Interactive Rendering of Complex 3D Models in Pipelined Graphics Architectures

Paulo W. C. Maciel

Technical Report

Department of Computer Science

Indiana University, Bloomington

May 23, 1994

Abstract

Interactive rendering of complex 3D environments is important for applications such as architectural design, CAD, flight simulation and virtual reality. These systems need to provide the user with both an illusion of real-time navigation and a realistic feel of the environment. The problem is that realistic looking models contain more graphics primitives then the interactive capabilities of graphics workstations. In this report we formalize the interactive navigation problem as a tree search problem and describe a prototype for interactive exploration of complex environments. In a pre-processing phase we create a hierarchical tree structure that represents the model in which each node is a hardware drawable representation of a set of objects in the model but takes less time to render then its children. In the interactive phase and according to a 'benefit' heuristic(contribution to model perception) and 'cost' heuristic(time to render a set of objects), we search this structure selecting drawable representations of objects that are then sent to a graphics pipeline for rendering. We also describe a framework that we will use to make the prototype more general and will also serve as a testbed for the implementation of specific solutions to some problems with the current prototype.

Thanks to my advisor Peter Shirley who started this research and contributed with many key ideas and proofread this document. Thanks to my sponsor, the Brazilian Government Agency CAPES (Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior) who gives me financial support for this research. Thanks to my research committee members Andrew Hanson, Dennis Gannon and Bruce Solomon. Thanks to the NSF grant that provided the graphics workstations that were used in this research.

Contents

List of Figures

Chapter 1

Introduction

In this document we present a technique that allows a user to navigate through a complex 3D environment at interactive frame rates. This is accomplished by replacing the complex geometry of the model with a simplified version of it that is within the interactive capability of a graphics workstation. We have investigated a two strategies for creating this simplified geometry and designed algorithms to decide which simplified version to display in real-time. We give a detailed description of a prototype implementation explaining why each given strategy was pursued together with the problems found with their implementations. Finally we present a design for a new framework that we will base the rest of this research which we hope will solve the problems with the current implementation.

1.1 Problem Statement

Interactive computer graphics rendering of very complex virtual environments are useful in applications such as architectural design, CAD, flight simulation and virtual reality.

For these systems to be effective however, they need to meet two competing criteria: 1) provide the user with an illusion of real-time exploration of the environment and 2) be "realistic", both in terms of the quality of the images produced and the complexity of the world they simulate. The reason these criteria are antagonic is that while an interactive system needs to achieve interactive frame rates of at least than 10 frames per second, realistic-looking models can contain hundreds of millions of polygons, far more than currently available workstations can render in an interactive fashion[1, 11, 15].

Current workstations are able to render one million polygons per second. Although this is an impressive number, at an interactive frame rate of 30 frames per second, it leaves us with a maximum of about 30K triangles per scene. For the sake of comparison, the IU campus database currently contains 70K polygons(and is unfinished) it represents most of its buildings(some of them just boxes) without limestone details, sidewalks, cars, people, trees[1] and so on.

From [4] we get the quote: "Initial results show the expected: many industrial virtual reality applications need one to two orders of magnitude of improvement in display performance". This is actually a conservative statement. An interactive walk-through a geometric database for an entire city, with vegetation, cars, buildings with furniture inside etc ... is much more than two orders of magnitude out of reach.

[1] In our current implementation we have a procedurally generated tree by Ken Chiu that contains around 7K polygons, and is not detailed enough to be viewed from close range.

Simply culling the model against the viewing frustum reduces the problem but doesn't eliminate it since more often then not the complexity of the visible scene is still too high to allow interactive walk-throughs. As we shall see in Section 2 next, techniques have been described that alleviate this problem to some extent.

1.2 Document Overview

In the next Section(2) we present an overview of the strategies that are currently being used to achieve interactive frame rates in walk-throughs.

In Section 3 we describe the new approach and the prototype that was implemented together with the lessons learned in the course of its implementation.

In Section 4 we propose a framework in which new ideas to solve the problems with the current implementation mentioned in 3 will be explored.

In the last Section(5) we summarize the work we have done so far and propose a plan to implement the framework and the ideas outlined in Section 4 to complete this research.

Chapter 2

Background

In this section we discuss previous approaches that to some extent cope with the hardware's inability to render realistic and arbitrarily large databases in real-time.

The attempts to solve the problem just described resort to one or both of the following techniques: *culling*, which eliminates from rendering considerations objects that fall outside the viewing frustum and *level of detail(LOD) Management*, that selects for display simplified versions of visible objects depending on how much they "contribute" to the final image.

2.1 Culling Techniques

At a given observer viewpoint and field of view, only parts of a given 3D model are visible and therefore we do not need to render objects that fall outside the user's viewing frustum. Examples of this technique can be found in [8, 16].

Funkhouser et al.[8], describe a technique for managing large amounts of data during interactive walk-through of an architectural model that uses spatial subdivision, visibility analysis and objects at multiple levels of detail to reduce the number of polygons to render per scene. In a pre-processing phase they perform a spatial subdivision of the model, do viewpoint independent lighting calculations using radiosity and ray tracing, and visibility computations. By using a variant of a tree structure they divide the whole model along the major opaque elements(like door frames, walls and floors) into cells and by identifying portals(transparent portions of boundaries) they build an adjacency graph. Then they do a series of visibility computations for each leaf cell, to determine if a cell is visible by another in the case where there is a sight line that traverses a portal sequence, called cell-to-cell visibility. The result of this computation is a stab-tree constructed from the adjacency graph. Then the set of objects that can be seen by an observer constrained to a given source cell, cell-to-object visibility, is computed and associated with the source and reached cell in a representation of the stab tree.

During the interactive phase, the cell containing the observer is identified and its cell-to-object visibility(a superset of the objects actually visible given its position in the cell)is accessed from a display database. The eye-to-cell visibility, the set of all objects incident upon a cell partially or completely visible to the observer, is computed. Finally, to estimate the eye-to-object visibility, a superset of the objects that are actually visible to the observer, they perform an intersection of the cell-to-objects and eye-to-cell sets.

They showed that for a particular model containing around 250K polygons and a particular viewpoint, with this strategy they were able to reduce the amount of polygons that need to be rendered to around 8% of the total. By further using objects represented at different LODs, they

were able to reduce this number even further to a total of 1.2% of the entire model, around 3K polygons.

The technique described in Greene et al.[16] combine three types of coherence inherent in the visibility computation, namely, object-space, image-space and temporal coherence. Object-space coherence, resolve the visibility of a collection of objects near to each other in space. Image-space coherence resolve the visibility of an object covering a collection of pixels and finally temporal-coherence where they use the visibility computation of the previous frame to speed-up the visibility computation of the current frame. The object-space coherence is exploited by dividing the model using an octree that is further combined with a Z-buffering strategy to eliminate objects from rendering considerations. By scan converting the faces of an octree cube they determine if the cube is visible or not. If the cube is not visible then the whole geometry inside it can be discarded.

In order to determine if the octree cube is hidden or not they scan convert each one of its 6 faces using a data structure that exploits image space coherence, a Z-pyramid. This pyramid is a structure that has the original Z-buffer at its finest level and combines 4 z-values in one level to get 1 at the next coarser level. Testing if a polygon is visible is done by finding a sample on the finest-level of the pyramid whose corresponding image region covers the screen-space bounding box of the polygon and if this value is closer than the nearest z-value of the polygon, then it is hidden. Temporal coherence is motivated by the fact that most of the octree cubes visible in the previous frame will still be visible in the current frame and is therefore explored by maintaining a list of the visible cubes in the previous frame. They first render this list to get the initial z-buffer and form the z-pyramid from it. By doing this, they managed to speed up the z-pyramid tests by proving with less recursion that octree cubes and polygons are hidden.

The authors showed that using this technique on a model containing half a billion polygons they were able to reduce the amount of polygons to be rendered to just around 40K polygons. However, even with the amazing reduction of complexity of this half a billion polygon scene to 40K polygons, the whole process still took 6.45 seconds to render in a top of the line SGI Crimson Elan, which is too much time for interactive rendering.

While the first of these techniques is suitable in cases where the entire model can be subdivided along major opaque elements possibly having portals, as in a building, it is not suited to outdoor environments. Although the second technique does not suffer from this problem, it would only allow interactive navigation of complex models if it would be implemented in hardware and if the model does not contain too many visible surfaces.

2.2 Level of Detail Management

Depending on its size and distance to the viewpoint, an object do not need to be rendered at full detail since the details will not be noticeable. This also apply to moving objects since sampling problems[6] will alias its image depending on its position from frame to frame.

Therefore, objects need to be described at different levels of detail(LOD) to reduce their rendering time. Figure 2.1 shows a book case and a book represented each at 3 LODs.

To that end, techniques have been developed to automatically obtain simplified polygonal versions of an object as well as to determine which of the object's representation should be rendered at a given point in the simulation.

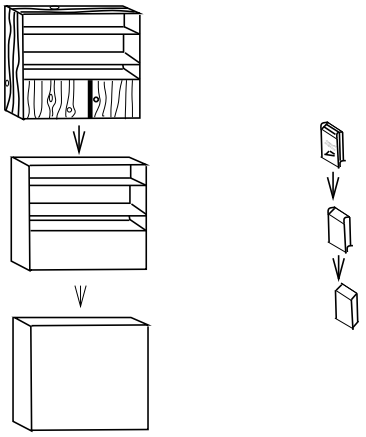

Figure 2.1: Two objects represented at 3 LODs

2.2.1 Polygonal Object Simplification

Hall et al.[10] describe a modeling system that is capable of generating objects at multiple levels of detail based on a user specified shape accuracy parameter. The number of accuracy levels depend on the type of the object. While a block can only be represented using one LOD, a sphere can have representations ranging from a box to a curved approximation. A complex object can also have a variable LOD representation, depending on the LODs of its component parts.

These tools are not suited to all applications. In scientific visualization for instance, one might want to generate multiple LODs for iso-surface models(medical), derived from volume data, or for surface molecular models(molecular graphics). In other applications, we may want to scan in a plastic model(with a laser scan) of an object(like an airplane) and have the system automatically generate the different LODs. In cases like this, an initial mesh has to be formed from the set of random 3D sampled points, and then simplified using mesh simplification techniques.

The technique devised by Turk in [23], is appropriate to generate curved surfaces. Initially, given a polygonal surface, a triangulation of it with a user specified number of vertices is created by randomly choosing a set of points in the planes of the original polygons, and by having each point repel each of its neighbors by means of a relaxation procedure. These points will be more densely distributed in regions of high curvature and will eventually become the vertices of a new triangular tessellation. A mutual tessellation is then created containing the old vertices of the original surface and these new points. Finally, the old vertices are removed from the mutual tessellation, one at a time, and the surface is locally re-tiled in a way that the new triangles accurately reflect the connectedness of the original surface. This technique is suited for modeling medical data and mathematical surfaces.

For generation of polygonal representations at different LODs of a real object from sample points obtained from a laser ranger device, Hoppe et al. devised a mesh generation method that can fit a mesh of arbitrary topological type to a set of data points[12], and a mesh optimization method [13] that improves the mesh's fit and reduces its number of faces, recovering sharp edges and corners common in objects such as machine parts. This optimization method can also be used to simplify an arbitrary mesh by sampling data points from the original mesh and use it as the starting point of the optimization procedure. The optimization method works by minimizing an energy function that captures the requirements of geometric fit and compact representation of the mesh. This energy function has three components. The first, represents the fact that the optimized

mesh is a good fit to the set of sampled points. The second term penalizes meshes with many vertices. The last one, represents a regularizing term that amounts to placing on each edge of the mesh a spring of rest length zero and some spring constant. This guarantees the existence of a minimum for the energy function.

2.2.2 LOD Selection

In [8], a display database representing objects at different LODs is described. The criteria used to select a given object LOD is static and based on a pre-determined size and speed threshold that are compared against the size in screen pixels of an average face of the object and on the objects relative speed to the observer, respectively. Although this strategy reduces the amount of polygons that have to be rendered in a scene, since the selection of LOD is static, the frame time can be arbitrarily large[7], since as the observer moves, many more objects can be visible than the machine hardware can render in real-time. This is static selection mechanism is also not good in situations where lots of objects become visible/larger/slower, like in an aircraft landing situation.

Commercial flight simulators[25] minimize this problem by means of computing a size threshold prior to rendering each frame based on the time needed to render the previous scene in an adaptive fashion. While this works well for flight simulation where there is a large amount of coherence from frame to frame, this strategy won't produce a uniform frame rate for applications like building walk-throughs where scene complexity can change dramatically e.g. when the observer moves from a corridor with very few objects to an auditorium with many objects.

To correct the problems caused by static and feedback LOD selection mechanisms, namely, arbitrarily large and non-uniform frame rates, Funkhouser [7] presented an algorithm that adjust image quality adaptively to maintain a user specified frame rate based on heuristics that estimate the computational "Cost" of rendering a scene versus its "Benefit" (the "quality" of the picture). In this algorithm, every object is associated to a tuple (O,L,R), where O is the object to be rendered, L is the LOD for the object and R is the rendering algorithm selected(like, flat/Gouraud shading, antialized, lighted ...). The idea is to maximize: $\sum Benefit(O, L, R)$, subject to $\sum Cost(O, L, R) <$ target-frame-time, but since this problem is a hard problem instead of devising an exact solution for the problem, they implemented a greedy approximation algorithm that selects tuples according to their Benefit/Cost ratios, in a descending order until the maximum cost (target frame time) is reached. Visible objects with low Benefit/Cost ratio that would make the total cost greater than the maximum are not displayed (or displayed with zero LOD).

The problem with this approach is that in many situations the visible geometry of a model is too complex to be rendered in real-time and since visible objects that don't fit into this frame time bucket are not rendered the whole simulation is impaired since it may not provide the required illusion of the virtual environment for the user running the application.

Incidental to the issue of LOD management is the selection of the method used to switch smoothly from one LOD representation to another in order to prevent "popping" of the object on the screen. Two methods have been used: One uses geometric interpolation as in Turk[23] and the other uses hardware dependent color blending as described in [19] and used in [25, 8]. Both approaches present problems. The first strategy may not be feasible in real-time if the number of vertices on the object is too large and the other has to be used with care since when one object fades-in the other has to gradually fade-out and until the transition is completed we are rendering both LODs and therefore increasing the frame time.

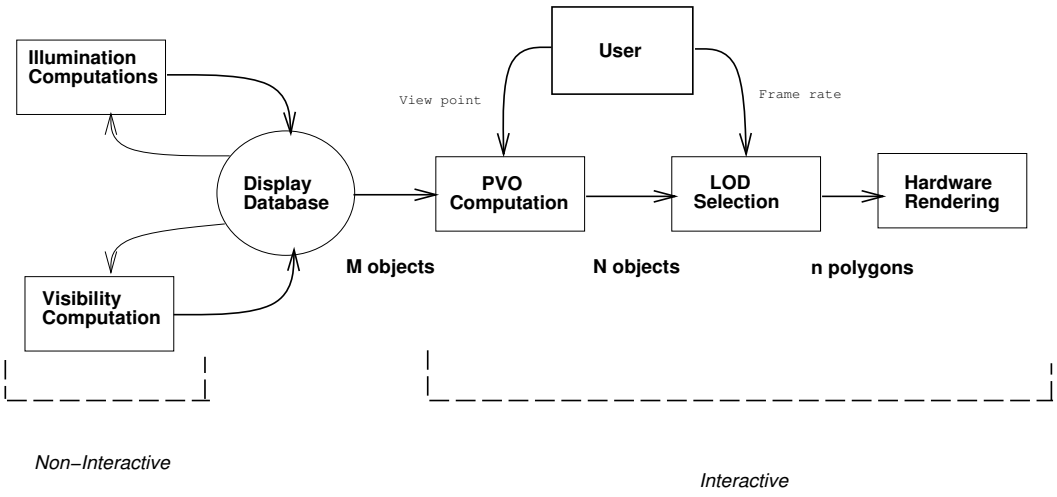

Figure 2.2: Rendering diagram

2.3 Summary of Previous Work

From the description given above we abstract in figure 2.2 the model that is currently being used to interactively render environments of medium size complexity(around a million polygons).

In a pre-processing phase, viewer independent visibility and illumination computations are performed and stored in a display database. In the visibility phase the structures that record the spatial relationship of objects in the model is built. These structures are later used to cull away hidden geometry in real-time. The illumination phase is used to compute the color of the objects in the model together with their shadows according to the light sources present.

During the interactive phase and for each frame, depending on the observer view point, a set of potentially visible objects(PVO) is computed. Usually, information obtained in the visibility pre-processing phase is combined with the observer viewing frustum to obtain the PVO. In the subsequent phase, models for each one of these objects at different levels of detail(LOD) are then chosen to meet a user specified target frame rate. Finally, the resulting polygons are displayed in a rendering system using a hardware z-buffer.

Chapter 3

Discussion of the New Approach

In this work we want to design a program that takes as input a display database composed of a set of objects and a set of modeling commands that place each object at a certain position and orientation in space, forming a complex 3D environment in such a way that would enable a user to navigate within it at interactive frame rates. Since more often than not there will be more objects visible from the observer viewpoint than the machine hardware can render in real-time this input data set will need to be pre-processed in a way that enables the walk-through program to select the best representation for objects and/or groups of objects that when rendered will not exceed a user-specified frame time. This is depicted in figure 3.1.

Conceptually, an individual object in the model can be displayed at several LODs. Each one of these representations give a certain contribution to the quality of the simulation and each has a rendering cost associated to it depending on factors such as the rendering algorithm used, its image space size and its number of polygons. Likewise, a set of objects can be represented on the screen by some approximation of the real objects that will hopefully produce an image similar to the one that would have been obtained if the actual objects were rendered but will take less time to render. Therefore, if we extend the idea of having each object being represented at different LODs to sets of objects, then we can form a hierarchy that represents the entire model by grouping objects together and obtaining a representation for the group that costs less to render than the actual objects.

At the lowest level of this hierarchy , we have the whole detailed model. Objects in subsequent levels are grouped together to form the next level of the hierarchy, until the whole model

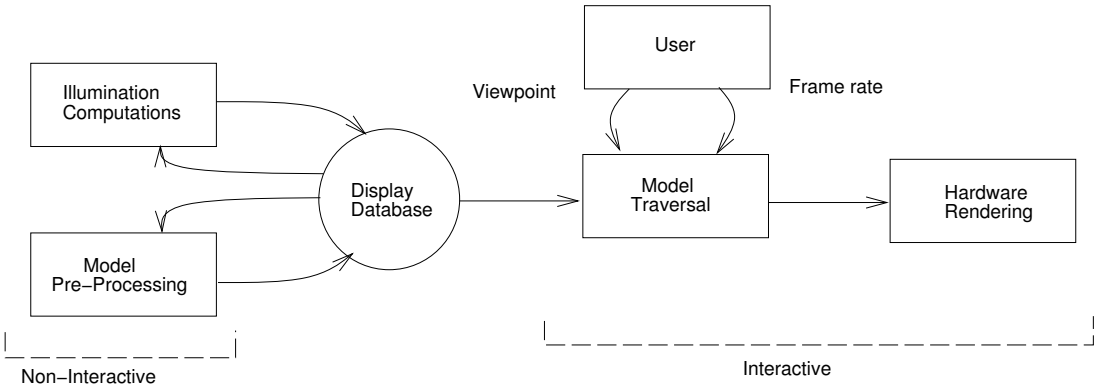

Figure 3.1: Proposed approach.

is represented by a single primitive. This allows interactive frame rates by displaying the "best" image possible given the available resources. By clustering together all the objects in the model and thereby arbitrarily reducing its complexity our simulation will never, in a sense, disregard any piece of the model as in the approach used by Funkhouser et al.. The rationale behind the construction of this hierarchy is given in section 3.4 below.

The approach just described, is also a variation of the predictive approach since instead of using the time to render the previous frame to select LODs to render the current frame as in the reactive approach, we are estimating the time and quality of the next frame before rendering it. This approach also assumes that interactivity is more important then image quality in situations where there are too many visible graphics primitives than the hardware is able to draw in real-time. Given this conceptual discussion, we now formalize this problem.

3.1 Formalization of the Problem

We begin by defining a cluster of objects to be any drawable entity that can be displayed on the computer screen and has associated to it:

- A rendering cost.
- A measure of its "contribution" to the quality of the image.
- An abstract representation of a group of objects in the model.

The model is defined to be a collection of conceptual objects at specific positions and orientations in the world forming the environment in which the user navigates.

A conceptual object is a hardware drawable entity that has a conceptual meaning for the particular walk-through such as a building, a car, a person and so on.

A cluster is just a set of hardware drawable entities that represents either a conceptual object or a set of of conceptual objects in the model that does not necessarily have a meaning.

The rendering cost of a cluster is defined to be the user time that a particular hardware takes to render the cluster on the screen and therefore is not influenced by variables such as the state of the operating system, interrupts or system load.

The contribution that a cluster provides to the image quality is a measurement of how much the cluster's representation of a portion of the model resembles the actual portion of the model. By the actual portion of the model we mean, a flat representation of the model with all objects at their highest LOD, rendered with the most accurate rendering algorithm available.

The model hierarchy is an arbitrary tree structure whose nodes are clusters of objects which provide multiple representations of the model each of which representing the model at a given rendering time and providing the user with a given perception of it. The root of this tree is the coarsest drawable representation of the entire model with the lowest possible rendering cost. Likewise each parent node is a drawable representation of its children that has a rendering cost less than the sum of the rendering cost of its children. While the root is the coarsest representation of the model with the lowest possible rendering cost, the leaves form the perceptually best representation of the model with the highest rendering cost.

Given these definitions, we state the walk-through problem as a tree search problem:

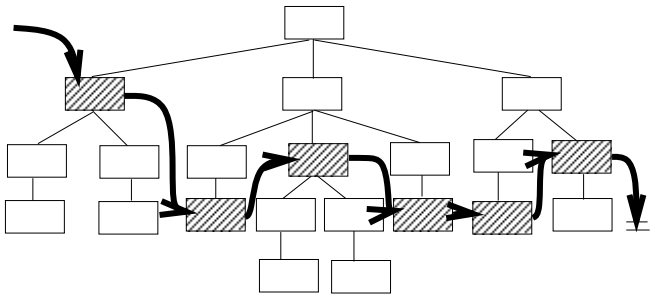

Figure 3.2: Tree representing the model hierarchy and the set of nodes to be rendered as a linked list.

"Select a set of nodes in the model hierarchy that provides the user with the perceptually best representation of the model", according to the following constraints:

1. The sum of the rendering cost of all selected nodes is less than the user specified frame time.

2. If one tree node is selected then none of its descendants can be selected since this node is already a representation of its descendants.

Figure 3.2 shows what the tree that represents the model hierarchy looks like together with a list of the nodes that will be sent to the graphics hardware for rendering.

This tree search is a little different from what can be found in the literature(see [18]) because it does not attempt to find a goal node with a certain property but instead it aims to find a set of nodes that is constrained to a certain property.

To solve this problem we need to look into the issues related to:

- The hardware that will run the application in order to get a model of a generalized rendering system that will run the application.
- Rendering cost.
- Scene perception metrics.
- The building of the model hierarchy.
- The tree search.

These issues are examined in the next sub-sections.

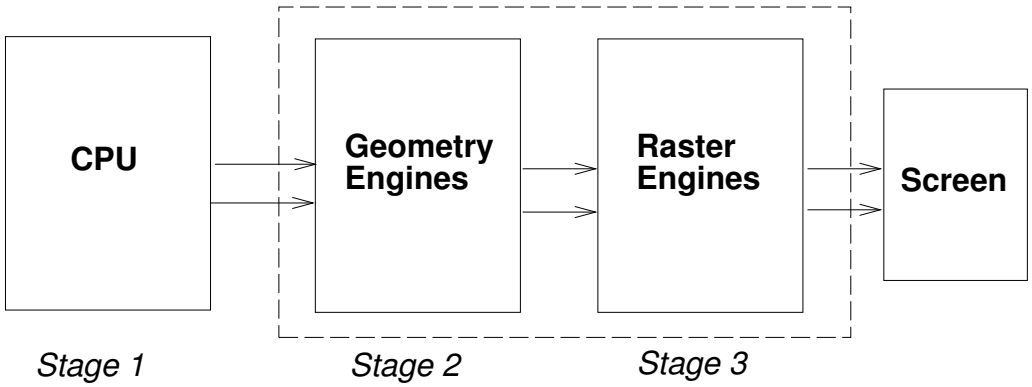

Figure 3.3: Stages of the graphics pipeline.

3.2 Hardware Assumptions

Although machine architectures differ from one manufacturer to another, and will continue to differ in the future in terms of the speed of their components, it seems unlikely that the pipelined architecture of the graphics engine of these machines will cease to exist in the near future since it is based on the conceptual rendering pipeline that is described in any introductory computer graphics book[6, 24]. Not only state-of-the-art hardware such as the SGI ONYX and RealityEngine [1] workstations(and workstations from other vendors such as Sun) use this pipelined architecture but also highly parallel architectures such as PixelFlow[15] use the same pipeline concept.

3.2.1 Pipeline Architecture

Assuming future machines will make use of a graphics pipeline we present a generalized rendering system model that for our purposes, can be applied to a wide variety of graphics workstations.

Based on the *Graphics Library Programming Tools and Techniques*[20] document from Silicon Graphics Inc., and similar to what was done in[7], we derive a model of a generalized rendering system that this research is based on. This rendering system is represented as a pipeline with 3 functional stages as shown in figure 3.3.

The first stage runs the application program that sends graphics requests to the graphics subsystem. The next stage does per-polygon operations such as coordinate transformations, lighting, depth-cueing, clipping and concave polygon decomposition. The final stage does per-pixel operations such as writing colors into the frame buffer, z-buffering and possibly alpha blending and texture mapping.

In this model, the amount of work done by different pipeline stages varies among applications. For instances, an application that draws a small number of large polygons will load the last stage of the pipeline and lightly load the first two stages whereas a program drawing large number of small polygons will certainly load the first stage. Since separate stages run in parallel and wait only if the next stage is not available to receive more requests, the speed of the system is determined by the speed of the slowest stage. Also, the performance of the first stage can be significantly improved in hardware architectures that allow the addition of extra processors since extra time will be needed to implement software solutions to reduce the work by the graphics hardware.

It is also interesting to notice that the graphics pipeline can do work in parallel with the cpu and we can take advantage of this parallelism by intelligently selecting which objects to send to the graphics pipeline. This situation gets even better in machines with more then one processor.

In this case while the graphics pipeline is rendering one frame of the simulation the cpu is already computing the next one and as long as this time does not exceed the frame time the graphics pipeline will not be delayed and the frame rate will not be affected.

3.2.2 Texture Mapping Capability

Texture mapping is a technique in which two dimensional images are pasted onto geometric objects yielding very realistic looking objects. This mathematical mapping is ideally suited for hardware implementation and many workstations nowadays have this feature implemented in their hardwares. As memory prices go down and cpu power go up, we can expect to see in the near future increases in the number of low cost machines(such as [5]) providing hardware textures and for those which already have this capability increasing texture memory size and rendering speeds.

3.2.3 Real-Time Features

In order to truly maintain a user-specified frame rate the cpu in which the application is running has to be relatively free of interrupts. Since interrupts are essential to the system it only makes sense disabling interrupts to a processor in a multi-processor systems. In these machines, one cpu can be dedicated to the real-time process while others can be used to handle the system and device interrupts. By means of using interprocess communication synchronization primitives like semaphores or by inserting directives(pragmas) for the multi-processing compiler or for tools that analyze and parallelize code such as [21], the program can be easily parallelized and by means of utilities such as those available for the IRIX operating system[22] such as *sysmp, systune, runon*[1] and others the real-time walk-through program can be locked alone into a relatively interrupt free processor while the work to compute the list of objects to be rendered under the control of this cpu can be divided among the other processors in the system.

3.3 Cost and Model Perception Metrics

The success of this approach depends on cost and perception heuristics being both accurate and computable within the available rendering time interval.

3.3.1 Cost heuristic

As mentioned before, the cost associated with rendering a drawable entity depend on factors such as:

1. Number, type of polygons and geometric operations such as lighting and depth-cueing.
2. Image space size of object, rendering algorithm and raster operations such as pixel blending and texture mapping.
3. Application overhead.

[1] Sysmp(sysadmi) is a system call(command) that provides multiprocessing control and information for miscellaneous system services. Systune allows the super-user to examine/configure kernel parameters. Runon assigns a specific processor to a given process.

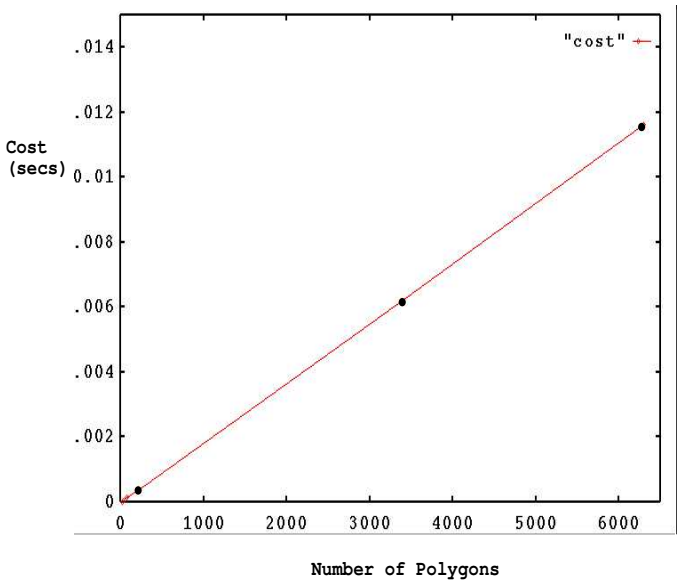

Figure 3.4: RealityEngine rendering cost plot.

In the generalized rendering system model described, 1 and 2 above will influence 2 different stages of the graphics pipeline namely, the geometry and raster stages, respectively.

The cost for an object to clear the first and the second stages of the graphics pipeline are then the functions:

- $G(pol, vert, go)$, where pol is the number of polygons in the object, $vert$ is its number of vertices and 'go' is the geometric operation associated with it.
- $R(pixels, ro)$, where $pixels$ is the image space size of the object and 'ro' is the raster operation associated with it.

Since in this generalized model the throughput of the pipeline depends on the slowest stage, the object's cost is the maximum of these 2 costs, that is $MAX(G(pol, vert, go), R(pixels, ro))$.

This cost function can be determined empirically by selecting geometric and raster operations and then rendering objects in the model. In [7] these functions are defined prior to the walk-through of the environment and computed in real-time. Instead of doing that we build a cost table in a pre-processing phase and just look it up in real-time. Also, to cope with the fact that the image space size of the object changes according to the viewpoint we can render the objects from varying distances to the viewpoint so that we can select, during the walk-through, its appropriate cost.

Figure 3.4 shows the cost function obtained for a RealityEngine for different objects containing a varying number of polygons, using Gouraud shading, no hardware lighting and no texture mapping. Three points are shown around 200, 3600 and 6300 polygons.

These timings were user time and do not include any application or interrupt overhead since it will vary from our cost measurement program to our walk-through program. Later, we can interactively fine tune our walk-through by measuring the overhead of the simulation and adjusting the cost of all the objects at once.

3.3.2 Benefit Heuristic

A good benefit heuristic should predict the amount and accuracy of the information caused by the rendering of the object, but it is hard to come by since it involves issues like the user perception of the model and how the user interprets what it sees.

Funkhouser et al.[7], use a benefit heuristic that takes into account 2 factors:

- The projected size of the object onto the screen, since large objects seem to appear to contribute more to the image
- Accuracy of the representation, measured in terms of the rendering algorithm and the number of polygons used to represent the object.

They also mention that a benefit heuristic should incorporate factors like:

- The importance of the object to the simulation. An enemy fighter plane should have its benefit enhanced although it may be relatively far from the viewpoint.
- The object's proximity to the center of the screen. This takes into consideration that objects in the periphery of the field of view are not seen by the eyes with full detail compared to the ones close to the line of sight(See [3]).
- The object's relative speed with respect to the viewpoint. This takes into account that fast moving objects will be seen blurred and there is no point in trying to render them at full detail.
- The fact that the object was rendered at a given LOD in the previous frame. This should reduce the annoying effect caused by frequently switching from one representation to another from frame to frame.

They quantify these factors by a real number between zero and 1, where 1 is the representation which is most beneficial to the simulation.

Besides all these factors there are some issues in the benefit heuristic that are worth mentioning. One of these issues regards the size of objects. If an object is small it may be reasonable to say that it contributes less to the image than if it were large but if we are talking about the information each object conveys to the scene then we can say that it is different but not less important. This is exemplified in figure 3.5. Both houses convey the information 'house with 5 windows and a door'. Each of them however convey the extra information 'close' for the bigger house and 'far' for the smaller one[2]. The conveyed information in each case is different but none is more beneficial than the other, that is, unless the image has a subpixel size or a little bigger, one can almost always recognize the features of the object. Nevertheless, if parts of the houses have texture maps then the details of the texture might only be observable on the closer one.

A benefit heuristic should also incorporate the fact that if some entity represents a group of objects then according to Gestalt psychology[9] the meaning conveyed by that representation may be more then merely the 'addition' of the meanings conveyed by each one of the objects alone, that is, the whole conveys more information then the sum of its parts. This can be exemplified in figure 3.6. Considered alone the objects in (a) are just objects but when positioned in a certain way and viewed as a whole as in (b) a new meaning is given to the picture.

[2]We do not immediately think that the first house is a real house whereas the second is just a miniature of a house.

Figure 3.5: The big and small houses convey the same information, i.e. 'house with 5 windows and a door'.

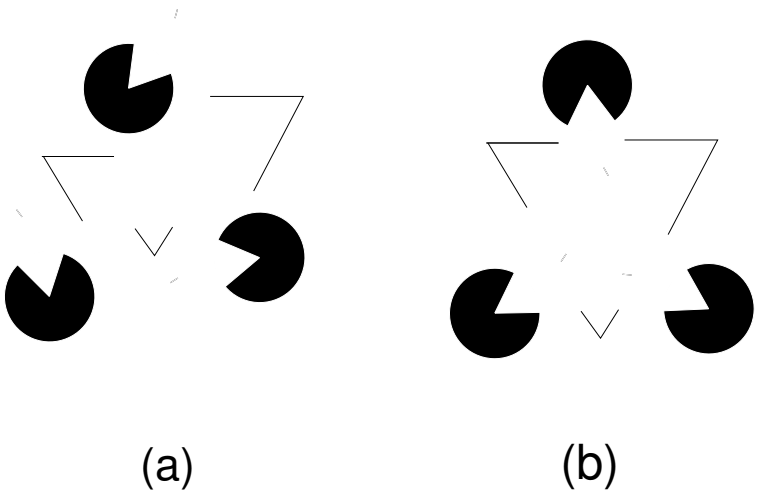

Figure 3.6: Subjective contour illusion from [9]. (a) Alone objects conveyng certain information. (b) Additional information is gained when we consider objects as a whole.

Although objects of different size contribute differently to the image though not more importantly, in an environment where the rendering time is at premium, it is reasonable to consider that the bigger object is more 'important' than the smaller one and therefore in our current implementation, the benefit heuristic takes into account only the distance of the object to the viewpoint[3] and display with more detail objects closer to it. Benefit of an object is currently just a real number.

Currently, the benefit of a cluster is just the greatest of all the benefits of the objects that it subsumes and we do not attempt to take into account the contribution to the perception of the model a cluster of objects provide to the entire scene.

3.4 Design of the Model Hierarchy

Given a reasonably complex(a few millions of polygons) 3D model we wanted to design a data structure that would enable a walk-through program to navigate in this environment in real-time(at least 10 frames per second) providing the user with a "reasonable" image of the environment at each frame.

In order to achieve that, the data structure that represents the entire model needs to have some properties:

1. Similar to what is done in techniques for realistic image rendering, such as ray-tracing, the object space of the entire model needs to be recursively partitioned in increasingly small cells containing fewer objects so that when the viewing frustum does not intersect a large cell, all the geometry inside that cell can be discarded.

2. The representation should be flexible enough to allow different models to be combined to form an arbitrarily large model.

3. It needs to support objects at different LODs since this allows the application to render the most "noticeable" features of the model opposed to all of them which would require a prohibitively rendering time.

4. Since we are interested in keeping a uniform frame rate regardless of the complexity of the model and viewpoint, the structure must be searched in real-time taking into account an estimated rendering time for each object in the model, until the user specified rendering time per frame is reached. When this occurs, all the elements found in this search need to be sent to the graphics engine to be rendered. A consequence of this is that at each given step of the search the list of objects to be displayed need to correspond to a 'as faithful as possible' representation of the model.

3.4.1 Tree Structure

Property 1 above, suggests a hierarchical tree structure in which the top level cell would contain a representation of the whole model and the leaves would contain the actual objects in the model. At each intermediate level, each node will have children that correspond either to a spatial division of its parent or to non-overlaping sets of object whose union is represented by the parent node.

This representation also complies with property 2, since the root of a model hierarchy can be combined with another by connecting them to a single root node that would represent a much larger model that could be further combined with other databases.

[3] More specifically, the square of the distance from the viewpoint to the closest point in the objects bounding box.

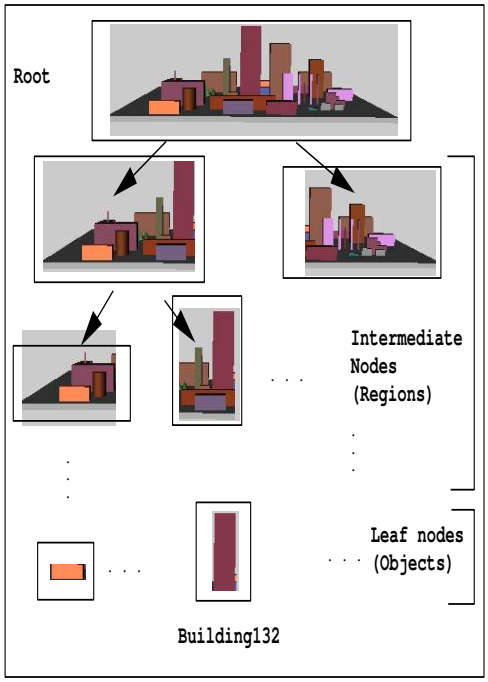

Figure 3.7: Currently implemented model hierarchy for a city. Here it is assumed that the representation at each parent node cost less to render then the sum of its children.

The third property, extends this representation by allowing the leaves of the tree to contain a hierarchy of LODs for the objects they represent.

Property 4 places a restriction on the amount of model information contained in each node of the tree structure. It states that, each node in the tree needs to be a drawable representation of all its descendant nodes. It also implicitly states that each node is a complete representation of a portion of the model and that its root is the coarsest representation of the entire model while the leaves correspond to the best possible representation of it.

In order to selectively choose a node to render, its rendering cost has to be strictly less than the sum of the rendering cost of its children. Tree nodes need to have an inexpensive to draw representation of the model that looks ok when it is being viewed from a reasonable distance(relative to its size).

Ultimately, this structure is a representation of the whole model containing coarse representations of groups of objects, and objects at different levels of detail and is therefore a generalization of the LOD approach used in previous approaches. In this way, we can select the representation that best fits the image quality requirements and the amount of rendering time we have available at each frame. Figure 3.7 presents the model hierarchy for a city.

3.4.2 Tree Node Contents

In the current implementation, each tree node has a drawable representation of a set of objects(or cluster) that it represents together with an associated rendering cost. Leaf nodes represent the conceptual objects at different LODs and have a benefit value associated to them. The root node and intermediate nodes represent non-overlaping sets of objects and do not currently have associated benefit values but instead they inherit the benefit values of the actual objects that they represent.

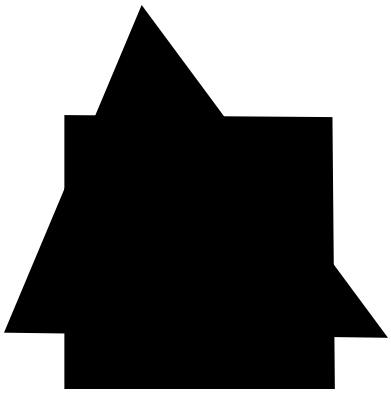

Figure 3.8: This picture is interpreted as a square overlapping a triangle(or vice-versa) and not some complex figure.

A drawable representation of a set of objects is merely an imposter for this set that as accurately as possible represents those objects on the screen with a lesser rendering cost.

We have tried 2 types of imposters for a set of objects. The first attempted was just a box with a color in each face that corresponds to the average color of the set of objects when they were viewed from each side left,right, front,back,top and bottom. This average color was obtained by orthographically projecting each face of the object onto the corresponding faces of its bounding box. This representation was the first to be attempted because it was easy to compute and its rendering cost is very small(compared, for instances, with the cost of rendering primitives such as spheres and cylinders) and we were hoping that it would only be selected to be rendered when very far from the viewpoint.

The second imposter tried makes use of the texture mapping hardware of the Reality Engine. Instead of computing the average color of each of the object's bounding box faces, the image of the projection of the object onto the screen was grabbed. A total of six texture maps are computed for each box corresponding to a set of objects. In addition to have a set of objects being represented by a box with the corresponding texture maps pasted onto its faces, another LOD was added to each geometric object in the model. In this case the lowest LOD of an object is just its bounding box with the appropriate textures on its faces.

3.4.3 Laws of Organization and Hierarchy Building

Up to now, we have only talked about the organization of the tree structure and what kind of information should its node contain but nothing has been said about the criteria used to group objects into clusters of objects represented by tree nodes.

It is worth taking a look into how humans tend to organize and perceive groups of objects. Gestalt psychology can give us some insights on this subject. In order to determine how your perception will be given certain stimulus conditions, psychologists came up with rules that they called "laws of organization"(see [9]). There are four of them: Simplicity, Similarity, Nearness and Good Continuation.

The first one, simplicity, says that every stimulus pattern is seen in such a way that the resulting structure is as simple as possible. For instances, figure 3.8 is perceived as a triangle overlaping a square and not as a 13 sided polygon.

The second is the similarity law, that states that similar things appear to be grouped together. In figure 3.9, circles appear to be grouped with other circles and squares are grouped with other

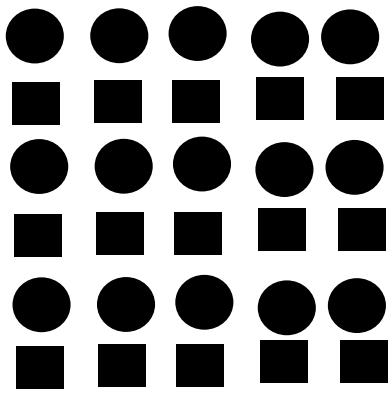

Figure 3.9: Figure perceived as alternating rows of circles and squares.

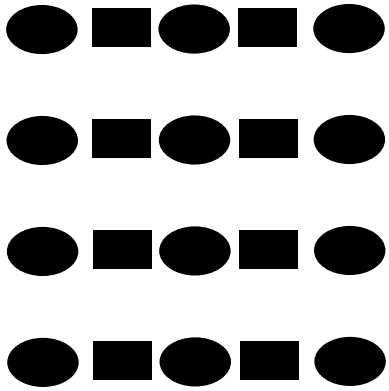

Figure 3.10: Figure perceived as rows of alternating circles and squares.

squares.

The third is the law of nearness, which states that objects near to each other appear to be grouped together. In figure 3.10 we tend to perceive rows of alternating circles and squares, because they are close to each other appear to be grouped in the same row.

Good continuation is the fourth of these laws and it states that points that form a straight or smooth contour when connected to each other seem to belong together. Lines tend to be seen in such a way as to follow the smoothest path as illustrated on figure 3.11.

Schifman [17] generalizes this law by saying that it is a special case of the general configuration principle that states an organizing tendency that encompasses other characteristics such as common fate, closure and symmetry. The common fate characteristic tends to group together objects that move in the same direction while closure and symmetry favors the perception of a complete figure instead of parts of it and the perception of symmetric objects, respectively.

Our current implementation only takes into account the third law[4] and our clustering program tend to cluster together nearby objects. Initially, it gets the bounding box of the entire model and divides it into a pre-determined number of cells of a regular grid. Next, starting with the smallest cell size, for each cell of this size in the model it checks what objects fall completely inside it and builds a bounding box for this set of objects that has on each face a texture map of the scene seen from a view point perpendicular to each face of the box in an orthographic projection. After all

[4]Other laws can also be considered although their use is not as straightforward as the nearness law.

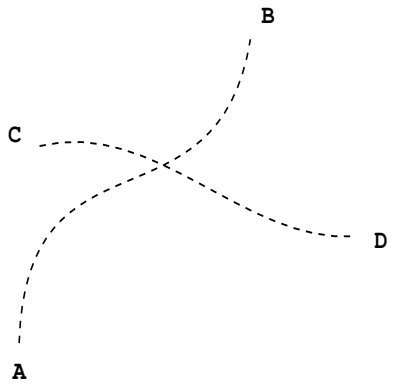

Figure 3.11: Figure is perceived as 2 intersecting curves AB and CD.

the cells of a certain size are checked and the objects completely within grouped together we loop again with a larger cell size until we obtain a cell size that has the same size as the bounding box of the entire model. If the smallest cell size is small enough, then we start grouping objects that are very close together.

Also, in order to reduce the error inherent in the grouping process, at each time we cluster objects we always take into account the actual objects that belong to the cell and not other clusters that might already been computed for a smaller enclosing cell.

To avoid duplicate nodes in the model hierarchy, whenever a large cell encloses only one cluster that has already been computed, before computing the cluster for that cell, we check if the bounding box of the objects in the cell is already a cluster.

The good thing about this process is that given an arbitrary model, the clustering program automatically builds the entire model hierarchy without human assistance.

3.5 Attempted Search Strategies

As mentioned in previous sections, we needed a fast algorithm that quickly(in less than the frame time)descends the model hierarchy selecting nodes to be rendered and accumulating rendering cost until the user-specified frame time is reached. When this occurs, the algorithm should stop and send the computed list to be rendered to the graphics pipeline.

Given the model hierarchy described in the previous section, we've experimented with 3 strategies that search the tree structure in a depth-first and best-first manners and a variation of the best-first strategy that pre-computes visibility, cost and benefit for the tree nodes prior to the actual search.

3.5.1 Depth-First Strategy

It seemed natural that this search should be guided by the benefit heuristic, since we wanted to render the most beneficial objects first. Therefore, the first strategy attempted was a top-down one in a depth-first manner.

Each node of the tree was either a bounding box of the objects inside it with equivalent average colors on its faces as in figure 3.12 or a geometric object at different LODs, and had a rendering cost and a benefit value associated to it. The rendering cost was an estimate of the time needed to render a box and the time required to render each LOD of each object. The benefit value of the

node was initially just the squared distance from the center of the bounding box of the object(s) to the viewpoint.

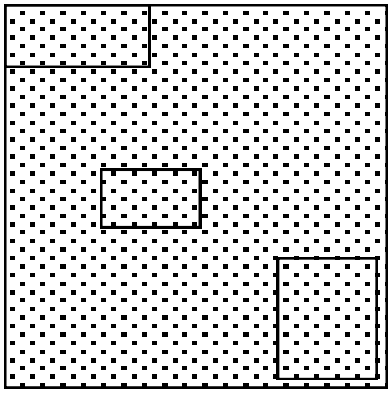

Figure 3.12: The color of the box's top face is the average color of the top face of all the objects it represents.

This initial metric didn't work since for large clusters there were cases where although the viewpoint was close to one of the boxes(cluster) its benefit value was very low since the center of the cluster was still far away from the viewpoint. The benefit of the cluster was then changed to be the square of the distance of the closest box vertex to the viewpoint, which to a certain extent solved this problem. Figure 3.13 illustrates both cases. In this figure, using the distance of the center of the box to the viewpoint will assign a low benefit to the box even though there is an object very close to the viewpoint.

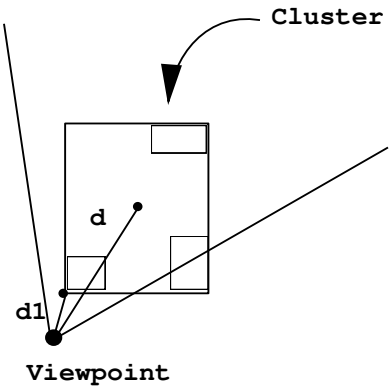

Figure 3.13: Two ways of measuring the distance of a box to the viewpoint. 'd1' is the distance to the center of the box while 'd2' is the distance to the closest box vertex.

The algorithm is recursive and works by examining the tree nodes, accumulating the rendering cost until the frame time is reached in the following way. Given a tree node we check if it is a cluster or an object. If it is a cluster we check it against the viewing frustum and subtract its cost from the total cost and exit the routine if it is not visible. If it is visible, we compute the benefit and cost for its children and check if the total cost plus the cost of the cluster's children is less than the frametime. In the negative case, we add the cluster to the list of objects to be rendered and return. Otherwise, we add the cost of the cluster's children to the total cost and call the algorithm again for each one of the cluster's child. If the node represents an object then we

```
Routine DepthFirstStrategy(node)
{
  if (node is a cluster) {
    if (node is visible) {
      Compute the cost of the node's children
      if (cost of children + total cost <= FRAMETIME)
        add cost of children to total cost
      else
        add cluster to the rendering list and return
    }
    else // Node is not visible.
      subtract the cost of the cluster from the total cost and return
    for(all the cluster's children)
      DepthFirstStrategy(child);
  }
  // Node is an object
  if (object is not visible) return
  add object to the rendering list and return
}
```

Figure 3.14: Pseudo-code for depth-first strategy

can either check its visibility or just add it to the rendering list and let the machine hardware do the culling. Figure 3.14 show the pseudo-code for the depth-first strategy.

Apart from the initial problem with the benefit heuristic, this approach turned out to be unsuccessful because more often then not the algorithm had to decide between two big clusters which to search first and if their benefits(the distances to the closest vertex of each bounding box to the viewpoint) were similar then because of the depth-first manner the algorithm would spend almost the entire rendering time in that branch leaving almost no rendering time for the other cluster which in theory should have received almost the same attention of the algorithm as the cluster that was actually selected.

Despite the above problems with this algorithm it served to show that the predictive approach works and to establish a path to a more successful algorithm, that is presented next.

3.5.2 Best-First Strategy

From the previous experience it seemed natural that at each moment in the search process the most promising node should be searched first. Using the same cost and benefit metrics of the previous algorithm, and using the same idea of accumulating rendering cost and using it to interrupt the search we've implemented a best-first variation of the previous algorithm.

This algorithm is not recursive and instead it uses a list of nodes ordered by their benefits and works as follows: it begins by assigning cost and benefit to the root node and inserting it to the list. It then executes the following loop until the total cost reaches the specified frame time or the list is empty. At each pass of the loop it removes the most beneficial node from the list and check

```
Routine BestFirstStrategy(node)
{
  Compute node's cost and benefit.
  Insert node into list.
  while ( list is not empty and total cost < FRAMETIME) {
        Remove node from list
        if (node is a cluster) {
          Compute the cost and benefit for all the node's visible children.
          if (child is an object) Compute its LOD
          if (total cost + cost of children < FRAMETIME) {
            Add cost of children to the total cost.
            Insert all visible child into the list and loop.
          }
        }
        // At this point the node is either a cluster that can not be
        // subdivided or it is an object.
        Insert node into the rendering list.
  }
  // If total cost has reached the frame time but list is not empty.
  while (list is not empty) {
        Remove node from list.
        Add node to the rendering list.
  }
}
```

Figure 3.15: Pseudo-code for the best-first strategy

its type. If the node is a cluster for all of its visible children it assigns cost, benefit and LOD(in case the child is an object) to the child and cost of the cluster's visible children. Then it compares the total cost plus the cost of the cluster's children with the frame time. If it is less then the frame time it adds the cost of the children to the total cost and inserts each child in the list, otherwise it inserts the cluster in the rendering list. If the node is an object then we optionally test for visibility and insert the object in the rendering list. The pseudo-code for the best-first strategy is given in figure 3.15.

This new algorithm although a little slower than the previous one, solves the problem of spending too much time exploring a branch of the tree and disregarding a branch with similar benefit and works reasonably well.

Due to the fact that the visibility check is done very often it had to be fast to compute. Currently, we check if the bounding box of the cluster in eye-coordinates intercepts the viewing frustum. This simple test can yield false positives in cases such as the one depicted in figure 3.16 and decide that the cluster is visible when none of its children are. In this case we are committing rendering time with a cluster that will never get rendered and thus preventing other visible objects to be selected for rendering. This problem is solved in the last implementation described.

This algorithm works reasonably well because usually the closest vertex of the cluster is also

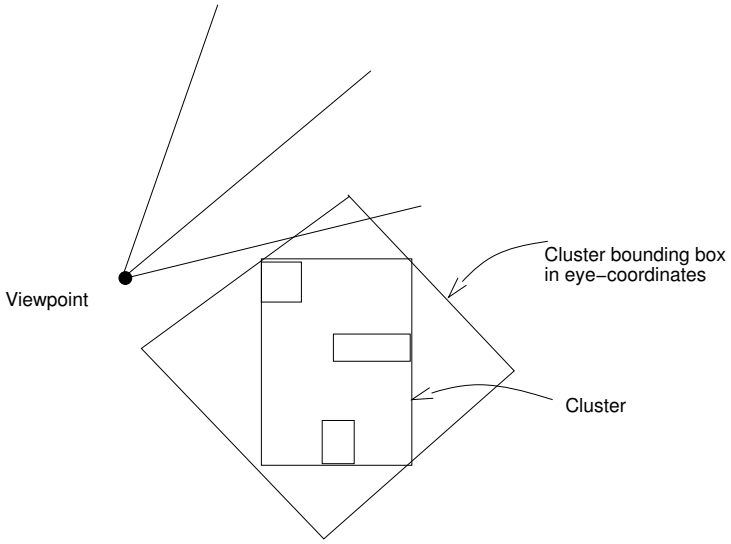

Figure 3.16: Visibility test yielding a false positive.

most of the time a vertex of the bounding box of the closest object inside the cluster. Figure 3.17 shows a cluster with 3 objects inside where the distance of the closest object inside the cluster to the viewpoint 'd2' is greater then the distance of the cluster(its closes vertex) to the viewpoint 'd1'. In this case, the algorithm fails to display at full resolution an object close to the viewpoint. Also, when we are examining an object it makes sense to attribute a high benefit to objects close to the view point, but for clusters what does its distance to the viewpoint mean ?

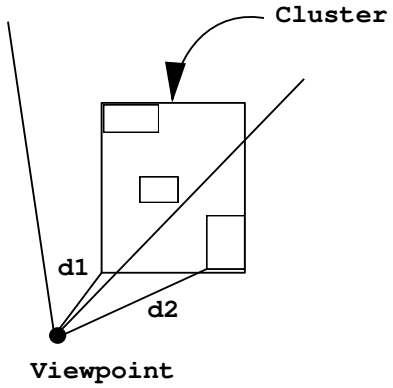

Figure 3.17: Distance of a cluster to the viewpoint.

The main problem with both algorithms presented however, is that what is guiding the search at the top level nodes of the hierarchy is actually the benefit of a cluster which doesn't actually reflect how much the rendering of the cluster contributes to the perception of the model.

We need a heuristic to decide which branch of the tree structure to search first and to define what is the actual contribution to model perception that the rendering of the cluster is giving to the simulation.

In the algorithm presented in the next section we tried to address the issue of deciding which branch to search first and left the benefit of the cluster issue to be discussed in section 4.

```
Routine AssignBenefitCostVisibility(node)
{
 if (node is invisible) Record this and return.
 if (node is an object)
    Assign cost, benefit and lod to the node and return.
 Assign cost to the node.
 for ( each children of the node ) {
     AssignBenefitCostVisibility(child);
     if (child is visible) {
        Update the visibility of the node.
        Update the benefit of the node to be the maximum between the
        computed benefit and the benefit of the child.
        Add cost of child to the cost of the node's children.
     }
 }
}
```

Figure 3.18: Pseudo-code for assigning benefit, cost and visibility to tree nodes

3.5.3 Two-Stage Best-First Strategy

In order to search the branch of the tree that would result in the most beneficial objects to be found first we need some way of making the benefit of the objects available to the intermediate nodes in the model hierarchy. This can be done if we split the search algorithm in two steps. In the first step we recursively descend the model hierarchy and assign visibility, cost and benefit to each node in the hierarchy. In the subsequent step we use this information to compute a list of objects/clusters to be rendered.

The first algorithm, for a given node it initially checks if its visibility. If the node is visible and is an object then it just assigns visibility, cost and benefit to the object and returns. Otherwise, it assigns to the node its cost, visibility and makes its benefit to be the maximum of all the benefit of its children. In this approach, the benefit of the cluster is equal to the benefit of its most beneficial object that the cluster subsumes. The pseudo-code for assigning benefit, cost and visibility to a tree node is given in 3.18.

The reason why we need to update the cluster's visibility when examining its children is to solve the visibility problem mentioned in the previous algorithms. Here, a cluster is visible if and only if at least one of its children is.

In the second stage, we make use of an ordered list and the information obtained in the previous stage to implement an efficient best first search. The algorithm is similar to the previous best-first algorithm and its pseudo-code is given in figure 3.19.

Fortunately, these two algorithms combined don't take much more time than the best-first algorithm alone since the later too needs to compute for each node, its visibility, cost and benefit.

```
Routine EfficientBestFirstStrategy(node)
{
  Insert node into list.
  while ( list is not empty and total cost < FRAMETIME) {
        Remove node from list
        if (node is not visible) continue.
        if (total cost + cost of children < FRAMETIME) {
          Add cost of children to the total cost.
          for (all of the node's visible children) {
              if (child is an object)
                   Insert child into the rendering list.
              else Insert child into the list.
          }
        }
        else
        // At this point the node is a cluster that can not be subdivided
        Insert node into the rendering list.
  }
  // If total cost has reached the frame time but list is not empty.
  while (list is not empty) {
        Remove node from list.
        Add node to the rendering list.
  }
}
```

Figure 3.19: Pseudo-code for the efficient best-first strategy

3.6 Discussion of the Problems with Current Approach

In this section we briefly mention the problems encountered during this research concerning the building of the model hierarchy and the real-time walk-through of the model.

The most important issue in building the model hierarchy is what representation to use for a group of objects. The initial idea of using a box with an equivalent color to represent the cluster has several drawbacks:

1. A box representation for a group of objects is only effective if its very distant to the view point and its projection on the screen covers a couple of pixels only. More often then not the clusters are selected to be rendered when they are still close to the viewpoint.

2. Representing two or more objects far apart by a box is not a good representation since the box fills in the empty spaces among the objects. Also the volume of a bounding box tends to be bigger than the actual object(s) inside it.

3. A box is not view independent(as a sphere is) and look different depending on the viewing angle and therefore it is a good representation only for box like objects.

Because the above problems the box with average colors representation was discarded except in the cases where it covers just a few pixel on the screen. Texture mapping the faces of the box with images corresponding to its sides, greatly improves the appearance of the model and to some extent solves the problem mentioned in 2 above. Unfortunately, there are problems with texture maps also:

1. In theory, the amount of texture memory that can be allocated to a single process is the amount of virtual memory allocated to it. In practice however, the machine has a limited amount of physical texture memory(4Mb for the Reality Engine) and therefore the total amount of texture memory should not exceed the machines texture memory limit. If the programed attempts to access a texture not in the texture memory it either will have to copy the texture from its main memory or bring it from disk, in the later case the frame rate will be greatly disturbed. This poses a constraint in the number of nodes of that the model hierarchy can have.

2. A texture mapped box representation is also not view independent and if the box is seen from its diagonal then we see 3 images of the same object/cluster instead of just one.

Despite the problems with texture maps, current hardwares are very optimized for them and can display filtered textures very efficiently and therefore instead of just using boxes with average colors we currently use boxes with texture maps pasted on their faces. These texture maps use transparency so that we can see through open spaces in the boxes.

Since we can substitute geometrically complex objects by texture maps with very good accuracy and efficiency it seems that the impostures that will be used in the final system will somehow make use of them.

With respect to the tree traversal the main problem is the fact that if the tree is too big, since at each step of the algorithm we commit portion of the frame time to the tree's node then it might be the case where when we reach the bottom of the tree we don't have enough rendering time left to render the object at the LOD it would need to be rendered according to its benefit. The solution to this would be to keep the depth of the tree small. If the depth of the tree is too small however,

clusters will be big and may not look okay. Besides that we need to decide whether to cull the objects contained in the cluster against the viewing frustum thereby increasing the computation time of the algorithm or let the graphics hardware do the culling in this case increasing the rendering time.

One other minor problem is some instability that the program presents when the user navigates among simple objects(just a few hundreds of polygons) and more complex objects(a few thousands of polygons). If the user is close to a simple object then the program has more time to render objects at the back then when it is close to a complex objects. As the user navigates in the environment, in one frame it can be closer to a simple object and a few frames later be close to a complex object. When this happens, clusters tend to appear and disappear from the scene causing a somewhat annoying switch of representations. While this is a common problem in systems that use LODs[8] to represent objects this problem can be eliminated by taking advantage of the temporal coherence that exists from frame to frame.

One final issue relates to a limitation of the system that is tied to the hardware in which the program runs. In order to guarantee that objects close to the view point or more generally speaking, with high benefit value, are displayed at their full resolution, the rendering complexity of the objects in the model can not exceed the machine's rendering capability. For instance, if the machine is able to draw 1 million polygons per second, if the simulation is to run at 10 frames per second then it has to draw 100,000 polygons per frame. Objects with a complexity greater than this will be represented by imposters even if the observer comes very close to them.

3.7 Implementation Details

This research has resulted in the implementation of 3 distinct programs, one for creating the model hierarchy, one for measuring the cost of the objects in the model and the interactive walk-through programs. All programs are implemented in C++ and make GL[19] calls to render objects in the scene. The walk-through program also has an X-Motif user interface with buttons, fields and slider to facilitate testing and evaluation of the simulation.

The program that builds the model hierarchy opens two windows. The first displays the objects/clusters and compute the texture maps for each of their sides and the second one shows the process of building the hierarchy.

In the first window, objects are orthographically projected and surrounded by a square that delimits the bounding box of that projection and the image inside the square is grabbed as the texture map for that object/cluster's face. The second window shows each grid cell, the objects that are inside it and the bounding box which is the result of grouping objects inside the cell. After the hierarchy is built each of its level is displayed and the hierarchy saved on disk.

The second program loads the model objects and display then on the screen a certain number of times to measure its rendering cost. It then saves these timings that will be loaded in by the walk-through program.

The third one reads one control file that causes five files to be read in order to produce an interactive walk-through of the environment. The first file contains the objects that are used in the simulation together with the number of LODs for the object and the respective rendering times that are loaded into a table that is accessed by the program in real-time. The second file positions each one of the objects inside the environment according to a translation vector and a rotation angle. The third file contains the model hierarchy built by the program. These three files are ascii files. The last two files are binary files containing the textures corresponding to the objects and the clusters.

This program displays a GL window inside an X Window so that the user can by using the mouse navigate inside the environment. Figure 3.20 shows what the interface of the walk-through program looks like. The top view shows a scene that is rendered using the model hierarchy and objects at different LODs. The bottom one is a rendering of the same scene withouth using the model hierarchy. Note that although the top image differs from the true image of the scene on the bottom, we are able to obtain interactive frame rates when the model hierarchy is used.

By filling the frame rate field with a 2 digit number, the user can interactively specify the desired frame rate for the simulation. With a slider the user can adjust the rendering time to account for the overhead of drawing the primitives since the rendering cost computed by the second program is the cost of actually rendering the objects and does not take into account the actual overhead of drawing them. By pressing a button we can activate/deactivate the use of the model hierarchy and therefore we can compare the performance and characteristics of using our approach oposed to having a simple flat structure that is rendered without a more complex processing except of the culling the object against the viewing frustum. The fields display the number of polygons that are being drawn and frames per second, the maximum and minimum number of polygons in the model, if all the objects were rendered at their lowest and maximum LODs, respectively, and how much time the machine is spending searching the model hierarchy and how much time it is spending drawing each frame. Some of these features can also be toggled by pressing the appropriate key when the cursor is inside the GL window. By pressing the 't' key the program shows a top view of the environment and we can actually see what clusters are being displayed at each frame.

In the next section we outline a framework of ideas that will be explored in order to eliminate the problems pointed out in 3.6 above which we hope will make the current implementation more realistic and effective.

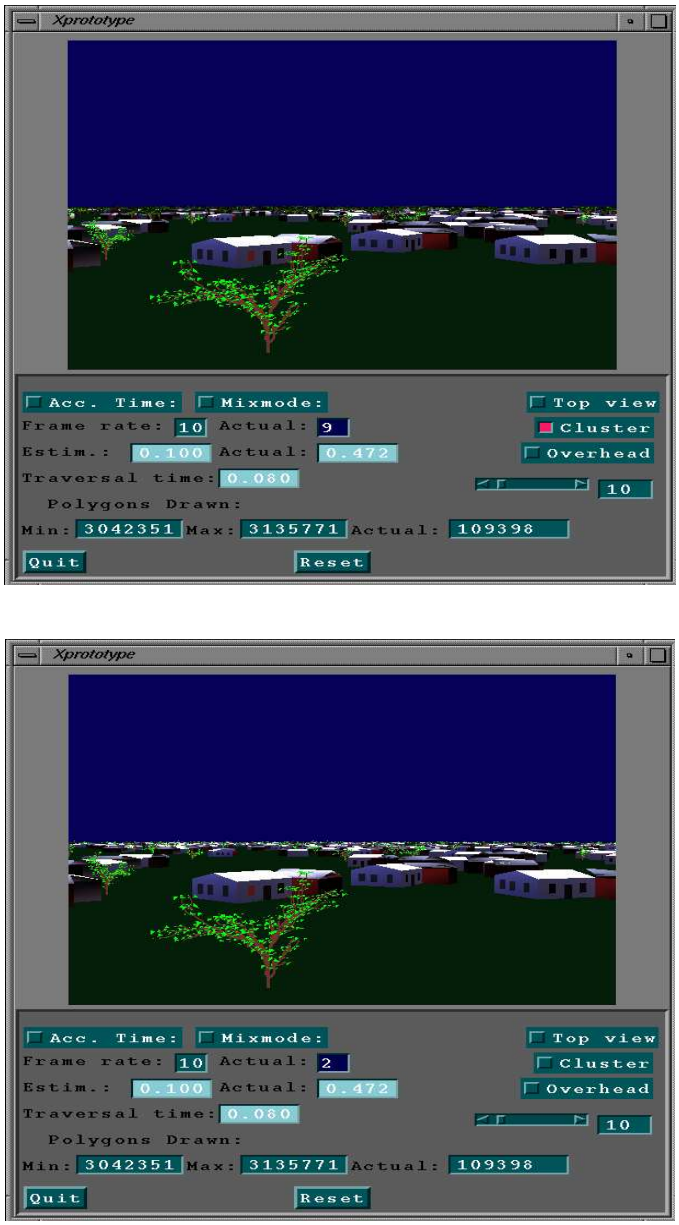

Figure 3.20: X interface of the walk-through program. The top image uses the model hierarchy whereas the bottom displays a flat list of objects in the model.

Chapter 4

Future Research

The aim of the walk-through program is to find the perceptually best representation of the model at each given frame subject to a drawing time constraint. Three problems with the current implementation are preventing that from happening.

1. We lack a general framework for the model hierarchy that enable particular solutions to be implemented and tested.

2. We currently do not cope with the view dependent nature of objects and clusters of objects. The projected image of objects onto the screen change as the observer moves through the environment, and as mentioned before, in the the current implementation, imposters are just bounding boxes with texture maps, that correspond to the images that an observer sees when its line of sight is perpendicular to each face of the box, pasted onto each side. This imposter representation look well for some kinds of objects such as box-like objects, but it works poorly for many objects particularly when they are looked at from the box's diagonal. Therefore it is imperative that our tree nodes contain view dependent representations, that is, representations that are appropriate according to the viewpoint.

3. Lack of an heuristic to determine what is the actual benefit of an imposter representation of a set of objects. The reason why such a benefit heuristic is important stems from the fact that if we are able to decide that a cluster representation is a good enough representation at that particular point in the simulation and because it costs less to render then the actual objects, the walk-through program can gain some rendering time by not bothering to render better and more costly representations for the objects in the cluster.

In the next section we develop a general framework in which particular solution paths can be explored. We then outline possible implementations of two of the functions defined in the framework that will attempt to solve the current problems.

4.1 Model Hierarchy Framework

If we recall Section 3.4.2 above, we are currently making a distinction between conceptual objects at different LODs and cluster of objects and we do not even provide for representations of sets of objects that are just a set of hardware renderable primitives. This is an unnecessary restriction to the model hierarchy. In the formalization of the problem in Section 3.1 it was stated that our tree node just need to contain some hardware renderable representation of a set of objects in the model.

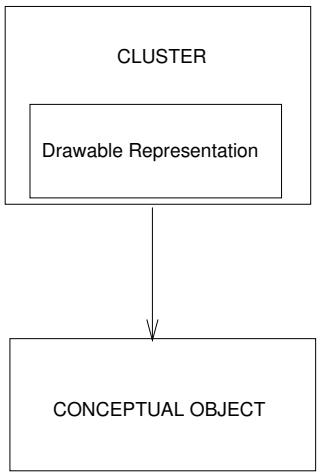

Figure 4.1: Main abstraction.

Therefore, given a 3D model for our approach to work we just need to build a general tree structures from it in which each node is a representation of its children but costs less to render then the sum of the cost of each child.

This requires an abstract design for our model hierarchy, i.e. we need to define a collection of abstract classes that are suitable for our particular walk-through application. More specifically, we will design a black-box framework as in [14, 2] in which we will specify the main abstractions together with their interfaces which will provide the desired behaviour of our application. Figure 4.1 shows the two main abstractions of our framework.

In this framework the cluster is an abstraction for any simplified representation of a set of objects, i.e. any hardware drawable representation that costs less to draw than all of its children in the model hierarchy, while the conceptual object is some object that has a conceptual meaning for the application like a car, a building, a person and so on. A conceptual object does not have children although it may contain simplified versions of the same object.

The interfaces for each of these classes are given in Table 4.1.

Both abstractions share the following information:

- An array of pointers to hardware drawable representations, that contain possible representation for clusters.
- The number of the selected representation to render in the array of hardware drawables. During the walk-through phase and based on the Cost and Benefit associated to each representation in the array the best representation for the particular viewpoint will be marked for rendering.
- The number of the last representation rendered and the number of the frame in which it was rendered. This is an attempt to make use of temporal coherence in the simulation based on the assumption that the scene does not change too much from frame to frame and that switching representations during the simulation is somewhat distracting.

A conceptual object share all this information with a cluster but is more specialized since it contains information about the name of the object as well as its position and orientation in the model.

Abstraction	Interface			
	Name	Description	Input	Data
Cluster	Draw(vp)	Displays a representation of a selected cluster on the screen.	Viewpoint	Array of hardware renderable representations, the current frame number, the number of the last rendered representation and its frame number.
	Cost(vp)	Computes the cost of rendering a cluster representation.	Viewpoint	
	Benefit(vp,frame)	Computes the contribution to image quality of a cluster representation. This is a virtual function, that is redefined by Conceptual Object.	Viewpoint and the current frame number.	
	Visibility(vp)	Checks the cluster against the viewing frustum and marks it as invisible if they do not intersect.	Viewpoint	
Conceptual Object	Benefit(vp,frame)	A redefinition of the cluster benefit that takes the meaning of the conceptual object in the simulation into account.	Viewpoint and frame number.	Name, position and orientation
	Draw(vp), Cost(vp), Visibility(vp)	Inherited from cluster.	Viewpoint	

Table 4.1: Main Abstraction definition

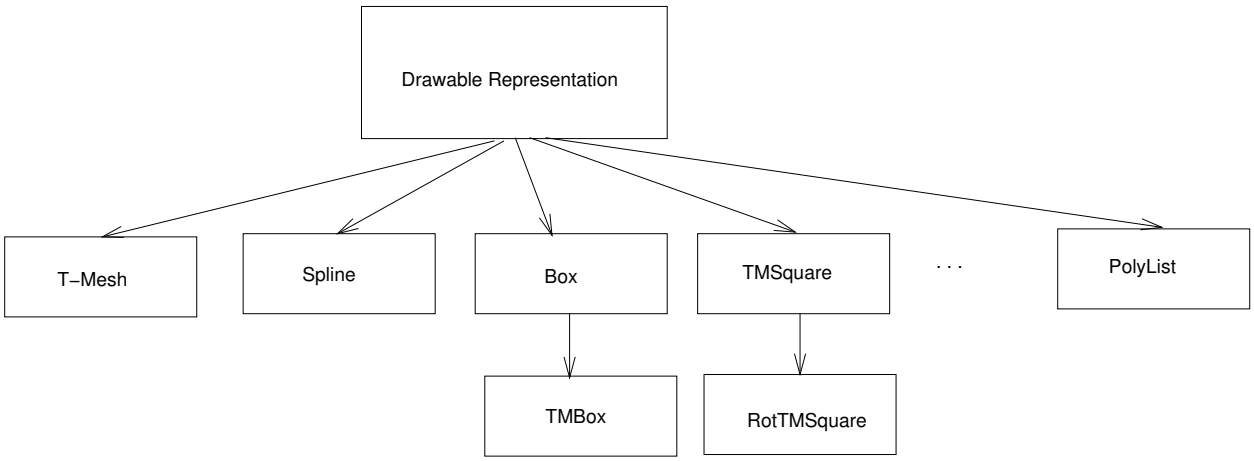

Figure 4.2: Abstraction for Hardware Renderable Objects.

In this framework a model hierarchy 'M' is a generic tree structure that can recursively be defined by the following rules:

1. A cluster that has no children is a model hierarchy with just one root.

2. Let $C_1, C_2 ... C_n$ be model hierarchies whose root nodes are the clusters $c_1, c_2 ... c_n$, respectively, that represent sets of drawable entities that do not overlap in object space. Let 'c' be a cluster that represents the union of c_i. 'C' is defined to be a model hierarchy if 'c' is the parent of c_i for $i = 1..n$.

Also, note that no restriction is being placed on to how these clusters are generated. Thus, they can be automatically generated by some clustering program or manually by the user. They just need to be representations that can be rendered by the hardware that the program will run on.

The taxonomy of hardware drawables mentioned above is depicted in figure 4.2 and described in Table 4.2.

The hardware drawable representation interface is composed of a single virtual draw function that is appropriately redefined according to its subclasses. All subclasses share a rendering algorithm variable which is used to select a representation and can be set to account for flat and Gouraud shading, hardware lighting, anti-aliased and texture mapping representations.

The subclasses depicted in the figure are either hardware primitives such as t-meshes and splines or constructs that use hardware primitives in a certain way such as a box, a texture mapped square and a list of polygons. Other classes may be added to this design as deemed necessary to solve a particular problem or to add a particular feature to the walk-through program. Two of these classes: TMBox and RotTMSquare will be described in more detail in section 4.2.1 and are attempts to solve problems with our current implementation.

To complete our framework we define auxiliary abstractions that will make the program more general by allowing different drawable representations to be input to it. These classes are depicted in figure 4.3. In this design we will have a class for each modeling software available(or at least the most important ones) that will basically contain a function to 'Read' a certain format and another that will 'Convert' that format to a list of hardware renderable primitives. Each of these classes contain a pointer to a hardware drawable representation. With this design it becomes possible the off line generation of simplified versions of clusters by artists without having to rely on more crude

Abstraction	Interface			
	Name	Description	Input	Data
DrawableRep	Draw(vp)	Displays a representation of a set of objects on the screen.	Viewpoint	Rendering Algorithm
T-Mesh	Draw(vp)	Draws a t-mesh.	Viewpoint	Array of points
Spline	Draw(vp)	Draws a NURB based on control points and weights	Viewpoint	Array of control points and weights
Box	Draw(vp)	Draws a box	Viewpoint	Eight points and a color for each of the box's faces.
TMBox	Draw(vp)	Draws a box with a texture map pasted on each face	Viewpoint	Eight points and 6 pointers to texture maps
TMSquare	Draw(vp)	Draws a texture mapped square	Viewpoint	4 Points and a pointer to a texture map
RotTMSquare	Draw(vp)	Draws a rotating texture mapped square that follows the viewpoint as it moves	Viewpoint	Angle of rotation
PolyList	Draw(vp)	Draws a list of polygons	Viewpoint	List containing a list of points and colors

Table 4.2: Definition of the Hardware Drawable Abstractions.

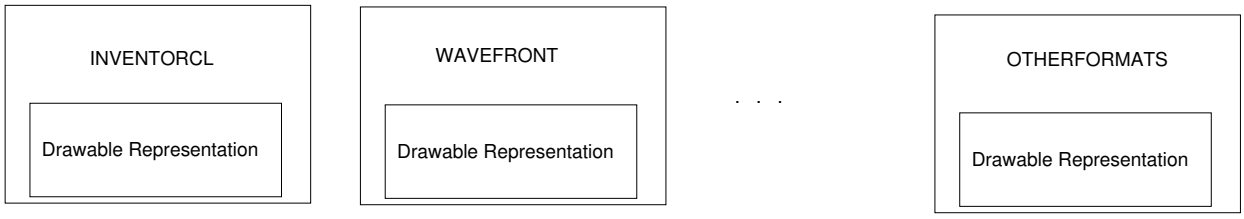

Figure 4.3: Abstractions for input models.

Conceptual Object	Shape
A building	Box-like
A planet	Spherical
A person	Cylindrical
A pine tree	Conic
A tree	Complex object with a symmetry axis
A bridge	Complex object

Table 4.3: Conceptual Objects.

automatic generation of clusters.

4.2 Object-Oriented Implementation

The approach described in the above framework makes the walk-through and hierarchy building programs suitable for object-oriented implementation. Each component of our abstract design will be made a C++ class with the appropriate interface.

In the next 2 sections, we describe a possible implementations for 2 of the virtual functions described namely, Draw and Benefit functions that explain the usage of the 'TMBox' and 'RotTM-Square' classes.

4.2.1 A Draw Function Implementation

For clusters that represent a single conceptual object and similar to what is done in computer vision[3] where objects have to be classified under certain parameters and stored in a database that is used to match seen objects in order to recognize them, given model objects can be stored and indexed by parameters such as shape and symmetry with respect to a main axis.

Table 4.3 is an attempt to classify conceptual objects according to some geometric properties:

Since the shapes of the objects in a simulation can be arbitrarily complex, we do not attempt to classify all of them and we leave a default type to represent arbitrary objects that do not fall into this classification.

Box-like objects that do not have holes can be drawn by texture mapping to each of its bounding box faces a view from each side of the object, i.e. they will be made instances of the 'TMBox' class.

Symmetrical objects, can have associated to them display information such as a texture map corresponding to a view of the tree that can be drawn by rotating the texture map so that it always face the observer, i.e. they will be made instances of the 'RotTMSquare' class.

4.2.2 A Benefit Function Implementation

Conceptual objects will have their benefit computed in the same way as described in Section 3.3.2 and as before it will continue to be used to guide the tree search so that the most beneficial objects can be rendered first.

To decide if a cluster representation is "good enough" however, we would need a perceptual model. We can develop a quantitative way of comparing the cluster representation of the objects with the actual objects that we hope will provide the simulation with good perceptual results.

Given an observer view position we want to determine if the imposter cluster representation is "good enough". This can be achieved by computing an error metric that we would obtain by comparing the image of the imposter for each viewpoint in space to the object(s) image at the corresponding position. This is not feasible since there are an infinite number of directions from which the observer could be looking at the object(s). Instead we will discretize the space of directions and get a measure of the discrepancy of the two images in each one of these directions. In this way we will build a 'table of differences' containing for each discretized direction a value measuring how much the imposter differs from the actual set of object(s). The imposter representation that falls within some specified threshold would be selected for rendering.

A simple error metric to decide how one image differs from another could be a simple pixel compare such that we could consider 2 pixels on each image being equal if their r, g and b components are within some pre-determined tolerance from each other.

Another more elaborate method would employ image analysis techniques in the following way: since the important features of an image have edges associated to them we would first pass an edge detector operator in each of them to obtain the contours of the objects in the image. The two images could then be segmented and finally compared to each other in order to get a measure of their discrepancy. Exactly how this whole process will be done is not totally clear at this point but techniques from the computer image/image processing literature could certainly be adapted to suit our purpose.

Therefore, for every imposter for the set of objects we will build these 'difftab' tables that show for each viewing position how the imposter image differs from the actual image. These 'difftab' tables will be pre-computed and then loaded into the walk-through program to be accessed in real-time.

By using these 'difftabs' we hope to select the best representation for a given viewpoint. As an example, consider a flight simulation where an airplane is flying at a considerable distance from the ground where a small town can be seen. Our cluster imposter representation selected to be rendered in this case could well be a big texture map representing the top view of this town. This representation would not differ much from the actual set of objects and yet would take much less time to render.

As yet another example, consider that the user is at some distance from a tree, it could be the case that a texture mapped version of it, for a particular distance to the viewpoint, is a "good enough" representation of the tree, and the program would not try to render the actual tree unless its benefit changes.

Chapter 5

Summary and Research Plan

In this report we have presented the description of a walkthrough prototype program for complex environments that enables the user to maintain a user-specified frame rate by selecting representations for sets of objects(clusters) that cost less to render than the actual objects that the cluster represents.

During the course of this description we explained why some strategies did not work and to which extent others did and stated the lessons learned in the development of the prototype. Finally we suggest ideas that will be tried to improve our results.

To complete this research I am planning the following activities:

1. Re-write my current code to implement the classes defined in the framework of Section 4.1 in C++. I also intend to implement the 'RotTMSquare' abstraction and possibly one of the auxiliary class formats.

2. Develop a view dependent method of comparing the images of a set of objects and its cluster representation. I will construct the 'difftabs' mentioned in Section 4.2.2 and use them in the walkthrough program.

3. Modify the current tree traversal algorithm to support the design specified in the new framework.

4. Add a temporal coherence mechanism to the traversal algorithm to avoid excessive 'popping'(frequent switching from one representation to another).

5. Since the system we are developing is specially good for outdoor environments[1] to enhance the realism of the simulation we will consider the sunlight. By adding one more button to our X interface we can turn hardware lighting and hardware shadows on and off.

With respect to number 2 above I anticipate that besides the fact that obtaining these discrepancy measures by using adapted image processing techniques will not be easy, it is difficult to predict how well they will work in our walkthrough program. Therefore, I am expecting this to be the most important unresolved issue in our research.

It is also difficult to estimate how much this new object-oriented framework will impact the performance of our prototype due to both inherent overhead of dynamically resolving references

[1] Depending on viewpoint in an outdoor environment there may be many more objects then can be rendered interactively and besides that we do not make use of visibility information from the environment like in a building in which objects in one room are only visible from certain rooms

to objects and to the addition of rendering actions to our objects(like rotating a tree to follow the viewpoint).

At the end of this research I hope to have built a system that allows the interactive navigation of a complex environment, by using a hierarchical version of it and with a 'reasonable' image quality, that could not otherwise be navigated in real-time if the model was just a flat list of objects.

I do not hope however, that this system works well for all possible environments containing any possible complex objects but with the framework described in the last chapter I hope that the system can be extended to provide the simulation with appropriate imposters for specific sets of objects.

Bibliography

[1] K. Akeley. Reality engine graphics. *Computer Graphics*, pages 109–116, 1993.

[2] P. K. Allen. A framework for implementing multi-sensor robotic tasks. *Proceedings of the Image Understanding Workshop*, 1:392–398, February 1987.

[3] D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[4] M. F. Deering. Making virtual reality more real: Experience with the virtual portal. *Graphics Interface*, pages 219–225, 1993.

[5] M. F. Deering and S. R. Nelson. Leo: A system for cost effective 3d shaded graphics. *Computer Graphics*, pages 101,108, July 1993.

[6] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[7] T. A. Funkhouser and C. H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environmnets. *Computer Graphics*, pages 247–254, July 1993.

[8] T. A. Funkhouser, C. H. Sequin, and S. Teller. Management of large amounts of data in interactive building walkthroughs. *Journal*, pages 11–20, 1992. Get name of the journal.

[9] E. B. Goldstein. *Sensation and Perception.* Wadsworth Publishing Co., Belmont, California, 1980.

[10] R. Hall, M. Bussan, P. Georgiades, and D. P. Greenberg. A testbed for architectural modelling. *Eurographics' 91*, pages 47–57, 91.

[11] C. B. Harrell and F. Fouladi. Graphics rendering architecture for a hight performance desktop workstation. *Computer Graphics*, pages 93–100, 1993.

[12] H. Hopper, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetsle. Surface reconstruction from unorganized points. *Computer Graphics*, pages 71–78, 1992.

[13] H. Hopper, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetsle. Mesh optimization. *Computer Graphics*, pages 19–26, 1993.

[14] R. E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, August 1991.

[15] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics*, pages 231–240, 1992.

[16] M. K. Ned Greene and G. Miller. Hierarchical z-buffer visibility. *Computer Graphics*, pages 231–238, 1993.

[17] H. R. Schiffman. *Sensation and Perception an Integrated Approach.* John Wiley & Sons, New York, 1990.

[18] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Co., Brown University, 1983.

[19] I. Silicon Graphics. *Graphics Library Programming Guide, Volumes I and II*, 1992.

[20] I. Silicon Graphics. *Graphics Library Programming Tools and Techniques*, 1992.

[21] I. Silicon Graphics. *IRIS Power C User's Guide*, 1993.

[22] I. Silicon Graphics. *IRIX System Programming Guide*, 1993.

[23] G. Turk. Re-tiling polygonal surfaces. *Computer Graphics*, pages 55–64, 1992.

[24] A. Watt and M. Watt. *Computer Graphics Animation and Rendering Techniques.* Addison-Wesley, first edition, 1992.

[25] J. K. Yan. Advances in computer-generated imagery for flight simulation. *IEEE Computer Graphics and Applications*, 5:37–51, August 1985.