approach is explained in more detail later in the section ‘4.6.4 Environment Mapping’.
This means that lighting, culling, and environment mapping operations now require a fixed amount of computation time and memory overhead and do not depend on the complexity of the object being rendered. This was one of the design requirements of this project.

4.2.2 Splat Kernels

Each point will be rendered to the screen as a circular disc (called a ‘splat’) which is contained in the plane orthogonal to the point’s normal and containing the point. It would be too costly to calculate the projection of the splat on the screen for each point during rendering, so a collection of masks are pre-calculated for splats with different normals and in different regions of screen space (as suggested in [Botsch et al. 2002]). The splat kernel lookup table need only be created once and may be used throughout the running of the renderer without modification. The index into this lookup table can be calculated at render-time depending on the screen coordinates of the point and normal of the point (see Figure 4.4).

Using the symmetry of the axes, the splat kernel lookup table contains only one quadrant of the x/y-axes. The splat masks are then inverted in the x and y axes (as appropriate) during rendering for other quadrants. There are 1024 different splats in total, requiring a 10-bit index into the table. The first and second bits indicate which quarter of the quadrant the splat is in. The third and forth bits indicate which eighth of the z-axis (between the front and rear of the object) the splat is in. The remaining six bits are an index into the normal lookup table for the normal of the splat after it has been transformed into screen space; the first three bits (specifying the octant of the normal) and the least significant bits are masked out to leave six bits of the normal index. These bits can be pre-calculated and cached in the normal table.

Each splat mask is a 16x16 monochrome projection of the disc onto the screen and is rendered to the screen in the colour of the point. The splat is also scaled as it is rendered to the screen, using the splat radius as explained below in the section ‘4.2.3 Splat Radius’. Each splat mask is 256 bits long (16x16) so the splat kernel lookup table requires ~32KB of memory in total.
4.2.3 Splat Radius

Each point requires a value for the radius of the disc (called a ‘splat’ once projected onto the screen), centered at the point and contained in the plane that is orthogonal to the point’s normal vector. A requirement of this project is that an object must be rendered with no holes in the surface, so the radius of the disc must be sufficient to contain all of the immediate-neighbouring points (when they are projected into the plane) but no larger than necessary. See the section ‘4.3 File Types’ for an explanation about how the disc radius is obtained from a polygon-mesh model stored in a PLY file. This value is stored as a 32-bit float in the PLY file data structure, but can be quantised to an 8-bit index into a lookup table containing 256 values between 0.0514 and 0.0002. It assumed that a splat radius will never exceed 5% of the size of the grid, so this range of quantisation values will be sufficient if (radius / grid-size) is indexed instead of just the radius (which could be any value). This does not incur any extra computation when extracting the value from the lookup table.

At render time this value must be converted to the radius of the splat to be drawn on the screen (in pixels). This value can be calculated with the following formula:

\[ r' = \frac{dr}{z} = (ds)(1/z)(r/s) \]

Where:
- \( r \) = original splat radius
- \( d \) = distance from viewpoint to view plane
- \( r' \) = projected splat radius
- \( z \) = z-coordinate of splat center in screen center
- \( s \) = grid-size

Figure 4.5(a) shows how this formula was derived. By the ratios of the lengths of the two similar triangles we have \( r'/d = r/z \). Note that the orientation of the disc does not need to be considered in this calculation. This is because the radius of the disc is directly proportional its length when projected onto the view plane (shown in Figure 4.5(b)) so the two can be interchanged in linear equations. Therefore, the disc normal can be assumed to be parallel to the viewing direction.

Figure 4.5: Derivation of projection of splat radius into screen space formula

The value of \((ds)\) can be calculated before rendering. The value of \((1/z)\) is known when the splat is drawn because it has already been calculated for the perspective transformation. The value of \((r/s)\) is obtained from the disc radius lookup table (index stored in the attribute data for the point). Therefore,
calculating the projected radius of the splat in screen space requires just two multiplications per splat.

Therefore, this lookup table helps to achieve two of the design requirements by: reducing the size of the data structure in memory; and by increasing rendering speed when calculating splat sizes in screen space.

### 4.3 File Types

Many high resolution models are available on the Internet in the PLY file format (developed by [Georgia 05]) modelled as polygon-mesh objects (sometimes with vertex colours). However, the PLY file format is flexible enough to support point based objects which contain data for point normals and disc radii.

If not specified explicitly in the PLY file data, point normals and point disc radii may be generated from the polygon-mesh in the following way:

- Point normal = normalise(sum of normals of all faces that have this point as a vertex)
- Point disc radius = max(lengths of all face edges that have this point as a vertex)

### 4.4 Rendering

The first objective of this project was to produce a fast and simple point-renderer. This would then allow shading techniques to be added on later while maintaining real-time operation. This module should contain functionality to render the point data (contained in the previously developed data structure) as a water-tight surface and perform and necessary culling.

The binary voxel grid that was used as a starting point for the data structure in this project is exploited further by [Botsch et al. 2002] to perform matrix transformations more efficiently, using the same spatial coherency strategy that the data structure employed. Matrix projective transformations (using 4x4 matrices) normally require 14 additions, 16 multiplications and 3 divisions per point;

\[ p_i' = Mp_i \]

However, in the uniformly partitioned octree the displacement from the center of a node to one of its octants (call it q) is not dependent on the data contained in the structure and can be pre-calculated and reused. Hence if \( p_i = p + q \) where \( p \) is the center of the parent node of the leaf node that represents point \( p_i \) and \( q \) is the displacement vector from the \( p \) to \( p_i \) then:

\[ Mp_i = M (p + q) = Mp + Mq = p' + q' \]

Where \( p' = Mp \) is known by the parent and \( q' = Mq \) may be pre-calculated.

Also, if depth is omitted from the calculations; if the 3x4 sub-matrix of \( M \) is used and assuming that the homogenous coordinate of the point is always 1, then number of calculations can be reduced further, i.e. \( M_{3x4}(x,y,z,1)^T = (u,v,w)^T \) where the 2D screen coordinates can be obtained by: \((x', y') = (u/w, v/w)\). Reusing results from parent nodes, the average number of operations being performed per point is 4 additions and 2 divides.
As mentioned, the vectors from parent node centers to child node centers can be pre-calculated. So if \( d_{i,j} \) is the vector from the center of a grid square to its \( j \)th octant in the \( i \)th level of the tree, then \( d_{i,j} \) is given by:
\[
d_{i,j} = \text{gridsize} \cdot 2^{-i-1} \cdot [\pm 1, \pm 1, \pm 1, 0]^T, j = 1, \ldots, 8
\]
For example, if a leaf node has a path through the tree (of depth \( k \)) of \( j_1, \ldots, j_k \) and the center of the voxel grid is at point \( c \), then the transformed point at the leaf node is given by:
\[
M_p = M ( c + \sum_{i=1}^{k} d_{i,j_i} ) = M c + \sum_{i=1}^{k} M d_{i,j_i}
\]
Where \( M_c \) and \( M d_{i,j} \) are all pre-calculated.

As explained in the Literature Review, a depth-first traversal of the voxel grid is more efficient and has a simpler algorithm. The algorithm for this method is as follows:

- Set \( M = \) object-to-screen space transformation 3x4 matrix.
- Pre-calculate \( d'_{i,j} = M^*d_{i,j} \) for all \( i=1, \ldots, \text{grid-depth}, j=1, \ldots, 8 \)
- Set \( c' = M^*c \) where \( c \) is the center point of the voxel grid.
- \( \text{render}(1, c') \)

Where the algorithm for the recursive method \( \text{render}(\text{level}, p) \) is as follows:

- if \( \text{level} \leq \text{grid-depth} \) then
  - set code = next byte-code from data stream
  - for \( j = 1 \) to \( 8 \)
    - if \( j \)th bit of code is set then \( \text{render}(\text{level} + 1, p + d'_{(\text{level},j)} \)
  - else
    - do back-face culling
    - get screen coordinates: \( q = (p.x/p.z, p.y/p.z) \)
    - get splat size from next element in attribute stream
    - do view frustum culling (using splat size and \( q \))
    - calculate colour of this point using attribute stream element
    - calculate splat index using its position on screen and normal orientation
    - draw splat at \( q \)

This is a simple recursive algorithm with \( O(n) \) complexity with respect to the number of points. Together with the reduced computational cost of matrix transformations it satisfies the design requirement of efficient rendering.

### 4.4.1 Culling

The binary voxel grid described so far does not allow branches of the tree to be culled away because each node inherits data about its position from its parent, and interrupting the data stream during tree-traversal would disrupt this. Therefore, culling must be carried out for every point in the data structure. Culling tests are performed at an early stage to prevent unnecessary calculations.

View frustum culling is performed by calculating the radius of the splat (call it \( R \)) and testing whether the \( R \times R \) square fits entirely within the viewing area – if it does then it is rendered, otherwise, it is discarded.
Back face culling is performed as a pre-calculation; Each time the scene is rendered, each normal in the normal lookup table is tested to see whether or not it faces the viewpoint (by testing if the dot product of the normal and the view direction is negative) and this normal is then flagged as culled or not. This operation is performed in world space, so there are no complications about the perspective of the camera. This is very effective at culling many points at an early stage because it only requires one test (of the culled flag for the normal) and it discards most of the points that will be culled during the rendering.

Occlusion culling is performed using a z-buffer. The performance of the z-buffer is further enhanced by storing 8 binary voxel grids for each object, which are rendered in order (closest to the screen first) to reduce the number of hidden splats that are rendered.

4.5 Background

In order to implement environment mapping and shadows, the object being modelled must be surrounded by a non-trivial background so that this background may be reflected on the surface of the object and so that shadows may be cast from the object onto the background geometry. The background is modelled with point primitives as a sphere that surrounds the object and a square plane positioned under the object (as shown in Figure 4.6).

The points of the sphere are generated using the normal lookup table (but with a higher precision for higher detail) and then scaled to encompass the object. The normal of each point is directed inwards so that the interior surface of the sphere is rendered. A panorama background (obtained from Hotel U Dómú website: http://www.udomu.360-panorama.net/ on April 1st 2006) image (with dimensions w x h) is then mapped to a point (x,y,z) on the sphere using the following formulas:

Texture y-coord = \((y + 1)*(h - 1)/2\)

Texture x-coord = \((\pi + \arctan(x/z))*w\) / \(2\pi\)

The material of the points on the sphere only has an ambient reflective component because it is assumed that the illumination of the background is contained in the image.
The precision of the normal lookup table required to generate points that are sufficiently dense to display the background image with adequate quality can be calculated as follows:

If the width of the screen is 640 pixels and we allow the background to be stretched by a factor of two then the screen contains 320 pixels from the background image across it.

Number of normals in view frustum = \(2^n\)

where \(n\) = number of refinements of the normal table – see Figure 4.7 for reasoning.

So \(2^n = 320 \Rightarrow n \approx 8\)

Hence normal table index length = \(3 + 2 \times 8 = 19\) bits

The square plane is then constructed as a 2D mesh of points at the base level of the object. The material of the plane points has diffuse and specular components this time so that it can be lit and so that shadows are visible on it.

All of the background data (points, colours, etc) are then placed into a PLY data structure so that a point grid can be generated for the background. This is a high-level module and uses much of the functionality already developed so far in the project so it will have a very simple implementation.

4.6 Shading

The main aim of this project is to explore how point-based modelling can be exploited to achieve different shading techniques in real-time. The following subsections describe the development of each of these techniques.

4.6.1 Lighting & Materials

Due to the time constraints of a real-time renderer and the assumptions of infinite light sources and viewpoint imposed by the normal lookup table pre-calculation strategy, a simplified Phong illumination model is appropriate. As explained in the section ‘4.2.1 Normals’, the lighting will be pre-calculated for each normal and simply read from the normal table during rendering.

In the Phong Illumination Model (for an infinite light source and infinite viewpoint), both the diffuse component and the specular component depend on the normal of the point. However, the diffuse component is not affected by the viewer so it can be pre-calculated once (each time the lights are modified) for each normal in the lookup table. The specular term must be calculated for each normal each time the view direction changes or the scene is modified. These values can be directly indexed at render-time and summed together to obtain the light intensity at each point. The lighting values are pre-calculated over all lights and indexed over all materials, then stored in the normal lookup table. See section ‘2.3.1 Lighting’ in the literature review for details about the Phong illumination model and formulas.
The material of each point may also be specified by adding a small field to the attributes of each point to index a list of materials with different reflective properties.

4.6.2 Shadows
Shadows are very important to the human eye; giving information about the relative positions of objects in the 3D space and information about the location of light sources. Therefore, functionality for generating shadows in a scene will be important to satisfy the design requirement of a level of realism.

Shadows may be generated by adapting the notion of a shadow map, originally proposed by [Williams 1978]. By viewing an object from the position of a light source in the direction of the light, it is possible to see which points are in the line-of-sight of the light and which are obscured, and hence; are in shadow. This view is obtained by rendering the object to a buffer in memory (called a ‘shadow buffer/map’). However, instead of the colour, a pointer to the point data is written to the shadow map so that the corresponding point responsible for filling each pixel can be identified. Each point that has a pointer in the shadow map may then be marked as ‘not in shadow’ and the remaining points marked as ‘in shadow’. The attributes for each point has a 1-bit flag for each light source that is set depending on whether or not the point is in the shadow of that light. Then, when the point is rendered, the light intensity at that point is calculated by summing the diffuse/specular components (from the normal lookup table) only for the lights which have been flagged as ‘not in shadow’.

This approach allows shadows to be generated in complex lighting environments with many lights and then rendered in real-time. This is because at render-time each point already knows if it is in shadow or not and no shadow maps need to be indexed.

This method assumes that every point that is in the line-of-sight of the light source has at least one pixel rendered in the shadow map. This does not happen in practice due to the resolution of the shadow map. In the example displayed in Figure 4.8, splats 1 and 2 are both in the line-of-sight of the light source but both project onto the same pixel in the shadow map. So splat 1 will not have a pixel in the shadow map and will be deemed to be in shadow. This will appear as a dark spot on a lit part of the surface.

Another source of aliasing arises from the resolution of the shadow map. An object rendered in front of another will have a pixelated edge in the shadow map, so the shadow cast on the second object will also have a pixelated edge. The extent of this pixilation is proportional to the resolution of the shadow map.
Therefore, increasing the resolution of the shadow map will reduce the amount of aliasing in the shadows (from both sources described before).

Unlike conventional shadow mapping, this procedure does not require the shadow maps to be retained after they have been processed (after each point has been set as being in or out of shadow). Therefore, several shadow maps can be generated from different viewpoints or directions. For an infinite light source, the shadow map can be generated from any point as long as the camera view direction is the same as the light direction. By creating multiple shadow maps from different viewpoints, there is a higher likelihood that points erroneously obscured in one of the shadow maps will be visible in another shadow map. A point is then set as ‘not in shadow’ if at least one shadow map contains a pixel corresponding to that point and ‘in shadow’ otherwise. Perturbing the position in this manner will also reduce the pixilation of shadow edges and could be used to create soft edged shadows. This approach to anti-aliasing could be applied to point light sources by creating multiple shadow maps from different view directions, while keeping the viewpoint fixed.

4.6.2.1 Casting shadows on to the background

When casting shadows onto the background object, an assumption can be made that significantly reduces shadow aliasing. As there are no points beyond the background, the background surface does not have to be hole-filled because no points are present that will show through the surface and be erroneously flagged as ‘not in shadow’. Therefore, the background may be rendered to the shadow buffer as single-pixel points. It is then less likely that multiple points will occupy just one pixel. This also reduces the time required to create the shadow map because setting one pixel is quicker than drawing a splat. Also, the colour of the point is not relevant for the shadow map, so lighting calculations and environment mapping, etc can be neglected.

In addition to the form of aliasing discussed earlier that creates pixelated shadow edges, the resolution of the shadow edge is also dependent on the proximity of the points in the background. Therefore, increasing the density of background points will increase the quality of the shadow edge. However, increasing the density of background points beyond the resolution of the shadow buffer will result in aliasing where multiple points are projected onto single pixels.

4.6.2.2 Shadow generation algorithm

The algorithm for generating shadows in the point-modelled scene will be as follows:

- Create shadow buffer and corresponding z-buffer (with high resolution)
- Set all points to in shadow for all lights
- Set splat radii of all points in background to minimum value in splat radius lookup table
- For each light source
  - For each perturbed light position
    - Clear shadow buffer and z-buffer
    - Render scene from perturbed light position
    - For each pixel (pointer to a point attribute) in shadow buffer
      - Ensure point is set as ‘not in shadow’ for this light source
- Restore splat radii of all points in background
This algorithm will have to be performed each time the lighting in the scene alters or an object in the scene moves. It involves rendering the scene and iterating through the shadow buffer once for each light source and for each perturbed light position. So the time taken to perform this algorithm is given by:

\[ T_{\text{shadows}} = (T_{\text{render}} + T_{\text{buffer}}) \times \text{light_count} \times \text{number_of_light_positions} \]

Where:
- \( T_{\text{shadows}} \) = total time required for shadow generation
- \( T_{\text{render}} \) = time required to render scene at high resolution
- \( T_{\text{buffer}} \) = time required to iterate through shadow buffer

With a maximum of 8 lights in the scene and 4 different light positions this would mean rendering the scene (at high resolution) and iterating through the shadow buffer up to 32 times. This is acceptable as a pre-calculation strategy before rendering begins, but could not be used in real-time. Therefore, this method could only be used in static lighting environments.

4.6.3 Textures

During the planning of this project it was decided that point-based rendering was most suited to static scenes of high complexity, such as statues. Given that statues are normally carved from stone and have high detail, it was decided that solid textures (proposed by [Perlin 1985] and [Peachey 1985]) were the most suitable texturing technique. It was shown in [Perlin 1985] that the Perlin noise function could be used to produce solid textures of natural materials such as wood and marble. Therefore, solid textures generated using the Perlin noise function will be used in this project to give objects a more interesting and realistic appearance.

The complexity of the Perlin noise function makes it unsuitable for calculating texture colours on-the-fly so three-dimensional texture buffers should be generated before the texture is used. The texture is then mapped to each point using the formula:

Point \( p=(x_1,x_2,x_3) \) maps to texture point \( t=(t_1,t_2,t_3) \) where:

\[ t_i = (x_i \times 2^{\text{grid\_depth}} / \text{grid\_size}) \mod \text{texture\_size} \text{ for i = 1,2,3} \]

This equates to one texel per voxel grid leaf node because the size of one voxel grid leaf is \( (\text{grid\_depth} / 2^{\text{grid\_depth}}) \). As each point stores its own colour in the data structure proposed, this mapping may be performed before rendering begins. The object may then be rendered as normal, with no additional costs at render-time.

Two common textures that can be generated with the Perlin noise function are wood:

Texture colour at \((x,y,z)\) = wood_colours[n + floor(n)]

With \( n = \text{perlin\_noise}(x,y,z) \times 20 \)

and marble:

Texture colour at \((x,y,z)\) = marble_colours[\(\text{COS}(x + \text{perlin\_noise}(x,y,z))\)]

The details of the Perlin noise function are outside the scope of this document and may be found in [Perlin 1985].
4.6.4 Environment Mapping

The range of materials that may be modelled can be extended to include mirror-like materials (such as metals) using environment mapping. This will increase realism of the renderer, which was a design requirement of this project.

Environment mapping is performed by reflecting a view ray (directed half-line in the viewing direction originating from the viewpoint) in the normal vector of a point on the surface of an object. The colour reflected from the environment at that point is then obtained by finding the closest intersection of the resulting ray with the surrounding surfaces and taking the colour at the intersection point. By ignoring the position of the surface point and viewpoint, the environment map colour is a function of only the point normal and the viewing direction. Therefore, environment mapping (with infinite viewpoint and background) can be pre-calculated on a per-normal basis by reflecting the view direction in the normal vector. The environment colour is obtained by intersecting the resulting ray (originating at the object origin) with the background and is stored in the normal lookup table so that it can be directly indexed.

The following derivation explains how to obtain the reflected unit vector \( \mathbf{R} \) in the surface with unit normal \( \mathbf{N} \) of the unit vector \( \mathbf{I} \).

\[
\begin{align*}
\vec{OC} &= -(\mathbf{N} \cdot \mathbf{I})\mathbf{N} \\
\vec{CB} &= \vec{AC} = \mathbf{I} + \vec{OC} \\
\vec{OB} &= \vec{OC} + \vec{CB} = \mathbf{I} + 2\vec{OC} \\
\Rightarrow \mathbf{R} &= \mathbf{I} - 2(\mathbf{N} \cdot \mathbf{I})\mathbf{N}
\end{align*}
\]

The construction of the background sphere (using the normal lookup table) makes finding the intersection of rays from the origin to the sphere very simple. When the normal table is created a list of points (origin + normal vector) that are uniformly distributed around a sphere is created. A parallel list of colours that correspond to the colour of the sphere at each point may also be created, using the formulas defined in section ‘4.5 Background’ to map the background image to each point. So by finding the normal table index of the reflected ray direction, the colour of the background at the point of intersection can simply be looked up with the same index.

When the point is rendered, the colour of the point is blended with the reflected colour of the environment (obtained form the normal lookup table) using a blending factor (in [0,1]) specified in the material for that point;

\[
\text{Final point colour} = \text{colour}_{\text{point}} \times (1 - \text{blending\_factor}) + \text{colour}_{\text{environment}} \times \text{blending\_factor}
\]

This is the only per-point computation and the calculation of the environment colour for each normal in the normal lookup table is independent of model complexity. Therefore this procedure can be implemented in real-time.

Ignoring point position and viewpoint position will mean that flat surfaces reflect only one colour. However, this method should perform adequately well for curved surfaces. The limited precision of the normal lookup table will also reduce the quality of the reflected image. However, the human eye is not very sensitive to imperfections in reflections on surfaces so these visual problems may not have a noticeable impact in most cases.
5 Implementation

This section describes any relevant issues regarding the implementation of the modules designed in the previous section.

5.1 Language and Platform

This project was written in C++ code due to its speed and control over memory management. It is also a well established language that may be compiled (with little or no modification) on any platform – allowing portability.

The code is designed to run on the MS Windows operating system using the DirectX application layer. The popularity of these platforms will allow the code to be run on most computers without installation.

5.2 Virtual Machine

It was decided that the application would be written for machines running Windows with DirectX but that the code should be easily portable to other operating systems. Therefore, all operating system and DirectX related code is contained in the virtual machine module. This module then provides abstract low level functions (such as converting and RGB triple to a colour value at the appropriate bit depth) and variables (such as the address of the back buffer). This module also provides functionality to write text to the screen and record the time taken to render a frame.

The application normally runs in a window, but can be run in full-screen mode as well. The screen resolution is set to 640x320 with a 16-bit colour depth.

5.3 Maths Engine

This module provides common maths functionality with a focus on speed. As the implementation is known, fewer checks will have to be performed on the parameters of functions. The maths engine consists of common mathematical constants (such as PI and matrix identities) and mathematical functions (such as EVEN, ODD, MODULUS, etc). It also houses more complex functions such as:

- Sine, cosine and tangent functions - implemented as lookup tables that are pre-calculated when the application starts using the Maclaurin Series:
  \[
  \sin x = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}, \quad \cos x = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!}, \quad \tan x = \frac{\sin x}{\cos x}
  \]
  The lookup tables are of length 360 and span the period of the functions. The symmetry of the functions is also used to reduce the number of computations: \( \sin(x) = \sin(180-x) = -\sin(180+x) = -\sin(360-x) \) for all \( x = 1, \ldots, 90 \).

- Heap Sort function – sorts an array of keys together with a parallel array of values corresponding to each key. The heap sort algorithm was selected mainly because it has a constant memory footprint (vital if sorting a set of data that only just fits into system memory) and also because it has a good worst case scenario complexity of \( O(n\log(n)) \).
• Vector min-max function – calculates the minimum and maximum values of an array of vectors. The algorithm works with pairs of points, making it more efficient than the naïve algorithm. Although both are $O(n)$.

A collection of matrix structures are defined together with some standard functions for addition and multiplication, etc.

In places, assembly code and FPU (floating-point unit) code is used to ensure that the code is optimised. All functions are written as macros, if possible, or otherwise as inline functions for speed.

This module also contains a Perlin-Noise (developed by Perlin 1985) function for generation of 3D procedural textures. This function takes a point in 3-space and return a random number based on that parameter. The output of this function is also low-pass filtered to make it ‘smoother’. The algorithm behind this function is described in more detail at Elias 2005 and its use in the project is explained further in the section: ‘4.6.3 Textures’.

5.4 Error Handling

This simple module provides methods to report error descriptions. These methods are for debugging purposes so are not expected to occur during the normal execution of the application. Therefore, they do not have to be efficient. An error is reported by concatenating an error code (a short unique string that can be searched for in the code to find the location of the error) with an error description (a string that describes the reason for the error) and writing it to a file. It is intended that the programmer will look in the error log if the application terminates unexpectedly.

5.5 Data Structures

The PLY files are read into a simple data structure that simply stores the coordinates, normals, disc radii, point colours in an uncompressed form. This is then deleted after it has been used to generate the point grid.

The point grid is stored in memory as a stream of bytes and a parallel stream of attributes (for leaf nodes only). The structure can be generated in either a breadth or depth first order (as this was very easy to implement), but can only be read in depth first order.

An object is comprised of eight point grids to provide an extra level of precision to the object point data and so that each grid can be sorted and rendered in order of their proximity to the viewpoint to aid the z-buffer in occlusions culling.

Each leaf node of the binary voxel grid data stream has a corresponding element in the attribute stream. The attributes of a node (point) are displayed in Figure 5.1.

<table>
<thead>
<tr>
<th>Normal Index</th>
<th>Point Colour</th>
<th>Material</th>
<th>Splat Radius</th>
<th>Padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 bits</td>
<td>16 bits</td>
<td>3 bits</td>
<td>8 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Figure 5.1: Layout of a point attribute structure
5.6 File Types

This module will not have full support for the PLY file format (developed by [Georgia 05]), as this is not a relevant field of study for this project. A simple parser is implemented as a finite state machine for each section of the PLY file format and it has support for point and face data in ASCII or little-endian binary format. A tokeniser class is also implemented to aid the parser when reading the PLY header and ASCII data. It is also able to read point colours, point normals, radii of point splats and material data (lighting coefficients, colour, environment mapping blending factor).

This module has the functionality to write PLY files. However, the face data is not retained in the PLY data structure so the normals and splat radii (and colour if present) are written along with the position of each point instead.

5.7 User Interface

When the application is run, the scene (described by the command line arguments) is setup. During this initialisation (normally just a few seconds) the string “Loading…” is displayed in the window to notify the user of inactivity during this period. After this initialisation, the camera is positioned at a suitable location to view the object being modelled. There are two camera movement models (that can be chosen with the command line arguments):

- Target Camera – has a constant target (the center of the object). Moving the mouse will alter the position of the camera while keeping the distance from the target constant. The direction of the camera will be updated so that it still points at the target point. Pressing the up and down arrow keys moves the camera position (along the direction vector) closer and further from the target respectively.

- Flying Camera – position stays constant while the mouse is moved. Moving the mouse will alter the direction of the camera, updating the target as it changes. Pressing the up and down arrow keys moves the camera position along the direction vector in the positive and negative direction respectively.

Pressing the space bar at any time will capture a frame and save it as a BMP image file. Pressing the escape key at any time will exit the application. Pressing the F1 key at any time will open the PDF help document (this is included in appendix 8.1C).

The optional parameters of the renderer are set using the following optional command line arguments:

- **-AVI**
  - Outputs an uncompressed AVI video file of all the frames rendered while the application runs. Also selects an automatic camera movement model that follows a preset path and exits the application at the end of that path.

- **-FLYING_CAMERA**
  - Specifies the flying camera model instead of the target camera model (default).

- **-IGNORE_POINT_COLOURS**
  - Renders the object with a default grey material with no textures or point colours.

- **-BACKGROUND <filename>**
  - Adds a background object to the scene. The optional
<filename> parameter overrides the default background image and specifies an alternative image to be mapped to the spherical background.

-**SHADOW**

Specifies that shadows should be added to the scene.

-**TEXTURE <name>**

Specifies that the point colours of the object and the plane in the background object should be overridden with texture mapped colours. The optional <name> parameter overrides the default texture (marble) and specifies the texture to be used for the object and may take the value of either “marble” or “wood”.

-**ENVIRONMENT_MAP**

Specifies that the environment may be mapped to the object if the material of the object requests it.

<filename>

The filename of the PLY file to open. If this parameter is not supplied, the default model will be loaded.

The material of an object can be specified in the PLY file. The following material parameters may be supplied: \(ka, kd, ks, power\) - Phong lighting coefficients, \(clr_r, clr_g, clr_b\) - material colour components and \(envMapFactor\) - the environment map blending factor (see section ‘4.6.4 Environment Mapping’). The default values for any parameters not specified are those of a matte (ambient reflection only with no environment mapping) grey material.

5.7.1 **Menu**

A separate windows application was also produced that provides a friendlier user interface for setting the renderer parameters. See Figure B.6 in the appendix for a screen shot of this application. The application simply, assembles the correct command line arguments, based on the user’s input and executes the command. The application was written in C#.net due to the ease of creating GUI applications with this language. Although .net languages are not completely portable to other operating systems (other than Windows), the simplicity of this application should mean that compiling if for other platforms will not be a problem, therefore maintaining the portability design requirement of this project.

5.8 **Rendering**

This module does all of the drawing to the screen, but it is also used to generate shadow buffers, etc so it must be generic enough to easily allow this flexibility. Therefore, the render method accepts screen buffers of any dimension and any bit-depth. The method is templated with respect to the colour bit-depth so that the method is compiled twice for 16-bit colours and 32-bit colours (and pointers – for the shadow buffer).
The colour of each point that is rendered to the screen is affected by many factors (light intensity, material, shadows, etc) so, for clarity, the algorithm for calculating the point colour at render-time for each point it given here:

- If creating shadow buffer
  - set colour = pointer to current point’s attributes
- otherwise
  - Set intensity = ka value for material of point
  - For each light
    - If point is not in the shadow of this light then
      - intensity += diffuse component for this light and this material from normal table entry for this normal
      - intensity += specular component for this light and this material from normal table entry for this normal
  - Set colour = colour of this point * intensity
  - Blend colour with environment colour from normal table for this normal using the environment blending factor specified in the material for this point

5.8.1 Splatting

The function to draw a splat to the screen is the most frequently called non-trivial function in the project. Therefore, it must be very efficient and heavily optimised.

For each point, the scale of the splat (in pixels) is calculated using the formula described in the section ‘4.2.3 Splat Radius’. If this scale is less than or equal to 1 then a single pixel is drawn to the screen. If the scale is less than or equal to 3 then a 3x3 square is drawn to the screen. If the scale is less than or equal to 8 then a splat is drawn to the screen, otherwise a scaled splat (scaled by a factor of a power of two: 2x, 4x, 8x, …) is drawn to the screen.

An index into the splat kernel lookup table is calculated at render time using the screen coordinates of the point, and the index of its normal (after it has been transformed into screen space). From this index, a 16x16 monochrome mask of the splat is obtained from the splat kernel lookup table in the form of a stream of bits (4 ints). The splat mask is rendered to the screen by iterating through the bit stream of the mask and setting the corresponding pixel on the screen if the bit is set to 1. The z-buffer is also checked in this algorithm.
6 Testing

The testing of this project will be presented in two parts; the testing of the design and implementation to check that it meets the requirements of the project; and the practical testing of the software produced to check that it gives the expected output for all variations of user input.

6.1 Requirements Testing

This section evaluates how well the project design meets the requirements laid out at the beginning of the project (in section '3 Requirements Specification').

6.1.1 Functional Requirements

All of the mandatory functional requirements are fulfilled; Model data can be loaded from files in the PLY file format. This point data may then be rendered to the screen as a surface with no holes and with no hidden surfaces rendered. A basic lighting model is also implemented and different colours and materials may be assigned to each point. However, the lighting model is a simple implementation of the Phong local illumination model (with infinite viewpoint and light positions) and therefore models a limited range of materials and does not achieve a high level of realism.

Texture mapping is implemented using procedural solid textures. Although this limits the range of textures that can be mapped in this fashion, it gives good results for natural materials such as wood and marble. The texture is mapped on a per-point basis instead of per-pixel. This means that the quality of the texture depends on how close the points are when projected onto the screen. For models with sparsely distributed points, the solid texture can be undersampled, and the appearance will not be very sharp. See Figure B.4 in the appendix for an example of the texture mapping in use.

Background geometry may be added to the scene in order to cast shadows from the object onto it and to reflect the background in the object. The background image is displayed with an adequately high resolution and is able to be rendered along with the object in real-time without a significant cost in time. The background consists of a sphere with a plane underneath the object. This limits the range of geometry that can be displayed in the background, but only minor alterations would be required for more complex background objects (modelled with point primitives).

Shadows may be cast from the object onto the background and upon the object itself. The pre-calculation strategy employed allows the shadows to be rendered very quickly at render-time, although the shadow generation method can take up to a couple of seconds to complete. This means that shadows cannot be calculated in real-time if the lights or objects are moving around the scene. Both forms of shadows produced (onto background and onto object) suffer from minor aliasing problems due to some points that are not in shadow failing to be rendered into the shadow buffer. This is due to point splats obscuring each other and the resolution of the shadow buffer. See Figure B.2 in the appendix for an example of this – there are some spots of dark on the top of the leg where some points are mistakenly thought to be in shadow. The shadow cast upon the background appears blocky as the pixels of the shadow buffer are projected onto
the surface. This is partly due to the finite resolution of the shadow buffer and partly to fact that the shadow is drawn on a per-point basis rather than per-pixel. This means that the resolution of the shadow is dependent on the density of the points on the surface. See Figure B.3 in the appendix for an example of this – notice that the edge of the shadow cast upon the plane is quite blocky.

The background may also be reflected in the surface of the object to give a shiny/mirror effect. This gives a good effect when the material is only a partial reflector and the original surface colour is blended with the reflection. As the environment mapping is done on a per-normal basis, the resolution of the reflected image is dependant on the precision of the normal table. So, even though the background sphere contains $2^{19}$ points, only $2^{13}$ of them can be mapped to the surface of the object. This results in under-sampling of the background and the reflections lose their sharpness and detail. See Figure B.5 in the appendix for an example of this. Another source of error in the environment mapping is caused by ignoring the position of the point and the viewpoint in the calculations. The effect of this is not very noticeable on curved surfaces but it causes flat surfaces to flash as the camera is moved because the surface is only reflecting one point of the background.

6.1.2 Performance Requirements

After extensive testing, the application proved to be completely stable and was able to remain responsive for every model that was loaded once rendering began. However, the application is not responsive before rendering begins (normally a few seconds) when the pre-calculations are being performed (i.e. when lookup tables are generated and shadows are created). The user is informed of this inactivity by a caption on the screen that says “Loading…”. This period of inactivity could be reduced by caching the results of these pre-calculations in files but it is not perceived to be a significant problem at the moment.

All models loaded could be rendered in real-time, although the precision of the point data had to be reduced (depth of point grid reduced) for larger models in order to achieve this. See Figure B.1 in appendix for an example of a model of high complexity that has had its level-of-detail bounded so that it may be rendered in real-time.

Rendering at progressive levels of detail when the viewpoint is stationary was not implemented because the depth first traversal of the point grid made this a very complex task and it was decided that the quality of the renderer was high enough after the first render (even when point precision was reduced in larger models).

Back-face culling was implemented to reduce unnecessary processing of point primitives. This was performed on the normal table and was a very efficient method of discarding around half of the point primitives each time the scene was rendered.

See later section ‘6.2 Software Testing’ for performance statistics.

6.1.3 Interface Requirements

All of the interface requirements have been met; the scene may be specified by passing the background image and object (PLY) file in the command line
arguments. Several renderer options may also be specified in the command line arguments and the user input is as required – see section ‘5.7 User Interface’.

6.1.4 Error Handling & Portability

All errors are reported and handled as required, i.e. all allocated memory is freed and an error message is written to an error log file before the application terminates.

The software code was written for machines running Windows with DirectX but is easily portable to other platforms. All operating system and DirectX related code is contained in the virtual machine module, which then provides abstract low level functions (such as converting and RGB triple to a colour value at the appropriate bit depth) and variables (such as the address of the back buffer). Porting the software to another operating system or graphics library only requires three code files in the virtual machine module to be rewritten.

6.2 Software Testing

This section tests the performance of the software produced to evaluate how it meets the design goals and how well it performs. This is achieved by running the software with different user inputs and assessing whether the output is as expected. Testing was performed on a system with an AMD Athlon 64 1.8GHz processor and 1GB of RAM.

6.2.1 Basic Behaviour

The system successfully loaded several different PLY files (details shown in Table 6.1):

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Size (KB/#points)</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere *</td>
<td>ASCII</td>
<td>2,177 / 32,768</td>
<td>Material, point, normal, disc radius</td>
</tr>
<tr>
<td>Bunny †</td>
<td>ASCII</td>
<td>2,963 / 35,947</td>
<td>Point, face</td>
</tr>
<tr>
<td>Bunny (colour) ‡</td>
<td>ASCII</td>
<td>2,785 / 35,947</td>
<td>Material, point, colour, normal, disc radius</td>
</tr>
<tr>
<td>Armadillo †</td>
<td>Binary</td>
<td>6,420 / 172,974</td>
<td>Point, face</td>
</tr>
<tr>
<td>Happy Buddha ‡</td>
<td>ASCII</td>
<td>41,621 / 543,652</td>
<td>Material, point, face</td>
</tr>
<tr>
<td>Manuscript †</td>
<td>Binary</td>
<td>86,241 / 2,155,617</td>
<td>Point, colour, face</td>
</tr>
<tr>
<td>Dragon †</td>
<td>Binary</td>
<td>133,949 / 3,609,600</td>
<td>Point, face</td>
</tr>
<tr>
<td>Statuette †</td>
<td>Binary</td>
<td>185,548 / 4,999,996</td>
<td>Point, face</td>
</tr>
</tbody>
</table>

* Self-generated
† Modified version of object obtained from [Stanford 2006]
‡ Obtained from [Stanford 2006]

The PLY file module also successfully saved a ply file (the objects: “Sphere” and “Bunny (colour)” were created using this functionality). Each command line argument was tested and found to give the expected behaviour and the keyboard and mouse inputs gave the expected outputs for each camera model. When invalid/missing ply files were loaded, the software exited and wrote an appropriate error message in the error log. The memory usage of the application was monitored (using the windows task manager) and it was concluded that all allocated memory was successfully freed when the application terminated. After extensive testing, the software was found to be very stable and did not freeze or exit unexpectedly. The visual appearance of the scene was also as expected for all shading options.
6.2.2 Data Structure

The main purpose of designing a special data structure was to reduce and bound the memory requirements and the rendering time for models of arbitrary complexity. To test if the data structure in this project meets these requirements, several models of varying complexity were loaded and the memory requirements and rendering times of the PLY data structure and the binary voxel grid structure were compared. The results of this test are displayed in the Table 6.2.

<table>
<thead>
<tr>
<th>Model</th>
<th>PLY data structure</th>
<th>Binary voxel grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford Bunny</td>
<td>35,947</td>
<td>35,933</td>
</tr>
<tr>
<td>Memory requirements (KB) *</td>
<td>431</td>
<td>647</td>
</tr>
<tr>
<td>Render Time (secs) †</td>
<td>0.0027972</td>
<td>0.0094848</td>
</tr>
<tr>
<td>Happy Buddha</td>
<td>543,652</td>
<td>260,920</td>
</tr>
<tr>
<td>Memory requirements (KB) *</td>
<td>6,524</td>
<td>9,786</td>
</tr>
<tr>
<td>Render Time (secs) †</td>
<td>0.0415150</td>
<td>0.0221110</td>
</tr>
<tr>
<td>Statuette</td>
<td>4,999,996</td>
<td>453,061</td>
</tr>
<tr>
<td>Memory requirements (KB) *</td>
<td>60,000</td>
<td>70,000</td>
</tr>
<tr>
<td>Render Time (secs) †</td>
<td>0.3914500</td>
<td>0.0332640</td>
</tr>
</tbody>
</table>

* The memory requirements for each data structure are split into three components: point data, attribute data, total size.
† The points were rendered as points only with no shading or culling. The time value is an average time of 100 executions of the render method only. It should be used for comparison purposes only.

For this test simple renderers had to be written for each data structure that just rendered the points directly to the screen with no culling or shading. The binary voxel grid was generated with a depth of 9 so that all models could be rendered in real-time with the binary grid structure.

Note that both memory requirements and the rendering times of the ply data structure increase linearly with the number of points, but increase at a decreasing rate for the binary voxel grid.

The ‘number of points’ row shows that as the spatial density of the points exceeds the resolution of the voxel grid (with fixed depth), the extra detail is discarded. When multiple points all occupy the space represented by one leaf node of the voxel grid, they are represented as just one point in the grid data structure. This bounds the memory requirements and the rendering time of the model, but results in a loss of detail and precision in the data. This demonstrates how the memory and render time can be arbitrarily reduced by reducing the depth of the grid. For example the statuette model was bounded to ~1/40 of the original total size in memory and was rendered in ~1/10 of the time (allowing real-time rendering speeds).

These results show that the data structure meets the design requirements by bounding both the memory usage and rendering time for models of arbitrary complexity.

6.2.3 Splat shape

In order to render the point data as a water-tight surface, each point must be rendered as a ‘splat’ – the projection of a disc centered at that point and orthogonal to the point’s normal. Previous study (see section 2 Literature Review) has shown that the quality of the image produced is effected by the shape of the splat, with normal-oriented circular discs producing better results.
than square splats. In this project, normal-oriented disc splats were implemented using a lookup table to store pre-calculated masks/images of several discs at different orientations and screen positions. The performance of this lookup table is tested now by comparing it with square splats and single pixel points. The results of this test are displayed in Table 6.3.

<table>
<thead>
<tr>
<th></th>
<th>Pre-calculation time (secs) *</th>
<th>Splat per second †</th>
</tr>
</thead>
<tbody>
<tr>
<td>One pixel points</td>
<td>0</td>
<td>3.83E+5</td>
</tr>
<tr>
<td>Square splats</td>
<td>0</td>
<td>3.74E+5 1.63E+4 5.33E+3</td>
</tr>
<tr>
<td>Normal-oriented disc splats</td>
<td>0.02261</td>
<td>3.43E+5 1.37E+4 4.60E+3</td>
</tr>
</tbody>
</table>

* Refers to the one-time pre-calculation time require when the application starts to generate lookup tables, etc.
† Split into three components for splats of radius: 1, 8, 16 so that scaled splats can be tested.

The results for the number of splats rendered per second were calculated by recording the time taken to draw 102400 splats to the screen (including the time taken to calculate the index into the splat kernel lookup table in the case of normal-oriented disc splats). The value for the number of splats rendered per second was then given by: 102400 / t.

These results show that the time required to draw a normal-oriented disc splat and the time required to draw a square splat only differs by a factor of ~1.2 on average. It is therefore concluded that the quality gained with the use of normal-oriented disc splats is worth the small increase in time required to render each splat (and the one-time pre-calculation time).

### 6.2.4 Culling

Back-face culling was a performance requirement of this project because it was expected that it would reduce the number of point primitives being rendered each frame and hence; increase the speed of the renderer. This hypothesis is tested now by comparing the rendering times for a standard example model (the Stanford Bunny model with two lights and shadow mapping) with and without back-face culling. Back-face culling is implemented in this project as a pre-calculation on each normal that must be performed each time the scene is rendered. Therefore, the time required for this pre-calculation is also measured for completeness. Note however that this pre-calculation time is constant, regardless of model complexity but the render time is not. The results of this test are displayed in Table 6.4.

<table>
<thead>
<tr>
<th></th>
<th>Pre-calculation time (secs) *</th>
<th>Render time (secs) †</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without back-face culling</td>
<td>0</td>
<td>0.0107518</td>
</tr>
<tr>
<td>With back-face culling</td>
<td>0.000594</td>
<td>0.0145252</td>
</tr>
</tbody>
</table>

* Refers to the pre-calculation time required before each frame is rendered.
† Average of 100 renders. These values should be used as a comparison only.

These results show that each frame is rendered around 25% faster with back-face culling enabled, and that the pre-calculation time is negligible in comparison. Although about half of the point primitives will be discarded in with back-face culling, the whole binary voxel grid tree must still be traversed. Therefore, it is not surprising that the speed increase of the culling is not closer to 50%.

These results support the hypothesis that back-face culling improves the performance of the renderer.
6.2.5 Shading

The main aim of this project was to achieve a high level of realism in real time by developing shading techniques that exploit the point-model. The visual quality of these techniques has been discussed earlier and now this section measures their performance in terms of speed. The tests are performed on two models of different complexity to see how well each technique scales with object complexity. Table 6.5 displays the results of these tests.

<table>
<thead>
<tr>
<th>Table 6.5: Performance costs of various shading techniques.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialise</strong></td>
</tr>
<tr>
<td>(secs)</td>
</tr>
<tr>
<td>1) 0 lights</td>
</tr>
<tr>
<td>2) 1 lights</td>
</tr>
<tr>
<td>3) 2 lights</td>
</tr>
<tr>
<td>4) self-shadow</td>
</tr>
<tr>
<td>5) background</td>
</tr>
<tr>
<td>6) full shadow</td>
</tr>
</tbody>
</table>

Each column contains two values; one for each test model. The first model is a sphere with 32,768 points and the second is a sphere with 131,072 points (4x the first sphere).

The ‘initialise’ column refers to the time required for any one-time calculations that are performed when the software starts. The ‘pre-calculate’ column refers to the time required for any calculations that must be performed each time the scene is rendered. The ‘render’ column is the time taken for the render function to be performed and the ‘total’ column is the total time taken to render the frame; from before the back buffer is cleared to after the back-buffer is flipped with the display buffer. The ‘frame-rate’ is the reciprocal of the total time (in reality, the frame-rate is clamped to 30fps by relinquishing the thread when the frame-rate exceeds 30fps).

Details about each test run are given below in Table 6.6.

<table>
<thead>
<tr>
<th>Table 6.6: Details of each run in shading test.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) 0 lights</td>
</tr>
<tr>
<td>2) 1 lights</td>
</tr>
<tr>
<td>3) 2 lights</td>
</tr>
<tr>
<td>4) self-shadow</td>
</tr>
<tr>
<td>5) background</td>
</tr>
<tr>
<td>6) full shadow</td>
</tr>
<tr>
<td>7) texture map</td>
</tr>
<tr>
<td>8) enviro-map</td>
</tr>
</tbody>
</table>

Screen shots of some of these shading options may be found in appendix ‘8.1B Screen Shots’ but for different models.

The results show that adding lights to the scene increases the time required for pre-calculation. They also show that the increase in pre-calculation time is not dependent on model complexity. This is because lighting is done on a per-normal basis instead of per point. Adding another light requires a further set of lighting calculations for each normal and material.

Notice that shadowing actually reduced the rendering-time. This is because the specular and diffuse components are summed for each light for each point at render-time. If a point is in shadow for a certain light, then these coefficients can
be ignored for that point. There are no further shadow calculations for shadows at render-time and no pre-calculation time required for each frame.

The texture mapping required an excessive time to be initialised. This is due to the complexity of the Perlin noise function and the number of texels in a 3D texture buffer. Since this test, the wood and marble texture buffers have been cached in files so that they may be loaded quickly when the application starts. However, after initialisation, the texture mapping required no further calculation because mapping is done during initialisation and then each point simple stores its new colour.

Unsurprisingly, the background geometry added a significant time cost to each phase of execution, due to the complexity of the object (≥2¹⁹ points). However, the interest that this adds to the scene justifies the cost in time. The complexity of the background object cannot be reduced without reducing the resolution of the background image.

Environment mapping increased pre-calculation time and slightly increased rendering time. However, the increase in pre-calculation time was independent of model complexity because it was done on a per-normal basis rather than a per-point basis.

These results show that all shading techniques implemented in this project are able to run in real-time – all have an interactive frame-rate and are independent of model complexity to a large extent.
7 Conclusion

This project explored how the point model could be exploited to provide various shading techniques in real-time. The product was a fast, versatile renderer capable of working with models of high complexity. This renderer is ideally suited to static objects of high detail such as statues. It offers functionality for textures, shadows (on object and background), environment mapping and backgrounds to add interest to datasets containing only positional data. However, the renderer would not be suitable for animated objects or simple objects that could be more efficiently modelled with polygons.

The main advantage of using points as a rendering primitive is that the large number of primitives means that many will share the same properties. Therefore pre-calculations can be performed on these common properties and the corresponding primitives can share the results by using lookup tables. This is most notably applied in this project with the use of a lookup table for normal vectors. This encompasses functionality for lighting, culling and environment mapping on a per-normal basis, rather than a per-point basis, reducing the impact of object complexity on the render-time.

Using the normal lookup table for lighting and environment mapping meant that the position of the points had to be removed from the calculations by making assumptions that light sources, viewpoints and backgrounds were infinitely far away. This reduced the quality of the lighting and environment mapping. However, the correctness of these processes is less noticeable by the human eye than other aspects of the image, such as the texture of the surface. The use of the normal table for environment mapping also resulted in the background being under-sampled (due to the precision of the normal table), giving low resolution reflections.

The point model was also exploited to create shadows by flagging each point to indicate whether or not it was in shadow. As well as reducing the computational cost of the shadow algorithm at render-time, this allowed multiple shadow maps to be created from different positions and then discarded, reducing aliasing. Another benefit of this procedure was that shadows could be calculated from many different light sources in a scene at no cost during rendering. The cost of this was a more time consuming shadow calculation algorithm that must be executed each time the object is moved or the lighting environment changes.

The density of the points (in comparison to the vertices of a polygon mesh) allowed textures to be mapped to the point surface before rendering by simply assigning a colour to each point. However, this can result in the texture being under-sampled if the points are not dense enough in space. The solid textures did not work as well as expected because of their large generation time and the difficulty in achieving realistic natural textures using Perlin noise. However, the purpose of this region of the study was to investigate how the point model could be exploited when applying textures - the method of mapping the texture to each point was largely irrelevant to this study.

The problem that caused the most significant reductions in quality and realism was due to shading being done on a per-splat basis, rather than a per-pixel basis. Each point was rendered as a splat of solid colour, which sometimes occupied several pixels. This reduced the sharpness of the images produced and gave a
noticeable mosaic effect at close proximity. This problem could be alleviated by
drawing each splat with a radial transparent gradient or by interpolating across
the splat using boundary colours. However the underlying problem is that colour
is only applied to finitely small regions of space which may be projected onto
multiple pixels on the screen. Per-pixel shading is difficult to achieve with the
data structure employed in this project because the world-space coordinates of
each point are never explicitly available during.

Overall, the project satisfies all mandatory design requirements and most of the
optional requirements. It successfully adapts and develops existing shading
techniques to work efficiently with the point-model. Therefore showing that
higher levels of realism can be achieved with real-time point renderers.

7.1 Improvements and Future Work

The mosaic splatting effect described earlier could be reduced by rendering
points as radial Gaussian splat kernels (as described in [Rusinkiewicz and Levoy
2000]) instead of solid blocks of colour. The alpha blending of each splat would
require two passes of the rendering algorithm and the splat kernel lookup table
would have to be enlarged to store alpha values instead of monochrome values.
The increase in quality would probably justify these costs.

For optimisation reasons, the point attribute structure was padded with 16
unused bits in order to bring it up to word length. These extra bits could possibly
be used to cache point colours after shading or to extend the possible colour
depth of the colour of each point (16bit ³ 32bit).

The most noticeable defect in the shadow functionality is the blocky edges of
shadows cast from the object on to the background. The current functionality for
generating multiple shadow maps at different positions could be extended to aid
this problem. By storing two (or more) bits for each light in the point attributes,
the point could be marked as ‘in shadow’, ‘out of shadow’ or ‘partially in
shadow’ to some extent. This would have the effect of softening the shadow
edges, simultaneously alleviating the problem of blocky edges and also creating
more realistic soft-shadows.

Currently, the shadow calculation algorithm is too time-consuming for real-time
use, making it unfeasible to move the object or change the lighting conditions.
These actions could be made possible by executing the shadow calculation
algorithm in a separate thread. This would allow the object/lights to be moved,
creating a small delay before the shadows are corrected.
8 Bibliography

8.1 References


http://freespace.virgin.net/hugo.elias/models/m_perlin.htm (November 1st 2005)

[Georgia 05] The PLY File Format (2005) [WWW]


Appendix

A Accompanying CD

The CD accompanying this dissertation (found on the back cover) contains the following:

- Software produced (Virus checked with Grisoft AVG Anti-Virus)
- Videos and screen shots from the project
- Complete code listing
- Electronic copies of this document and the project proposal

The CD has an autorun program that launches a menu in the form of a webpage. If this page fails to appear, it can be located at \index.html in the root folder of the CD.

B Screen Shots

Figure B.1: Statuette Model (4,999,996 points)

Figure B.2: Stanford Bunny model and two lights without shadow (left) and with shadow (right).
Figure B.3: Stanford Bunny model with two lights, point colours, background and shadows.

Figure B.4: Happy Buddha model with marble texture on square plane with wood texture.
Figure B.5: Sphere (32,768 points) with two lights and environment mapping with blending factor of 1.0.

Figure B.6: Simple menu application to provide friendlier user interface for setting renderer parameters.

C Help Documentation

The help document that is opened when the user hits the F1 key is displayed on the following page. It is almost identical to the section: ‘5.7 User Interface’.
Point Grid Renderer User Guide

When the application is run, the scene (described by the command line arguments) is setup. During this initialisation (normally just a few seconds) the string “Loading…” is displayed in the window to notify the user of inactivity during this period. After this initialisation, the camera is positioned at a suitable location to view the object being modelled. There are two camera movement models (that can be chosen with the command line arguments):

- Target Camera – has a constant target (the center of the object). Moving the mouse will alter the position of the camera while keeping the distance from the target constant. The direction of the camera will be updated so that it still points at the target point. Pressing the up and down arrow keys moves the camera position (along the direction vector) closer and further from the target respectively.

- Flying Camera – position stays constant while the mouse is moved. Moving the mouse will alter the direction of the camera, updating the target as it changes. Pressing the up and down arrow keys moves the camera position along the direction vector in the positive and negative direction respectively.

Pressing the space bar at any time will capture a frame and save it as a BMP image file (at the location C:\frame.bmp). Pressing the escape key at any time will exit the application. Pressing the F1 key at any time will open this help document.

The optional parameters of the renderer are set using the following optional command line arguments:

- **-AVI**
  Outputs an uncompressed AVI video file of all the frames rendered while the application runs (at C:\test.avi). Also selects an automatic camera movement model that follows a preset path and exits the application at the end of that path.

- **-FLYING_CAMERA**
  Specifies the flying camera model instead of the target camera model (default).

- **-IGNORE_POINT_COLOURS**
  Renders the object with a default grey material with no textures or point colours.

- **-BACKGROUND <filename>**
  Adds a background object to the scene. The optional <filename> parameter overrides the default background image and specifies an alternative image to be mapped to the spherical background.

- **-SHADOW**
  Specifies that shadows should be added to the scene.

- **-TEXTURE <name>**
  Specifies that the point colours of the object and the plane in the background object should be overridden with texture mapped colours. The optional <name> parameter overrides the default texture (marble) and specifies the texture to be used for the object and may take the value of either “marble” or “wood”.

- **-ENVIRONMENT_MAP**
  Specifies that the environment may be mapped to the object if the material of the object requests it.

  <filename>
  The filename of the PLY file to open. If this parameter is not supplied, the default model will be loaded.

The material of an object can be specified in the PLY file. The following material parameters may be supplied: \(ka, kd, ks, power\) - Phong lighting coefficients, \(clr_r, clr_g, clr_b\) - material colour components and \(envMapFactor\) – the environment map blending factor. The default values for any parameters not specified are those of a matte (ambient reflection only with no environment mapping) grey material.

These parameters may be selected in a more user friendly environment using the menu application.
Appendix

D Code Listing

This section contains a subset of the code written in this project to provide details of the implementation. The following code samples have been included:

- Virtual Machine
- PLY File
- Binary Voxel Grid
- Object
- Renderer
- Drawing a Splat
- Normal Lookup Table
- Splat Kernel Lookup Table
- Background
- Shadows

Virtual Machine

The main execution loop of the system lies in the vm.cpp file which has three functions. The code is as follows:

```cpp
//default values for command line args
int recordAVI=0, flyingCamera=0,ignorePointColours=0,showBackground=0;
int backgroundHasPlane=0, haveShadow=0, haveTexture=0, haveEnvironmentMapping=0;
char * backgroundFilename = "D:\release\backgrounds\hotelroom.bmp";
char * plyFilename = "D:\release\objects\bunny.ply";

//z-buffer memory
float _zBuffer[WINDOW_WIDTH * WINDOW_HEIGHT];
SCREEN_BUFFER z_buffer;

//object data and camera
Camera cam;
Object * obj;
BACKGROUND * background = (BACKGROUND *)0;

/* Called once when the application starts. Initiates all objects */
int VM_Init(int argc, char ** argv)
{
    //initialise lookup tables
    InitTrigTables();
    InitDiscRadLookupTable();
    InitLights();
    InitMaterials();
    GenerateTextures();
    GenerateNormalTable();
    GenerateSplatKernelTable();

    //parse command line arguments
    for(int i=0; i<argc; i++)
    {
        if(argv[i][0] != '-')
            plyFilename = argv[i];
        else if(strcmp(argv[i],"-AVI") == 0)
            recordAVI = 1;
        else if(strcmp(argv[i],"-FLYING_CAMERA") == 0)
            flyingCamera = 1;
        else if(strcmp(argv[i],"-IGNORE_POINT_COLOURS") == 0)
            ignorePointColours = 1;
        else if(strcmp(argv[i],"-BACKGROUND") == 0)
        {
            showBackground = 1;
            if(argc >= (i+2)) backgroundFilename = argv[++i];
        }
        else if(strcmp(argv[i],"-SHADOW") == 0)
            haveShadow = 1;
        else if(strcmp(argv[i],"-TEXTURE") == 0)
            haveTexture = 1;
    }
```
Appendix

```c
/*
 * haveTexture = 1;
 * if(argc >= (i+2)) textureName = argv[++i];
 */
else if(strcmp(argv[i],"-ENVIRONMENT_MAP") == 0)
    haveEnvironmentMapping = 1;
}

//initialise avi for recording
if(recordAVI)
{
    InitAviRecording("c:\test.avi",20);
}

//initialise l/z-buffer
SetupScreenBuffer( zBuffer,WINDOW_WIDTH,WINDOW_HEIGHT,&z_buffer);
float * ptr = _zBuffer+(WINDOW_HEIGHT>>1)*WINDOW_WIDTH + (WINDOW_WIDTH>>1);
z_buffer.origin = ptr;

//add lights
LIGHT * light1 = AddLight();
LIGHT * light2 = AddLight();
light1->direction.Set(1.0f,-3.0f,1.0f);
light2->direction.Set(-1.0f,-3.0f,1.0f);
light1->intensity = 200;
light2->intensity = 200;

//do some lighting pre-calculations
InitLighting();

//initialise object
PLY_DATA * ply_data = LoadPlyFile(plyFilename);
if( ! ply_data) return 0;
if(ignorePointColours) {
    ply_data->material->usePointClr = 0;
}
if(haveTexture) {
    ply_data->material->usePointClr = 1;
    ply_data->material->texture = GetTextureByName(textureName);
}
obj = new Object(ply_data);

//generate background
if(showBackground)
{
    background = Background_Generate(backgroundFilename, obj);
}

//free ply file data as it will not be necessary any more
delete ply_data;

//pre-calculate shadows
if(haveShadow)
{
    CalculateShadows(obj, background);
}

//initialise camera
cam.Initialise(obj, &back_buffer);
return 1;
}

/* Main method of application. Called once for every frame. */
int VM_Main()
{
    //do environment mapping and backface culling
    if(haveEnvironmentMapping)
        DoEnvironmentMappingAndBackfaceCulling(&cam,background);
    else
        SetCulledNormals(&cam);
    //do lighting calculations for each normal
```
DoLighting(&cam);
//Clear 1/z-buffer
ClearScreenBuffer(z_buffer);
//render background to back buffer
if(showBackground)
{
    Render_PointGrid(background->grid, &cam, background->position, &back_buffer, &z_buffer, 0);
}
//render object to back buffer
obj->Render(&cam, &back_buffer, &z_buffer);
//render to avi and move camera
if(recordAVI)
{
    RecordAviFrame();
    //update camera position based on predefined path
    if(cam.MoveAsAutomaticCamera())
    {
        //exit application if predefined camera has ended
        return 0;
    }
    else
    {
        //update camera position from user input
        if(flyingCamera)
            cam.MoveAsFlyingCamera();
        else
            cam.MoveAsTargetCamera();
    }
    //if spacebar pressed, capture frame
    if(keyboard_state[KEY_SPACE])
    {
        SaveAviFrameAsBmp("c:\frame.bmp");
    }
    //if F1 key pressed, open help file
    if(keyboard_state[KEY_F1])
    {
        Launch("D:\release\help.pdf");
    }
    return 1;
}
/* Called when the application exits. Frees any memory in use. */
int VM_Shutdown()
{
    //finish recording avi
    if(recordAVI) FinaliseAviRecording();
    //delete objects
    delete obj;
    if(background) delete background;
    return 1;
}

**PLY File**

The definition for the PLY file data structure is given in the following code sample:

```c
typedef struct _ply_data
{
    int point_count;   //number of points in this structure
    POINT3D * points;  //array of point data
    VECTOR * normals;  //array of normal data
    float * discRadii; //parallel array of splat radius data
};
```
unsigned short * colours; //parallel array of colour data
MATERIAL * material;  //material for this object (NULL if none)
~_ply_data()   //destructor
{   
  if(this->points) delete this->points;
  if(this->normals) delete this->normals;
  if(this->discRadii) delete this->discRadii;
  if(this->colours) delete this->colours;
}
} PLY_DATA;

/* Reads a PLY file and returns a PLY_DATA structure
* containing the 3D-point data in the file. Returns
* NULL if an error occurs.
*/
PLY_DATA * LoadPlyFile(char * filename);

/* Saves a PLY file to a new location.
* Returns 0 if the file already exists or if the method fails.
*/
int SavePlyFile(PLY_DATA * ply, char * filename);

Binary Voxel Grid
The following code sample generates a binary voxel grid from ply data:

#define MAX_GRID_DEPTH 9

typedef struct _point_attribute
{
  unsigned int normal:3 + 2*NORMAL_TABLE_REFINEMENT_DEPTH; //index into normal lookup table
  unsigned int colour:16; //colour of this point
  unsigned int material:3; //index of the material of this point
  unsigned int inShadow:8; //each bit is a boolean that indicates whether the point is in view of a particular light source
  unsigned int discRad:8; //radius of splat
  unsigned int padding:12; //unused bits to bring struct up to word boundary
} POINT_ATTR;

typedef struct _point_grid
{
  POINT3D location;  // location (object space) of center of this grid
  float size;  //the edge length of this grid
  char isDepthFirst; //breadth first if zero and depth first otherwise
  unsigned char gridDepth; //the depth of the grid
  unsigned char * data; //start of the grid data stream
  POINT_ATTR * attrData; //attribute stream
  int dataLen; //length of data stream
  int attrLen; //length of attribute stream
  
  ~_point_grid()   //destructor
  {   
    if(this->attrData) delete this->attrData;
    if(this->data) delete this->data;
  }
} POINT_GRID;

/* Spatially partions a set of irregular 3D point samples
* into a grid. A Grid regularly partions a region of 3-space
* and acts as a characteristic function that returns boolean
* values for points in 3-space.
* Parameters:
Appendix

* ply - parallel array of points, normals, disc-radii, colours
* origin - origin in world-space of points.
* min - any points smaller than min are not added to the point grid
* max - any points larger than max are not added to the point grid
* isDepthFirst - ordering of grid
* grid_depth - depth of the point grid to be generated

*/

POINT_GRID * GeneratePointGrid(PLY_DATA * ply, VECTOR origin, VECTOR min, VECTOR max, int isDepthFirst, int grid_depth)
{
    POINT_GRID * grid = (POINT_GRID *)0;
    // uncompressed data stream that represents the binary voxel grid:
    unsigned char * tempGrid;
    // sizes (number of nodes) of sub trees in the voxel grid octree
    int subGridSizes[MAX_GRID_DEPTH+1];
    int pcount = ply->point_count; // local copy of number of points
    // error check
    if( ! ply->points || pcount <= 0)
    {
        WriteError("GRD001","cannot generate a grid with no points");
        return grid;
    }
    if(grid_depth < 1 || grid_depth > MAX_GRID_DEPTH)
    {
        WriteError("GRD002","grid depth must be between 1 and MAX_GRID_DEPTH inclusive");
        return grid;
    }
    // size of cube that contains all points in array
    float grid_size = MAX(MAX(max.x-min.x,max.y-min.y),max.z-min.z);
    // initialise texture step size for possibility of textures
    // texture_step = grid_size / (2^grid_depth)
    // this equates to one texture voxel per leaf node
    float textureStep = 0.0f;
    MATERIAL * material = ply->material;
    if(material && material->texture)
    {
        textureStep = 2.5f * grid_size / (float)(1 << grid_depth);
    }
    // initialise temporary grid
    subGridSizes[0] = 1;
    int pow = 8;
    for(int i=1; i<=grid_depth; i++)
    {
        subGridSizes[i] = subGridSizes[i-1] + pow;
        pow *= 8;
    }
    int tempGridSize = subGridSizes[grid_depth-1];
    tempGrid = new unsigned char[tempGridSize];
    for(int i=0; i<tempGridSize; i++)
    {
        tempGrid[i] = 0;
    }
    // initialise temporary attribute stream
    // stores the index of the node in the temporary grid stream:
    int * attr_order = new int[pcount];
    // attributes of each point
    POINT_ATTR * attr_stream = new POINT_ATTR[pcount];
    int attr_index = 0; // index into attr_stream array
    // set value for minimum splat size with this grid depth
    float minSplatSize = 1 / (float)(1 << grid_depth);
    // add points to temporary grid
    for(int i=0; i<pcount; i++)
    {
        // translate point so that its origin matches that of the voxel grid
        VECTOR p;
        Vector_Sub(ply->points[i],origin,p);
        // skip point if it is not within interval [min,max]
if(p.x<min.x || p.x>=max.x || p.y<min.y || p.y>=max.y || p.z<min.z
|| p.z>=max.z)
{
    continue;
}

//translate point again to make min the origin
Vector_SubFrom(p,min);

int index = 0; //address of current block in tempGrid
int n = 0;  //id of octant that p is in

//for breadth-first index finding
int n_stack[MAX_GRID_DEPTH];
int n_stack_ptr = 0;

//dimensions of blocks
float half_grid_size = grid_size;
VECTOR block_min, block_max, block_mid;
block_min.Set(0.0f, 0.0f, 0.0f);

//traverse tree to find leaf for p
for(int j=0; j<grid_depth; j++)
{
    //find bounds of current block
    if(n & 1)
        block_min.x += half_grid_size;
    if(n & 2)
        block_min.y += half_grid_size;
    if(n & 4)
        block_min.z += half_grid_size;
    block_max.Set(block_min.x + half_grid_size, block_min.y +
    half_grid_size, block_min.z + half_grid_size);
    half_grid_size /= 2;
    block_mid.Set(block_min.x + half_grid_size, block_min.y +
    half_grid_size, block_min.z + half_grid_size);

    //find which octant p is in
    if(p.x < block_mid.x)
    {
        if(p.y < block_mid.y)
            n = (p.z < block_mid.z) ? 0 : 4;
        else
            n = (p.z < block_mid.z) ? 2 : 6;
    }
    else
    {
        if(p.y < block_mid.y)
            n = (p.z < block_mid.z) ? 1 : 5;
        else
            n = (p.z < block_mid.z) ? 3 : 7;
    }

    //ensure nth bit of block is set to 1
    tempGrid[index] |= (1 << n);

    //find address of next block in tempGrid
    if((j<(grid_depth-1))
    {
        if(isDepthFirst)
        {
            index += 1 + n*subGridSizes[grid_depth-j-2];
        }
        else
        {
            //nodes to the left on current tree tier
            int left_side = 0;
            //nodes to the right on previous tree tier
            int right_side = 0;
            n_stack[n_stack_ptr++] = n;
            pow = 1;
            left_side += n_stack[n_stack_ptr - 1];
            for(int m=n_stack_ptr-2; m>=0; m--)
            {
                right_side += (7 - n_stack[m]) * pow;
            }
        }
    }
}
Appendix

pow *= 8;
left_side += n_stack[m] * pow;
}
index += right_side + left_side + 1;
}

//set attributes of this point
attr_stream[attr_index].normal = GetNormalIndex(ply->normals[i]);
if(material->usePointClr)
{
  if(material->texture)
    attr_stream[attr_index].colour = GetTextureColour(&p,material->texture, textureStep);
  else
    attr_stream[attr_index].colour = (ply->colours) ? ply->colours[i] : material->colour;
}
else
  attr_stream[attr_index].colour = material->colour;
attr_stream[attr_index].material = material->index;
attr_stream[attr_index].inShadow = 0;
float splatRad = ply->discRadii[i] / grid_size;
attr_stream[attr_index].discRad = GetDiscIndex(MAX(splatRad,minSplatSize));
attr_order[attr_index] = 8*index+n; //set to index in tempGrid
attr_index++;

//reorder attribute stream using attr_order as keys
int attr_count = attr_index;
HeapSort(attr_order,attr_stream,sizeof(POINT_ATTR),attr_count);

//replace multiple entries with an average of the attribute
int prev_index = attr_order[0];
attr_index = 0;
float maxDiscRad = 0;
VECTOR norm;
norm.Set(0.0f, 0.0f, 0.0f);
for(int i=0; i<attr_count; i++)
{
  if(prev_index == attr_order[i])
  {
    Vector_AddTo(norm,NormalTable[attr_stream[i].normal].normal);
    if(maxDiscRad < DiscRadLookupTable[attr_stream[i].discRad])
      maxDiscRad = DiscRadLookupTable[attr_stream[i].discRad];
  }
  else
  {
    attr_stream[attr_index] = attr_stream[i-1];
    attr_stream[attr_index].normal = GetNormalIndex(norm);
    attr_stream[attr_index].discRad = GetDiscIndex(maxDiscRad);
    attr_index++;
    Vector_Copy(&NormalTable[attr_stream[i].normal].normal, &norm);
    maxDiscRad = DiscRadLookupTable[attr_stream[i].discRad];
  }
  prev_index = attr_order[i];
  if(tempGrid[i] > 0) compressed_size++;
}
//create new grid and add compressed data
grid = new POINT_GRID;
Vector_Copy(&min, &(grid->location));
grid->size = grid_size;
grid->isDepthFirst = (char)isDepthFirst;
grid->gridDepth = grid_depth;
grid->data = new unsigned char[compressed_size];
grid->attrData = new POINT_ATTR[attr_index];
Appendix

```c
grid->dataLen = compressed_size;
grid->attrLen = attr_index;
int ptr = 0;
unsigned char * d = grid->data;
for(int i=0; i<tempGridSize; i++)
{
    if(tempGrid[i] > 0) d[ptr++] = tempGrid[i];
}
memcpy(grid->attrData,attr_stream,attr_index*sizeof(POINT_ATTR));
//free temporary arrays
delete attr_stream;
delete attr_order;
delete tempGrid;
return grid;
}

Object
An object is then represented by the following class:

class Object
{
private:
    Object() {}     //private simple constructor
public:
    float size;     //y-coord of lowest
    float min_y;  //y-coord of lowest
    POINT_GRID grids[8]; //octree of PointGrids that are rendered in order
    POINT3D position; //translation of this object in world-space
    ~Object();  //destructor
    Object(PLY_DATA * ply)
    {
        //find min/max bounds of object
        VECTOR min, max, mid, a,b;
        VectorMinMax(ply->points,ply->point_count,&min,&max);
        Vector_SubFrom(max,min);
        Scalar_Vector_Multiply(0.5f,max,mid); //mid-point of object space
        size = MAX(MAX(max.x,max.y),max.z);  //total size of object
        //calculate appropriate depth of point grid
        int grid_depth = MAX_GRID_DEPTH;
        if(ply->point_count > 3000000) grid_depth--;
        if(ply->point_count > 4500000) grid_depth--;
        //split obj space into 8 octants & generate point-grid for each one
        a.z = 0.0f;  b.z = mid.z;
        for(int z=0; z<2; z++)
        {
            a.y = 0.0f;  b.y = mid.y;
            for(int y=0; y<2; y++)
            {
                a.x = 0.0f;  b.x = mid.x;
                for(int x=0; x<2; x++)
                {
                    int gridIndex = (z<<2) + (y<<1) + x;
                    grids[gridIndex] = GeneratePointGrid(ply,
                    min, a, b, l, grid_depth);
                }
            }
        }
        //set translation of this object
        Vector_Copy(&VECTOR_ZERO, &position);
    }
    void Render(Camera* cam,SCR_BUFFER* screen_buffer,SCR_BUFFER* depth_buffer)
```
Appendix

#include "gl.h"
#include "math.h"

//decide rendering order of grids – closest to viewpoint first
int distances[8];
float size_div4 = this->size / 4;
for(int i=0; i<8; i++)
{
    VECTOR v;
    v.x = grids[i]->location.x + size_div4 - cam->position.x;
    v.y = grids[i]->location.y + size_div4 - cam->position.y;
    v.z = grids[i]->location.z + size_div4 - cam->position.z;
    distances[i] = (int)Vector_NormSquared(v);
}

HeapSort(distances,grids,sizeof(void*),8);

//render grids in order of closeness to viewpoint
for(int i=0; i<8; i++)
    Render_PointGrid(grids[i],cam,position,screen_buffer,
                   depth_buffer,0);
}

Normal Lookup Table

The code for the normal lookup table is as follows:

typedef struct _normal_table_entry
{
    VECTOR normal; //the unit normal for this table entry
    float dotLight[MAX_LIGHTS]; //the dot product of this normal and each of the light directions
    unsigned int diff_light[MAX_LIGHTS][MAX_MATERIALS]; //unsigned int spec_light[MAX_LIGHTS][MAX_MATERIALS];
    int culled:1; //if normal is back-face culled, 0 otherwise
    int splatKernelIndex:5; //part of index into splat kernel table
    unsigned short envMapClr; //environment map reflected colour
    int padding:10; //unused
} NORMAL;

#define NORMAL_TABLE_REFINEMENT_DEPTH 5
#define NORMAL_TABLE_LENGTH (1 << (3 + 2*NORMAL_TABLE_REFINEMENT_DEPTH))

NORMAL NormalTable[NORMAL_TABLE_LENGTH];

VECTOR a,b,c; // vertices of the current triangle in the uniformly divided sphere

/* Sets the values of the vectors a,b,c to the vertices of a triangle
   * in the octant given by index (3 bits)
   */
void SetABC(int index)
{
    switch(index)
    {
        case 0:
            a.Set( 0.0f, 0.0f,-1.0f);
            b.Set( 0.0f,-1.0f, 0.0f);
            c.Set(-1.0f, 0.0f, 0.0f);
            break;
        case 1:
            a.Set( 1.0f, 0.0f, 0.0f);
            b.Set( 0.0f,-1.0f, 0.0f);
            c.Set( 0.0f, 1.0f, 0.0f);
            break;
        case 2:
            a.Set(-1.0f, 0.0f, 0.0f);
            b.Set( 0.0f, 1.0f, 0.0f);
            c.Set( 0.0f, 0.0f,-1.0f);
            break;
        case 3:
            a.Set( 0.0f, 0.0f,-1.0f);
            b.Set( 0.0f,-1.0f, 0.0f);
            c.Set(-1.0f, 0.0f, 0.0f);
            break;
        default:
            a.Set( 1.0f, 0.0f,-1.0f);
            b.Set( 0.0f,-1.0f, 1.0f);
            c.Set(-1.0f, 1.0f, 0.0f);
            break;
    }
}
Appendix

```c
void UpdateABC(int index)
{
    VECTOR templ, temp2;
    switch(index)
    {
    case 0:
        Vector_Copy(&a, &temp1);
        Vector_Copy(&b, &temp2);
        Vector_Average(b, c, a) //a = avr(b, c)
        Vector_Average(temp1,  c, b) //b = avr(a, c)
        Vector_Average(temp1, temp2, c) //c = avr(a, b)
        break;
    case 1:
        Vector_Average(a, b, b) //b = avr(a, b)
        Vector_Average(a, c, c) //c = avr(a, c)
        break;
    case 2:
        Vector_Average(a, c, a) //a = avr(a, c)
        Vector_Average(b, c, b) //b = avr(b, c)
        break;
    case 3:
        Vector_Average(a, b, a) //a = avr(a, b)
        Vector_Average(b, c, c) //c = avr(b, c)
        break;
    }
}
/*
 * Updates the values of the vectors a,b,c to the vertices of the subtriangle of the original a,b,c values given by index (2 bits)
 */
void UpdateABC(int index)
{
    VECTOR templ, temp2;
    switch(index)
    {
    case 0:
        Vector_Copy(&a, &temp1);
        Vector_Copy(&b, &temp2);
        Vector_Average(b, c, a) //a = avr(b, c)
        Vector_Average(temp1,  c, b) //b = avr(a, c)
        Vector_Average(temp1, temp2, c) //c = avr(a, b)
        break;
    case 1:
        Vector_Average(a, b, b) //b = avr(a, b)
        Vector_Average(a, c, c) //c = avr(a, c)
        break;
    case 2:
        Vector_Average(a, c, a) //a = avr(a, c)
        Vector_Average(b, c, b) //b = avr(b, c)
        break;
    case 3:
        Vector_Average(a, b, a) //a = avr(a, b)
        Vector_Average(b, c, c) //c = avr(b, c)
        break;
    }
}
/*
 * Fill normal table with unit vectors uniformly distributed around a sphere.
 */
void GenerateNormalTable()
{
    memset(NormalTable, 0, NORMAL_TABLE_LENGTH*sizeof(NORMAL));
    GenerateNormalTableIn(&NormalTable[0].normal, sizeof(NORMAL),
        NORMAL_TABLE_REFINEMENT_DEPTH);
}
/*
 * Fill an array with unit vectors uniformly distributed around a sphere.
 */
void GenerateNormalTableIn(VECTOR * table, size_t entrySize, int refinementDepth)
{
    int count = (1 << (3 + 2*refinementDepth));
    int shift = 2 * refinementDepth; //number of bits for refinement
    for(int i=0; i<count; i++)
    {
        //find which octant normal is in
        int mask = (1 & (7 << shift)) >> shift; //bits: 1-3
```
Appendix

SetABC(mask);

// refine
for(int bits=2*refinementDepth-2; bits >= 0; bits -= 2)
{
    mask = (i & (3 << bits)) >> bits;
    UpdateABC(mask);
}

// set normal as average of the three vertices
table->x = (a.x + b.x + c.x)/3;
table->y = (a.y + b.y + c.y)/3;
table->z = (a.z + b.z + c.z)/3;
Vector_Normalise(table);

// set pointer to next normal in table
if(table = (VECTOR *)((char *)table + entrySize))
{
    /* Returns the normal from the lookup table that most closely matches n */
    int GetNormalIndex(VECTOR n)
    {
        int index = 0;
        // find which octant n is in
        if(n.x < 0)
        {
            if(n.y < 0)
            {
                index = (n.z < 0) ? 0 : 4;
            }
            else
            {
                index = (n.z < 0) ? 2 : 6;
            }
        }
        else
        {
            if(n.y < 0)
            {
                index = (n.z < 0) ? 1 : 5;
            }
            else
            {
                index = (n.z < 0) ? 3 : 7;
            }
        }
        // define bounds (a,b,c)
        SetABC(index);
        // refine normal
        for(int i=0; i<NORMAL_TABLE_REFINEMENT_DEPTH; i++)
        {
            // calculate dot products
            float aDot = Vector_Dot(a,n);
            float bDot = Vector_Dot(b,n);
            float cDot = Vector_Dot(c,n);
            // find which sub triangle n is in
            VECTOR mid;
            float max,dist;
            int code = 0;
            if(aDot > bDot)
            {
                if(aDot > cDot) // aDot is max
                {
                    Vector_Average(b,c,mid);
                    max = aDot;
                    dist = (a.x-n.x)*(a.x-n.x) + (a.y-n.y)*(a.y-n.y) +
                    (a.z-n.z)*(a.z-n.z);
                    code = 1;
                }
                else // cDot is max
                {
                    Vector_Average(a,b,mid);
                    max = cDot;
                    dist = (c.x-n.x)*(c.x-n.x) + (c.y-n.y)*(c.y-n.y) +
                    (c.z-n.z)*(c.z-n.z);
                    code = 2;
                }
            }
            else
            {

            }
        }
    }
}
if(bDot > cDot) // bDot is max
{
    Vector_Average(a,c,mid);
    max = bDot;
    dist = (b.x-n.x)*(b.x-n.x) + (b.y-n.y)*(b.y-n.y) +
    (b.z-n.z)*(b.z-n.z);
    code = 3;
}
else // cDot is max
{
    Vector_Average(a,b,mid);
    max = cDot;
    dist = (c.x-n.x)*(c.x-n.x) + (c.y-n.y)*(c.y-n.y) +
    (c.z-n.z)*(c.z-n.z);
    code = 2;
}

// find whether mid or max vertex is closest to n
Vector_Normalise(&mid);
if(Vector_Dot(mid,n) > max)
{
    code = 0;
}

// add code to index
index = (index << 2) + code;
UpdateABC(code);
Vector_Normalise(&a);
Vector_Normalise(&b);
Vector_Normalise(&c);
}
return index;

// only do lighting/culling on subset of normals - this variable
// specifies how many normals to skip:
int skip = 1 << (2*NORMAL_TABLE_REFINEMENT_DEPTH - 10);

/* Flags all normals that are back-facing the camera direction.
*/
void SetCulledNormals(Camera * cam)
{
    VECTOR view;
    Vector_Copy(&cam->unitDirection, &view);
    for(int i=0; i<NORMAL_TABLE_LENGTH; )
    {
        float dot = Vector_Dot(NormalTable[i].normal, view);
        int culled = (dot < 0.0f) ? 0 : 1;
        #if NORMAL_TABLE_REFINEMENT_DEPTH > 5
            for(int l=0; l<skip; l++)
            NormalTable[i++].culled = culled;
        #endif
    }

    /* Precalculate some dot products and non-view-specific lighting calculations.
    */
    void InitLighting()
    {
        // ensure all light directions are normalised
        for(int j=0; j<light_count; j++)
        {
            Vector_Normalise(&LightSources[j].direction);
        }

        // precalculate dot products and non-view-specific lighting calculations
        for(int i=0; i<NORMAL_TABLE_LENGTH; i++)
        {
            for(int j=0; j<light_count; j++)
            {
                NormalTable[i].dotLight[j] = -
                Vector_Dot(NormalTable[i].normal, LightSources[j].direction);
                for(int k=0; k<material_count; k++)
                {
                }
            }
        }
    }
NormalTable[i].diff_light[j][k] =
Diffuse_Component(NormalTable[i],Materials[k],j);
}
}
}

/* Apply a lighting model to each normal so that lighting values can
 * be looked up based on a points normal during rendering. */
void DoLighting(Camera * cam)
{
    for(int i=0; i<NORMAL_TABLE_LENGTH; i += skip)
    {
        for(int j=0; j<light_count; j++)
        {
            for(int k=0; k<material_count; k++)
            {
                int spec =
Specular_Component(NormalTable[i],Materials[k],j,cam);
                int l=i;
                #if NORMAL_TABLE_REFINEMENT_DEPTH > 5
                int max = i + skip;
                for(; l<max; l++)
                #endif
                NormalTable[l].spec_light[j][k] = spec;
            }
        }
    }
}

/* Transform normal and store result as 6-bit index into
 * normal table. This can then be used as part of an index
 * to a splat kernel lookup table. */
void SetSplatIndexes(Camera * cam)
{
    MATRIX4X4 trans;
    cam->Set_World_to_Perspective_Transform(&trans);
    for(int i=0; i<NORMAL_TABLE_LENGTH; )
    {
        //get transformed normal index
        VECTOR new_norm;
        Matrix3x3_Vector_Multiply(trans,NormalTable[i].normal,new_norm);
        int index = GetNormalIndex(new_norm);
        //mask out least significant bits
        index &= 63 << (2*NORMAL_TABLE_REFINEMENT_DEPTH - 6);
        //shift right to leave bits free for remaining bits of lookup index
        index >>= (2*NORMAL_TABLE_REFINEMENT_DEPTH - 10);
        #if NORMAL_TABLE_REFINEMENT_DEPTH > 5
        for(int l=0; l<skip; l++)
        #endif
        NormalTable[i++].splatKernelIndex = index;
    }
}

/* Reflect the view direction in each plane (orthogonal to each normal)
 * and use the result to lookup the colour from the environment map.
 * Also flags all normals that are back-facing the camera direction. */
void DoEnvironmentMappingAndBackfaceCulling(Camera * cam, BACKGROUND * bg)
{
    VECTOR D;
    Vector_Copy(&cam->unitDirection, &D);
    for(int i=0; i<NORMAL_TABLE_LENGTH; i++)
    {
        VECTOR N;
        Vector_Copy(&NormalTable[i].normal, &N);
        // R = camera direction reflected in this normal
        //   = D - 2N(D.N)
        VECTOR R;
        float s = 2.0f * Vector_Dot(N, D);
        R.x = D.x - s*N.x;
Appendix

R.y = D.y - s*N.y;
R.z = D.z - s*N.z;

// perform backface culling on normal
NormalTable[i].culled = (s < 0.0f) ? 0 : 1;

// set colour of environment
int index = GetNormalIndex(R) << 2*(BACKGROUND_REFINEMENT_DEPTH - NORMAL_TABLE_REFINEMENT_DEPTH);
NormalTable[i].envMapClr = bg->colours[index];
}
}

Splat Kernel Lookup table

The code for the generation of the splat kernel lookup table is as follows:

typedef struct _splat_kernel
{
    int splat[4]; // 16x16 bit monochrome image of a splat
    // each int is 4x(start,end) pairs (4-bits each)
} SPLAT_KERNEL;

// splat kernel table data
#define SPLAT_KERNEL_TABLE_LENGTH 1024
SPLAT_KERNEL SplatKernelTable[SPLAT_KERNEL_TABLE_LENGTH];

/* Generate the splat kernel lookup table. */
void GenerateSplatKernelTable()
{
    // clear splat kernel table
    memset(SplatKernelTable,0,SPLAT_KERNEL_TABLE_LENGTH*sizeof(SPLAT_KERNEL));

    // fill table
    for(int norm=0; norm<64; norm++)
        for(int x=0; x<2; x++)
            for(int y=0; y<2; y++)
                for(int z=0; z<4; z++)
                {
                    int splatIndex = (norm << 4) + (x << 3) + (y << 2) + z;

                    // set actual index into normal lookup table
                    int normIndex = norm << (2*NORMAL_TABLE_REFINEMENT_DEPTH - 6);

                    // set center of splat
                    VECTOR p;
                    p.Set(z*x*16.0f+8.0f, z*y*16.0f+8.0f, z*32.0f+32.0f);

                    // calculate p.normal
                    float pDotNorm = Vector_Dot(p,NormalTable[normIndex].normal);

                    // for each pixel of splat kernel image
                    for(int col=0; col<16; col++)
                    {
                        int start = 1;
                        int end = 0;

                        // find start and end of non-zero bits in this row
                        for(int row=0; row<16; row++)
                        {
                            // get direction of view
                            VECTOR d;
                            d.Set(z*x*16.0f+row, z*y*16.0f+col, z*32.0f+32.0f);

                            // calculate normal.view
                            float normDotView = Vector_Dot(d,NormalTable[normIndex].normal);

                            // set d to point where view vector intersects plane
                            float lamda = pDotNorm / normDotView;
                            Scalar_Vector_MultiplyTo(lamda, d);
                        }
                    }
                }
}
Appendix

//if distance from p to d squared is <= 16 then
pixel (row,col) is in splat (splat radius = 4)
Vector_SubFrom(d,p);
if(Vector_NormSquared(d) <= 16.0f)
{
if(start)
{
    start = 0; end = 1;
    int mask = EVEN(col >> 1) ?
    (EVEN(col) ? row : (row << 8)) : (EVEN(col) ? (row << 16) : (row << 24));
    SplatKernelTable[splatIndex].splat[col >> 2] |= mask;
}
else
{
    if(end)
    {
        end = 0;
        int mask = EVEN(col >> 1) ?
        (EVEN(col) ? ((row+1)<< 4) : ((row+1) << 12)) : (EVEN(col) ? ((row+1) << 20) :
        ((row+1) << 28));
        SplatKernelTable[splatIndex].splat[col>>2] |= mask;
    }
}
}
}
//add start/end indices of nonzero bits of row to splat mask
if(start == 0  &&  end == 1)
{
    int mask = EVEN(col >> 1) ? (EVEN(col) ? (15 << 4) :
    (15 << 12)) : (EVEN(col) ? (15 << 20) : (15 << 28));
    SplatKernelTable[splatIndex].splat[col>>2] |= mask;
}
}
}

Renderer

The code to render a binary voxel grid to a buffer is as follows:

//globals for Render_PointGrid
VECTOR d[20][8];
unsigned char * data_ptr;
POINT_ATTR * attr_ptr;
void * screen;
int scr_pitch;
float scr_right,scr_left,scr_bottom,scr_top,scr_near,scr_far;
int screen_width_half, screen_height_half, screen_width_quart,
    screen_height_quart;
int outputColours;
float * zBuffer;
int zBuffer_pitch;
camDist_gridSize; //for finding splat scale = camera.distance * grid.size

/* The recursive step for the Render_PointGrid method.
 * Templated for different colour bit depths
 */
template <class CLR>
void depth_recurse(int level,float dx,float dy,float dz)
{
    if(level)
    {
        #define RECURSE(j) depth_recurse<CLR>({level},dx+di[j].x, dy+di[j].y, dz+di[j].z)
        level--;
        VECTOR * di = d[level];
        int code = (*data_ptr++);
        if(code & 1) RECURSE(0);
        if(code & 2) RECURSE(1);
        if(code & 4) RECURSE(2);
        if(code & 8) RECURSE(3);
        }
if(code & 16)  RECURSE(4);
if(code & 32)  RECURSE(5);
if(code & 64)  RECURSE(6);
if(code & 128) RECURSE(7);
else
{
    NORMAL * normal = &NormalTable[attr_ptr->normal];

    //back face culling
    if(normal->culled)
    {
        attr_ptr++;
        return;
    }

    //get screen coordinates
    float dz_inv = 1.0f/dz;
    float y = dy * dz_inv;
    float x = dx * dz_inv;

    //get radius of splat disc (in pixels)
    float splatRad = camDist_gridSize * dz_inv *DiscRadLookupTable[attr_ptr->discRad];

    //view frustum culling
    if(x < (scr_right-splatRad) && x > (scr_left+splatRad))
    {
        if(y < (scr_bottom-splatRad) && y > (scr_top+splatRad))
        {
            if(dz <= scr_far && dz >= scr_near)
            {
                //get screen coords
                int _x, _y;
                asm
                {
                    FLD y
                    FISTP _y
                    FLD x
                    FISTP _x
                }

                //get colour of this point
                CLR clr;
                if(outputColours)
                {
                    //get material of point
                    int matIndex = attr_ptr->material;
                    MATERIAL* mat = &Materials[matIndex];

                    //do lighting(if point not in shadow)
                    unsigned int intensity = mat->ka;
                    if(intensity < 255)
                        for(int i=0; i<=light_count; i++) {
                            if((attr_ptr->inShadow & (1<<i)) ==0) {
                                intensity += normal->diff_light[i][matIndex] + normal->spec_light[i][matIndex];
                            }
                        }

                    clr = Colour_Multiply(attr_ptr->colour,intensity);
                }

                //blend in environment colour
                if(mat->envMapFactor)
                {
                    clr = Colour_Interp(clr, normal->envMapClr, mat->envMapFactor);
                }
                else
                {
                    //set colour to attribute index
                    clr = (CLR)attr_ptr;
                }

                //get index into splat kernel lookup table
                int splatIndex = normal->splatKernelIndex;
                splatIndex |= (dz>0.5f) ? ((dz>0.75f) ? 3:2) : ((dz>0.25f) ? 1:0);

                //draw point to screen
                DrawSplat(_x, _y, dz_inv, clr, splatRad, splatIndex);
            }
        }
    }
}
Appendix

void Render_PointGrid(POINT_GRID * grid, Camera * cam, POINT3D position, SCREEN_BUFFER * screen_buffer, SCREEN_BUFFER * depth_buffer, int outputAttrIndexes)
{
    //this method can only traverse depth first grids
    if(! grid->isDepthFirst) return;

    //local copies of screen/depth buffer variables for faster/global access
    screen = (unsigned short *)screen_buffer->offset;
    scr_pitch = screen_buffer->pitch;
    zBuffer = (float *)depth_buffer->origin;
    zBuffer_pitch = depth_buffer->pitch;

    //set screen clipping bounds and translate screen pointer
    screen_width_half = screen_buffer->width >> 1;
    screen_height_half = screen_buffer->height >> 1;
    screen_width_quart = screen_width_half >> 1;
    screen_height_quart = screen_height_half >> 1;
    scr_right = (float)screen_width_half - 2.0f;
    scr_left = -(float)scr_right + 1.0f;
    scr_bottom = (float)screen_height_half - 2.0f;
    scr_top = -(float)scr_bottom + 1.0f;
    scr_near = cam->near_clip;
    scr_far = cam->far_clip;
    screen = (char*)screen + screen_buffer->sizeof_entry*(screen_width_half + screen_height_half*screen_buffer->pitch);

    //get object to screen transform
    MATRIX4X4 objectToPerspective;
    cam->Set_World_to_Perspective_Transform(&objectToPerspective);
    Matrix_AddTranslation(objectToPerspective,position);

    //set some globals
    //distance from the camera to the viewplane * width of grid:
    camDist_gridSize = cam->distance * grid->size;
    //specifies whether attr indexes or colours should be written to the screen
    outputColours = ! outputAttrIndexes;
    data_ptr = grid->data;   //start of the point grid data stream
    attr_ptr = grid->attrData;
    POINT3D loc;    //location of grid center
    Vector_Copy(&grid->location, &loc);
    Matrix3x4_Vector_MultiplyTo(objectToPerspective,loc);

    float h = grid->size / (float)(2 << grid->gridDepth);
    d[0][0].Set(-h,-h,-h);
    d[0][1].Set( h,-h,-h);
    d[0][2].Set(-h, h,-h);
    d[0][3].Set( h, h,-h);
    d[0][4].Set(-h,-h, h);
    d[0][5].Set( h,-h, h);
    d[0][6].Set(-h, h, h);
    d[0][7].Set( h, h, h);
    for(int j=0; j<8; j++)
        Matrix3x3_Vector_MultiplyTo(objectToPerspective,d[0][j]);

    for(int i=1; i<grid->gridDepth; i++)
    {
        for(int j=0; j<8; j++)
            Scalar_Vector_Multiply(2.0f, d[0][j], d[1][j])
    }
Appendix

Drawing a Splat

The code for drawing a splat to the buffer is as follows:

```c
/* Draws a row of a splat defined by an 8-bit segment of an integer: rows
* scaling is achieved by modifying the startShift and endShift values.
* define used instead of inline because there are too many parameters.
* NOTE: xAdd, yAdd = + or -
*/
#define DRAW_SPLAT_ROW(startMask, startShift, endMask, endShift, xAdd, yAdd) \\
{ 
    end = scr xAdd ((rows & endMask) >> (endShift));
    nextRow = scr yAdd scr_pitch;
    nextBufRow = zBufIndex yAdd zBuffer_pitch;
    int start = ((rows & startMask) >> (startShift));
    scr xAdd= start; zBufIndex xAdd= start;
    while(scr != end) \\
    { 
        if(zBuffer[zBufIndex] < z) \\
            { 
                *scr = colour; zBuffer[zBufIndex] = z;
            } 
        scr = scr xAdd 1;
        zBufIndex = zBufIndex xAdd 1;
    }
    scr = nextRow;
    zBufIndex = nextBufRow;
}

//same as above but start := ((rows & startMask) << (startShift));
#define DRAW_SPLAT_ROW_invStartShift ...
```

/* Draw a splat to the screen and update z-buffer
 */
template <class CLR>
inline void DrawSplat(int x, int y, float z, CLR colour, float scale, int splatIndex)
{
    //choose splat size/type
    if(scale < 3.0f)
    {
        if(scale < 1.0f) //draw single point
        {
            int zBufIndex = y*zBuffer_pitch + x;
            if(zBuffer[zBufIndex] < z)
            {
                *((CLR*)screen)[y*scr_pitch + x] = colour;
                zBuffer[zBufIndex] = z;
            }
        }
        else  //draw 3x3 square
        {
            int zBufIndex = (y-1)*zBuffer_pitch + x;
            CLR *scr = (CLR*)screen + scr_pitch*(y-1) + x;
            if(zBuffer[zBufIndex-1] < z)
            { scr[-1] = colour; zBuffer[zBufIndex-1] = z; }
            if(zBuffer[zBufIndex ] < z)
            { scr[ 0] = colour; zBuffer[zBufIndex ] = z; }
            if(zBuffer[zBufIndex+1] < z)
            { scr[+1] = colour; zBuffer[zBufIndex+1] = z; }
            scr += scr_pitch;
            zBufIndex += zBuffer_pitch;
            if(zBuffer[zBufIndex-1] < z)
            { scr[-1] = colour; zBuffer[zBufIndex-1] = z; }
```
if(zBuffer[zBufIndex] < z)
    { scr[ 0] = colour; zBuffer[zBufIndex] = z; }
if(zBuffer[zBufIndex+1] < z)
    { scr[+1] = colour; zBuffer[zBufIndex+1] = z; }
scr += scr_pitch;
zBufIndex += zBuffer_pitch;
if(zBuffer[zBufIndex-1] < z)
    { scr[-1] = colour; zBuffer[zBufIndex-1] = z; }
if(zBuffer[zBufIndex] < z)
    { scr[ 0] = colour; zBuffer[zBufIndex] = z; }
if(zBuffer[zBufIndex+1] < z)
    { scr[+1] = colour; zBuffer[zBufIndex+1] = z; }
}
} //draw splat
int diff = 8; //radius of splat
int factor = 1; //scaled splat = factor * origional splat radius
int shift = 0; // 2^shift = factor, shift = ciel(shift)

//do scaling of splat if necessary
if(scale > 8.0f)
{
    //round (scale/8) upto nearest power of two for its log2
    int scale_div8 = ((int)scale) >> 3;
    factor = 2;
    shift = 1;
    while(factor <= scale_div8 && shift < 4)
    {
        factor <<= 1;
        shift++;
    }
    diff <<= shift;
}

//complete splat index and draw splat
if(x < 0)
{
    if(y < 0)
    {
        if(-x > screen_width_quart) splatIndex |= 16;
        if(-y > screen_height_quart) splatIndex |= 8;
        x+=diff; y+=diff;
        int zBufIndex = y*zBuffer_pitch + x, nextBufRow;
        CLR *scr = (CLR*)screen + scr_pitch*y + x;
        for(int i=0; i<4; i++)
        {
            unsigned int rows =
                SplatKernelTable[splatIndex].splat[i];
            CLR *end, *nextRow;
            DRAW_SPLAT_ROW_invStartShift(15,shift,240,4-shift,-,-);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(3840,8-shift,61440,12-shift,-,-);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,-,-);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(251658240,24-shift,4026531840,28-shift,-,-);
        }
    }
    else
    {
        if(-x > screen_width_quart) splatIndex |= 16;
        if( y > screen_height_quart) splatIndex |= 8;
        x+=diff; y-=diff;
        int zBufIndex = y*zBuffer_pitch + x, nextBufRow;
        CLR *scr = (CLR*)screen + scr_pitch*y + x;
        for(int i=0; i<4; i++)
        {
            unsigned int rows =
                SplatKernelTable[splatIndex].splat[i];
            CLR *end, *nextRow;
            DRAW_SPLAT_ROW_invStartShift(15,shift,240,4-shift,-,-);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(3840,8-shift,61440,12-shift,-,+);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,-,+);
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(251658240,24-shift,4026531840,28-shift,-,+);
        }
    }
}
else
{
    if(-x > screen_width_quart) splatIndex |= 16;
    if( y > screen_height_quart) splatIndex |= 8;
    x+=diff; y-=diff;
    int zBufIndex = y*zBuffer_pitch + x, nextBufRow;
    CLR *scr = (CLR*)screen + scr_pitch*y + x;
    for(int i=0; i<4; i++)
    {
        unsigned int rows =
            SplatKernelTable[splatIndex].splat[i];
        CLR *end, *nextRow;
        DRAW_SPLAT_ROW_invStartShift(15,shift,240,4-shift,-,+);
        for(int j=0; j<factor; j++)
            DRAW_SPLAT_ROW(3840,8-shift,61440,12-shift,-,+);
        for(int j=0; j<factor; j++)
            DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,-,+);
    }
}
}
Appendix

```cpp
for(int j=0; j<factor; j++)
    DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,-,+);
    for(int j=0; j<factor; j++)
    DRAW_SPLAT_ROW(251658240,24-shift,4026531840,28-shift,-,+);
}
else
    if(y < 0)
    {
        if( x > screen_width_quart) splatIndex |= 16;
        if(-y > screen_height_quart) splatIndex |= 8;
        x-=diff; y+=diff;
        int zBufIndex = y*zBuffer_pitch + x, nextBufRow;
        CLR *scr = (CLR*)screen + scr_pitch*y + x;
        for(int i=0; i<4; i++)
            unsigned int rows =
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW_invStartShift(15,shift,240,4-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(3840,8-shift,61440,12-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(251658240,24-shift,4026531840,28-shift,+,-);
}
else
    {
        if( x > screen_width_quart) splatIndex |= 16;
        if( y > screen_height_quart) splatIndex |= 8;
        x-=diff; y-=diff;
        int zBufIndex = y*zBuffer_pitch + x, nextBufRow;
        CLR *scr = (CLR*)screen + scr_pitch*y + x;
        for(int i=0; i<4; i++)
            unsigned int rows =
            for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW_invStartShift(15,shift,240,4-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(3840,8-shift,61440,12-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(983040,16-shift,15728640,20-shift,+,-);
                for(int j=0; j<factor; j++)
                DRAW_SPLAT_ROW(251658240,24-shift,4026531840,28-shift,+,-);
}
}
}
}
}
}
}
}

/* Draw square Splat with size (scale x scale) and update z-buffer. */

template <class CLR>
inline void DrawSquareSplat(int x, int y, float z, CLR colour, float scale)
{
    //cast scale to an int
    int nScale;
    _asm
    {
        FLD scale
        PISTP nScale
    }
    //create a square (scale x scale) splat
    int scrIndex = y*scr_pitch + x;
    int zBufIndex = y*zBuffer_pitch + x;
    int scrYStep = scr_pitch - nScale;
    int zBufYStep = zBuffer_pitch - nScale;
    int scrYEnd = scrIndex + scr_pitch*nScale;
```
while(scrIndex < scrYEnd)
{
    int scrXEnd = scrIndex + nScale;
    while(scrIndex < scrXEnd)
    {
        if(zBuffer[zBufIndex] < z)
        {
            ((CLR*)screen)[scrIndex] = colour;
            zBuffer[zBufIndex] = z;
        }
        scrIndex++;
        zBufIndex++;
    }
    scrIndex += scrYStep;
    zBufIndex += zBufYStep;
}
}

**Background**

The code to generate the background geometry is as follows:

```c
//number of points in the background object, generated with normalTable code
#define BACKGROUND_REFINEMENT_DEPTH 8
#define BACKGROUND_POINT_COUNT (1 << (3 + 2*BACKGROUND_REFINEMENT_DEPTH))

typedef struct _background
{
    int normal_count; //number of points on the sphere of this background
    unsigned short * colours;//background pixel colour at respective normal
    POINT_GRID * grid; //a point grid object to render the background
    POINT3D position; //translation of background in world-space
} _background;

~_background() //destructor
{
    if(this->colours) delete this->colours;
    if(this->grid) delete this->grid;
}

/* Generate the background geometry - a sphere that encompasses obj
* with an image mapped to it and a plane at the base level of obj.
*/
BACKGROUND * Background_Generate(char * imageFilename, Object * obj)
{

    //attempt to load the background image
    BITMAP * bmp = LoadBmpFile(imageFilename);
    if( ! bmp)
    {
        return (BACKGROUND *)0;
    }

    //get point count for background
    float scale = 2.0f * obj->size;
    int half_plane_width = (backgroundHasPlane) ? 100 : 0;
    int point_count=BACKGROUND_POINT_COUNT+4*half_plane_width*half_plane_width;

    //create new background object that this method will return
    BACKGROUND * newBG = new BACKGROUND;
    newBG->normal_count = BACKGROUND_POINT_COUNT;
    newBG->colours = new unsigned short[point_count];
    VECTOR * normals = new VECTOR[newBG->normal_count];

    //generate normals with normal lookup table function
    GenerateNormalTableIn(normals,sizeof(VECTOR), BACKGROUND_REFINEMENT_DEPTH);

    //assign colours by pointing normal at background image
    for(int i=0; i<newBG->normal_count; i++)
    {  
        VECTOR v;
        Vector_Copy(&normals[i], &v);
        int bmp_x,bmp_y, r,g,b;
```
float x_angle = (PI - atan2f(v.x,v.z)) / PI2;
bmp_x = (int)(x_angle * (float)(bmp->width-1));
bmp_y = (int)((v.y+1.0f) * 0.5f*(float)(bmp->height-1));

int bmp_index = (bmp_y*bmp->width + bmp_x)*(bmp->bpp >> 3);
b = (int)bmp->raster[bmp_index++];
g = (int)bmp->raster[bmp_index++];
r = (int)bmp->raster[bmp_index];
newBG->colours[i] = RGB16BIT565(r,g,b);
}
delete bmp;

//generate point grid with scale*normals for points
PLY_DATA * ply = new PLY_DATA;
ply->point_count = point_count;
ply->points = new POINT3D[point_count];
ply->normals = new VECTOR[point_count];
ply->colours = newBG->colours;
ply->discRadii = new float[point_count];
VECTOR diff;
Vector_Sub(normals[0],normals[1],diff);
Scalar_Vector_MultiplyTo(scale,diff);
float discRad = 1.5f * Vector_Norm(diff);
ply->material = AddMaterial();
ply->material->ka = 255; ply->material->kd = 0; ply->material->ks = 0;
ply->material->usePointClr = 1;
for(int i=0; i<newBG->normal_count; i++)
{
    Scalar_Vector_Multiply(scale, normals[i], ply->points[i]);
    Scalar_Vector_Multiply(-1.0f, normals[i], ply->normals[i]);
    ply->discRadii[i] = discRad;
}

//set points for plane for the object to sit on
discRad = Vector_Norm(diff) * 0.9f;

for(int y=1-half_plane_width; y<half_plane_width; y++)
for(int x=1-half_plane_width; x<half_plane_width; x++)
{
    int i = newBG->normal_count+(y+half_plane_width-1)*2*half_plane_width+x;
    ply->points[i].Set((float)x*discRad, plane_y, (float)y*discRad);
    Vector_Copy(&up, &ply->normals[i]);
    ply->discRadii[i] = discRad;
    if(haveTexture)
    {
        POINT3D p;
        p.Set(ply->points[i].x, ply->points[i].z, ply->points[i].y);
        ply->colours[i] = GetTextureColour(&p,wood, textureStep);
    }
    else
    ply->colours[i] = plane_clr;
}

VECTOR min,max;
min.Set(-scale,-scale,-scale);
max.Set(scale, scale, scale);
newBG->grid = GeneratePointGrid(ply,VECTOR_ZERO,min,max,1,MAX_GRID_DEPTH);
ply->colours = (unsigned short *)0;
delete ply;

//set material of plane
MATERIAL * planeMat = AddMaterial();
planeMat->ka = 75; planeMat->kd = 100; planeMat->ks = 80;
planeMat->power = 1.5f;
planeMat->usePointClr = 1;
for(int i=0; i<newBG->grid->attrLen; i++)
{
    if(newBG->grid->attrData[i].normal == NORMAL_UP_INDEX)
    {
        newBG->grid->attrData[i].material = planeMat->index;
    }
}
Appendix

Shadows

The code for shadow generation is as follows:

```c
//declare shadow buffer as twice the resolution of the screen
#define SHADOW_BUFFER_WIDTH (WINDOW_WIDTH << 1)
#define SHADOW_BUFFER_HEIGHT (WINDOW_HEIGHT << 1)

/* Flag each point in scene as being 'in shadow' or 'out of shadow' as appropriate for each light.*/
void CalculateShadows(Object * obj, BACKGROUND * background)
{
    //initialise dummy screen buffer
    POINT_ATTR ** dummyScreen = (POINT_ATTR **)(new int[SHADOW_BUFFER_WIDTH * SHADOW_BUFFER_HEIGHT]);
    SCREEN_BUFFER dummyScreenBuffer;
    SetupScreenBuffer(dummyScreen,SHADOW_BUFFER_WIDTH,SHADOW_BUFFER_HEIGHT,&dummyScreenBuffer);

    //initialise dummy 1/z-buffer
    float* dummyZbuffer_array=new float[SHADOW_BUFFER_WIDTH*SHADOW_BUFFER_HEIGHT];
    SCREEN_BUFFER dummyZBuffer;
    SetupScreenBuffer(dummyZbuffer_array,SHADOW_BUFFER_WIDTH,SHADOW_BUFFER_HEIGHT,&dummyZBuffer);
    float * ptr = dummyZbuffer_array + (dummyZBuffer.height >> 1)*dummyZBuffer.width + (dummyZBuffer.width >> 1);
    dummyZBuffer.origin = ptr;

    //set all points to 'in-shadow'
    for(int i=0; i<8; i++)
    {
        int attr_count = obj->grids[i]->attrLen;
        POINT_ATTR * attr_array = obj->grids[i]->attrData;
        for(int j=0; j<attr_count; j++)
        {
            attr_array[j].inShadow = 255;
        }
    }

    //set new disc radii to smallest to minimise aliasing
    int oldRad=0, newRad = GetDiscIndex(0.0f);
    if(background)
    {
        for(int i=0; i<background->grid->attrLen; i++)
        {
            //set plane point to 'in shadow'
            if(background->grid->attrData[i].normal == NORMAL_UP_INDEX)
            {
                if(oldRad==0) //backup old disc rad of plane points
                oldRad = background->grid->attrData[i].discRad;
                background->grid->attrData[i].inShadow = 255;
            }
            background->grid->attrData[i].discRad = newRad;
        }
    }

    //foreach light source
    for(int lightIndex=0; lightIndex<light_count; lightIndex++)
    {
        //perform multiple passes to reduce aliasing
        for(int u=0; u<2; u++)
        {
            for(int v=0; v<2; v++)
            {
                for(int w=0; w<2; w++)
                {
                }
            }
        }
    }
}
```
Appendix

// clear screen, depth and origin buffers
ClearScreenBuffer(dummyScreenBuffer);
ClearScreenBuffer(dummyZBuffer);

// initialise camera (from lightsource)
// add small perturbation to camera direction for each pass
VECTOR dir;
Vector_Copy(&LightSources[lightIndex].direction, &dir);
float dist = 2.5f * obj->size;
dir.x *= dist;
dir.y *= dist;
dir.z *= dist;
shadowCam.Initialise(obj, &dir, &dummyScreenBuffer);
shadowCam.position.x *= (1.0f + u*0.1f);
shadowCam.position.y *= (1.0f + v*0.1f);
shadowCam.position.z *= (1.0f + w*0.1f);
shadowCam.UpdateTransforms();

// performing backface culling precalculations
SetCulledNormals(&shadowCam);

// render background to shadow buffer
if(background)
{
    Render_PointGrid(background->grid, &shadowCam,
        background->position, &dummyScreenBuffer, &dummyZBuffer, 1);
}

// for each sub grid
for(int i=0; i<8; i++)
{
    // render to dummyScreen, writing attribute indexes
    Render_PointGrid(obj->grids[i], &shadowCam, obj->position, &dummyScreenBuffer, &dummyZBuffer, 1);
}

// set flag of each attribute of each point in shadow buffer
for(int i=0; i<dummyScreenBuffer.size; i++)
{
    if(dummyScreen[i])
    {
        unsigned int mask = (1 << lightIndex);
        // set point attribute to not in shadow
        if((dummyScreen[i]->inShadow & mask) == mask)
        {
            dummyScreen[i]->inShadow -= mask;
        }
    }
}

// restore disc radii of background
if(background)
{
    for(int i=0; i<background->grid->attrLen; i++)
    {
        background->grid->attrData[i].discRad = oldRad;
    }
}

// free memory
delete dummyScreen;
delete dummyZbuffer_array;