



D 8.4. Variable Quality Rendering Techniques

Project Number:	ACTS Project N. AC040
Project Title:	COVEN - Collaborative virtual environments
Deliverable Type:	P*

CEC Deliverable Number:	A040-UCL-CS-DS-P-084.b1
Contractual Date of Delivery to the CEC:	January 1 st , 1998
Actual Date of Delivery to the CEC:	
Title of Deliverable:	Variable Quality Rendering Techniques
Workpackage contributing to the Deliverable:	WP8
Nature of the Deliverable:	R**
Author(s):	UCL, SICS

Abstract: One important factor for achieving realism in VR is the display of images at a high and steady rate. As the models we are developing for the COVEN scenarios grow larger this becomes an increasingly difficult task. In this report we present some of the methods that were developed and implemented on the COVEN platform for accelerating the rendering of large models.

Keyword list: rendering, real-time, view volume culling, level of detail, visibility culling.

*Type: P-public, R-restricted, L-limited, I-internal **Nature: P-Prototype, R-Report, S-Specification, T-Tool, O-Other

1. Introduction	4
2. Frame Rate Control Module	5
2.1. DIVE Summary	5
2.2. Overview of the Rendering Module.....	5
2.2.1. Queue	6
2.2.2. Render Loop	6
2.2.3. Frame Optimiser	6
2.3. Initial Version (Level of Detail Control)	7
3. Scaleable Geometry Primitives in DIVE	9
3.1. Introduction	9
3.2. Traditional High-Level Primitives.....	9
3.3. Point Sets	10
3.4. Level of Detail	10
3.5. Billboards	12
3.6. Switches	13
3.7. Animated Textures	14
3.8. Some Examples	14
4. Probabilistic View Volume Culling.....	16
4.1. Statistical Object Representation	17
4.2. Concepts of the Culling Algorithm.....	19
4.2.1. The Basic Algorithm	19
4.2.2. Algorithm	20
4.3. Implementation Heuristics	21
4.4. A BSP Tree Implementation.....	22
4.5. Results.....	23

5. Fast Approximate Quantitative Visibility for Complex Scenes	28
5.1. Discrete visibility	28
5.2. Depth Tree.....	30
5.2.1. The construction of the depth tree	32
5.2.2. Retrieving the visibility	33
5.3. Results.....	33
5.4. Conclusion.....	35
6. Conclusion.....	37
7. References	38

1. Introduction

One of the main factors for presence in virtual environments is the frame rate. Ideally it has to be close to video frame rate and it should be kept fairly constant at all times. In terms of rendering, this means that the geometry passed to the graphics hardware should never exceed what it is capable of rendering within the desired frame time.

One way of achieving this is to have a rendering system that can modify what is being rendered and at what precision in order to keep up the frame rate. For example, when the graphic system is “stressed”, lower levels of detail (LODs) could be used or “insignificant” objects could be omitted. We would like to implement such a system on the COVEN platform. In the next section we describe our first design and a very basic implementation where the system will only be in charge of the LOD selection. For this we have ported a continuous LOD algorithm to DIVE. In the work done here we use DIVE as our platform for developing the techniques to go in the final COVEN platform.

In Section 3, we describe a number of tried and tested rendering acceleration techniques which were added by SICS to the DIVE system. Their implication on networking and multi-user issues is discussed.

In Sections 4 and 5, we present two new algorithms developed for COVEN. These are for view volume and occlusion culling respectively and they differ from previous methods in that their run-time cost can be varied depending on the stress of the system by reducing their precision.

2. Frame Rate Control Module

2.1. DIVE Summary

DIVE is an experimental platform for the development of virtual environments, user interfaces and applications based on shared 3D synthetic environments. DIVE is especially tuned to multi-user applications, where several networked participants interact over an internet.

DIVE is based on a peer-to-peer approach with no centralised server, where peers communicate by reliable and non-reliable multicast, based on IP multicast. Conceptually, all peers share a common state that can be seen as a memory shared over a network. Processes interact by making concurrent accesses to the memory, which is in reality replicated at each site. The DIVE architecture is described in [Carls93a], [CARLS93B] and [HAGSAND96]

The DIVE architecture is centred around a hierarchical database of what is called *entities*. DIVE entities can be compared to objects in object-oriented approaches, although DIVE is written in plain ANSI C. The database contains the representation of the scene. However, DIVE entities hold much more than only graphical information. In particular, DIVE entities can enclose user-defined data and autonomous behaviour descriptions.

2.2. Overview of the Rendering Module

The rendering module will run in parallel to the rest of the system. On start-up, there will be an initialisation step where some extra information is created on the model. This information will be used later to speed up individual frames. Dive is responsible for reading the input, building the object hierarchy and dealing with user interactions. The module shares the memory with the rest of the system. It reads the object hierarchy (along with the graphics context and the camera parameters) from the common database and renders it. It also needs to be notified separately about any creation/modification events to the database. This is necessary because at initialisation it generates some extra data structures, such as the cell partitioning graph or octrees for continuous LOD support.

Note that the module does not generate it's own copy of the hierarchy but rather it adds any object specific information to the hierarchy shared by the rest of the system. If this is part of a multi-user system then the rendering module of each user is responsible for generating and maintaining the extra information locally. This gives the flexibility of adapting the graphical information generated to the capabilities of the user's machine. For example when running on a slower machine, it might choose not to construct and maintain any data structures for improved illumination.

The overall structure and flow of information is shown in Figure 1. It consists of three separate threads, initiated on start up, and a local queue.

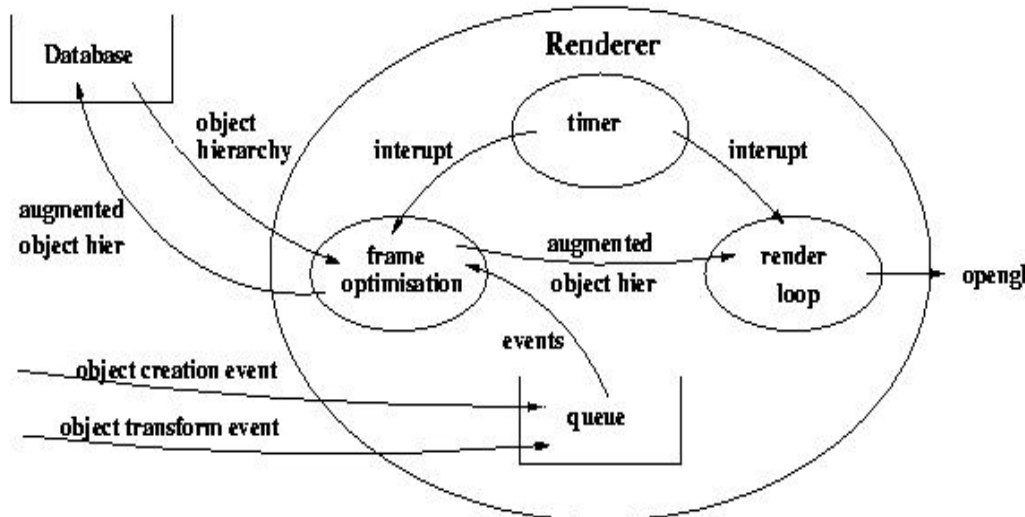


Figure 1 Description of the rendering module

2.2.1. Queue

On initialisation the module will register callbacks showing interest in the creation or modification of any part of the geometric database. All such events received will be accumulated in the *queue* (see Figure 1). We need to know these events so as to update the module's data structures. However, in an attempt to limit the amount of processing to only the relevant parts of the geometry, these events are not be applied immediately. When the frame optimiser decides that an object is relevant for the current frame, then the events for that object are taken out of the queue and applied. Otherwise they remain in the queue grouped together per-object. (An example of an operation that should be postponed is the update of a BSP tree. If an object, which is not visible, is moving, then we keep it in the same position in the BSP and only delete and re-insert it when it becomes visible.) Note that a *world creation* event is added to the queue just like all other events.

2.2.2. Render Loop

This loop does nothing more than feed the graphics hardware with the objects specified by the frame optimisation, at the level of detail specified. It takes as input the current camera parameters, the graphics context and the geometry and renders the image.

2.2.3. Frame Optimiser

The frame optimiser reads the object hierarchy and camera parameters and process the hierarchy to select a suitable part of the geometry that is small enough to render within the specified frame period and yet provides maximum realism. This is where all the available acceleration techniques, such as approximate culling, LOD, image based methods etc, are brought together. It is also responsible for reading the queue for event in order to update the relevant data structures of the visible objects.

A special case is the world creation event. Such an event will only be generated when the system is starting up (or when we migrate to a new world). Here we can afford to take the extra time to process all objects and create the relevant data structures for saving time later. All the pre-processing for the illumination and rendering is computed by this process.

2.3. Initial Version (Level of Detail Control)

Level of Detail (LOD) Control is a technique which has long been used to reduce complexity of areas of a scene which require it least. This reduction of complexity can result in several benefits:

- Higher frame rates due to reduced complexity
- Frame rate consistency due to complexity control
- Redistribution of detail to areas which require it more

An LOD algorithm can typically be separated into three modules:

- Off line pre-processor LOD model provider
- Run-time LOD model provider
- LOD Controller

The exact functions which (a) and (b) perform are dependent on the type of multi-resolution LOD model provision being used. The first of these modules performs any off line processing, such as data structures for the Run-time LOD model provider. Many systems generate a small number of discrete LODs off line, which are used by a caching scheme implemented by the Run-time model provider. Various other systems generate data structures off line which calculate differencing information to morph the lowest resolution model into the next higher resolution, and so on to the highest.

The LOD Controller has the task of distributing complexity to objects in the scene. It must make a global assessment, deciding which objects are more visually and semantically more important than others, to distribute the complexity well. It takes into account the static visual importance of objects, e.g. based on surface curvature and local scale, and combines this with dynamic visual importance metrics, such as size under a perspective projection (distance & 3D size), angular velocity, and position in periphery. Task specific semantic importance may bias importance to objects being examined or interacted with. These object specific metrics must be combined and used by the controller to provide global LOD control. Very little work has currently been done in the field of LOD controllers.

LOD model providers are generally categorised in two ways. Methods which analyse an original object to produce a small number of fixed resolutions are termed "Discrete" LOD systems. The differences between such models can be quite noticeable as they can be arbitrarily different geometrically. A phenomenon known as "Popping" can be seen when transitions are made between models. If error bounds are known, the LOD controller can include terms to compensate for this. Other category is "Continuous" providers. These tend to make smaller changes to objects to provide more resolutions, such as edge collapse operations. Because changes are small, the overhead of switching LODs is temporally more even, permitting more approximation to be performed by the Run-time model provider module. An object at one resolution can be altered to another resolution and linearly morphed between the two states, resulting in a truly continuous method, which minimises the popping effect.

The technique implemented by the VR Group at UCL lies somewhere between the discrete and continuous forms, providing about 40 resolutions for a typical object containing about 2500 triangles. It can decompose and reconstruct an object at runtime. The off line provider

module constructs a dynamically allocated Octree of the object's vertices. The triangulation is restructured such that a set of triangles is associated with each vertex, based on connectivity. A compact runtime data set is then generated for use by the run-time module. It contains a compressed version of the Octree structure, which does not rely on dynamic allocation for nodes. It resides in a single memory allocation. In the runtime data set, an array of objects stores the vertex related data for each level of the Octree. Each of these objects contains scalar attributes such as position, mean normal vector and colour. An array of indices to the start of each Octree level's data is also kept.

The Octree structure is traversed breadth first, and the groupings specified by the branching are used to dictate which vertices are approximated with which other neighbours. A parent volume containing at most eight child volumes is collapsed in three stages, one for each axis forming an eight to four to two to one sequence of sub volumes in the parent. At any time, only one vertex can exist in these child nodes. The data associated with the vertices are approximated using an averaging function. The differencing data is also stored to permit full reconstruction. The total amount of vertex related data in memory is constant. This process is similar to a Wavelet theory using a Haar mother Wavelet.

When vertices merge, the sets which represent the surface connectivity are also grouped together into another set, which represents the set for the approximated vertex which now represents the approximated vertices. This vertex set method is memory efficient and reversible for reconstruction. At any time, the atomic, low level sets can be retrieved from the grouped data structure. Again, this method requires no dynamic allocation.

The triangulation for a resolution is reconstructed by evaluating which triangle indices are in the intersection of all atomic sets in the set for an approximated vertex. This can be performed in $O(n)$ time where n is $P*V$ where P is the number of triangles, and V is the number of vertices. by attempting to reconstruct the original triangulation by entering vertex indices into triangle data structures. If a vertex index is entered more than once into a triangle definition, the triangle can not exist in the current resolution, as the descendant vertex is an approximation of at least two vertices which defined the triangle originally. The triangle is marked as obsolete, and is not rendered.

The system appears to work well on scenes with many objects of about 2000 polygons on system with a high CPU / 3DPipeline speed ratio such as the Sun Ultra 1. On systems with a lower ratio such as the SGI Onyx, the system is not efficient. The main problem is that whole Octree levels are being approximated at once. The system could be augmented to approximate only parts of an Octree level at a time, spreading the CPU burden over time and reducing the popping effect.

3. Scaleable Geometry Primitives in DIVE

3.1. Introduction

Most of the rendering graphic engines used for presenting three-dimensional information to users in current Virtual Reality (VR) platforms are only able to handle polygon meshes. These engines repetitively project the polygons onto the visualising plane, frame after frame. The performance of the graphic engines is commonly limited by two different factors:

- *the pixel-fill rate*, i.e. the number of pixels representing all the projected polygons and the final image that have to be transferred from the rendering module to the rendering window.
- *the geometric transformation rate*, i.e. the number of geometric transformations that have to be performed to represent the whole scene. Typically, these transformations will involve matrix operations to determine the final colour and position of the different pixels corresponding to a polygon.

These sections present a few techniques used in Distributed Interactive Virtual Environment (DIVE) in order to increase the frame rate and thus achieve geometry scalability. Most of these techniques have long been used in regular three-dimensional toolkits. We will focus on their implications on networking and multi-user issues.

In general, the techniques described here deal with some specific geometry primitives. These primitives aim at rendering simplicity while holding visual complexity, in order to keep the *illusion* that the visualised scene is complex and contains many details. Many primitives are exploited at their best with texture capable rendering engines.

3.2. Traditional High-Level Primitives

Using traditional high-level primitives such as spheres, boxes, cylinders, ellipses, cones, elevation grids, texts, allows DIVE to use more optimised algorithms for rendering, collision and intersection. Less usual primitives are detailed more thoroughly in sections 3.4 to 3.7.

DIVE is able to decide a way of rendering each high-level primitive depending on the distance between the viewer and the object, reducing at the same time the number of polygons needed to achieve a visually acceptable result.

For example, a cylinder can be visualised as an extruded hexagon (described using only 12 vertices and 8 polygons) when its size on the screen is sufficiently small. When its size on the screen is bigger, more polygons and vertices will be needed to give the impression of a smooth and round object. Figure 2 shows a detailed cylinder to the left and a simplified cylinder to the right.

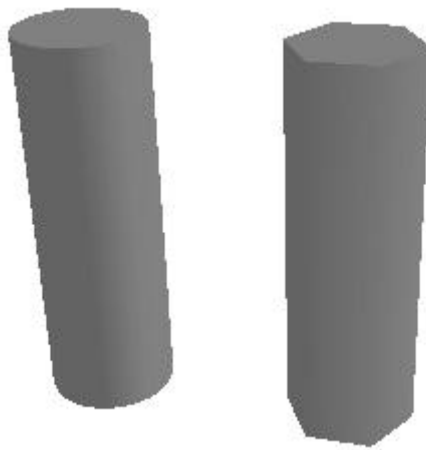


Figure 2: Two high-level primitives (cylinders) shown at two different detailing levels. The cylinder to the right is normally rendered using so few polygons only from a distance; it is brought close to the viewer for clarification purposes.

Many modelling packages are not able to output files that effectively contain high-level primitives. Instead, the files produced contain polygon meshes which overall shape is the same as the high-level primitives. However, DIVE will not be able to aggregate the polygons back into the original shape, thus, other techniques will have to be used. Usually, these techniques utilise more computational resources and are less accurate, as sketched in section 3.4.

3.3. Point Sets

Point Sets are a concise way of representing polygonal-based information. Point sets use indirection into sets of vertices, normals and texture co-ordinates to build their composing faces, without duplicating the vertices. Indirection is achieved using sets of indices that refer to the component sets.

Point sets reduce the data size, thus decreasing network distribution of new objects. Furthermore, point sets ease deformation techniques by allowing the programmer to modify vertices without modifying the polygon formation.

3.4. Level of Detail

Level Of Details (LOD) are alternative representations of the same logical object, each with a different level of detail, i.e. usually with less polygons, smaller textures, simpler primitives, etc. There are two different types of LOD in DIVE:

- **Distance LOD** — These are the most common LOD and DIVE will choose the appropriate representation of the logical object depending on the distance that separates the viewer from the object. Distance LOD are generally used to simplify the representation of an object as seen from a distance. An example of these LOD is depicted in Figure 3. It is often a good idea to use high-level primitives for the less detailed LOD, since DIVE will provide rendering optimisations for these primitives, as described in section 3.2. In some cases, the “worst” LOD can even be an empty representation.

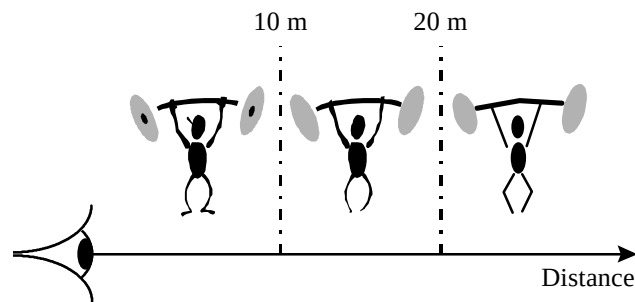


Figure 3: An example of distance LOD. The figure is progressively simplified as its distance to the viewer increases. Note that the last representation only uses four ellipses and five (poly)lines.

- Angular LOD — DIVE will choose the logical object representation depending on the angle between the object's orientation axis and the viewer. An example of angular LOD is depicted in Figure 4.

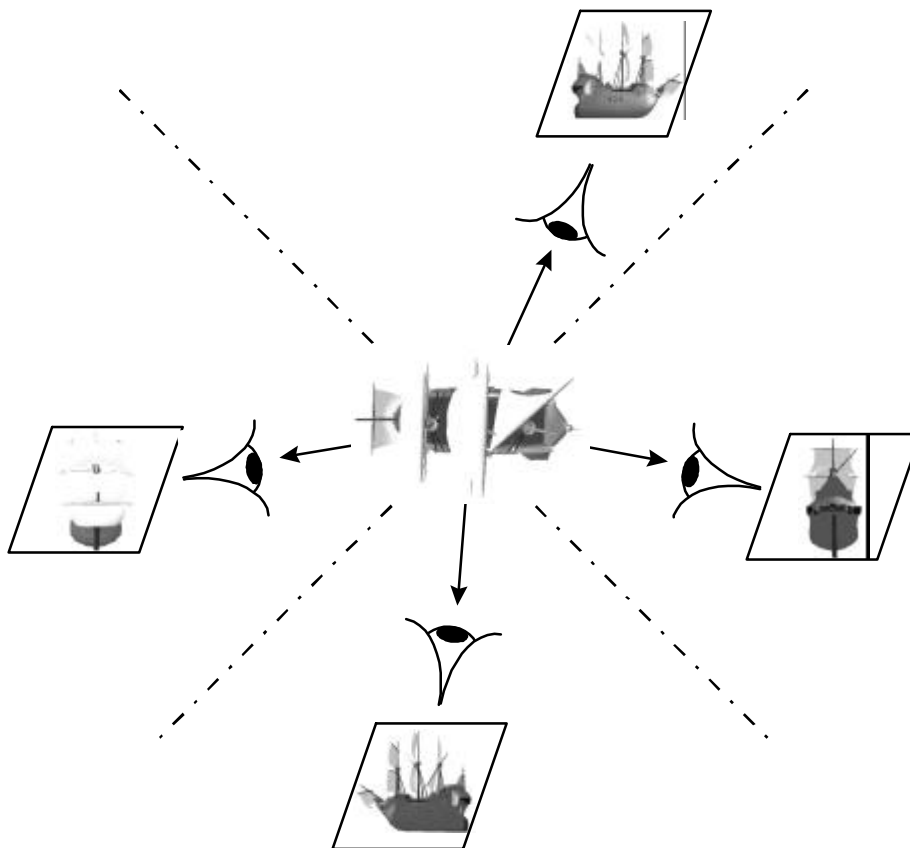


Figure 4: An example of angular LOD. The boat is represented differently depending on the quadrant in which the user is situated.

Some authoring tools provide automatic LOD generators that produce different representation with decreased polygon count and complexity. Figure 5 presents an example of such an automatic generation: The pirogue at the top is the original object. Under it, close to the user, is one of its LOD, zoomed and brought close to the viewer in order to show the

reduction of polygon count. Finally, in between both pirogues is the reduced version from a distance, as a user would effectively see it.



Figure 5: A three-dimensional example of automatic LOD generation.

Using LOD in a multi-user VR system such as DIVE raises different problems that do not necessarily occur in a single-user system:

- DIVE allows any application or user to modify (almost) any entity of the database. Problems occur with objects that have different LOD representations. Indeed, the different representations have no logical connections between each other. Thus, an application that is willing to modify the appearance of such an object will have to modify *all* the different representations separately. Automatic LOD generation from a single original mesh would be a more adequate solution since it would store a single logical object. Smooth and automatic LOD are discussed in [CERTAIN96], [HOPPE96] and [SCHMALSTIEG97].
- Ray casting¹ raises similar problems. Indeed, LOD are user dependent and sending a ray in direction of an object that has different LOD will hit a different representation depending on *who* sends the ray. DIVE has chosen to restrict itself to the first representation of distance LOD, considered to be the most detailed, most accurate and the “real” representation of the logical object.
- LOD have network implications as well, because of the bandwidth needed to transfer all the different representations of the same object.

3.5. Billboards

A *billboard* is an object that automatically rotates around an axis or a point in order to always face the user. The principles behind billboards are depicted in Figure 6. One particularity of DIVE billboards is that they do not need to lie in a single plane. Billboards can have some depth information when necessary. DIVE billboards are rotated locally at each peer in order to face each remote user separately.

¹ Ray casting is a technique that consist in sending a ray from a given three-dimensional point in a given direction and in gathering the different objects that were hit and the target positions. Ray casting has many applications such as ground following, gravity, object collision, user interaction with mice or three-dimensional pointing devices, etc.

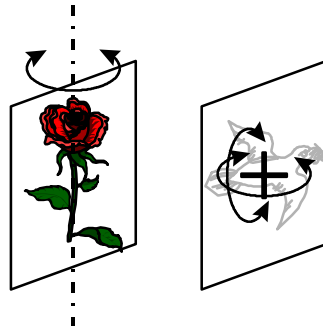


Figure 6: Two billboard examples with planes. The rose to the left rotates around its vertical axis, while the bird to the right rotates around its centre point.

Billboards are often used together with transparent textures in order to represent complex objects that would have requested many polygons in order to achieve a result that is equivalent to a picture. Usually, billboards are utilised for enhancing the background scene. Examples of billboards include plants and animals, as in Figure 7.



Figure 7: Some billboards representing trees in a textured environment.

In a multi-user environment like DIVE, billboards suffer from ray casting problems similar to those raised by LOD. However, because a billboard is represented by a unique physical object in the shared database, only the intersection points of ray casting differs depending on the site, and thus the user, from which the ray was sent. While this raises semantic problems for the content of interaction events, which contain the intersection location, it is considered of minor importance.

3.6. Switches

A *switch* is an object that chooses one or none of its sub objects to be used for rendering, intersection, collision and interaction. The initial idea behind switches is to offer a support for the techniques that are used in cartoon animation, i.e. if the different sub children of a switch represent a similar group of objects that differ slightly from one to the other and if the value of the switch is changed sufficiently rapidly, the succession of “images”, or in the VR case, geometry descriptions will look like if it was animated. This is depicted in Figure 8. Switches have other applications such as presenting multiple choices to the users.

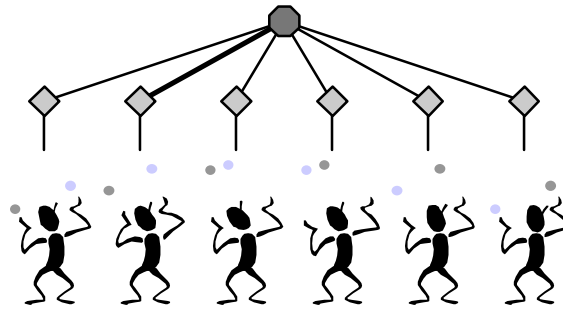


Figure 8: An example of a switch: the switch entity is represented by a hexagon and each sub branch of the hierarchy is introduced by a diamond. The current value of the switch is “one” and the item rendered is the second figure. While changing the switch value rapidly from “0” to “5”, the different figures will be shown and a juggling character will be “animated”.

Switches are especially useful when the animation to be performed involves slight changes from one key frame to the other. Typically, when such animation is implemented, each position and rotation modification generates a network message. This can result in high bandwidth use and thus do not scale to more than a few simultaneous peers. Switches can be used to hard-code the different steps of the animation. As a result, one network message — the switch value modification — will replace many network messages — all position and rotation modifications that would be necessary to jump from one key frame to the next one.

Though they can improve network efficiency when *running* an animation, switches can be a network burden when *initialising* an animation. Indeed, if there are n key frames in the animation, there will be $n - 1$ times more objects than if the animation had been coded using regular position and rotation modifications. Indeed, all the objects necessary to the animation are copied and positioned and rotated in advance. Transferring the different sub branches of a switch in-between peers can have serious implications on bandwidth use.

Hopefully, since switch values are coherent and identical throughout all the different connected peers, switches do not face the same semantic problems as the ones described for LOD in section 3.4.

3.7. Animated Textures

Animated textures are based on bitmap formats that, instead of containing a single image contain a series of images that constitute a simple and short film. An example of such a format is the second version of the Graphics Interchange Format (GIF) [Gif89]. Texture animation can be used to give life to the background scene, such as animating a cloudy sky, the tides on the sea, water that streams, etc.

Texture animation is performed locally to each peer, without any synchronisation. Thus, these textures do not increase the network load. However, the lack of synchronisation might have an influence on the multi-user aspects.

3.8. Some Examples

This section aims at showing how the different primitives and techniques described above can be combined in order to achieve appealing and realistic results:

- A technique to simplify complex objects which only have a few openings to the outer world, e.g. houses with windows, is to replace them from a distance by a LOD constructed using the following trick: In the simplified object, all internal objects are removed and the openings are replaced by opaque polygons on which a texture with a snapshot of what should normally be seen through the opening. This technique is effective on texture workstations; however, it totally hides the content of the complex object to the outer participants and thus users, which are navigating and interacting within the object will be invisible to far avatars.
- While billboards give realistic results for symmetrical objects, they can not represent asymmetrical objects. However, this problem can be solved by combining billboards with several angular LOD. The quality of the illusion increases with the number of sub representations of the LOD.
- The previous trick can be enhanced using switches or animated textures to perform some crude animation. This structure is adequate to represent three-dimensional sprites.
- Another use of angular LOD is by totally switching off the representation of an object that is totally occluded by another object, typically a wall, under some angles. However, no object may be moved, unless the angular LOD values are recalculated.

4. Probabilistic View Volume Culling

For any frame in an interactive walkthrough of a dynamically changing scene, the set of objects forming the scene data base is partitioned into three subsets: (I) those objects that are entirely inside the current view volume, (O) those that are completely outside, and (B) those that are intersected by at least one polygon belonging to the boundary of the view volume.

The aim of view volume (VV) culling is to efficiently identify these sets, O, I and B. Objects in set I may be rendered (without the necessity of clipping), those in B are rendered but require clipping, and those in C may be ignored. Clipping is an expensive operation, but in practice for a large scene, only a small fraction of the total number of objects actually intersect the boundary of the view volume at any frame therefore requiring clipping. Blinn (Blinn91) remarked that clipping seems to be a wasted operation most of the time - having no visible effect on the rendered image; for the vast majority of objects, clipping will either determine that they needn't have been "clipped" at all (I), or remove them altogether (O). One of the advantages of VV culling is that it enables targeting of the clipping process toward only those objects that need to be clipped.

This partition of the scene data base changes dynamically for two reasons: the view frustum changes (translates, rotates), and objects move. However, in most circumstances there will be coherence from frame to frame based on spatial and temporal locality. Spatial locality implies that whichever of the three sets to which an object belongs in a particular frame, its neighbours are likely to belong to that same set. Temporal locality implies that whichever set an object is in during this frame, it is likely to be in the same set in the next frame. Since the sets correspond to regions of space relative to the VV, another way of expressing these localities is that objects tend to be in the same regions of space as their neighbours (obvious) and that from frame to frame they tend to stay in the same region of space. If we consider the region of space corresponding to the set O, we can also consider the partitioning of this space into subspaces which are classified according to their "distance" from the VV. The principles of temporal and spatial locality will apply also within these subspaces.

An ideal VV culling algorithm would determine the partition of the objects into these sets efficiently, based on temporal and spatial locality, and during the rendering process objects in O would not be touched at all. Avoiding references to objects in O could be important, since for a very large data base objects in O may be stored on disc, so that accessing them may be extremely expensive. Of course, however, it is not possible to avoid all references to O, since the set membership would change over time thus requiring an iteration through the data base in order to check for possible changes of allocation of objects to the sets.

One way to avoid an iteration through all objects is by exploiting hierarchy (Clark76; Rohlf94). For example, with hierarchical bounding volumes, as soon as any volume is determined as being wholly in I or O, then all of its children must be in the same set. The problems here are the overheads in building the hierarchy, and more importantly, of maintaining it as the scene itself changes through time.

An alternative approach is to subdivide world space into a uniform grid of voxels, with each voxel maintaining a list of identifiers of objects that intersect it. This method also has several drawbacks. It could require a lot of memory to store the grid and could be expensive since every voxel intersecting the view volume must be examined. Nevertheless, this approach has the characteristics of a genuine VV culling algorithm in that it can in principle

satisfy the fundamental requirement - to process only those objects inside or on the boundary of the view volume, and to not touch at all those that are outside the view volume. The problem with the space subdivision approach though is that it has undesirable characteristics from the point of view of implementation.

In the VV culling algorithm presented here, at any moment objects are classified into the three sets I, B and O, and the set O is partitioned into subsets $O_1, \dots, O_n$ based on the distance of their contained objects from the view frustum. The objects in set O_i are sampled in each frame, with probability of selection inversely proportional to i , and checked to examine if they have changed set. The probabilities are updated according to changes in position and orientation of the view frustum, so that when the view frustum is moving away from the region corresponding to a particular set, its corresponding sampling probabilities are reduced. Moreover we employ a statistical representation scheme for objects which allows for efficient computation of the relationship between objects and the planes forming the view frustum.

4.1. Statistical Object Representation

Objects are represented by their statistical properties, in particular their mean vector and covariance matrix. We adopt a paradigm that supposes that objects are in an ideal sense probability distributions over a vector-valued random variable (x,y,z) (which has, for example, a tri-variate Gaussian distribution). Samples from this particular distribution would be points in 3D space, and a large number of points would form an ellipsoid cluster with decreasing density towards the periphery. In this paradigm any region of space has an associated non-zero probability of containing a part of the object - so that theoretically, every object is distributed over the whole of space, though with greatest probability density corresponding to where the object actually “is” (in the normal sense). A 2D analogue is shown in Figure 9.

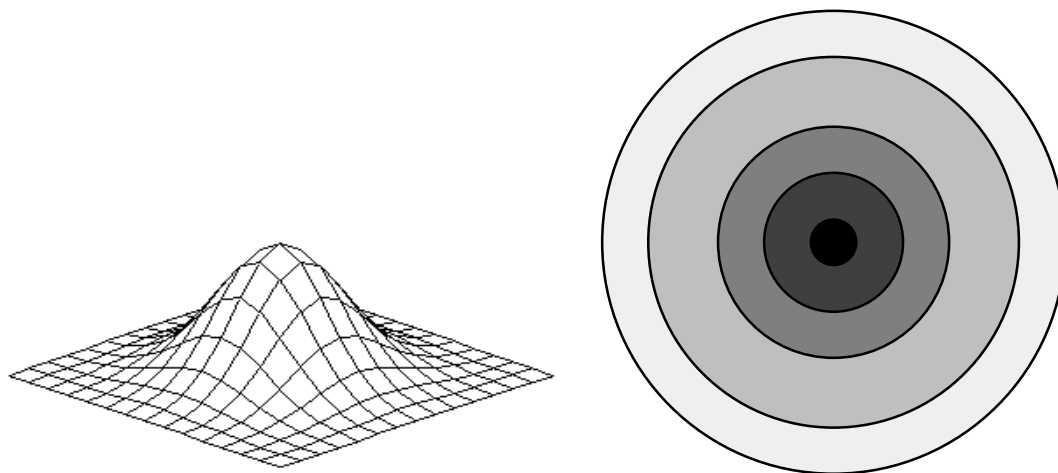


Figure 9 Left, Bivariate Gaussian distribution and right, samples on Bivariate Gaussian, observation density decreases towards the periphery

A probability distribution can be completely characterised by its moments, provided that these are all finite. In practice, the first two moments (the mean and variance for a scalar variable) are very informative, providing information about location, dispersion and correlation between x , y and z . Hence an object can be approximated by its mean vector

(location) and covariance matrix (dispersion and correlation). This idea has been used in (Gottschalk96) where the eigenvectors (or principle components) obtained from the covariance matrix are used to determine an object's oriented bounding box. In our algorithm the object is approximated by a bounding ellipsoid.

The following are some important relationships that we use in the construction of the algorithm. Let μ_p be the mean vector for the random variable p , and Σ_p the covariance matrix. The main diagonal of the covariance matrix consists of the variances of x , y and z respectively, and the off-diagonal terms the covariances. In the case of a Gaussian distribution, if all the covariances were zero and all the variances equal, then a large number of points sampled from this distribution would form a spherical shape.

Suppose that the object is transformed by an affine distribution, so that any point p becomes $q = pA + v$ where A is a 3×3 non-singular matrix and v is a translation vector. Then $\mu_q = \mu_p A + v$ and $\Sigma_q = A^T \Sigma_p A$, where A^T is the transpose of A .

Let $ax + by + cz + d = 0$ be a plane equation, with normal $n = (a, b, c)$. Then the perpendicular distance D of any point p to the plane is $(n^T p + d)/|n|$. If we consider p to be a random point on an object distribution, then the mean and variance of the distribution of the scalar *distance* D of that object from the plane is given by:

$$\mu_D = (n^T \mu_p + d)/|n|$$

$$\Sigma_D = n^T \Sigma_p n / |n|^2$$

A fundamental observation for our algorithm is that we can use this result to characterise the likely relationship of an object to the (typically) six planes of the view volume. Suppose that $(n^T p + d) = 0$ is the oriented equation of a clipping boundary, where normal n points in the "outside" (or "invisible") direction. Then if for an object we find $\mu_D > 0$ (the mean distance of this object from the plane is positive) this indicates that the object is likely to be outside of that clipping plane, and similarly, inside when $\mu_D < 0$.

In particular, if the distribution is taken to be Gaussian as mentioned above, then the distribution of D is also Gaussian, and approximately 95% of the distribution of D lies within bounds $\mu_D \pm 2\sigma_D$ where $\sigma_D = \sqrt{\Sigma_D}$.

Up to now we have not specified object representation. However, suppose as usual that our actual representation of objects is as polyhedra. Then we estimate the mean vector μ_p by taking the means of the vertices of the polyhedra, and the covariance matrix by forming the estimated covariance matrix:

$$S_{ij} = \Sigma \Sigma (u_i - m_i)(u_j - m_j) / N \quad (i, j = 1, 2, 3).$$

where N is the number of vertices, u_1 is the x -variate, u_2 is the y -variate, u_3 is the z -variate with m_1, m_2, m_3 being the corresponding estimated means.

Our paradigm inherently considers objects as solids, whereas the observations are only on the boundaries. Hence the estimation of the variances including Σ_D will be inflated. For the purposes of the algorithm we are interested in finding the clipping plane (if any) for which the (positive) mean distance is maximum. This corresponds to the clipping plane which the object is "most outside". (For example, the object could be inside all planes except for the

far and right clipping planes. In such a case four of the mean plane distances would be negative, and two positive. We are interested in finding the maximum positive distance, and the corresponding plane). For this purpose, instead of using μ_D , which does not take account of the possible variation in D , we use $\mu_D - k \cdot \sigma_D$ as the measure of distance (for some positive constant k - for example, in the ideal Gaussian case k would be 2 to capture 95% of the distribution). If $\mu_D > 0$ then obviously $\mu_D + k \cdot \sigma_D > 0$, so we do not need to consider this side of the interval. On the other hand if $\mu_D - k \cdot \sigma_D > 0$, then this is strong evidence that the entire object is outside of the clipping boundary, whereas if $\mu_D > 0$ but $\mu_D - k \cdot \sigma_D < 0$ we would take this as evidence that the object at least partly intersects the boundary.

To summarise this section:

- We “idealise” objects as tri-variate continuous (point-mass) probability distributions in 3D World Coordinate space.
- An object is characterised by its mean vector and covariance matrix, which provides information about location, dispersion and orientation (i.e., the covariances amongst x , y and z).
- The relationship between an object and the view volume is characterised by the plane if any which has the maximum distance from the object, measured by $\mu_D - k \cdot \sigma_D$ amongst those planes for which this value is positive.

We next use these results to present the algorithm.

4.2. Concepts of the Culling Algorithm

4.2.1. The Basic Algorithm

The algorithm maintains three data structures: (I) the set of identifiers of objects that are completely inside the view volume (and which therefore do not require clipping). (B) the set of identifiers of objects that intersect at least one boundary (and therefore require clipping).

The third data structure is a two dimensional array: for each of the six clipping boundaries we have an array of sets of identifiers of objects, where the members of a set have their “distances” (as defined above) within a certain interval. These intervals are constructed using a “width” (w) which would be chosen to be greater than the expected or typical distance of a camera move from frame to frame. Let $B[i,j]$ be the set of objects corresponding to plane i ($i = \text{Left, Right, Bottom, Top, Front, Back}$) at level j . The levels are characterised by $(j-1)w \leq \text{distance} < jw$ ($j = 1, 2, \dots, L$) where L is the number of levels. We also refer to the elements of this array as “cells”.

Now any object will either have been determined to be in the visible set (I), or in the boundary set, or at least one of its distances from the clipping planes is positive. In this case find the plane (i) for which this distance (d) is a maximum (Figure 10), and put the identifier for the object into the sequence $B[i,j]$ where $(j-1)w \leq d < jw$.

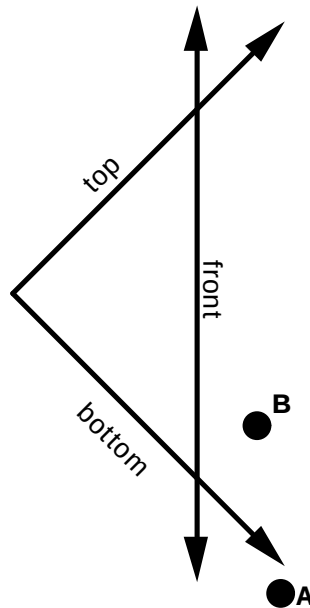


Figure 10 Points Outside the Clip Boundary. Point A is assigned to the bottom plane, and B to front.

Finally, associated with each $B[i,j]$ is a value $p_{ij} \in [0,1]$, representing the probability that an object in this set will be selected for checking to determine whether it has moved into another set (another plane, or another level). These probabilities are constructed so that within each plane i they are inversely proportional to j . A first overview of the algorithm is as follows:

4.2.2. Algorithm

Find the estimated mean vector and covariance matrix for all objects using the object vertices. This step is only carried out once, since if an object is transformed, the new mean and covariance matrix can be found by transformation.

Render all objects, recording which ones are actually clipped, forming the initial sets I and B .

For each camera change:

1. Obtain the new clipping planes in World Co-ordinates.
2. Update the probabilities p_{ij} according to how the VV has changed.
3. For each plane i , for each level j , use the p_{ij} to select objects for checking, and move those objects to their new sets as required. Objects that appear to move into set I are never moved directly into I but into B first of all.
4. Render all objects in I without clipping, but checking if any should be moved to B .
5. Render all objects in B with clipping, but checking if any should be moved into I or into the $B[i,j]$.

For any object that is transformed:

- a. Transform its mean vector and covariance matrix.
- b. If it is in $B[i,j]$ check it and move it to the appropriate new set.

Updating the Probabilities

An important aspect of the algorithm is that the probabilities adapt to how the viewpoint (and therefore view volume) is changing - so that, for example, when the VV is moving away from a set of objects their selection probability is decreased, and when moving towards the set it is increased. Such updating must take account of rotation of the view volume. One way to do this would be to keep an overall mean and covariance matrix for all objects in, for example, $B[i, _]$, and use these to compute a corresponding $\mu_D - \sigma_D$ with respect to plane i . Suppose d_k and d_{k+1} are these distances for two successive camera specifications, then the change $d_{k+1} - d_k$ could be used to update the probabilities p_{ij} ($j=1..L$). However, this scheme would require a lot of computation to update the overall mean vectors and covariance matrices whenever objects joined or left a particular cell.

Another possibility would be to base this on an arbitrary single object in a cell. The computation would be light and necessary anyway, but in the case of rotation, if this object happens to be near the part of the plane that is rotating towards it, then the distance would have seen to have decreased, but the opposite should be the case for objects that are have moved away from the plane because of the rotation.

Instead we use the following scheme. We define a virtual object that initially corresponds to the view volume expressed in World Co-ordinates. This object never changes. However, as the camera changes and the actual view volume changes, the changing distances of this virtual object from the planes of the view volume are used to update the probabilities. This takes into account not just change in actual distance of one view volume boundary from its successor, but also changes in orientation. The particular scheme used is described in the next section.

4.3. *Implementation Heuristics*

In this section we present some of the heuristics used for the current implementation. First we consider the probability scheme. Initially, each $B[i, _]$ is assigned a probability of $\text{prob} = \rho$, and this is halved for each successive level. Let $\Delta d_k = d_{k+1} - d_k$. Then if prob is the current probability for a particular cell, we use $\text{prob} := \text{prob} * (1 - \Delta d_k / w)$ as the updating formula (but clamped to the interval $[0,1]$). Hence when the distance increases the probability of selection decreases, and vice versa.

Next, the cells do not actually contain sets, but sequences, and we do not actually do a random selection within a cell, but start from the beginning of the sequence. For example, if in cell $B[i,j]$ there are N_{ij} objects, and the probability is p_{ij} then we take the first $\lceil N_{ij} \times p_{ij} \rceil$ (ceiling) objects (so that always at least one object is checked). When a selected object is checked, it is moved to the tail of the sequence to which it has been assigned. Hence, if it stays in the same cell, it is simply moved to the back of that cell.

This has several potential advantages: First, there is not the computational overhead involved in “random selection from a set”. Next, there is no special reason to suppose (initially) that the first object assigned to a cell is special in any sense in terms of its relationship to the clip region boundaries, compared to any other object in that cell - so the first might as well be

selected. Third, there is also a possible “principle of locality” that is being exploited here - it is likely that in scene construction objects are not defined in arbitrary order, but that there is some organisation to the construction. Hence this scheme may exploit the fact that whatever happens to one object is likely to happen to its neighbour in any representation scheme.

Finally, if it is found that the last object checked in a sequence has changed, then we continue to check along the sequence beyond $\lceil N_{ij} \times p_{ij} \rceil$, until the first non-changed object is encountered, or the end of the sequence is reached.

4.4. A BSP Tree Implementation

The most natural implementation of this algorithm is in the context of a Z-buffer visibility algorithm. In fact this is the version of the algorithm discussed above, apart from one problem. For the algorithm to be most effective, clearly objects in set I should be rendered without any clipping, with clipping only used for objects in B. Systems such as OpenGL do not support such a possibility - for clipping to the view frustum is always enabled. Moreover, it is not possible in the context of such standard rendering systems to perform special purpose operations - such as not actually clipping an object but at least checking if any of its vertices are outside the clip boundaries. Of course this can be implemented, but the implementation cannot make use of the renderer, so that much of what the renderer does has to be duplicated.

Our initial implementation was, therefore, in a more challenging environment, that is, in the context of a Binary Space Partition (BSP) tree visibility algorithm (Fuchs80). A BSP tree may be constructed by choosing a root polygon, and using its (oriented) plane to form a partition of all other polygons in the scene into three subsets: those on the “front” side of the root, those on the “back” side, and those on the same plane as the root. Polygons that intersect the root plane are split and assigned appropriately. The subdivision process is then applied recursively to each of the front and back sets. The BSP tree allows a back-to-front ordered display of the polygons in the scene: for any subtree, if the viewpoint is in the front half-space defined by the plane of the root then display the subtree referenced by the back child, then render the polygons at the root, and finally display the subtree referenced by the front child. If the viewpoint is in the back half-space, then first display the front subtree, then the root node, and then the back subtree.

The VV culling algorithm operates at the level of objects (i.e., polyhedra) rather than individual polygons. However, each polygon maintains a reference to the object to which it belongs. Therefore, when a polygon is encountered during traversal of the tree while producing a new frame, its corresponding object can be accessed. The object stores information about its visibility status - whether it is in the completely visible region, the boundary region, or in the invisible region (i.e., one of the cells). If the polygon is in the invisible region, it is not rendered. If it is in the visible region, it is rendered without clipping, but during this process there is a simple check to determine whether it should be clipped. If it is in the boundary region, it is rendered with clipping, and as a result there would be information about whether or not it was actually clipped by the boundary.

It is not so straightforward as described, however. Suppose that a polygon is in the boundary region, it is rendered and clipped, and the clipping determines that its status has changed (it is now completely inside, or completely outside). At this time the object status cannot be changed, for there may be subsequent polygons in the BSP tree belonging to this object. In order to determine whether or not the entire object has been processed while traversing the

tree, each object also maintains an identification number (PIN), and there is a PIN also associated with each frame. Initially, all objects have the same PIN as the first frame. The frame PIN is incremented with each successive frame. When an object is encountered that has a different PIN from the frame PIN, this indicates that it is the first time that this object has been encountered for the current frame (in other words all of its polygons would have been processed during the previous frame).

Completely Outside	Intersected by Boundary	Completely Inside
visible = false	visible = false	visible && = true
invisible &&= true	invisible = false	invisible = false

Table 1 Updating the visible and invisible flags associated with an Object: Results of Clipping an Individual Polygon

Two boolean flags are stored for each object. The first is called “invisible”, which has the value true if and only if all polygons so far processed for this object have been in the invisible region. The second is called “visible”, which has the value true, if and only if all polygons so far processed have been in the visible region. Both are initially set to true. The method for updating these flags is shown in Table 1.

Suppose all polygons of an object in the boundary region had been in the visible region - then visible would have the value true, invisible would have the value false, and the object may be reassigned to the completely visible region (I). If all polygons were in the invisible region, then visible would have the value false, and invisible would have the value true, and the object may be reassigned to one of the cells in the invisible region. Finally, if some polygons were visible, and some invisible, so that, of course, some would also be intersected by the boundary, then both flags would return false. In this case the polygon would be displayed with clipping, and the object would remain in the boundary region.

In fact, for the BSP tree implementation there is no need to keep an explicit set corresponding to the boundary region, since all objects are in any case visited (though obviously not necessarily rendered) during traversal of the tree. In one sense, since all polygons are “touched” during traversal of the tree, this does not represent the ideal for a VV culling algorithm - which should only consider at all objects in the visible and boundary set.

4.5. Results

The BSP tree version of the algorithm was tested in order to obtain information about two aspects of performance: first, the extent of the error which the algorithm produces, and second, its time performance compared with the hierarchical bounding box approach.

A test scene (S1) was a representation of a laboratory in the Computer Science Department at UCL. This consisted of 307 objects composed of 11,752 initial faces, and the size of the corresponding BSP tree was 23,648 faces. (The tree was constructed using Fuch’s method

as described in (Fuchs83)). A random walkthrough was constructed by pre-selecting a sequence of points (uniformly) at random inside the overall bounding box of the scene, and the camera moved in small increments along the straight lines joining successive points, with its view plane normal vector corresponding to the direction of the line segment. At the end of such a line segment, the camera would interpolate through the angle between the previous line segment direction and the next, so that once again, its view plane normal vector would face along the current line segment. So the simulation consisted of the camera flying around the scene (translating and rotating) much as a virtual reality observer might fly around such a scene.

There were 588 such camera updates, and therefore frames, altogether, so that the total number of frame-polygons involved was $588 \times 23648 = 13,905,024$.

Polygons Processed	Frequency S1	S1 %	Frequency S2	S2 %
Displayed without clipping	627,258	34	1,440,015	49
Displayed with clipping	1,220,648	66	1,524,831	51
Total	1,847,906	100	2,964,846	100
Number Processed	1,847,906	13.3	2,964,846	15.3
Overall total	13,905,024	100	19,394,550	100

Table 2 Number of Polygons Processed

A second test scene (S2) consisted of a two layers of this laboratory, one on top of the other. This had 23,448 initial faces and 45.850 in the BSP tree. For the second animation the number of frames was 423, and the total number of frame-polygons was 19,394,550.

Table 2 shows the overall results. Out of the total number of frame-polygons, only 13% and 15% were actually passed down the viewing pipeline (i.e., rendered with or without clipping). Table 3 considers only those polygons that were clipped. Approximately one quarter of these turned out to be completely inside the view volume, whereas approximately 10 per cent intersected the boundary.

Clipped	Frequency	S1	Frequency	S2
----------------	------------------	-----------	------------------	-----------

Polygons	y S1	%	S2	%
Completely visible	285,382	23	440975	29
Completely invisible	830,771	68	935749	61
Intersecting boundary	104,495	9	148107	10
Total clipped	1,220,648	100	1,524,831	100

Table 3 Classification of Clipped Polygons

Category of Error	Frequency S1	% total S1	Frequency S2	% total S2
Incorrectly not clipped	7,519	0.05	7,519	0.04
Visible but not displayed	32,542	0.23	32,542	0.17
Unnecessarily clipped	1,137,992	8.18	1,137,992	5.87

Table 4 Error Rates

Table 4 shows error rates. The algorithm may clip polygons unnecessarily (because they were completely outside or completely inside the view volume). This was the largest category of error, but also the “safest” error to make (since the rendered scene is not incorrect if this occurs). The next largest category, is more serious, that is where a polygon is visible, but was not displayed at all. Finally, the case where a polygon should have been clipped but was not, was the category of error that occurred the least.

An attempt has been made to calibrate the algorithm with respect to the parameters:

- k - the multiplier used to measure distance $\mu D - k \cdot \sigma D$ (for these results $k = 1.5$);
- ρ - the initial selection probability assigned to each boundary (here 0.1 was used);
- w - the width of each cell (5 times the camera step distance);
- L - the number of levels for each boundary ($L=10$).

Trial and error indicated that a value of k of about 2 gives good results amongst a number of object shapes, where by inspection, the ellipsoid approximating the object fitted the object.

The above experimental data used a value of $k = 1.5$, which provides too small an approximation for many shapes. However, this ellipsoid approximation is not accurate where there is an uneven distribution of vertices around the object - in this case the density is distributed closer to the region of the greatest density of vertices. This was pointed out by (Gottschalk96) who attempted to improve this situation for their oriented bounding box approximation by evening out the data distribution. In the context of collision detection this is important, but less important for VV culling - where anyway the scheme is probabilistic for rapid walkthrough. However, this is an area where, after the event, we have realised that an alternative scheme could be used. This would involve making a direct estimation of the best fitting ellipsoid that covers all or most of the vertices, and then using this ellipsoid to derive the mean and covariance matrix for the object. Unlike the case of real statistical estimation, we are not interested in the ‘true’ mean and covariance matrix of the object distribution (which anyway is only a fiction) but in using an idealised representation that has useful properties for rapid computation. We are investigating this alternative approach at the time of writing, which should further reduce the error rates (which are already very small).

We carried out a multiple regression analysis to examine the influence of the parameters ρ , w and L . The design had all 27 combinations of three levels of each of the parameters, ρ ($= 1/9, 1/3, 1/2$), $w = (1, 2, 5)$ times the camera step distance, $L = (5, 10, 20)$. There was no significant influence of any of these three parameters over the overall time performance of the algorithm. However, w and L (but not ρ) were highly statistically significant with respect to the error rate, in the case of both those incorrectly not clipped, and those incorrectly not displayed. In each case, over the range of data considered, the error rate was linearly and negatively correlated with both w and L , though L always the most significant. (In each case the square of the correlation coefficient, that is, the variation in the error rate explained by the linear model, was approximately 80%).

S1

	Bounding Box	Probabilistic
Overall Time	137s	89s
No. of times clip called	2,580,314	1,224,708

Table 5 Profile Times for the Hierarchical Bounding Box and Probabilistic Culling Algorithms

S2

	Bounding Box	Probabilistic
Overall Time	232s	139s
No. of times clip called	2,470,686	1,541,888

Table 5 shows the comparative timings, of two culling algorithms, the hierarchical bounding box, and the probabilistic culling algorithm. The time is the total time for the scene walkthrough, but excluding actual rendering time (the rendering occurred on an X11 server

running as a separate process). The profiling was carried out on a SUN Ultra 2 UPA/SBus (2 X UltraSPARC 168mhz) with 192M available memory, under SunOS Release 5.5.1. The table also shows the number of calls to the clip function generated by the two algorithms. The new algorithm, for these particular scenes and these implementations completes the animation in about 60-65% of the time of the bounding box method, and with about 50-60% of the calls to the main clip function. However, it is to be emphasised that these are results, for one type of scene only, and one set of parameters controlling the operation of the new algorithm.

5. Fast Approximate Quantitative Visibility for Complex Scenes

Efficient visibility computations have formed the core of many algorithms in the field computer graphics. Visibility information is necessary for displaying a 3D scene from a specified point of view with hidden surfaces removed, as well as for performing global illumination computations in order to produce photorealistic images.

The goal of the work in this section is to develop an algorithm that given a complex scene *rapidly* computes answers to *quantitative visibility queries*, such as: (i) what is the average visibility between a point (a differential area element) and a finite area or volume; (ii) what is the average visibility between two finite areas or volumes. We define average visibility as the ratio of unoccluded rays to the total number of rays in the ray pencil defined by the query; thus, the answer to such a query is a real number in the interval $[0,1]$. In practice, computing this ratio exactly is infeasible; we are therefore interested in reliable and efficient approximate algorithms.

There are several very important applications requiring the efficient computation of quantitative visibility queries. However the most immediately applicable to our rendering system is in the form of an approximate occlusion culling method. From a given viewpoint we want to answer queries such as how much of a certain object or region can we see? If the answer is none or not much, then we can avoid the cost of rendering it with out introducing much visible error to the image.

In order for the algorithm to handle extremely complex scenes, it must be relatively insensitive to the number of geometric primitives comprising the scene. Since ray-casting algorithms can produce visibility estimates in time roughly $O(\log N)$, our goal is to develop an algorithm whose observed complexity is sub-logarithmic in the number of scene primitives. The actual cost of a quantitative visibility query should depend on the desired accuracy: low accuracy requirements should result in very fast queries; when a more accurate approximation is required the query could take longer to compute.

We describe a new image-based approach towards attacking the quantitative visibility problem. The proposed approach utilises a novel discretization of the space of bounded rays in the scene, constructed in a preprocessing stage. Once computed, this data structure is able to quickly provide approximate answers to visibility queries. Because visibility queries are computed using a discretization of the space, the execution time is effectively decoupled from the number of geometric primitives in the scene. A potential hazard with the proposed approach is that it might require large amounts of memory, if the data structures are designed in a naive fashion. We discuss ways for substantially compressing the discretization, while still allowing rapid query evaluation. Preliminary results obtained via a partial implementation demonstrate the effectiveness of the proposed approach.

5.1. Discrete visibility

As stated above, our approach consists of discretizing the space of bounded rays in the scene. Once such a discretization has been constructed, we can quickly classify any bounded ray in the scene as obstructed or unobstructed. Because we use the discretization of the space instead of the original geometry of the scene, the classification time no longer depends on the number of geometric primitives in the scene. The price that we pay for this increase in

classification speed is that the result of the classification is no longer guaranteed to be precise: certain obstructed rays might be classified as unobstructed, and vice versa. However, since our ultimate goal is to handle quantitative visibility queries for finite pencils of rays, rather than binary ray queries, the accuracy of individual ray queries is not crucial. In designing our discrete visibility data structure, we are guided by the following two requirements: (i) the data structure should support fast classification of visibility along a bounded ray; and (ii) the data structure should be easy to compress.

Consider the simplest possible discretization of the space of bounded rays. Since each bounded ray is defined by its two 3D endpoints, the space of all such rays is six-dimensional. We could simply discretize this six dimensional space, resulting in a 6D binary array. This discretization supports ray classification in constant time, but its high dimensionality makes it difficult to get the memory requirements down to manageable amounts. It is also not clear how well we can compress such a volume, while still providing rapid retrieval of answers to ray queries.

Instead of using a 6D volume, we propose a novel 5D discretization of the space. The reduction of one dimension is traded off for a slight increase in the query retrieval time. The 5D data structure can be thought of as a 2D array of 3D arrays. Each element of the 2D array corresponds to a direction, which can be associated with a point on a unit sphere with polar co-ordinates (θ, ϕ) . For each direction (θ, ϕ) , we have a $n \times n \times d$ 3D array, which can be thought of as a stack of $n \times n$ binary images. Each image is obtained by projecting the contents of a slice of the scene perpendicular to the direction (θ, ϕ) (the projection is parallel to the direction).

Given a bounded visibility ray we first represent the ray in the form $(\theta, \phi, i, j, d_1, d_2)$, where (θ, ϕ) is the direction of the ray, (i, j) are the Cartesian coordinates of the point where the ray intersects a plane perpendicular to the direction (θ, ϕ) positioned in the middle between the ray endpoints, and (d_1, d_2) are the ray endpoints along the given direction. Intuitively, this representation allows us to choose the element(s) in the 2D array, whose direction is closest to that of the ray. Once the discrete direction has been chosen, we can classify the visibility of the ray by examining the (i, j) -th element in all slices between d_1 and d_2 of the corresponding 3D array.

Generally, the direction of the visibility ray will not coincide with one of the directions that we have chosen for our discretization. In this case, we can simply use the nearest direction. A more sophisticated solution would be to examine the 3 closest directions on the sphere, and to combine the results using a weighted average. Having said that, in the remainder of this discussion we will concentrate on the representation of the 3D array corresponding to one particular direction, since the compression and the visibility classification is performed in the same manner for each direction.

The 5D discretization described above is a binary array, in which the ray classification time is $O(d)$, where d is the number of scene slices along each direction. In the next section we describe a modification of this data structure that will allow us to both compress it effectively, and to classify ray queries faster (in time $O(\log d \log n)$, where n is the resolution of the images in each direction). We refer to the compressed 3D data structure corresponding to each direction as a *depth tree*.

5.2. Depth Tree

As said above we are now concentrating on the representation of the visibility corresponding to a particular direction. Given a ray (i, j, d_1, d_2) and a binary discretization of the scene, the ray visibility can be classified in time linear in the number of slices by examining the slices between d_1 and d_2 . Instead we show how the binary discretization is translated to a binary-tree of depth images which can be compressed well and still, the visibility classification is faster.

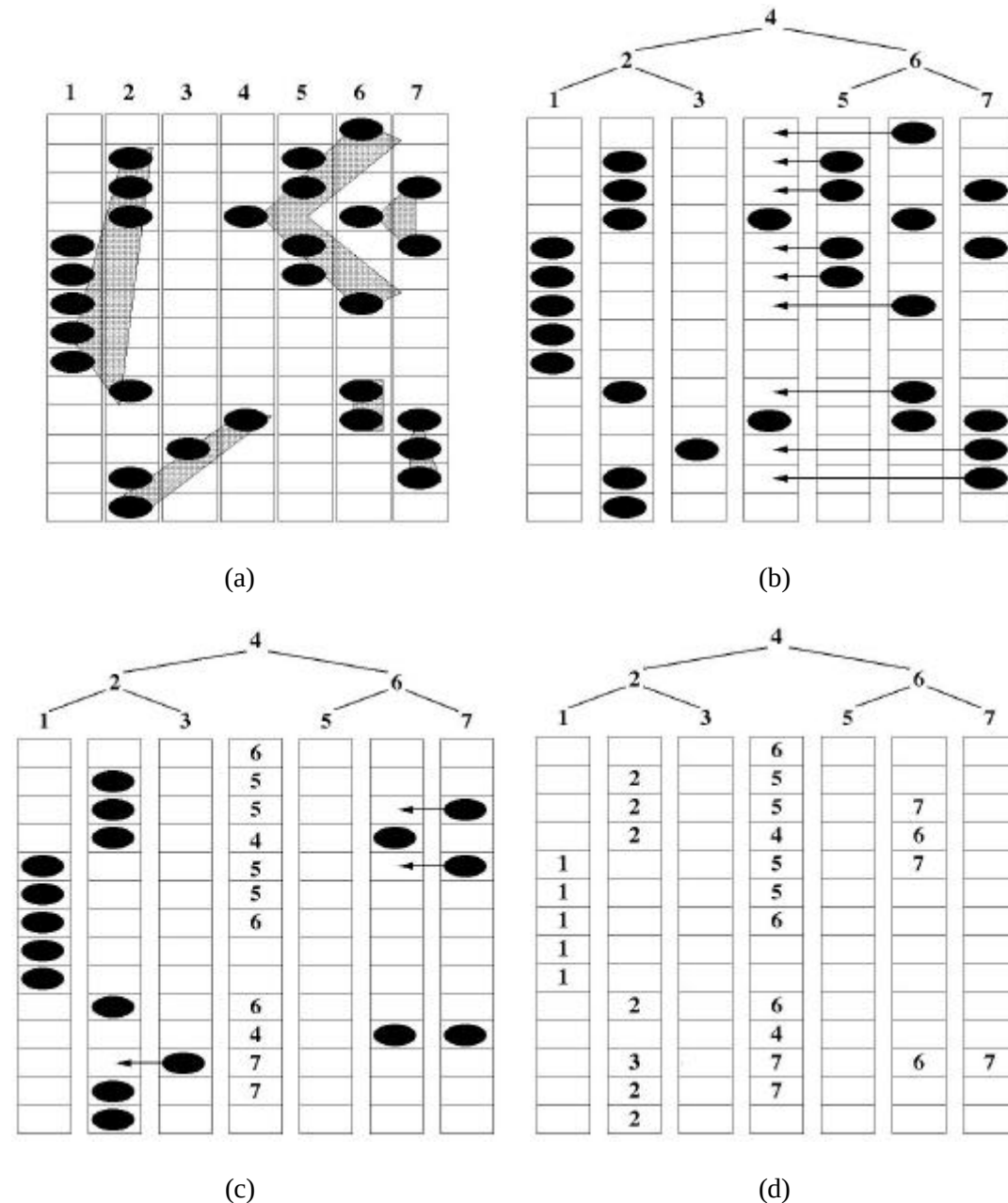


Figure 11 The binary depth image.

Figure 11, illustrates a 2D example where the slices are one dimensional - the vertical columns, and the depth axis is horizontal. We can see in (a) seven binary slices (columns), where the black ovals correspond to non-empty pixels. In (d) the their corresponding seven

depth columns (slices), where each non-empty entry in the binary slices gets the depth values associated with a left-to-right view. Note that the depth image is still spatial coherent as the binary image, and can be well compressed along the axes perpendicular to the slices (columns in the figure). However, it would be much better to have a representation which has a 2D spatial coherence, since it could then be compressed even better. Note that depth image consists of many zero values which guarantee a 2D spatial coherency in the slices along the depth axes. Thus, the slices can be compressed using a quadtree which exploits the 2D spatial coherency. This mean that the access to a particular depth value requires $O(\log n)$, where n is the resolution of the slices. Figure 13, shows six binary slices of the “5 trees” scene containing five trees shown in Figure 12. Figure 14 shows the two depth images and their corresponding quadtrees.

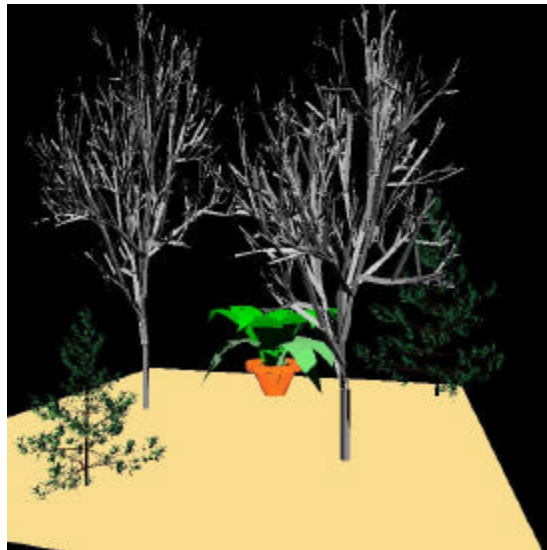


Figure 12 The “5 trees” scene

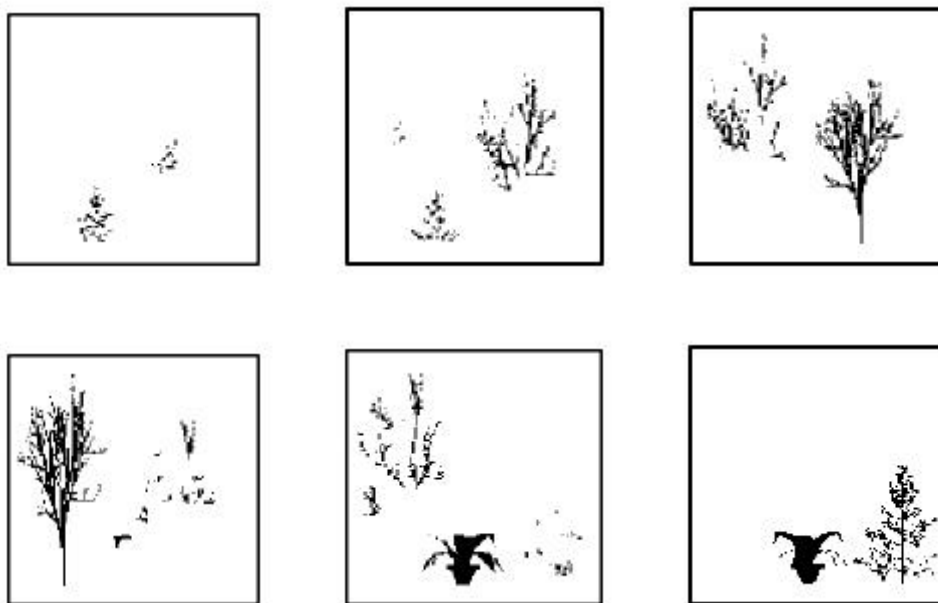


Figure 13 Six binary slices of the “5 trees” image

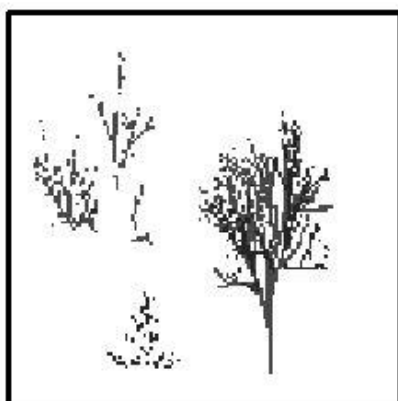
As can be seen in Figure 11 (b), the slices (or columns in Figure 11) can be regarded as nodes of a perfectly balanced binary-tree. This suggests that it may be possible to access the depth values in logarithmic time rather than linear. As we shall show below, the classification of a ray requires a search down the depth tree. That is $O(\log d)$ steps, where d is the number of slices. Including the access into the quadtree the classification of a ray takes $O(\log d \log n)$. However, as we shall see, the depth trees are transformed such that most of the information in the tree is pushed as close as possible to the root of the tree so that on the average the number of steps down the binary-tree is better than logarithmic. See the fourth column in the binary-tree in Figure 11 (d)- it is more loaded than its direct sons (nodes 2 and 6), and much more loaded than the leaves (nodes 1,3,5 and 7). Now we need to show (i) how to construct the depth tree and (ii) how to answer a query.

5.2.1. The construction of the depth tree

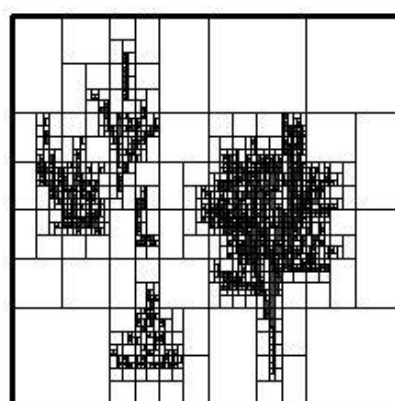
Given a binary image and a depth image the depth tree image is constructed by shifting data upwards the tree while leaving behind zero values. To explain it let's look at a given row in the depth images. Denote $h(x)$ and $v(x)$ the level and the value, respectively, at the x column in the binary-tree image. Note that the tree is balanced, x is the in-order index of the tree, and that $h(x)$ is number of least significant consecutive zeros in binary representation of x .

Initially $v(x) = x$ if and only if the voxel at x is not empty. All other values are null (Figure 11(c)). Now, the value of $v(x)$ is shifted upwards the tree to the entry of x' , $v(x') = v(x)$ provided that (i) $x' < x$, (ii) $h(x) < h(x')$ and (iii) $v(x') = 0$. That is, a depth value shifts up to an empty entry of a higher level in the tree, but only from a right subtree. This *pull* operation starts from the root of the tree, pulls in as much information from its right subtree as possible under the above constraints. Figure 11 (b) shows the initial values of the depth tree before applying the pull operations Figure 11(c) shows the values which are shifted towards the root of the tree.

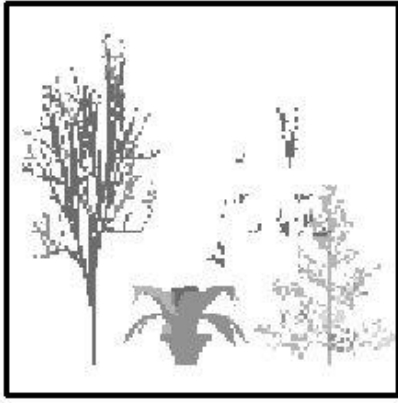
The final depth tree contains no "duplicate" information. Moreover, most of the information tends to be as close as possible to the root. Thus, the lower levels - the leaves or close to the leaves (more than half of the data) are mostly empty and can be well compressed. Note that columns 3 and 5 in Figure 11(d) are empty. This is typical to the leaves of the depth tree.



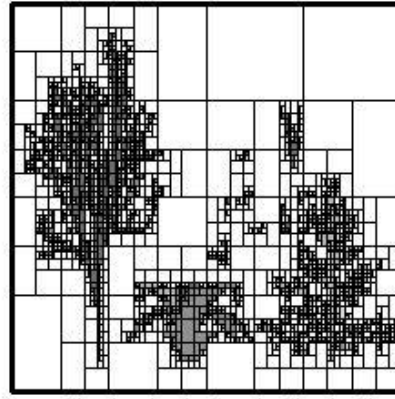
(a)



(b)



(c)



(d)

Figure 14 (a) and (c) are two depth images (2 and 4) of the “5 trees” scene after the pull operation, and their corresponding quad-trees in (b) and (d)

5.2.2. Retrieving the visibility

Once the depth tree has been constructed it can be used for fast ray classification. Given a ray (d_1, d_2) the visibility is retrieved by descending the tree top down. Denote by $d(tree)$ the depth of current tree node, and by $C(tree)$ the value stored in the node. Each node has two subtrees - the left subtree and the right subtree.

There are two cases to consider: the first case is when d_1 , the start-point of the ray is to the left of $d(tree)$. If d_2 , the end-point is to the right of the first occluder, then the ray is occluded. Otherwise we recurse in the left subtree, since there is no more useful information here. The second case is when d_1 is on or to the right $d(tree)$. If the first occluder is null (there are no occluders to the right of the current node) then the ray is unoccluded. Otherwise, there is an occluder at $C(tree)$. Now, the ray may start to the right of it ($d_1 > C(tree)$) so we should recurse down to the right subtree. If the ray ends to the left of the first occluder ($d_2 < C(tree)$) then it is unoccluded. Finally if the ray's endpoints are on either side of the occluder ($d_1 < C(tree) < d_2$) then the ray is occluded.

This visibility classification requires at most $(\log d)$ steps, where each step requires at most one access to $C(tree)$ ($O(\log n)$) and only few comparisons. Note that the classification procedure traverses the tree down to node of d_1 . If that value is at a leaf then it is $(\log d)$ levels away, since our tree is perfectly balanced. However, the value of d_1 could be higher up the tree or we might acquire sufficient information that allow to conclude the visibility and stop before this value is actually reached.

5.3. Results

The data structure that we have created allows a fast point sampling of large and complex scenes by accessing a precomputed compressed representation of the visibility. We have applied the algorithm to complex scenes like the “pine tree” scene which consists of 7,779 polygons, or the “5trees” scene in Figure 12, which consists of 25,816 polygons. These type of scenes are complex enough so that the generation of their shadows is too expensive to be

created in real-time by casting shadow rays. Our experimental tests where applied to one direction only. For the given direction the scenes where discretized, compressed, and the visibility queries where applied only for parallel rays.

These scenes were discretized employing conventional rasterization hardware, which also yields the depth images on the z-buffer. For each slice, the scene was rendered with a front and back clipping planes defining the slice width. The non empty pixels of the slices (see Figure 13) where given their depth value by rendering the scene where the back clipping plane lay on the slice and no front clipping plane. Our implementation of the technique is not optimized. We use no culling technique which could trivially reject most of the scene at each slice, so we re-render everything for every image. Also our implementation of the quad-tree is rather naive. But still the construction of a $511 \times 512 \times 512$ buffer and its compression takes about 5 minutes only. Almost half of that time is spent on rendering and most of the rest on the construction of the quad-trees.

Slice resolution	128	256	512
Number of slices			
127	130	410	758
255	164	510	906
511	202	610	1,119

Table 6 the number of KBytes of memory required for “pinetree” scene, 7,779 polygons

Table 6 and Table 7 show the number of Kbytes required for the “pinetree” and the “5 trees” scenes. Note the direct effect of the resolution of the slice, and the sub-linear dependency in the number of slices. This suggests that it pays off to have more slices. On the other hand, as can be seen in Figure 15, the slice resolution is not that visually critical to the quality of results. One should account the fact that typically such a ray query is only one sample out of many in a ray pencil so that it must not be accurate or in other words, the slice resolution must not be high. Note also that the size of the compressed depth tree is sub-logarithmic in the volume resolution. A 512^3 volume of 128Mbytes is compressed down to 1,506Kbytes, about the same size as a 256^3 volume of 16Mbytes which is compressed down to 1,118Kbytes.

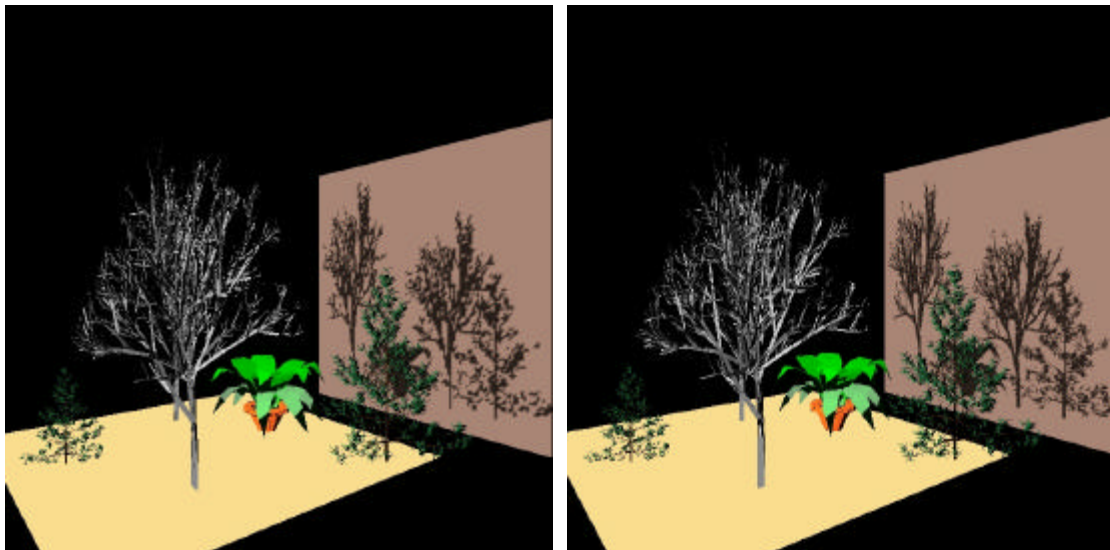
Slice resolution	128	256	512
Number of slices			
127	178	258	829
255	250	716	1,118
511	336	962	1,506

Table 7 the number of KBytes of memory required for the “5 trees” scene, 25,816 polygons

To measure timings we sent 1 million random rays through the two different scenes at different resolutions and they always seem to take the same amount of time. For example, the “pinetree” at resolution 128x128 with only three slices took 4.4sec for all the million rays, and the “5trees” at resolution 512x512 with 511 slices took only 5.5sec. We can see that as we increase the resolution the increase in the time taken to classify the rays is much less than logarithmic, even though our data structures have a logarithmic bound. This is because most rays are classified very high up near the root of the tree and only few are traced down to the leaves.

5.4. Conclusion

We have presented a mechanism for rapidly replying to ray visibility queries. Since it is based on the discretization of both the scene and the ray, the visibility of a single ray is just an approximation. As we mentioned earlier, our final goal is quantitative visibility queries for finite pencils of rays, rather than binary ray queries. When the pencil of rays is very large it would require many ray queries where the accuracy of each individual ray is less significant. Thus, we intend to extend this work and construct a hierarchy of discretization of the scene on which we would be able to query on low resolution rays. Such coarse rays will yield a quantitative visibility that approximates the results of several finer rays. However, the success of replacing many finer rays by a single coarser ray is very much scene dependent. Thus, we can associate with each ray a confidence value which will indicate the probability that the approximation is successful. Such mechanism will provide means for trading off speed for accuracy.



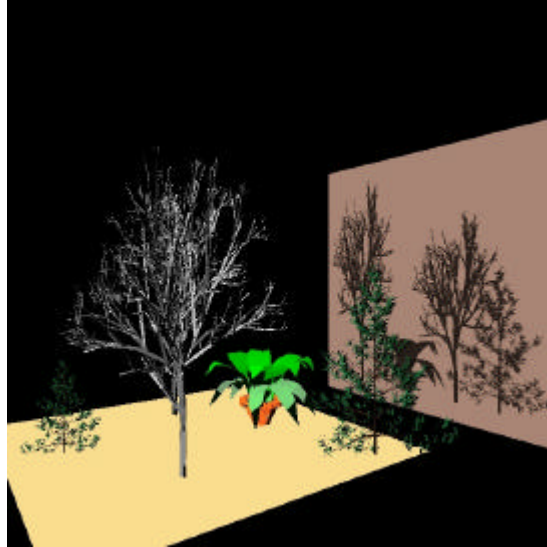


*Figure 15 in this images we see the effect of the slice resolution.
Increasing from 128 to 256 to 512*

6. Conclusion

For the rendering layer of a VR system to be scalable to the kinds of models we are developing in COVEN, without slowing down to unacceptable frame rates, a frame-rate-control system needs to be in place with a number of acceleration techniques at its disposal. Through this workpackage we have suggested such a system and also developed and implemented a number of new techniques for the COVEN platform.

However there is still a lot of work to be done. The frame-rate-control system has not been fully implemented and the probabilistic culling as well as the approximate visibility methods have not been properly integrated. Also even greater accelerations could be achieved if image based methods (image caching) are added to the system.

7. References

- [Blinn91] Blinn, J.F. *A Trip Down the Graphics Pipeline- Line Clipping* IEEE CG&A Jan, 11(1) 98-105.
- [CARLS93a] C. Carlsson and O. Hagsand, “*DIVE — A Multi User Virtual Reality System*”, IEEE VRAIS, Sept, 1993.
- [CARLS93b] C. Carlsson and O. Hagsand, “*DIVE — A Platform for Multi-User Virtual Environments*”, Computers and Graphics 17(6), 1993.
- [CERTAIN96] A. Certain, J. Popovic, T. DeRose, T. Duchamp, D. Salesin, W. Stuetzle, “*Interactive multiresolution surface viewing*”, SIGGRAPH '96, August, 1996, pp. 91-98.
- [Clark76] Clark, J.H. *Hierarchical Geometric Models for visible surface algorithms*, Communications of the ACM, 19(10), 547-554.
- [Fuchs:80] Fuchs, H., Kedem, Z.M., Naylor, B.F. *On visible surface generation by a priori tree structures*, Computer Graphics (SIGGRAPH), 14(3), 124-133.
- [Fuchs:83] Fuchs, H. Abram, G.D., Grant, E.D. *Near real-time shaded display of rigid objects*, Computer Graphics (SIGGRAPH) 17(3), 65-72.
- [GIF89] “*GIF™ — Graphics Interchange Format™ — A standard defining a mechanism for the storage and transmission of raster-based graphics information*”, Version 89a, CompuServe, <http://www.w3.org/pub/WWW/Graphics/GIF/spec-gif89a.txt>.
- [Gottschalk] Gottschalk, S., Lin, M.C., Manocha, D. *OBBTree: A Hierarchical Structure for Rapid Interference Detection*, Computer Graphics (SIGGRAPH Proceedings), 171-180.
- [HAGSAND96] O. Hagsand, “*Interactive Multi-User VEs in the DIVE System*”, IEEE Multimedia Magazine, Vol. 3, No. 1, 1996.
- [HOPPE96] H. Hoppe, “*Progressive Meshes*”, Proceedings of SIGGRAPH'96, August 1996, pp. 99-108.
- [Rohlf] Rohlf, J. and Helman, J. *IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics*, Computer Graphics (SIGGRAPH Proceedings), 381-394.
- [SCHMALSTIEG97] D. Schmalstieg, G. Schaufler, “*Smooth Levels of Detail*”, VRAIS '97, Albuquerque, New Mexico, March 1-5, 1997, pp 12-19.