VISUAL NAVIGATION OF LARGE ENVIRONMENTS

USING TEXTURED CLUSTERS

by

Paulo William Cardoso Maciel

April 1995

This dissertation is described here in less than 350 words using no footnotes, diagrams, references or outside anything.

Contents

List of Figures

List of Tables

1

Introduction

Computer-generated visual simulations are used in many areas such as flight simulation, building walkthroughs, computational fluid dynamics and video games, with the purpose of training, evaluation, education, and entertainment.

Since current general-purpose graphics workstations allow the interactive display of tens of thousands[1] of 3D polygons [2], these simulations have become more common and accessible to a variety of users such as scientists and educators.

Visual simulations will ultimately reach mainstream users at their homes using the information superhighway [17] and interactive television [8]. As graphics performance increases and its cost decreases, the new generation of users will demand more complex and more realistic animations. Such animations will require real-time performance at approximately constant high frame rates so that the user has the feeling of actual immersion in the virtual

---

[1]The Freedom Series of graphics accelerators from Evans & Sutherland can display up to 4 million meshed triangles per second and are available on many general purpose workstations.

world [32, 6]. User requirements will, for the foreseeable future, be ahead of what can be delivered by graphics workstations and therefore software solutions that extract the maximum performance from the still very expensive graphics subsystems should be explored [6].

In this dissertation we describe the design of a system for a particular kind of visual simulation, namely, visual navigation of complex environments. With this system a user can navigate inside a virtual environment by using a device such as a mouse. An example application for such a system is the construction and commercialization of a condominium complex. Such a system is useful in: planning the area to be built, understanding the effect it will have on people living in it, and in allowing its exploration by potential buyers.

The main issue regarding a visual navigation system is that its effectiveness is intimately connected to two key concepts: the realism with which it portrays the environment and the way the user interacts with it.

"Realism" here is used in the sense of Chiu et al. [7], in which the mental image formed by the user running the simulation of the virtual environment is similar to the image he would form by interacting with the real environment in such a way that the user becomes "convinced" that his experience running the simulation is real. The key to realism is the complexity of the scene both in terms of the geometry of the model and in terms of how the interaction of light in the virtual environment mimics its counterpart in a real environment.

The forms of user interaction with the system can vary from a simple mouse/keyboard interface to more sophisticated ones such as data gloves [14] and enclosures such as those used in flight simulation [51] and in virtual reality applications [10]. Regardless of the way

this interaction takes place, the key to good user interaction is real-time response, that is, how fast the system responds to user requests.

In summary, visual navigation systems need to: 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.

1.1   Problem Statement

The reason that the criteria mentioned above are mutually incompatible is that while an interactive system needs to achieve interactive frame rates[2], realistic-looking models can contain hundreds of millions of polygons, far more than currently available workstations can render in an interactive fashion [2, 20, 33]. A balance between realism and interactivity is required.

Deering [11] writes:

> 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. Although current workstations are able to render about one million polygons per second, at an interactive frame rate of 30 frames per second we are left with a maximum of about 30K meshed triangles per scene. An interactive

---

[2]Usually, computer animations and movies use 30 and 24 frames per second (fps), respectively.

walkthrough of a geometric database for an entire university campus[3], with vegetation, cars, buildings with furniture, etc., is much more than two orders of magnitude out of reach.

Traditional approaches to this problem use a hardware graphics pipeline and attempt to minimize the number of polygons sent to the system by extracting visibility information from the model and using this information to cull objects against the viewing frustum and by rendering geometrically coarse representations (levels-of-detail, or LODs) of single objects to keep a high frame rate.

As we shall see in the background Chapter 2, Funkhouser et al. [15] adapted these techniques for building walkthroughs by determining in a preprocessing phase sets of objects that are potentially visible from partitions of the model and giving high rendering priority to objects that "contribute" the most to the scene. In cases where there are too many unoccluded primitives inside the viewing frustum so that even if objects were rendered at their lowest LOD, the target frame time could not be achieved, objects are not rendered. In outdoor scenes more often then not the complexity of the visible scene is likely to be too high to allow interactive walkthroughs and therefore using their system would cause "blank areas" to appear on the screen. Also, visibility information can not easily be extracted from an outdoor environment.

In this Thesis, we address the problem of how to navigate through a large model, in which visibility information is hard to extract, maintaining a high and approximately constant frame rate without resorting to not rendering visible objects.

---

[3]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 and so on.

1.2   Thesis Contributions

In this document we present a system that allows a user to interactively navigate through a complex 3D environment at an approximately constant user-specified frame rate. What is meant by complex environment is one that has the potential for more unoccluded primitives inside the viewing frustum than the graphics workstation can display interactively. This system targets walkthroughs of large outdoor environments, where this definition of complexity is most appropriate.

This technique is based upon the use of *impostors* [4]. An impostor is a drawable representation for an object that takes less time to draw than the *true object*, but retains the important visual characteristics of the true object. In this context, traditional LODs are just particular cases of impostors.

The interaction at high/constant frame rate is accomplished by creating a hierarchy of representations for the entire model in which groups of objects at the bottom of the hierarchy are subsequently replaced by impostors until a single, coarse set of representations is obtained for the whole model. In this way impostors for objects (or the objects themselves) can be used near the viewpoint, and cluster impostors can be used when groups of objects are far away.

Throughout this research, we have investigated strategies for creating this simplified

---

[4]According to Webster's dictionary: one that assumes an identity or title not his own for the purpose of deception.

representation for objects/groups of objects as well as algorithms[5] to decide which simplified version to display in real-time. We present a framework that can be used by any interactive navigation system to improve its performance and give a detailed description of the architecture of the navigation system that we have developed.

The new results of this research are:

1. An extensible object-oriented framework in which individual objects and groups of objects are treated as hardware drawable entities, the impostors (Section 5.1).

2. A technique that uses a graphics hardware to automatically generate some of these representations and guidelines used to choose which among them is rendered at a particular frame (Sections 5.2.2 and 5.2.3).

3. A heuristic to compute an object's "contribution" to image quality that takes into account the view-dependent nature of objects. A sample of how image processing techniques can be used to compare different representations of objects against the original highly complex object is presented (Section 4.3).

4. A technique to preprocess a complex environment (one that has the potential for more unoccluded objects inside the viewing frustum than the state-of-the-art hardware can draw in real-time) and produces an interactive walkthrough at approximately constant and high-frame rates (Chapter 6).

---

[5]In a technical report [29] we present the initial algorithms that were used to decide what to render, their flaws and the motivation to design the framework presented in Chapter 5.

More specifically, an extensible object-oriented framework was implemented in C++[28, 46, 47, 31] within which both objects/groups of objects can have multiple representations that can be drawn by a graphics hardware. In this framework we present certain types of view-dependent as well as view-independent representations of objects and groups of objects and show how they can be automatically generated as well as the criteria that should be used to choose which representation to render in real-time. Since this framework allows the inclusion of several different (possibly view-dependent) representations for a single object in the model, it is a generalization of the LOD concept.

We incorporate view-dependent information into a "benefit heuristic" by showing an example of how image processing techniques can be used to compare views of an object or group of objects with the image of its drawable representation. By deciding how closely these two images match, we can select during the walkthrough the best representation for a specific viewpoint.

The problem outlined in Section 1.1 is formalized as an NP-complete tree traversal problem and a hierarchical structure and a traversal algorithm that allow the uniform frame rate interactive navigation of a complex environment are presented. This interactive navigation renders impostors for groups of objects to maintain a user-specified frame rate and tries to avoid disregarding most of the visible objects at each frame. This has not been achieved by previous systems.

1.3 Document Overview

In Chapter 2 we present an overview of the techniques that are currently being used in visual navigation systems and in Chapter 3 we state the assumptions about the class of hardware on which this research is based and show how the rendering time to draw objects can be estimated.

In Chapter 4 we take a look into the perceptual issues involved when designing benefit metrics for visual navigation systems as well as the perceptual issues involved in rendering groups of objects.

In Chapter 5 we describe a framework that can be used by any visual navigation system to improve performance. This framework can be viewed as providing a generalization of the level-of-detail concept.

In Chapter 6 we describe the design of our interactive navigation system. We formalize the problem of navigation of largely unoccluded environments and explain how to build a model hierarchy from a given 3D model, and navigate through it in real-time.

In Chapter 7 we describe the details of our implementation, including the performance of our system and its limitations. Finally, the conclusion summarizes the work we have done and give directions for future improvements of the technique we have developed.

Chapters 2 and 3 provide background knowledge and establish a basic foundation for our research. Some of the new material is presented in Chapter 4 and Chapters 5, 6 and 7, contain the rest of the innovative work.

2

Interactive Navigation Techniques

This Chapter is a background chapter that introduces some of the important concepts and techniques that are used in interactive rendering of large environments.

Since our system architecture uses the system developed by Funkhouser et al. [15] as a base model we explain it in more detail. Their system maintains an approximately constant frame-rate by selectively rendering objects at varying degrees of geometrical complexity including no geometry at all, that is, a visible object may not be rendered. This is the main weakness of their system that is addressed in this research.

Section 2.1 describes how some systems avoid rendering objects that are not potentially visible from the observer's viewpoint. Section 2.2 describes how to manage the complexity of an object in order achieve a balance between interactivity and image quality. Section 2.3 presents the techniques that are used to compute the illumination of large environments. Section 2.4 presents a diagram illustrating based on the model of Funkhouser that shows how the techniques described in sections 2.1, 2.2 and 2.3 are used. The final section summarizes

this chapter and points out the main unsolved problem in interactive rendering of complex environments, which is, how to keep an approximately constant frame rate even in situations where there are too many unoccluded objects inside the viewing frustum than the graphics hardware can render in real-time.

2.1   Culling Techniques

In order to achieve high frame-rates when navigating through large models we need to avoid rendering objects that are not potentially visible and therefore in this Section we examine a few techniques that are currently been used to attain this purpose.

The simplest and most intuitive technique to not render objects that are not visible is culling the objects agains the viewer's field-of-view and thereby reducing the number of polygons to be drawn per frame. Unfortunately, the complexity of the scene inside the viewing frustum can still be beyond the interactive capacity of the hardware. Examples of this technique can be found in [36, 16, 1].

Funkhouser et al.[16] describe a technique for managing large amounts of data during interactive walkthrough 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, i.e. around 3K polygons out of the 250K.

The technique described by Greene et al.[36] combines three types of coherence inherent in the visibility computation, namely, object-space, image-space and temporal coherence. Object-space coherence can be used to resolve the visibility of a collection of nearby objects.

Image-space coherence can be used to resolve the visibility of an object covering a collection of pixels and finally temporal-coherence can be used to speed-up the visibility computation in a sequence of frames with nearby viewing parameters. 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 consideration. By scan-converting the faces of an octree cell they determine if the cell is visible or not. If the entire cell is not visible then the whole geometry inside it can be discarded.

To determine if the octree cell is hidden or not they scan convert each one of its six faces using a data structure that exploits image space coherence, a z-pyramid.

The z-pyramid is a structure that has the original Z-buffer at its finest level and combines four z-values in one level to get one at the next coarser level. A polygon's visibility is tested 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 are 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 one image in an SGI Crimson Elan.

While Funkhouser et al's technique is suitable in cases where the entire model can be subdivided along major opaque elements possibly having portals, as in an office building, it is not suited to outdoor environments or indoor environments such as hotel atriums and sports arenas. Although Greene et al's technique does not suffer from this problem, it would only allow interactive navigation of complex models if it would be implemented in hardware and, like Funkhouser et al's approach, the model is mostly invisible from any given viewpoint.

2.2   Level of Detail Management

Depending on its size and distance to the viewpoint, an object does not need to be rendered at full resolution if the details will not be noticeable. This also applies to moving objects since sampling problems[50, 14] will cause aliasing in the image depending on its position from frame to frame.

Objects can 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 three LODs.

To that end, techniques have been developed to automatically obtain simplified polygonal versions of an object as well as to determine which and how an object's representation
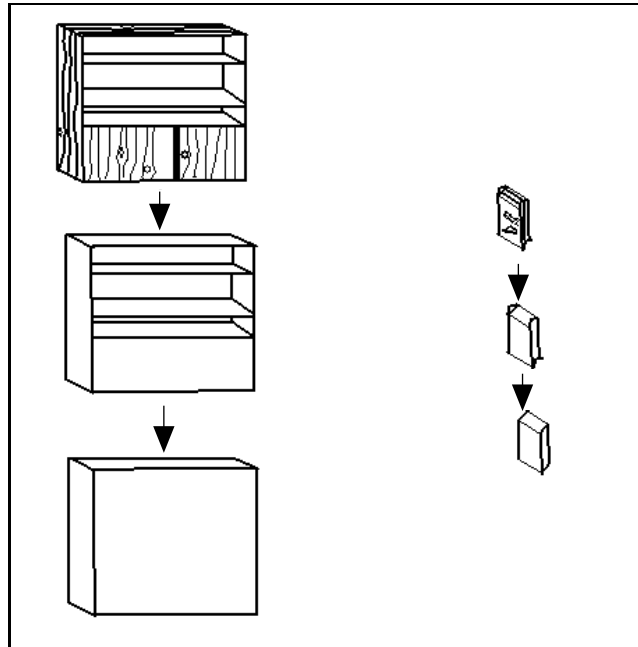
Figure 2.1: Two objects represented at three LODs

should be rendered at a given point in the simulation. These issues are examined in sections 2.2.1, 2.2.2 and 2.2.3.

2.2.1 Polygonal Object Simplification

If we want to use objects represented at different LODs we need tools and techniques to generate them either manually or automatically. Some of these techniques and tools are presented in this Section.

Hall et al.[19] 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

14

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. For instance, the head of a human being due to its importance and detail could be modeled with five LODs while its torso could use just two or three LODs.

The kind of tool described in Hall et al. is 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 sampled 3D points, and then simplified using mesh simplification techniques.

A technique devised by Turk [49] 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 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

15

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 range scanner, Hoppe et al. devised a mesh generation method that can fit a mesh of arbitrary topological type to a set of data points[22], and a mesh optimization method [23] that improves the fit of the mesh 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 third is 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

Given objects represented at different levels of geometric complexity we need criteria to decide which of them we will render at a certain point in the visual navigation. In this Section we present the main LOD selection paradigms: Static, Feedback and Predictive.

In [16] the criterion used to select a given object LOD is static, i.e. 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[15], 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[51] 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 feedback approach works well for flight simulation where there is a large amount of complexity coherence from frame to frame, this strategy won't produce a uniform frame rate for applications like building walkthroughs 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 [15] presented an algorithm that adjusts 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 predictive approach, 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 (e.g. flat/Gouraud shading, antialized, lighted ...). The idea is to maximize: $\sum Benefit(O, L, R)$, subject to $\sum Cost(O, L, R) < \Delta t$, where Δt is the target frame time, but since this problem is NP-Complete, 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).

A problem with Funkhouser et al's approach is that in situations where the visible geometry of a model is too complex, visible objects that do not fit into this frame time bucket are not rendered. This means that portions of the database may simply "disappear" rather then being "blured" geometrically.

2.2.3 LOD switching

An important issue of LOD management is the selection of the method used to switch smoothly from one LOD representation to another since if this transition is not smooth enough object will suddenly appear on the screen[1].

Two methods have been used: one uses morphing(or geometric interpolation) as in Turk[37, 26, 49] and the other uses hardware dependent color blending as described [37, 41, 16, 51]. Both approaches present problems. While morphing may not be feasible in real-time if the number of vertices on the object is too large, color blending has to be used

---

[1]This is known as "popping" and produces a distracting effect.

with care since when one object fadesin the other has to gradually fadeout and during the

transition we are rendering both LODs and therefore increasing the frame time.

2.3   Illumination Techniques

2.3.1   View Independent Illumination Techniques

2.3.2   Real-time View Dependent Illumination

2.3.3   Shadows

2.4   Rendering Model

Figure 2.2 shows how the modules that implement the LOD and illumination techniques

described in the previous sections are put together into a single system and reflects the work

done by Funkhouser et al.

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,
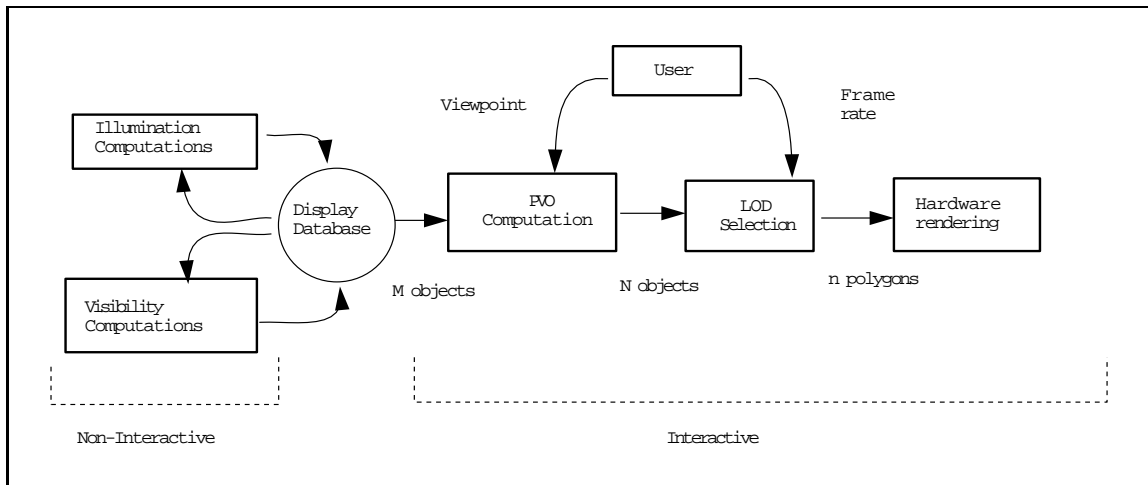
Figure 2.2: Rendering diagram

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 frame rate. Finally, the resulting polygons are displayed in a rendering system using a hardware z-buffer.

2.5 Summary

In this chapter we have discussed techniques that to some extent cope with the hardware's inability to render large databases in real-time.

They improve the rendering time of visual navigation systems and resort to one or both of the following techniques: *culling*, which eliminates from rendering considerations objects that are not potentially visible from a given viewpoint, 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.

We also discussed techniques that are used to compute the illumination of large environments for interactive rendering purposes that contribute to the realism of the simulation.

It is important to note that the model presented in this chapter is useful for rendering relatively large environments but it breaks down when there are more unoccluded objects inside the viewing frustum then the graphics hardware can render in real time and simply discards visible objects that do not fit in the time allowed per frame.

3

Hardware Assumptions and Rendering

Cost Measurement

In order to design an effective visual navigation system we need to not only understand the class of hardware that our system will run on but also take advantage of hardware features such as texture mapping and real-time processing capability to extract maximum performance from the graphics subsystem.

Since the predictive approach to LOD selection explained in Section 2.2.2 is the one that produces the most uniform frame rates, this is the approach we adopt for our system. Therefore, understanding how we can measure the rendering cost of objects and how this cost is affected by texture mapping is of paramount importance.

In Section 3.1 we present the main assumption on which this research is grounded and in Sections 3.2 and 3.3 we address the use of hardware texture mapping and of real-time system features, respectively. Based on the model described in Section 3.1 we consider

how the rendering cost of objects can be measured bearing in mind that this cost can be affected by unpredictable graphics interrupts generated for instance by the indiscriminate use of texture mapping as well as system interrupts inherent to the use of a multi-processing hardware.

This Chapter does not present new results but instead serves as the basis for the predictive system design presented in Chapter 6.

3.1   Pipeline Architecture

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 classic conceptual rendering pipeline that has been used for two decades [14, 50]. Not only state-of-the-art hardware such as the SGI RealityEngine [2] workstations (and graphics systems from other vendors such as Sun [12]) use this pipelined architecture, but also highly parallel architectures such as UNC's PixelFlow [33] use the same pipeline concept.

Based on the *Graphics Library Programming Tools and Techniques* [42] document from Silicon Graphics Inc., and similar work by Funkhouser et at. [15], we present a model of a generalized rendering system that this research is based on. This rendering system is represented as a pipeline with three functional stages as shown in Figure 3.1.
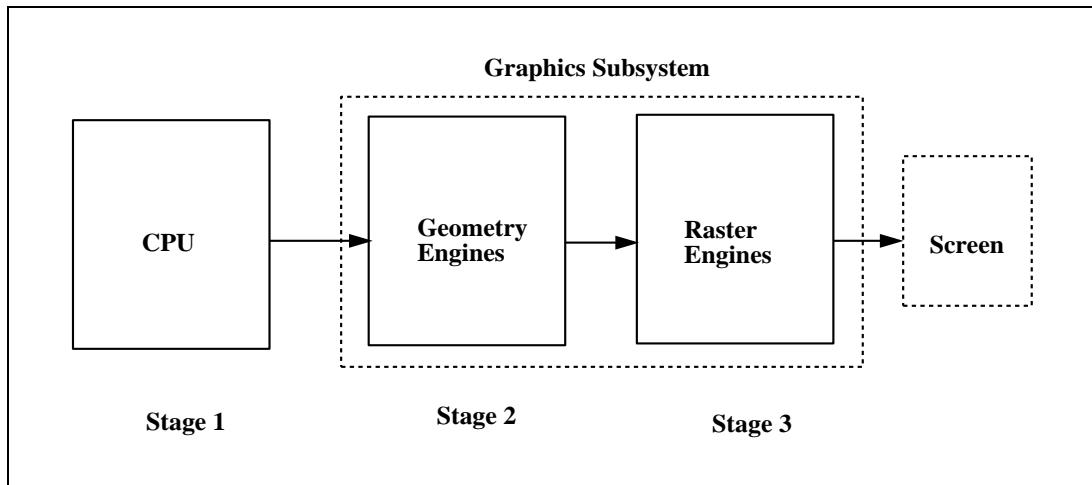
Figure 3.1: Stages of the graphics pipeline.

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 instance, an application that draws a small number of large polygons will heavily 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 issuing graphics commands and returning to do computations so that the graphics subsystem is constantly kept busy. 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   Texture Mapping Capability

Texture mapping is a technique in which two dimensional images are pasted onto geometric objects to yield realistic looking objects. This mathematical mapping is ideally suited for hardware implementation and many current workstations have this feature implemented in hardware. As memory prices go down and CPU power goes up, we can expect to see more low cost machines (such as [12]) providing hardware textures. For those machines that already have this capability, we can expect increases in texture memory size[1] and rendering speeds.

However, texture maps need to be used judiciously. To be rendered, texture maps must be cached in a texture memory and therefore if a texture cache miss occurs an interrupt will

---

[1]The Freedom Series of graphics accelerators from Evans & Sutherland can be configured up to 16 Mb of texture memory and is available on many general purpose workstations.

be generated which will delay the rendering of the object using that texture map. While predicting when and if these kinds of interrupts will occur is not practical, we can try to avoid them by drawing all the objects that use a specific texture map in sequence, not exceeding texture mapping memory, using small texture maps and repeating them while drawing, making sure texture maps dimensions are powers of two and keeping the texture size below the recommended maximum specified by the hardware manufacturer. In addition to these measures, for the specific case of SGI machines, a number of other strategies to improve performance when rendering texture maps is recommended in [41, 42].

3.3 Real-Time Features

In order to 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, they can only be disabled in 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. Using interprocess communication synchronization primitives like semaphores or by inserting directives (pragmas) for the multi-processing compiler and using tools that analyze and parallelize code such as [43], the program can be easily parallelized. By using utilities such as those available for the IRIX operating system [44] (*sysmp, systune, runon, etc.*)[2], a real-time 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

---

[2]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.

can be divided among the other processors in the system. A more detailed description of real-time mechanisms specific to Silicon Graphics hardware, such as processor isolation, interrupts redirection, processor locking and so on can be found in a technical report in the bibliography [24].

3.4   Rendering Cost Metric

The success of any visual navigation system that attempts to maintain a fixed frame rate using the predictive approach described in Section 2.2.2 depends on an accurate and efficient rendering time (cost) heuristics.

As mentioned before, the cost associated with rendering a drawable entity depends 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.

4. State of the graphics subsystem, e.g., state of graphics queues, swapping of texture maps from/to texture memory, etc.

In the generalized rendering system model described, one and two above will influence

Geometric Stage

Model Transf. | Viewing Transf. | Lighting Comp. | Projection Transf. | Clipping and Screen Transf.

A   B   C   D   E   F

Map triangle to frame buffer | Compute color, depth, texture coord. | Hidden surface removal | Blend pixel color with texel color

F   G   H   I   J
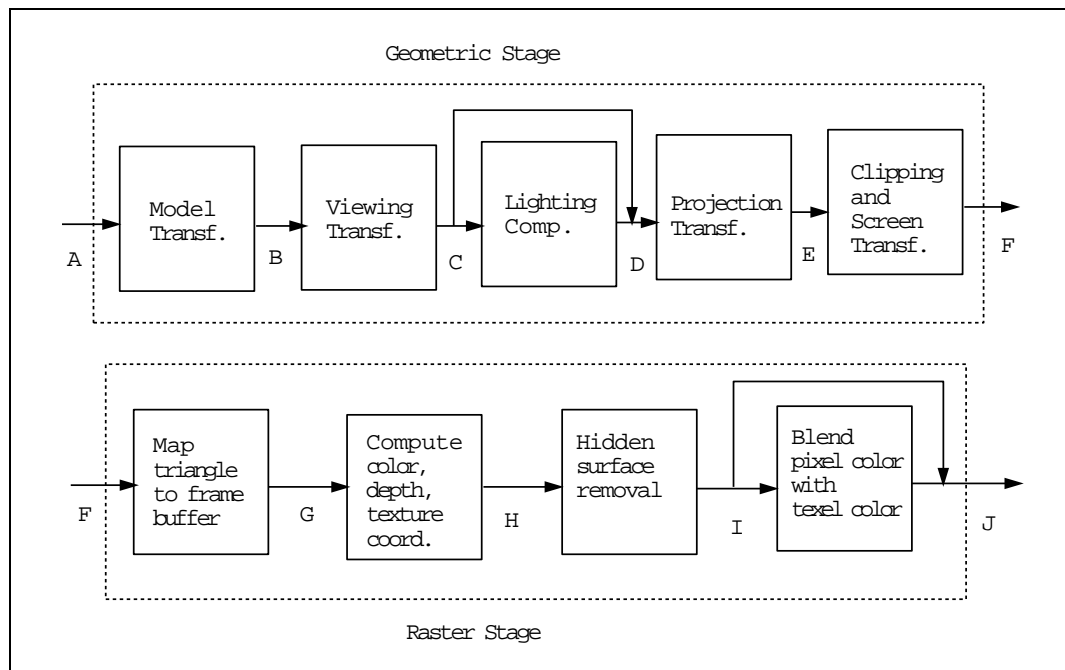
Raster Stage

Figure 3.2: Conceptual rendering pipeline.

the geometry and raster stages of the graphics pipeline, respectively. In order to obtain formulas to compute the rendering cost of each stage of the pipeline in Section 3.1, we examine what happens when a triangle is sent through the corresponding conceptual pipeline [14]. This conceptual pipelines is shown in Figure 3.2.

Entering the geometry stage at A, a triangle is first transformed from object coordinates to world coordinates in B and then to eye coordinates in C by a model and viewing transformations. If lighting is specified, its effect is computed for each vertex of the triangle. This lighted triangle in D is then projected and normalized to the space of a unit cube in E, i.e., each coordinate is in the zero to one range. It is then clipped with respect to this cube and transformed to screen coordinates in F, where each vertex of the triangle keeps its z-value. At this point the triangle is ready to be rasterized. Initially, the frame buffer

pixels that cover the triangle are determined. For each pixel of the triangle entering G, a color, depth and texture coordinates are linearly interpolated from the colors and depths at the vertices. Then the incoming z-values are compared to the z-values already in the frame buffer, to determine visibility in I. If texture is specified, the pixel color is blended with the texel color to produce a final image of the triangle (if it is not subsequently covered by another polygon).

Analyzing the first stage, we see that a triangle to get from A to C, its vertices need to be multiplied by a transformation matrix. This is a fixed cost operation. Lighting is computed using a local illumination model which computes an ambient, a diffuse and a specular component. The ambient component is determined by multiplying the material reflectance to the ambient light. The diffuse component is proportional to the dot product of the normal at each vertex for Gouraud shaded triangles (or the triangle normal for flat shading) with the vector from the light source to the vertex. The specular component is proportional to the dot product of the vector from the light source to the vertex and the vector from the viewpoint to the vertex. Since it uses dot products and multiplications, this cost is also constant. From D to E another matrix multiplication is required. Finally, from E to F a simple linear mapping is needed. Therefore the cost for n triangles to clear the first stage of the pipeline is: $C_1 \times n$, where C_1 is a constant that depends on lighting being turned on/off.

The second stage involves a simple linear mapping from F to G at constant time. From G to H the cost will depend on the number of pixels covered by the triangle. Since the

29

operation here is just a linear interpolation it also takes constant time. From H to I, a simple test is performed for each pixel at constant time. Again from I to J a linear interpolation is again applied to all pixels in constant time. That is, to go from F to J the cost of rasterizing a triangle is simply: $C_2 \times p$, where C_2 is a constant that depends on textures being turned on/off, and p is the number of pixels covered by the triangle. Clearly, since this cost is dependent upon the image-space size of the triangle, it cannot be determined precisely, unless we know the exact image size of the triangle in advance.

To complete this analysis, we note that each stage does work in parallel with respect to the other stage, so the raster stage can be processing triangle one while the geometric stage is processing triangle two. Therefore the total cost of rendering an object is the maximum of the cost of the two stages. Suppose stage one costs two time units per triangle and stage two costs one. When the first triangle clears the first stage, another triangle is admitted to the pipeline. The first triangle will leave the pipeline at time three, while subsequent triangles will leave in time five, seven and so on. For instance, if 100 triangles are sent to the graphics pipeline the rendering time will be: $99 \times 2 + 3 = 201 \approx 100 \times 2$. The same reasoning applies if the cost of the stages are interchanged.

While rendering an object with many different size triangles, at a particular point in time, we do not know which of the stages is the bottleneck of the system and in general we cannot predict the exact cost of rendering the object. However, if an object O is composed of n triangles and has an image size of p pixels, than a good approximation for its cost is: $Cost(O) = MAX(C_1 \times n, C_2 \times p)$, where C_1 and C_2 are as described above.
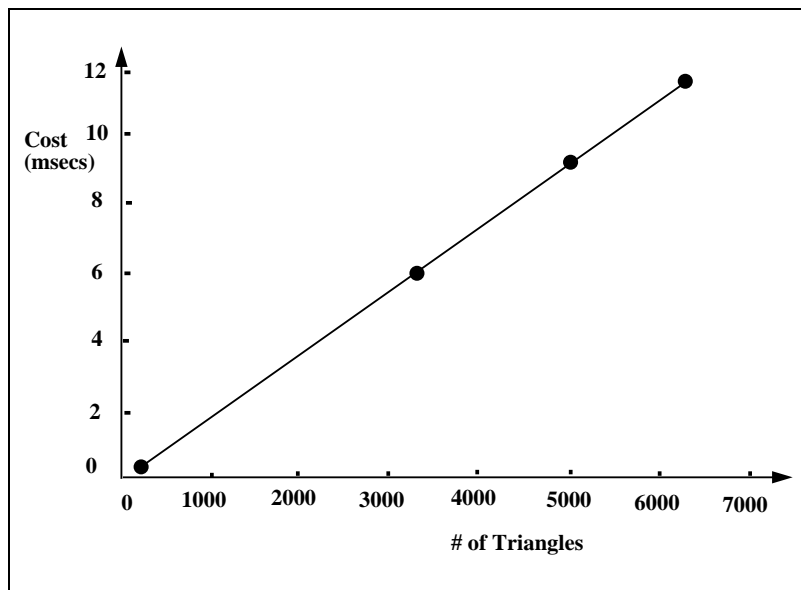
Figure 3.3: RealityEngine rendering cost plot.

These constants can be obtained by benchmarking the machine. Figure 3.3 shows the cost function obtained for a RealityEngine for different objects containing a varying number of triangles, using Gouraud shading, no hardware lighting and no texture mapping. The slope of this line is our constant C_1. Four points are shown around 200, 3400, 4800, and 6300 triangles. These timings were user time and do not include any application or interrupt overhead.

Based on the above discussion we can feel confident that the predictive LOD selection approach works since our conceptual analysis of the graphics pipeline indicates that it basically works in a linear fashion (as illustrated by Figure 3.3) and therefore we can get reasonably good approximations for the rendering cost of objects.

It is important to notice that in principle, since unpredictable graphics and process

31

interrupts can defeat the purpose of a predictive system we need to pay special attention to this problem. To minimize graphics context switches all texture maps should fit in texture memory, no clock should be enabled and no other graphics meters or applications should be running on the graphics processor. Minimization of process context switches, can be achieved by using operating system commands such as those described in Section 3.3 and by running the application as super-user. Therefore, at least one processor should be assigned the task of computing the scene to render while another processor should be assigned to render the computed frame. In this case, the time to compute the scene must not exceed the target frame time. The effect of interrupts are shown in Figure 7.7 in the implementation Chapter 7.

3.5 Summary

In this Chapter we have presented the class of hardware and its features (texture mapping and multiprocessing) on which this research is based, and explained that the rendering time associated to drawable representations for objects composed of triangles can be approximated.

One of the main hardware features that will be explored in this Thesis is texture mapping since as we shall see in Chapter 5, view-dependent representations for objects can be generated and rendered at a low cost using texture mapping.

Although texture mapping is a useful resource, it can impair the purpose of a system that uses the predictive approach to LOD selection by generating unpredictable graphics

interrupts that will alter the rendering time for the entire scene.

In general, graphics and process context switching along with excessive time to compute a single frame are the main factors that can reduce the efficiency of a predictive system. Fortunately there are ways to minimize these effects.

4

Model Perception

Visual navigation systems as explained in Chapter 2 use different representations (LODs) of an object to improve the performance of the simulation. We start this chapter by making a few observations about the main characteristics of these representations according to the intended purpose of the simulation in which they are used in Section 4.1.

The effectiveness of these systems lies on the balance between interactivity and realism. Therefore, they need to be able to decide how objects contribute to the overall "feel" of the simulation. This contribution can be estimated by a benefit heuristic which attempts to take into consideration factors associated to an object as well as to its representations such as, amount of information that the object conveys and the accuracy of the representation that is used to render the object, respectively. This heuristic is presented in Section 4.2.

Since different representations for an object might contribute differently according to the angle from which they are viewed (e.g. a roadside billboard has a low benefit when viewed from the side), in Section 4.3 we examine how view dependency can be added to the

benefit of an object.

The benefit heuristic as described in Section 4.2, as well as the one described by Funkhouser et. al., does not address how humans perceive a collection of objects seen as a whole. Briefly, if two objects A and B are represented by C and have benefits B_a and B_b what should the benefit B_c of C be?. As we shall see B_c is not simply the sum of B_a and B_b since A and B when viewed as a group might give a different contribution (meaning) to the simulation then the objects alone would, i.e., the benefit of all the objects in a scene does not translate into the benefit for the entire scene. Perceptual issues that account for how a group of objects is perceived (its semantics) are addressed in Section 4.4, and Section 4.5 summarizes the Chapter.

4.1 Characteristics of Representations

The designer of a visual navigation system which render objects as well as their LOD representations need to determine their main characteristics and how they contribute to the perception of the model. While the former is examined in this Section the latter is examined in Section 4.2.

An object has intrinsic properties that distinguish it from other objects in the database such as, form, detail, and color, which need to be taken into account when generating these representations. Form and detail for instance, are basic attributes associated to an object that our visual system uses to classify it as pertaining to a certain category of objects, while color is an attribute of an object that greatly contributes to its realism.
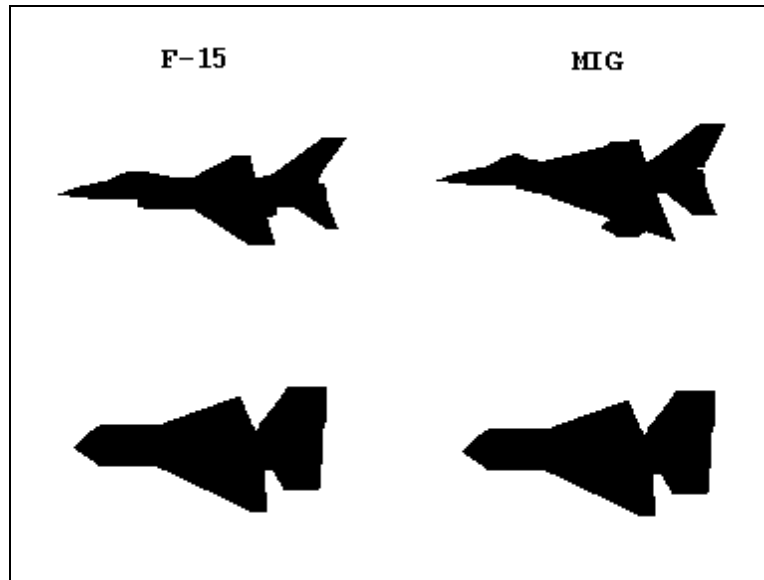
Figure 4.1: Two different objects with the same representation.

One other property which is inherently dependent upon the kind of simulation the object is taking part is its semantics. The semantics determines what role the object plays in the simulation and therefore the representation needs to convey the same meaning as the actual object.

Consider for instance a flight simulation application that has two different plane models stored in its database, an F-15 and a MIG, with identical representations as in Figure 4.1. Since the representations are identical, displaying them in a military exercise could result in erroneous consequences whereas for a commercial application one could be just interested in avoiding a collision with other planes, regardless of nationality. Each application area of visual navigation systems has different representation requirements so that the purpose of the particular simulation is achieved.

Training systems such as combat simulation require that representations for objects like

tanks have at least a form close to the real object and the appropriate colors (so that it can be distinguished from an enemy one). If such an application is considered from the strategic point of view then an icon representing the tank would be sufficient although visceral realism will not be achieved in this way.

Flight and driving simulators need to be able to recognize the conditions of an airport runway or a highway under different weather conditions, respectively. On a rainy day for instance the geometrically simpler representation for a road needs to have a reflectivity similar to that of the actual road so that the specular reflection of the light coming from a car in the opposite side of the road is the same.

Architectural walkthrough systems should display pleasant environments and therefore not only are the forms of buildings, furniture, etc., of concern, but also the different color shades and the reflectivity properties of the materials are extremely important.

Navigation system for planetary exploration may be more concerned with form and reflective properties of craters, rocks, debris and so on, while in volume rendering applications such as computational fluid dynamics and MRI data sets visualization the most important thing may be colors associated to turbulent fluid behavior and features such as tumors and contrast absorption by different tissues, respectively.

As we see from these observations what will ultimately determine the representations for objects is the kind of application that they will be used in.

4.2 Benefit of Objects

A good benefit heuristic should predict the amount and accuracy of the information conveyed to the user when the object (or one of its representations) is rendered. Although a good benefit heuristic is hard to design since it involves issues related to both low-level vision processing and high-level interpretation by the brain of the stimuli that come from the visual system, several factors should certainly contribute to the overall perception of an object.

We have divided this contribution in two components, one that is intrinsic to the object, the object's benefit, and one that is intrinsic to a representation of the object, the accuracy with which it represents the full detail object, that is, the overall contribution to image quality of an object can be viewed as a function of the object and the representation used to render the object. While the object's benefit heuristic can be used to decide which of the objects will have priority in receiving rendering time the accuracy/cost ratio can be used to determine which representations of a given object gives the best contribution to the simulation in a constrained frame time situation.

The object's benefit has to deal with the perception and semantics of the object in the particular simulation. While the perception of the object (or its contribution to the feel of the simulation) varies during the visual navigation its importance is established prior to the simulation.

The benefit of an object, as in the work by Funkhouser et al. [15], should incorporate

factors such as:

- The projected size of the object onto the screen (its image space size), since large objects seem to contribute more to the image.

- 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[1]. This assumes the center of the screen to be the focus of attention and 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 [4]).

- The object's relative speed with respect to the viewpoint. This takes into account that a fast moving viewpoint will result in objects being blurred in the scene and therefore there is no point in trying to render them at full detail.

- The fact that the object was rendered using a different representation in the previous frame. This is included here because of the annoying "popping" effect caused by frequently switching from one representation to another from frame to frame.

Each of these parameters can be computed to lie in the zero to one range and the benefit of an object can be computed in ways that can range from a simple multiplication [15] of all the above factors to their weighted average. We particularly prefer the second way,

---

[1]We assume that the user's eyes are looking at the center of the screen and in this case this parameter can be measured as the distance of the object to the line of sight.

since we believe it gives us good flexibility when adjusting benefit according to particular applications.

Note that while the benefit of an object (except for its semantic) can only be determined in real-time and therefore is inherently dynamic, the other component of an object's contribution to the simulation, the accuracy of one of its representations, is statically determined and does not vary during the simulation (only its perception does).

The accuracy of a representation can be obtained by comparing the representation against the full detail version of the object. If levels of detail are used, a formula that takes into consideration the number of polygons present in the representation and the rendering algorithm used to draw it can be used as in [15]. If however view-dependent representations are used, then more elaborate accuracy measures are required. This is the topic of Section 4.3.

4.3 View Angle Dependent Benefit Calculation

The benefit heuristic in Section 4.2, does not incorporate the view dependent nature of an object and its representations. Consider the case where we want to assign benefits to three different representations of a house as shown in Figure 4.2 where the first of these representations is the house object at full detail, the second is a low level of detail representation and the third is just a single polygon with a texture map representation of the front of the house. We classify the third representation as *view-dependent* and the first two as *view-independent* meaning that the *view-dependent* would only be considered
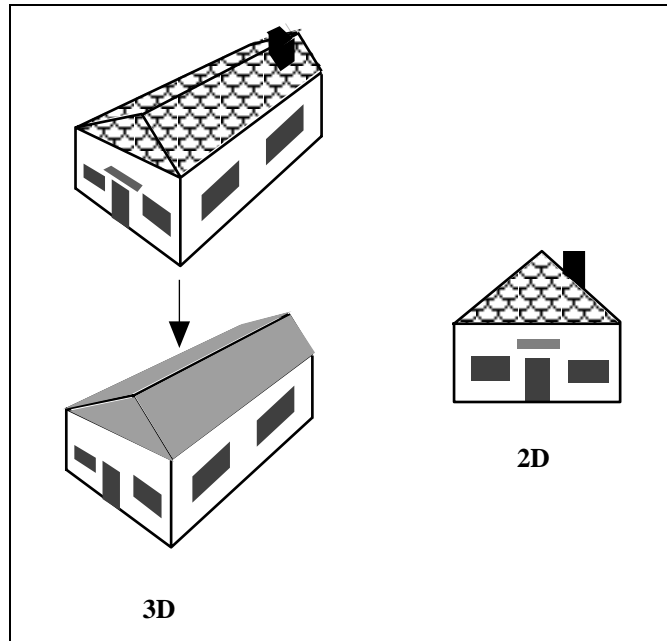
Figure 4.2: Three representations for a house. The left two are view-independent while the one on the right is view-dependent.

for a subset of all possible viewing directions, while the *view-independent* LODs would be considered for all viewing angles.

The first of these representations has the highest accuracy regardless of view angle but we might not want to render it since it is also the most expensive to render. The benefit that should be assigned to the other two will depend upon the user's view angle and view distance. If the viewpoint is in front of the house looking straight to it then the view-dependent texture map should probably be preferred if the viewpoint is relatively close to the house. On the other hand, if the user is looking to the side of the house then the view dependent texture map would have a zero benefit. If the viewpoint is far from it then the untextured LOD will cause less aliasing.

To determine the total contribution to the feel of the simulation caused by rendering of an object, we need to incorporate view dependent information into the object's benefit heuristic and determine not only what representations are suitable for given view angles but also how faithfully these representations are to the full detail object.

A 3D object such as the house in Figure 4.2, when projected on the screen, will result in a different image-space size for different view angles. One simple way of incorporating this view-dependency into the object's benefit heuristic is to determine which face of the object's bounding box is facing the viewpoint. We can then approximate the image size of the object (an important component of the object's benefit heuristic) by computing the projection of this bounding box face onto the screen. This may be a poor approximation for certain objects and view angles and therefore it might be necessary to approximate this size by the sum of the image-space sizes of all faces that are visible to the observer.

Another way of incorporating view dependent information into the total benefit heuristic of an object is to measure the accuracy of each of the object's representations according to each viewing direction possible. However, since the space of possible viewpoints and viewing directions is infinite, we discretize it into a finite set of viewing directions, and assume that the view distance is infinite (we use an orthographic projection). To tabulate directional benefits, we sample the hemisphere of directions around the object as in Figure 4.3 and calculate the accuracy of each representation for each sample direction.

The number and location of these samples will depend on the representations that the object has associated with it and the possible viewpoints during the walkthrough. If in the

case of Figure 4.3 the house only has five view-dependent representations (one texture map for each of the four lateral faces and one for the top) we try to distribute the space around each of these representations into regions such that when the viewpoint is in one of these regions than we consider the texture map associated to that region. In this case, what we are basically interested is in determining if this view-dependent representation has a better accuracy than a possible view-independent representation suitable for that situation. We therefore need to sample those directions around the line perpendicular to the texture map and compute the accuracy of that representation for those view angles. The number of samples will depend on the behavior of the accuracy as a function of the view angle around the line perpendicular to the texture map, so that we can get an as good as possible picture of the overall shape of the accuracy function. Of course, we do not need sample directions that would require the viewpoint to be under the house.

We sample each of the viewing directions and measure the accuracy of each representation and construct a table that has one entry for each pair (representation, viewing direction). Each of these entries contains a similarity value (accuracy) of the representation measured with respect to the full detail object for the particular viewing direction. During the walkthrough, the accuracy component of the benefit heuristic for a given representation and viewing direction can be obtained by accessing this table. The values in these table entries range from 0, meaning that the representation is not suitable for that viewing direction, to 1, meaning that it is the most "accurate" representation for that viewing direction.

43
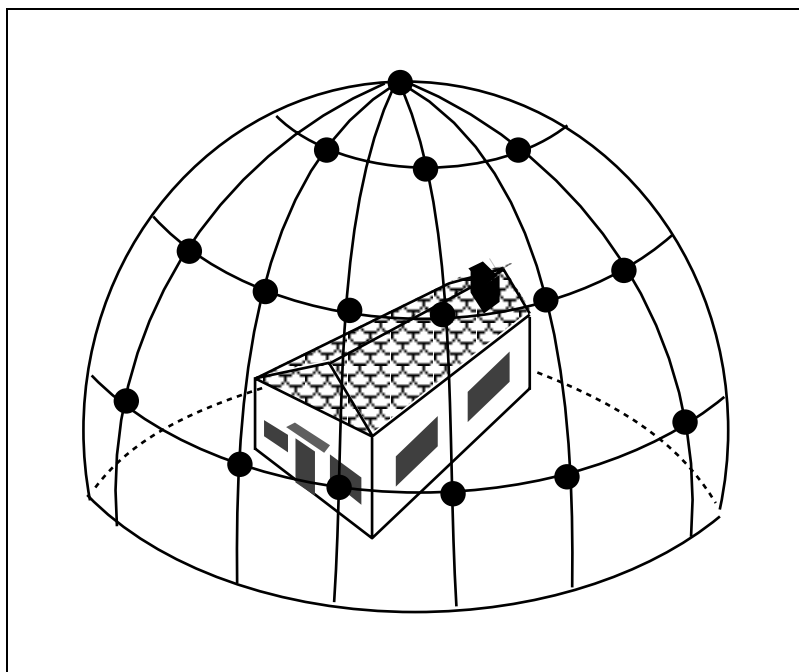
Figure 4.3: Discretizing the space of viewpoints around an object.

Since "images" are being used as view-dependent representations for objects we need more appropriate ways of measuring their accuracy, that is, for a given viewpoint we need to compare the representation's image for the object(s) with the image of the actual object(s), i.e., the "ideal" image.

Many ways of comparing two images can be devised but what one needs to be concerned with is what are the features of interest in the images that are important to the application. Are the colors on the images the most important features? Is the overall shape of the object(s) depicted on the image close to the shape of the real object(s)? How much detail does the image contains compared to the real object and to what extent are these details important?

Ideally the accuracy of an image with respect to an "ideal" image should be obtained by a perceptual comparison of the two images but since we are in search of automatic ways to determine similarity we resort to computational techniques. Here we suggest that image processing techniques can be used to measure how similar two images are and present a sample of what can be accomplished using these techniques, assuming that a more accurate procedure would be substituted in the future.

If colors are important then techniques that incorporate the histogram of two images may be suitable since a simple pixel-by-pixel comparison would not work because of precision problems and the fact that one image might be a little shifted with respect to the other. If details are important then a procedure that incorporates edge detection mechanisms on the two images might be required. If we assume that the apparent shape of the object(s) in the image is important then we can use image processing techniques to roughly simulate some low-level processes that the human visual system performs to detect shape.

The routine described below and illustrated in Figure 4.4 should not in any way, be viewed as a near-optimal way of comparing two images.

1. We avoid dealing with color, since slightly shifted R,G, and B values would result in the similarity of two very similar images being zero. Therefore, we first turn the two color images in the upper left corner of Figure 4.4 into gray scale images by simply averaging the RGB components at each pixel resulting in the images on the upper right corner of the same figure.

2. Since we are assuming that we are interested in the most important features of the

image that our low-level vision can detect, we extract the contour of objects by applying an edge detector to both images. This is done by convolving the images obtained in the previous step with a 5x5 Laplacian operator and computing its zero crossings. Pixel values that fall below a threshold are set to zero (black) and the others to 1 (white) as in the lower left part of Figure 4.4.

3. We want to detect the features present in both images even if they are shifted, scaled or rotated by small amounts. For instance, imagine that an image has an edge at (1,0) and that same edge in the other image is in (1.2,0). If we do a pixel by pixel comparison than these two edges will never correlate. Therefore, we blur the images so that the edges become thicker (the edge at one might extend from 0.9 to 1.1 while the edge at 1.2 might extend from 1.1 to 1.3) and can be correlated. This blurring is accomplished by first shrinking the images by convolving them with a Gaussian operator and then expanding them to their original sizes. A Gaussian pyramid [13] can be used to achieve several degrees of blurring.

4. In the last step, we do a pixel by pixel comparison of the two resulting images. The lower right images on Figure 4.4 show the common points (dark) on the two blurred versions of the edge detected images on the left.

As we mentioned, the process described above is designed to compare images that are slightly different and is far too simple to mimic human image processing. What is really needed is a set of more powerful image processing techniques that would be able to extract

46
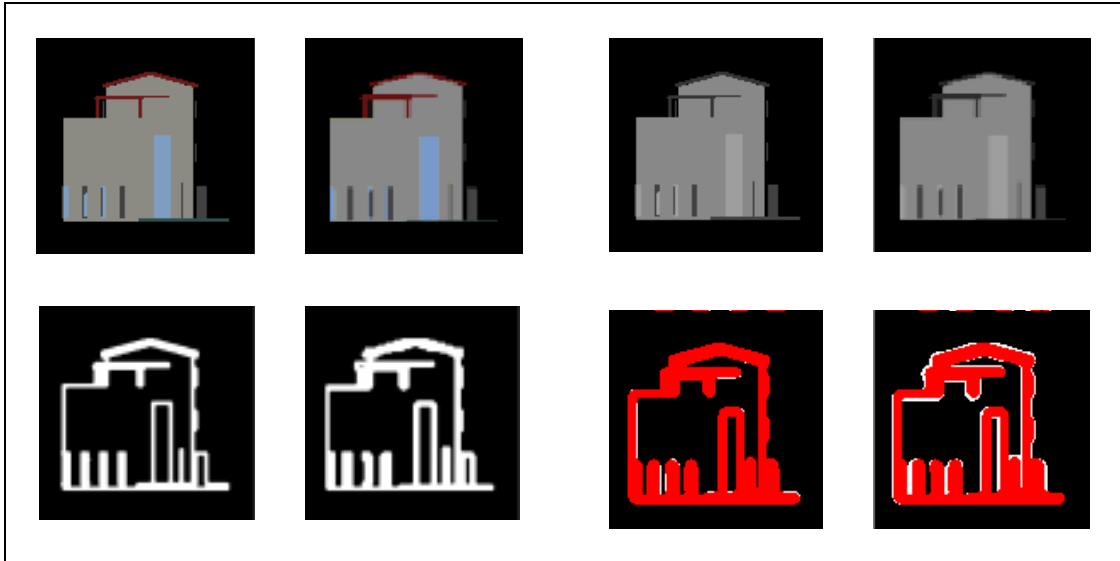
Figure 4.4: Top left original images. Top right gray scale version. Bottom left images are shown after an edge detector. Bottom right blurred versions of the images on the left showing in red the common points.

features in one image and locate these features in similar places of the other image. Nevertheless, it does serve as a placeholder in our system that can be replaced later with a module that performs better by using segmentation and high level processing.

As we shall see in Chapter 7 the reason why being able to measure the accuracy of a given representation is important is that if more than one representation is suitable for a given viewpoint, since both view-dependent and view-independent representations can be used, we want to render the one that most accurately represents the object in a constrained rendering time situation. In fact, since each representation has a cost associated to it, we can select the one with the best accuracy/cost ratio to render.

4.4 Benefit of Clusters

In this Section we examine visual organization rules that have been used by graphics designers for centuries [14] and codified by Gestalt psychologists (see [18]) in terms of "Laws of Organization" that appear to account for how humans perceive groups of objects and consider how the benefit of objects change when they are viewed as a whole. We conclude that it is hard to make any formal statement about this subject and we avoid drawing too many conclusions about combining benefits. The ideas in this section are meant to highlight that much more research needs to be done on how benefit heuristics can draw on perceptual behavior, that is, how we can produce benefit heuristics that also take into account the surroundings of a particular object.

There are four laws of organization that appear to be unconsciously used by humans to interpret collection of objects: Simplicity, Similarity, Nearness and Good Continuation.

1. Simplicity, states that every stimulus pattern is seen in such a way that the resulting structure is as simple as possible. For instance, Figure 4.5 is perceived as a triangle overlapping a square (or vice-versa) and not as a 13 sided polygon.

2. Similarity, states that similar objects appear to be grouped together. In Figure 4.6, circles appear to be grouped with other circles and squares are grouped with other squares, and what we perceive are alternating rows of circles and rows of squares, opposed to columns.

3. Nearness, states that objects near to each other appear to be grouped together. In
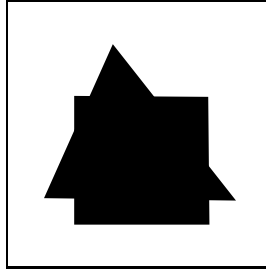
Figure 4.5: This picture is interpreted as a square overlapping a triangle (or vice-versa) and not some complex figure. Adapted from [18].
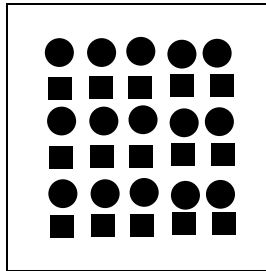
Figure 4.6: Figure perceived as alternating rows of circles and squares. Adapted from [18].

Figure 4.7 we tend to perceive rows of alternating circles and squares, because since they are close to each other they appear to be grouped in the same row.

4. Good continuation, 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 4.8.

Schifman [38] generalizes the good continuation 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

49
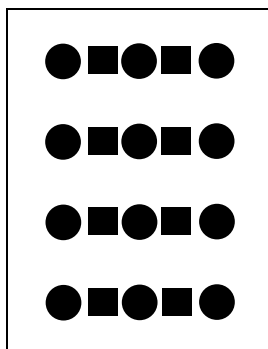
Figure 4.7: Figure perceived as rows of alternating circles and squares. Adapted from [18].
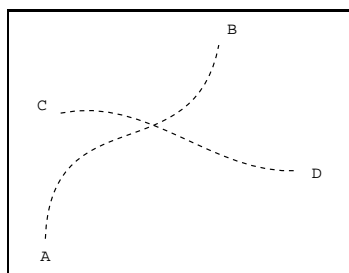
Figure 4.8: Figure is perceived as two intersecting curves AB and CD. Adapted from [18].

of symmetric objects, respectively.

As a practical application of the continuation law, consider the lights that demarcate an airport runway. If these lights are considered alone then because of their sizes (just dots in a flight simulation application) they might be assigned very low benefit and some of them (if not all) might not even appear in the simulation[2]. If instead we have a representation for these lights as a group and use the continuation law then we would have a higher benefit associated to this representation that would encourage its rendering.

Another example would be to consider a walkthrough of a battle field containing many

---

[2]In this case, the brightness of each light might also be considered as one of the components of the benefit heuristic.
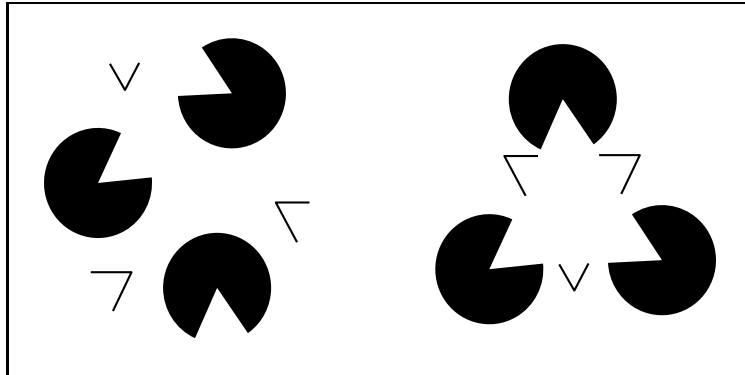
Figure 4.9: Subjective contour illusion adapted from [18]. Left, a set of objects. Right, additional information is gained when objects are rearranged and viewed as a whole.

soldiers and guns. We can assign individual benefits to both a soldier and a gun, but if the soldier is actually holding the gun then the benefit that should be assigned to both objects in tandem would probably exceed that of the sum of the two benefits, particularly if the soldier is pointing the gun toward the user of the system.

Examining these laws, we can conclude that to determine the benefit of an object in some cases is undecidable without knowing what surrounds it. The meaning conveyed by an object 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. In Figure 4.9, objects on the left are just isolated drawings but can convey additional information when positioned in a certain way and viewed as a whole.

While realizing that it is extremely difficult to account for how objects interact in a scene we still use a per-object benefit heuristic although it may not be suitable for some groupings of objects. Despite the limitations of current use of benefit heuristics, we rely on them because even imperfect benefit heuristics have proven very useful in practice.

4.5 Summary

In this Chapter we presented the important factors that a benefit heuristic should incorporate. Although this heuristic is very similar to the one used by Funkhouser et. al. [15] they differ conceptually. The difference is that we regard the benefit of an object to be some property intrinsic to the object that is determined dynamically while accuracy is a property associated to one of the object's representations and is statically determined. The benefit of the object can be used to determine the priority to allocate frame time to it whereas the accuracy of a given representation as well as its cost can ultimately be used to determine which representation to render.

We pointed out that this benefit heuristic should incorporate two important factors: the semantics of a group of objects and the view dependent nature of objects and their representations. The semantics of a cluster of objects is different from the combination of the semantics of each object and therefore the benefit of a cluster is not the sum of the benefits of all the objects in the cluster, that is, the equation Benefit(A) + Benefit(B) = Benefit(A + B) does not hold. Despite this drawback, in this work we still use a per-object benefit heuristic since it can efficiently be computed in real-time and in practice have proven to yield in good image quality, especially in navigation of static databases.

The other important factor is the fact that objects as well as their representations look different from different viewing directions. This is especially true if representations are view-dependent and are only suitable for certain viewing directions.

While incorporating semantics into cluster representations would involve developing computable ways to account for the "laws of organization" and to "understand" the meaning of groups of objects, which is beyond the scope of this work, we showed how the view dependent component can be incorporated into the benefit heuristic.

We have outlined a method that discretizes the space of viewing directions surrounding an object, obtain the image of the full detail object(s) and that of one of its representations and checks how similar they are. We argued that this comparison can be made by using image processing techniques and we present a sample of how this comparison can be done. The output of this method is a table that can be accessed in real-time to determine the accuracy component of the benefit heuristic for a particular representation for given view angles.

Although the incorporation of semantics for clusters would be extremely hard, we can still use the benefit heuristic with some confidence provided that we have incorporated the view dependency factor.

5

Framework for Visual Navigation Systems

Conventional walkthrough systems usually rely on the designer to produce different levels-of-detail (LODs) for the objects in the model in order to reduce the rendering time of a particular frame. In many cases, these representations are either not readily available, expensive, or time consuming to generate. Furthermore, the user needs to concentrate on the application and not on generating different LODs whose only purpose is to improve rendering speed and which are not directly related to the application.

Since these LODs are simply representations of the actual objects they do not necessarily need to be versions of the same object with fewer geometric primitives (or drawn with a less accurate rendering algorithm such as flat shading instead of Gouraud shading) but rather representations that can be drawn on the computer screen in less time than the actual object and provide the simulation with a feel similar to that obtained by using the full detail object. Although this same idea also applies to a group of objects (cluster), the details of how clusters are used are left to Chapter 6.

54

In Section 5.1 we develop an extensible object-oriented framework as in [27, 3, 25] within which objects (or groups of objects) can have multiple drawable representations, which we have been calling impostors, that can be drawn by a graphics hardware.

A framework as intended here is the definition of the behavior of a collection of objects for use in visual navigation systems. These objects form the building blocks that can be used, extended, or customized for particular software solutions. We have tried to make it as complete as possible in the sense that it provides the basic functionality needed by walkthrough systems. We also tried to make it flexible and extensible so that future users of this framework can easily add and modify its functionality. The actual implementation of this framework is in C++ and is described in Chapter 7.

We then discuss in Section 5.2 the types of impostors that our framework supports and how they can be automatically generated. We also give guidelines that can be used to determine what impostors are allowed to be used at a certain point in the simulation and a summary of the Chapter in Section 5.3.

5.1 Object-Oriented Design

In this Section we specify the main abstractions that should be used by a visual navigation system together with their interfaces. Figure 5.1 shows the main abstraction of our framework. The "Conceptual Object" class is an abstraction for any object in the model whereas the "Drawable Representation" class represents a variety of hardware drawable representations that are associated to a conceptual object. The "Conceptual Object" class
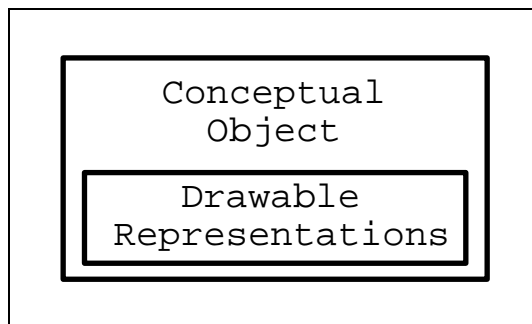
Figure 5.1: Main abstraction.

Interface			
Function	Description	Input	Data Used
void Draw(vp)	Renders a selected rep. according to the current rendering algorithm. Calls a virtual function in the contained class.	Viewpoint	A hardware drawable rep.
real Cost()	Accesses the cost of rendering the selected rep. A virtual function.		A hardware drawable rep.
real Benefit(vp)	Computes the contribution to image quality of an object.	Viewpoint	The items described in Section 4.2
boolean Visibility(vp)	Checks the object against the viewing frustum and marks it as invisible if there is no intersection. Can return false positives.	Viewpoint	Bounding box of the object.

Table 5.1: Interface of the conceptual object abstraction

has the interface illustrated in table 5.1.

The "Drawable Representation" abstraction is depicted in Figure 5.2 and its interface described in Table 5.2. This abstraction also contains, cost and accuracy values that are instantiated by the particular subclass it represents. It contains a table of accuracies of the given representation for selected viewpoints. During the walkthrough, depending on the viewpoint with respect to the representation, the appropriate accuracy value is accessed.

56
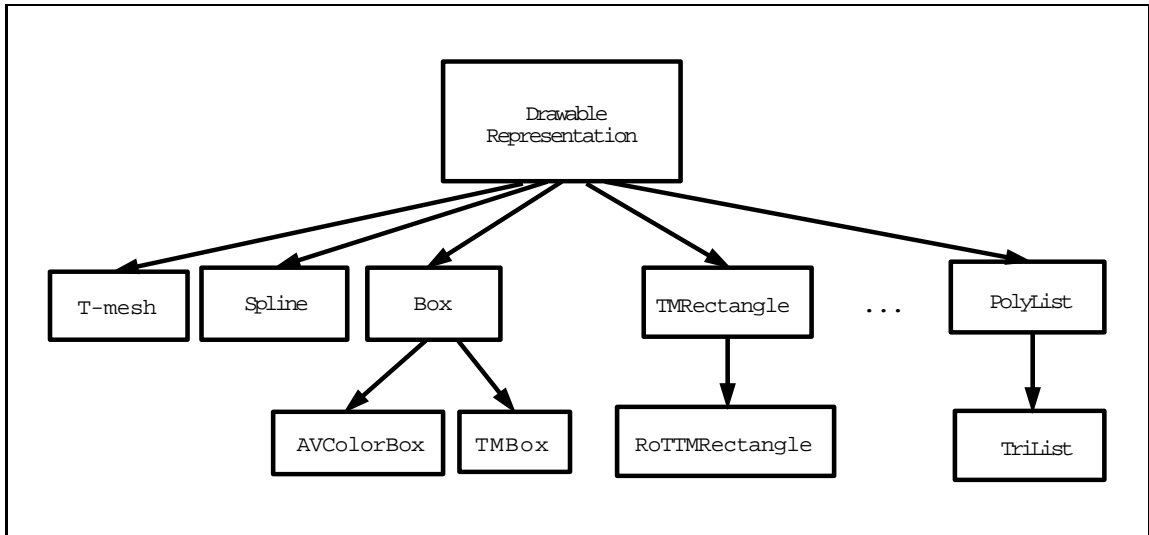
Figure 5.2: Abstraction for hardware drawable representations

Interface			
Function	Description	Input	Data Used
void Draw(vp)	Renders a drawable rep. A virtual function.	Viewpoint	A hardware drawable rep.
real Cost()	Accesses the rendering cost associated to a rep.		A hardware drawable rep.
real Accuracy(vp)	Accesses the accuracy of a rep. with respect to the full detail object.	Viewpoint	A hardware drawable rep.

Table 5.2: Interface of the hardware drawable abstraction

In Table 5.3 some of the subclasses of a drawable representation are shown together with the data they use. Associated to each of these subclasses there is a "Draw" function that takes the viewpoint as input and draws the representation on the computer screen. The subclasses depicted in Figure 5.2 are either hardware primitives such as t-meshes and splines or impostor types such as those mentioned in Section 5.2.1 below. 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.

57

Abstraction	Interface	
	Description	Data
T-Mesh	Draws a t-mesh.	Array of points
Spline	Draws a NURB based on control points and weights	Array of control points and weights
Box	Draws a parallelepiped wireframe	Eight points.
AvColorBox	Draws a solid box with colors	Eight points and six colors.
TMBox	Draws a box with a texture map pasted on each face	Six pointers to texture maps and eight points
TMRectangle	Draws a texture mapped rectangle	4 Points and a pointer to a texture map
RotTMRectangle	Draws a rotating texture mapped rectangle that follows the viewpoint as it moves	Same as above plus the angle of rotation
PolyList	Draws a list of polygons	List containing a list of points and colors

Table 5.3: Some hardware drawable abstractions.
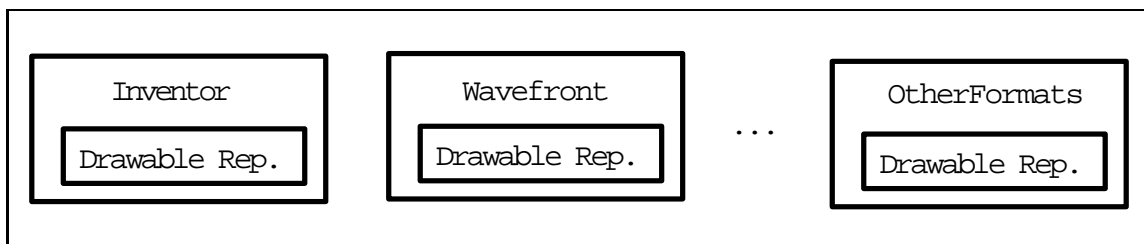
Figure 5.3: Input models abstractions.

The hardware drawable representation interface is composed of a single virtual draw function that is appropriately redefined according to its subclasses. This allows the design of dedicated and therefore very efficient rendering routines for each particular representation and thus allowing the extraction of the maximum performance of the expensive graphics subsystem. 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.

To complete our design 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 5.3. In this design we will have a class for each modeling software supported that will contain a function to read a certain format and convert it to a list of hardware renderable primitives.

5.2 Impostors

In this Section we present the kind of impostors that our framework currently supports, explain how they can be automatically generated and what are the criteria used to select them during the visual simulation.

5.2.1 Types of Impostors

Our framework allows for two kinds of impostors: view-dependent and view-independent. LODs are impostor representations for an object that are usually geometrically simplified user generated versions of the object that preserve as much as possible the characteristics of the highest LOD of the object such as its form, details and colors. The perception of these representations are not affected by the viewing direction on which they are looked upon and therefore are called view-independent impostors.

Other impostors in turn are view-dependent meaning that they might not be suitable for given view angles. Given a complex object, a view-dependent impostor representation for it can just be a texture map containing the view of the object from a certain viewpoint.

We now enumerate the impostor types, classified as view-dependent and as view-in-dependent representations of a given object, that are currently being used in our system implementation. Other impostor types can be easily added to the system according to our object-oriented approach in Section 5.1.

5.2.1.1 *View-dependent Impostors.*

- TMRectangle: This impostor type corresponds to a texture map that is pasted onto the appropriate face of the object's bounding box.

- RotTMRectangle[1]: This is a variation of the TMRectangle type above in which the TMRectangle instead of being pasted to one of the faces of the object's bounding box it is centered at the object's center and it is made to rotate to follow the viewpoint. It is particularly suited to represent symmetric objects such as pine trees.

- PseudoTMRectangle: This is yet another variant of the TMRectangle type. Since meshed triangles (or meshed quadrilaterals) are much faster to draw[2] then regular polygons we create a mesh in which each pair of meshed triangles (or a single quadrilateral) corresponds to a pixel in the texture maps. This type can be used in machines that do not support texture mapping or in cases where the texture maps in the system are causing excessive page faults to happen thus disturbing the interactivity of the system.

---

[1]This type is also known as billboard in [37]

[2]It is around 4 times faster to draw than triangles on the RealityEngine

- LODs: These impostors are the conventional levels-of-detail that are used to represent objects.

- AvColorBox: This impostor is a representation for an object which is just a box with average areas and average colors at each of its faces. It is useful to represent objects whose image-space size during the walkthrough is just a couple of pixels.

- TMBox: This representation is a variation of the TMRectangle and corresponds to a bounding box of an object onto which faces texture maps are pasted. This impostor is suitable to represent box like objects such as a box shaped building.

5.2.2   Impostor Generation

View-independent impostors can usually be obtained by using the techniques described in Section 2.2.1 or manually by the database designer. View-dependent impostors such as the TMRectangle type texture map can be automatically obtained in the following way with the help of a graphics hardware:

1. Set up a viewpoint, a viewing direction, an orthographic projection matrix and a window size according to the resolution of the texture map to be obtained for the object.

2. Draw the object in the appropriate window on a completely black background, adjust the resolution of the texture by scaling the object, and grab the resulting image. What

ultimately determines the resolution of the texture map is the complexity (or granularity of details) that the object exhibits from the particular direction it is viewed.

3. Set the alpha component of black pixels to zero, so that if the object has holes in it then we can see through when it is rendered.

Average color boxes (AvColorBox type) can also be obtained with the help of a graphics hardware:

1. Repeat step one above.

2. For each face of the object's bounding box, draw the object onto a black background and grab the resulting image. The average color for that face is just the sum of rgb components of all non-black pixels divided by the number of non-black pixels on that face. The average area is the number of non-black pixels in the image.

3. Compute the dimensions in world coordinate of the average area box.

Let $x_s, y_s, z_s, A1_s, A2_s, A3_s$ and $x_w, y_w, z_w, A1_w, A2_w, A3_w$ be the x,y and z components and the yz, xy, xz areas of the boxes in Figure 5.4 in screen and world coordinates, respectively. Then from the equations: $y_s z_s = A1_s$, $x_s y_s = A2_s$ and $x_s z_s = A3_s$, we get:

(I) $x_s = \sqrt{A2_s A3_s / A1_s}$, $y_s = \sqrt{A2_s A1_s / A3_s}$ and $z_s = \sqrt{A2_s A3_s / A1_s}$ and analogous equations for the x_w, y_w and z_w dimensions.

Let N_x, N_y and O_x, O_y be the dimensions of the viewport and the orthographic viewing volume, respectively. Let K be $N_x N_y / O_x O_y$. Then we have:
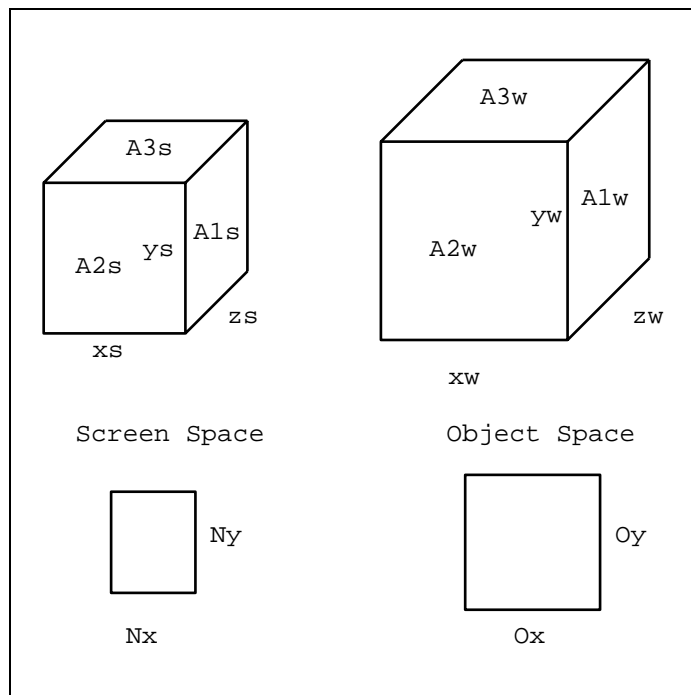
Figure 5.4: Boxes in screen and object spaces.

(II) $A1_s = KA1_w$, $A2_s = KA2_w$ and $A3_s = KA3_w$.

From (I) and (II) above we get $x_w = \frac{1}{\sqrt{K}} x_s$, $y_w = \frac{1}{\sqrt{K}} y_s$ and $z_w = \frac{1}{\sqrt{K}} z_s$.

The generation of pseudo-texture maps (PseudoTMRectangle type) involves a pre-processing of the original texture map. As stated before, the pseudo-texture map is an image that is pasted onto a triangular or quadrangular mesh (t-mesh and q-strip in the SGI jargon) in which every pair of triangles forming a rectangle corresponds to a pixel in the image. Therefore, if resolution of the image is high then too many meshed triangles will be needed making the rendering speed gain of using meshes disappear. To solve this problem we successively shrink the original image a user-specified number of times by convolving it with a Gaussian filter that averages the RGB components of the pixels. The new and

63
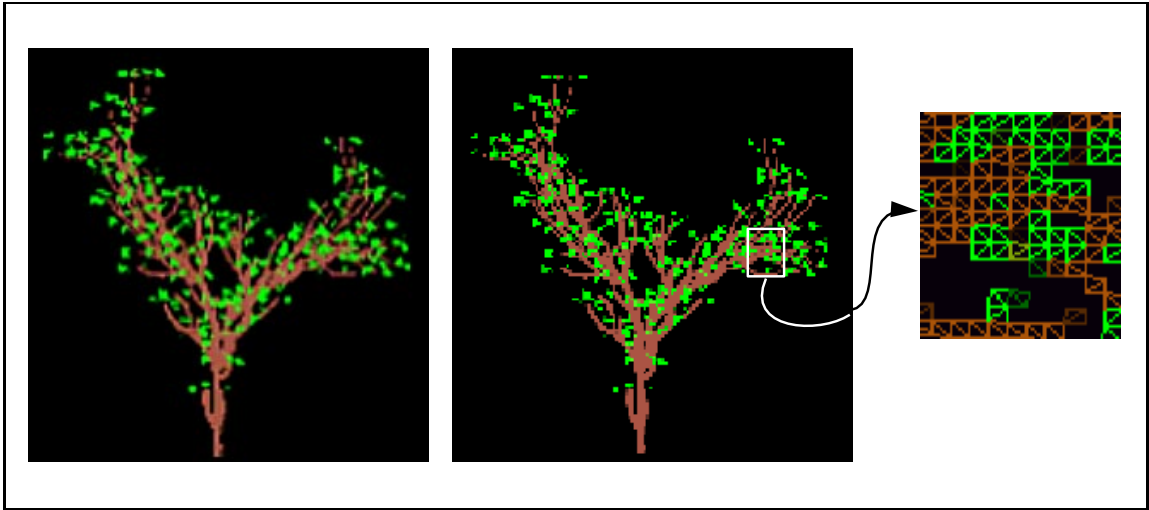
Figure 5.5: An example of a pseudo-texture map. Left the original texture map (TMRectangle). Middle a pseudo-texture map (PseudoTMRectangle). Right detail of the t-mesh.

smaller image resolution is then used to compute the triangular mesh shown in Figure 5.5. Although the process of shrinking the image is a little slow, it takes place when the model data is read into the walkthrough program and before the actual navigation of the model. As with the TMRectangles, pseudo-texture maps are pasted onto the faces of the object's bounding box.

The shrunk image can also be used as a texture map but the resulting image can appear more blurred than the pseudo-texture map since flat shaded meshed triangles are drawn without any further filtering (except the one already used to shrink the image).

Since when shrinking an image we are removing its high frequency components (its edges) this is a smoothing process, and for small image-space sizes the image flickers less when rendered as a pseudo-texture map then when rendered using the texture mapping hardware.

5.2.3 Guidelines for Impostor Selection

There are certain cases where specific impostors must be used and others where more then one impostor is suitable. We use two criteria to select impostors: The image-space size of the object and the viewpoint in object coordinates.

The size of an object in image-space N is the number of pixels resultant of the projection of the object onto the screen and can be approximated by: $\frac{N_x N_y}{W}\omega$, where N_x and N_y are the window dimensions, W is the maximum solid angle, i.e., is the solid angle that covers the entire viewing frustum and ω is the solid angle subtended by the object.

The solid angle ω subtended by an object can be approximated by $\frac{A\cos(\theta)}{d^2}$, where A is the area of the appropriate face of the objects bounding box, θ is the angle the faces normal make with the line of sight and d is the distance of the object to the viewpoint.

The maximum solid angle W can be approximated by: $\frac{A}{n^2}$, where A is the area of the intersection of the near clipping plane with the viewing frustum and n is the distance from the origin of the coordinate system to the near clipping plane. We compute A as follows:

(I) $\frac{h}{w} = \frac{N_y}{N_x}$ and (II) $\tan(\frac{fov}{2}) = h/n$ and (III) $A = 4hw$, where h and w are as in Figure 5.6 and fov is the field of view. Substituting h and w from (I) and (II) into (III) yields in $A = \frac{4n^2 \tan^2(\frac{fov}{2})}{a}$ where a is the window's aspect ratio $\frac{N_y}{N_x}$.

An expression for the image-space size N of an object is then: $N = \frac{N_y{}^2}{4\tan^2(\frac{fov}{2})}\omega$. Note that only the object's solid angle needs to be computed in real-time.

The viewpoint in object coordinates can easily be calculated by a simple inverse matrix.
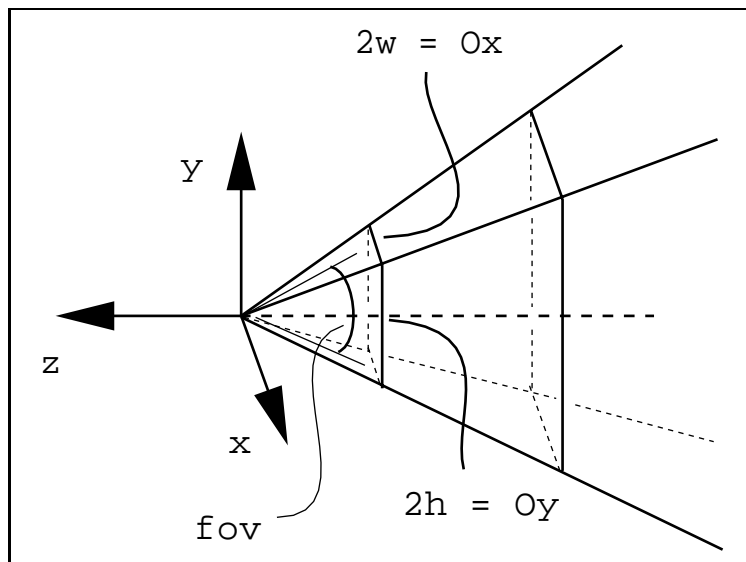
Figure 5.6: The viewing frustum after the viewing transformation.

For instance, if the modeling commands that place an object in world coordinates is a rotation, say, $R(\theta)$, followed by a translation $T(x, y, z)$, then the transformation that takes a point in world coordinates to object coordinates is just: $M = T(-x, -y, -z) \cdot R(-\theta)$. Therefore to obtain the viewpoint in object coordinates we just left multiply it by transformation matrix M which can be obtained directly from the graphics hardware.

Assuming that we have computed the image-space size N of the object and the viewpoint with respect to the object's coordinate system we give the following guidelines to select view-independent representations.

If N is less then a few pixels then the representation that must be used is the AvColorBox. There is no point in rendering any representation even slightly more complex than this since details will not show on the screen. If, however, the solid angle is greater then a pre-fixed percentage of the maximum solid angle W then this means that the object is right

in front of the viewpoint and needs to be rendered in full detail. So in this case the actual object (its highest LOD) must be selected. If different LODs are present in the model, then different image-space size thresholds for the objects may be used to select the appropriate LOD to be displayed.

Each view-dependent representation is only suitable for a particular viewpoint. If one texture map is provided for each face of an object's bounding box, we can determine what representation can be used as follows:

1. Along with each view-dependent representation that is generated in a pre-processing phase we associate a number corresponding to the region in object space the viewpoint was in when the representation was generated. These regions are illustrated in Figure 5.7. The number of regions used will depend upon the object, the application and the memory requirements of the machine on which the application will run.

2. During the walkthrough we determine the viewpoint with respect to the object's coordinate system and thus the region it is in. We then use this region number as an index in a table of regions associated to each object.

5.3 Summary

The object-oriented framework developed in this Chapter is based on the idea that an object (or cluster) can have many different representations associated to them, including
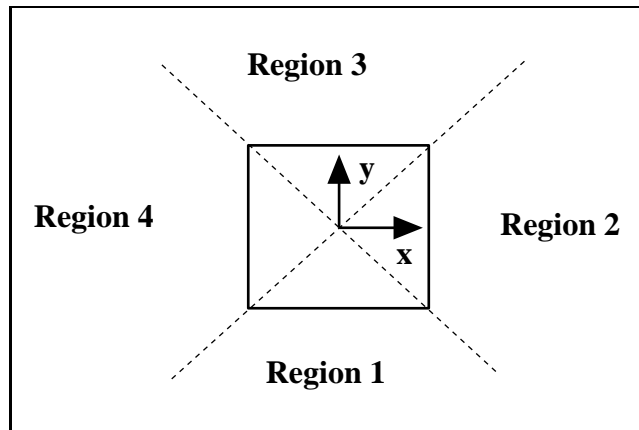
Figure 5.7: Possible viewpoint regions in object coordinates.

conventional LODs and special types of representation called impostors. Thus this frame-work provides a generalization of the LOD concept.

We have presented ways of automatically generating representations for objects and clusters, mainly the view-dependent ones, and criteria to select them in real-time.

Since this framework places no restriction upon the representations allowed for objects/clusters it accommodates both automatically generated and user provided ones.

Although any visual navigation system can benefit from using the framework described here, the problem of having more unoccluded objects inside the viewing frustum then the hardware is able to render interactively, is only addressed when we also use representations for clusters. This is the subject of Chapter 6.

6

Navigation System Design

The basic goal of this work is design a visual navigation system that is able to keep a user-specified uniform frame rate when displaying a complex environment. What we mean by a complex environment is one that at any given time may contain more visible graphics primitives inside the viewing frustum then the machine that the system is running on is able to render during the available frame time.

Before solving this problem by using drawable representations for objects as well as for groups of objects we first formalize it as an NP-complete tree traversal problem in Section 6.1 and present an outline of the overall design in Section 6.2.

The considerations upon which a hierarchy representing the model (or the model hierarchy for short) is built along with its construction is given in Section 6.3 while the algorithms used in its traversal are presented in Section 6.4.

Since the system we have designed is a predictive system that uses benefit and cost heuristics we present in Section 6.5 a methodology to analyze and validate our design. The
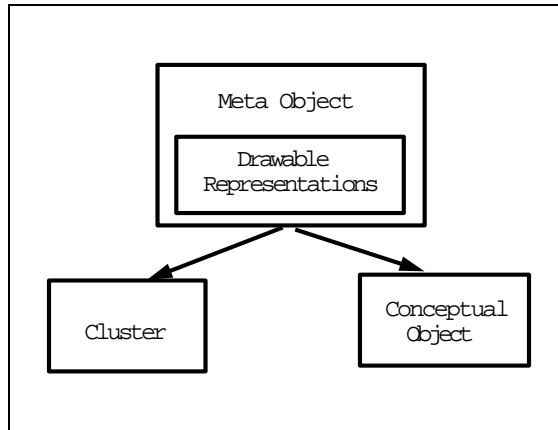
Figure 6.1: The meta-object abstraction.

last Section presents a summary of the Chapter.

6.1   Formalization of the Problem

We begin by defining a meta-object to be an entity that has associated to it one or more hardware drawable representations as in the framework described in Section 5.

A conceptual object is an instance of a meta-object that has a well-defined meaning to the walkthrough application, such as a building, a car, and so on, while a cluster of objects is an instance of a meta-object that represents a set of conceptual objects. This distinction between conceptual objects and clusters is made because conceptual objects are the building blocks which can be instanced by the user to assemble the virtual world by specifying their positions and orientations in space whereas clusters represent groups of conceptual objects after they have been positioned in space. Figure 6.1 shows the conceptual object and cluster abstractions connected to the meta-object abstraction by an 'is-a' relationship.

Again, a hardware drawable representation is an entity that can be rendered by the graphics hardware and is an abstraction that represents a conceptual object or a cluster of objects and has associated to it a rendering cost and a measure of its "contribution" to the simulation.

The rendering cost of a meta-object is defined to be the user time that the graphics hardware takes to render one of its representations on the screen and therefore is not directly influenced by variables such as the state of the operating system, interrupts or system load.

The contribution that a meta-object provides to the simulation refers to the contribution of a conceptual object and its representations . It can be measured by a per-object benefit heuristic such as described in Section 4.2 that incorporates view dependency as in Section 4.3, possibly incorporating the meaning of objects when viewed as a whole.

The model is then defined as a collection of conceptual objects at specific positions and orientations in space that form the environment in which the user navigates.

The model hierarchy is defined to be a tree structure whose nodes are meta-objects which provide multiple representations of the model, each representing it at a given rendering time and providing the user with a given perception of it. The root of this tree contains the coarsest drawable representations of the entire model with the lowest possible rendering costs. Likewise each parent node contains drawable representations of its children that have rendering cost less than the sum of the rendering cost of its children. While the root contains 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

71

cost.

More formally, the model hierarchy M is a tree structure that can recursively be defined by the following rules:

1. A meta-object that has no children is a model hierarchy with just one node, the root node.

2. Let $M_1, M_2 M_n$ be model hierarchies whose root nodes are the meta-objects $m_1, m_2 \ldots m_n$, respectively, that represent sets of conceptual objects and have associated to each of them the sets $r_1, r_2 \ldots r_n$ of drawable representations. Let m be a meta-object that represents the union of m_i and has associated to it a set r of drawable representations such that $Cost(Max(r)) < \sum_{i=1}^{n} Cost(Min(r_i))$, where $Max(r)$ is the representation that has the highest cost among those in r, $Min(r_i)$ is the representation that has the lowest cost among those in r_i and $Cost(x)$ is the rendering cost of representation x. M is then defined to be a model hierarchy if m is the parent of m_i for $i = 1..n$.

Given these definitions, we state the walkthrough problem as a tree traversal problem:

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

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

2. Only one node can be selected for each path from the root node to a leaf node, since each node already contains drawable representations that represent all its descendant nodes.

The problem as described by this formalization is an NP-complete tree traversal problem and is a variant of the "Knapsack problem". The candidate sets from which only one element will be selected to be put in the knapsack are the set of representations associated to each meta-object. The knapsack size is the frame time per frame that the selected representations must not exceed. The cost of each element is the rendering cost associated to a representation. The profit of an element is the accuracy of the representation plus the benefit of the object with which it is associated.

To solve this problem we use the framework described in the previous Chapter, appropriately extended by the meta-object abstraction as shown in Figure 6.1 and develop a model hierarchy building and traversal strategies. These issues as well as an outline of the whole system are examined in the next Sections.

6.2 System Outline

As in the model described in Section 2.4 the design of our navigation system also acknowledges the need of a pre-processing phase where the illumination of the model can be computed and a representation for the entire model can be built. This representation however allows for situations where there can be more visible objects for a given viewpoint than

can be rendered in real-time and assumes that no visibility information can be extracted from the model.

The illumination computation of the environment can be done by using the radiosity technique described in Chapter 2.3. In this case we would generate a new object that is, geometry and shading, for each object in the model and store it in an object database[1]. If hardware lighting is enough then we keep only the original objects in the database, instancing them when appropriate. The pre-computation of the global illumination of the environment however, is not addressed in this work.

The model hierarchy is designed so that during the walkthrough phase, geometry can be culled away without disregarding any visible objects inside the viewing frustum. This hierarchy uses and extends the framework described in Chapter 5 such that in the same way objects can have many different representations so can sets of objects. However, to actually gain rendering time, a representation for groups of objects needs to take less time to render then the sum of all actual objects that it is a representation of.

The model hierarchy can also be viewed as an extension of the LOD concept for the entire model viewed as a whole. 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 is represented by a single primitive. 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 previous approach

---

[1]This would increase the memory requirements for the walkthrough program and a strategy to swap objects in and out of memory may be required

described in [15].

During the interactive phase of the navigation system the model hierarchy is traversed and the selected nodes are sent to the graphics pipeline. As we shall see in Section 6.4 this will be done in two steps. First the visibility of each node (intersection with the viewing frustum) is determined, an initial drawable representation selected and its cost and benefit computed. In the second phase, a best first traversal algorithm descends the model hierarchy, examining the most "beneficial" nodes first, selecting nodes to render and accumulating rendering cost until the user specified frame time is reached.

The entire system is depicted in Figure 6.2. This approach is also a variation of the predictive approach since instead of using the time to render the previous frame to select representations to render in the current frame as in the reactive approach, we are estimating the time and quality of the next frame before rendering it. It also assumes that interactivity is more important then image quality in situations where there are too many visible graphics primitives inside the viewing frustum than the hardware is able to draw in real-time.

The rationale behind the construction of this hierarchy is given in Section 6.3 whereas how this hierarchy is traversed is presented in Section 6.4.

6.3   Design of the Model Hierarchy

To achieve interactive navigation of a complex model we need first to pre-process it in a way that conforms to our formalization of the problem in Section 6.1. Intuitively, we need
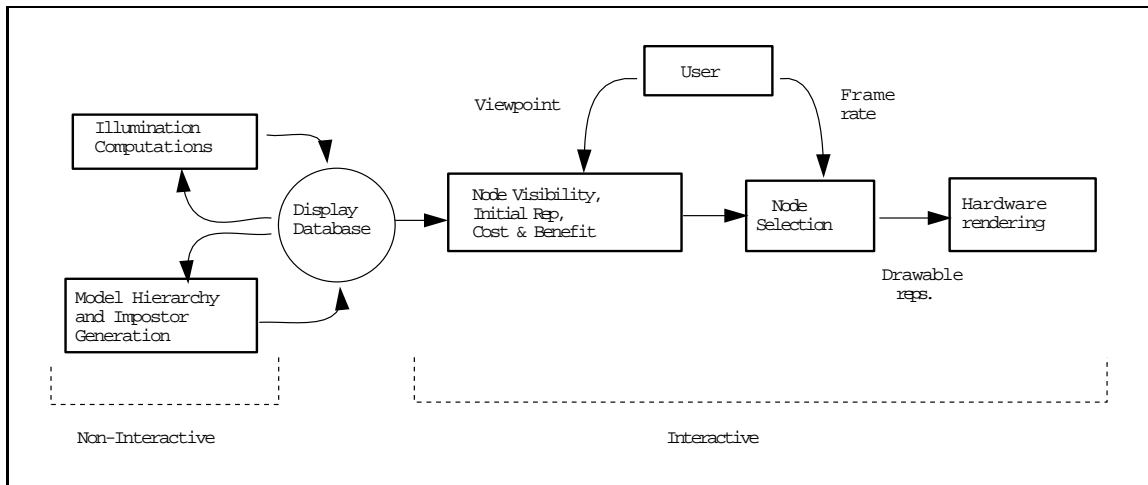
Figure 6.2: Diagram for the proposed system.

to design a data structure that possesses the following characteristics:

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

2. It needs to support objects as well as groups of objects with sets of low-cost hardware drawable representations associated with them, so that when there is not enough rendering time to draw the objects we can select representations for entire groups of objects to render.

76

6.3.1 Tree Structure

The characteristics above suggest a hierarchical tree structure that subdivides the entire 3D space into cells in which each cell (node) contains drawable representations of the clusters it represents.

The top level node of this hierarchy has to contain a representation of the whole model and the leaves have to contain, among their representations, the actual objects in the model. Each intermediate node contains representations of entire groups of objects that do not overlap and are faster to render than the sum of the rendering cost of its children as in our formalization in Section 6.1.

The proposed structure contains representations for the entire model at each level and can be viewed as a generalization of the level-of-detail concept for the entire model. In the top level or root node, we have a very coarse, very-low cost representation for the whole model. At the next level down we have a better representation (though more expensive) of the model and so on, so that at the leaves we have the best and most expensive representations for the model. 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.

Based on the discussion above, the tree structure that we decided to use a variant of an octree [21] in which instead of having actual objects associated to its nodes we have sets of hardware drawable representations for the objects that it spatially subsumes.

The octree we actually use in our implementation allows us to have a different number

of subdivisions for each x, y and z axis, and is constructed in a bottom-up fashion instead of the traditional top-down recursive way. The reason why this hierarchy is constructed in a bottom-up fashion is just to facilitate the monitoring of the hierarchy construction. As we shall see in Chapter 7 our hierarchy building implementation shows what objects are being grouped together into clusters. If the clusters contain too many or too few objects we can stop the program and input a better octree cell size.

This octree provides a hierarchical spatial partitioning of the entire model into cells and is also a bounding volumes hierarchy in which each node corresponds to the bounding volume of all the objects that are completely inside it. Also, since each node contains drawable representations for object(s) this tree has the dual purpose of serving as a rendering aid and that of culling sets of object against the viewing frustum. Figure 6.3 presents the model hierarchy for a city.

6.3.2 Hierarchy Building

Up to now, we have only talked about the organization of the tree structure and what information its nodes contain. Nothing has been said about the criteria used to group objects into clusters of objects represented by tree nodes. Our current implementation only takes into account the nearness law of organization presented in Section 4.4 [2] and our model hierarchy building program is designed to cluster together nearby objects first as shown in Figure 6.4. Note that representations are generated only if there are more then one object

---

[2] Other laws can also be considered although their use is not as straightforward as the nearness law.
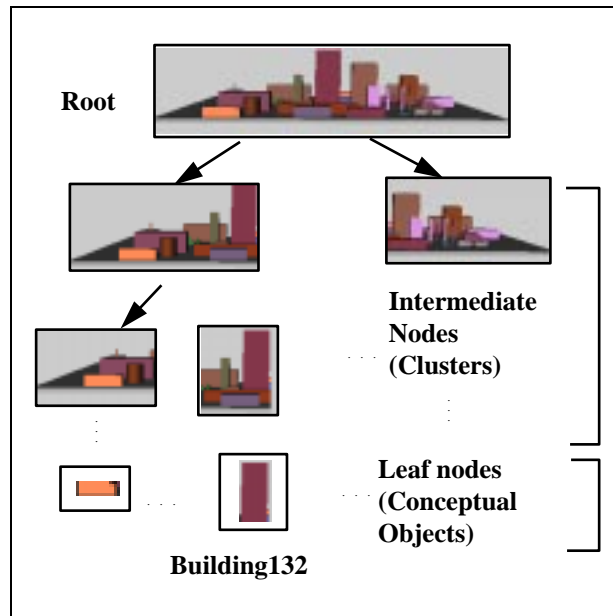
Figure 6.3: Currently implemented model hierarchy for a city. Here it is assumed that the representation at each parent node cost less to render then the sum of its children.

totally inside each cell. Single objects inside a cell as well as objects on cell boundaries will be grouped in the next levels up in the hierarchy. Figure 6.5 shows the structure of subtree A depicted in Figure 6.4.

Note that the size of an octree node is usually smaller then the octree cell size since it corresponds to the bounding box of the objects totally inside the cell. This has the advantage that if there is a group of objects whose bounding box is much smaller then the cell size then it is more likely that the cell will be discarded when checked against the viewing frustum.

It is important to note that each time we cluster[3] objects together we always take into account the actual objects that the cell subtends instead of previously computed clusters.

---

[3]What is meant by clustering is basically the generation of impostors for the group of objects.
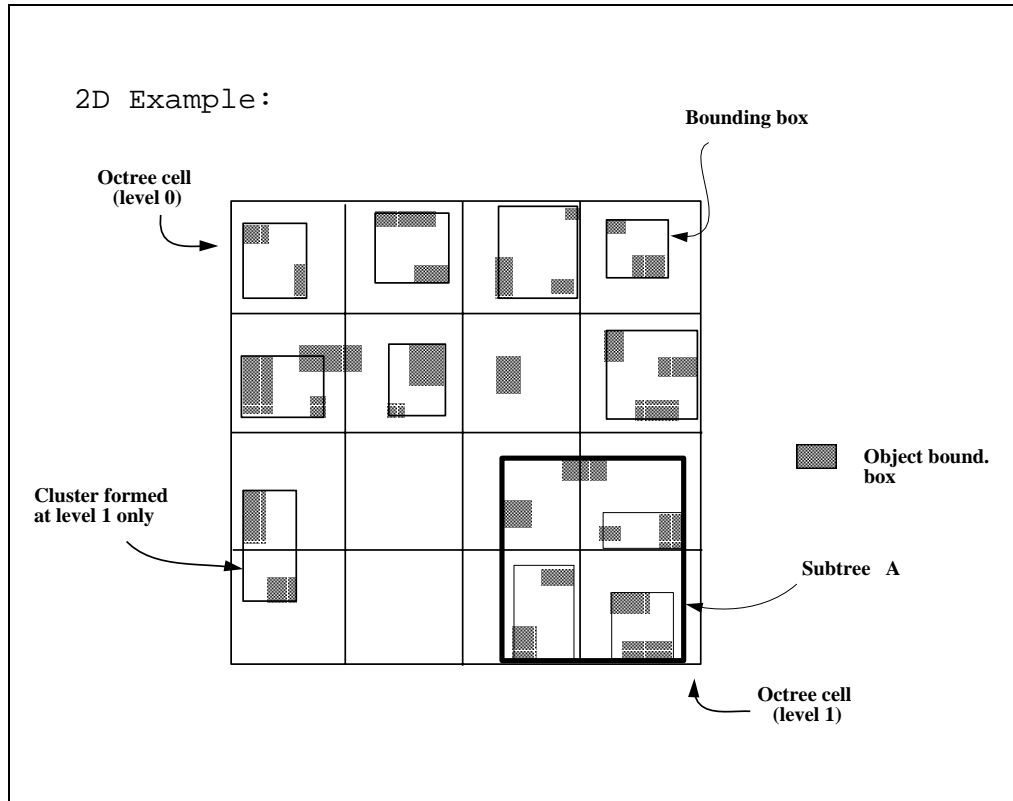
Figure 6.4: Generating the model hierarchy octree. Representations are generated for cells with more than one object.
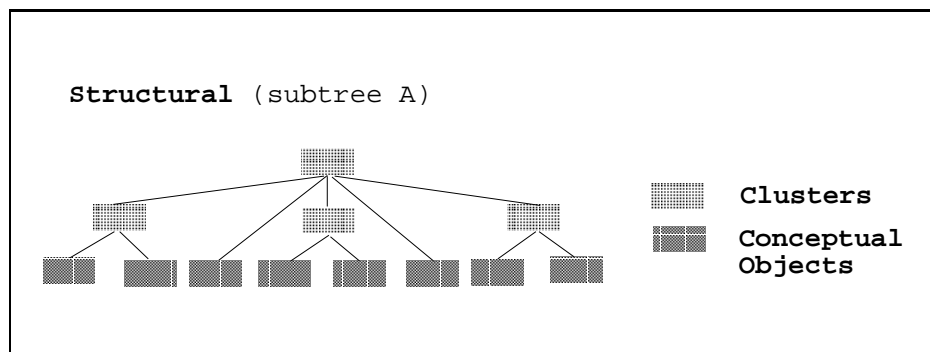
Figure 6.5: Subtree A as depicted on Figure 6.4.

The model hierarchy building algorithm works as follows: we start by creating what we call a child's list that contains all the conceptual objects in the model with their bounding boxes. This initial list will correspond to the leaves of the tree. Then we compute an initial cell (and thus the tree height) size based on the proximity of the objects in the x, y and z directions, since we want to start grouping objects that are close together. The initial cell size is a function of the bounding box for the entire model and the number of divisions of the x, y and z axis.

The child's list is used to generate the next level of the tree in a bottom-up fashion as follows: For each level of the tree and for each cell in that level, we get the set of objects that are completely inside the cell. If this set is empty we move on to the next cell. Otherwise we compute the bounding box of the objects in the cell and discard it if the bounding box is already in the child's list, since impostors for that set of objects has already been computed. If it is not in the list we create impostor representations for the objects inside the cell and associate them to the cell.

In our implementation these impostors are texture maps (textured clusters when they refer to groups os objects) obtained in the same way as described in the previous chapter, although they can be any drawable representation, even ones generated by an artist, since our only requirement is that the costs to render a node is less then the sum of the rendering cost of its children. After impostors are computed, we make the node point to its children and remove them from the child's list. We then add the new node to the end of the child's list and repeat the process until we obtain a single cell with an impostor representation for

the entire model. It is important to note that at each time we cluster objects we always take into account the actual objects that the cell subtends instead of previously computed clusters.

The octree constructed by the above algorithm has the following properties:

Height "H":

$H = MAX(divx, divy, divz)$, where $divx$, $divy$, and $divz$ are parameters such that the number of divisions in x, y, and z axis of the bounding box for the entire model, are 2^{divx}, 2^{divy}, and 2^{divz}, respectively.

Number of nodes "$N(l)$" at level "l":

$N(l) = 2^{MAX(0,divx-(H-l))}2^{MAX(0,divy-(H-l))}2^{MAX(0,divz-(H-l))}$, where the root node corresponds to level 0. In the case where the number of divisions is equal in all axis, we get the usual octree, where $N(l) = 2^l2^l2^l$.

Number of leaves "$N(H)$":

$N(H) =$ number of conceptual objects in the model.

6.4 Traversal of the Model Hierarchy

Due to the NP-complete nature of selecting representations to render from the model hierarchy, we have devised a heuristic algorithm that quickly (in less than the frame time) traverses the model hierarchy. This algorithm selects representations to be rendered, accumulating rendering cost until the user-specified frame time is reached. When this occurs,

```
Routine BuildModelHierarchy(divx,divy,divz)
{
  GetInitialCellSize(divx,divy,divz,bb);

  ni = 2^{divx}; nj = 2^{divy}; nk = 2^{divz}; // Get # cells in each dimension

  Generate bounding box for all objects in model and insert in
  the child's list(chl).

  pass = 0;
  Repeat (until size of chl = 1) {
      // Compute the x, y and z limits for the current pass.
      limi = ni/2^{pass}, limj = nj/2^{pass} and limk = nk/2^{pass}.

      GetCellIncrements(); // In x, y and z dir.

      x = minimum of model bounding box in x.
      for i = 0 to limi {
          y = minimum of model bounding box in y.
          for j = 0 to limj {
              z = minimum of model bounding box in z.
              for k = 0 to limk {
                  Get objects entirely inside cell and
                  the cluster's bounding box.
                  if (bounding box inside chl) continue;
                  Compute the impostor representation.
                  Remove children from chl and add to current cell.
                  Add cell (bbox) to the end of chl.
                  Increment in z;
              }
              Increment in y;
          }
          Increment in x;
      }
      Increment pass;
  }
}
```

Figure 6.6: Pseudo-code for building the model hierarchy octree.
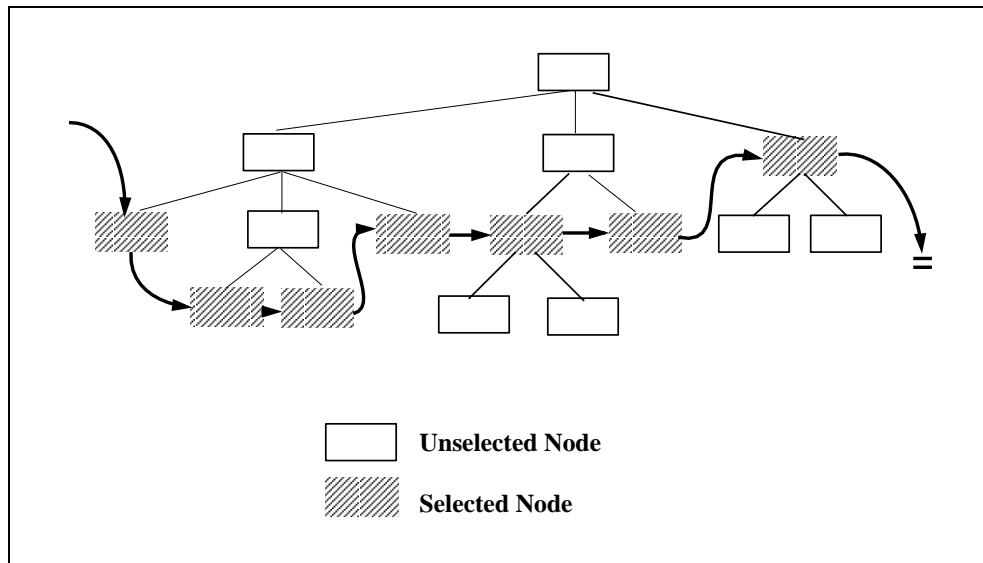
Figure 6.7: Tree representing the model hierarchy and the rendering list.

the algorithm stops and sends a list of representations to the graphics pipeline. Figure 6.7 presents an example of our model hierarchy together with a list of the nodes selected to be sent to the graphics hardware for rendering.

Intuitively, since we want to spend most of the frame time rendering drawable representations of objects/group of objects that contribute the most to the perception of the model, the traversal of the model hierarchy should be guided by a benefit heuristic.

While the benefit heuristic for a conceptual object can be measured by quantifying the issues mentioned in Section 4.2, the benefit of a group of objects is more subtle and difficult to quantify. Since at each point of the tree traversal we wanted to explore the branch where the most "beneficial" objects are, we made the benefit of a cluster be the benefit of its most "beneficial" object.

Since our tree traversal should begin at the coarsest representation of the model and

84

descend its children in order to get better representations according to the available frame time, at each given node we need to know the benefits of its children to select the best branch to descend first. The problem is that the benefit heuristic associates benefit not to clusters but to conceptual objects that are at the very bottom of the tree on its leaves and high up in the hierarchy we do not know to which branches of the tree the most beneficial objects belong. Because of this, we have decided to break this selection task into two phases. First, we recursively descend the tree computing the visibility of each node, assigning benefit, and initial representations to be rendered. In the second phase, at any given time we descend the most promising branch of the tree structure in a best-first fashion, accumulating frame time as representations to render are selected. If at any point in this traversal frame time is still available then a better representation is selected. On the other hand, if frame time is running out, cluster representations are selected for rendering.

6.4.1 Assigning Representation, Cost, Benefit and Visibility.

In this first phase of the selection process, we recursively descend the model hierarchy in a depth-first manner, associating visibility to each node in the tree, and an initial drawable representation and benefit to visible nodes.

The visibility of nodes are computed by checking if the bounding box in eye-coordinates of the bounding box of the object intersects the viewing frustum as described in Section 7.3.3. We determine if a node is visible if at least one of its children is visible.

Since the leaves represent single objects, their benefits are computed as a weighted

average of the factors intrinsic to objects as described in Section 4.2. The benefit value associated to an intermediate tree node is taken to be the maximum value of all the benefits of its children[4].

At a given point in the simulation a view dependent and a view independent representation for an object is selected using the criteria specified in Section 5.2.3. The rendering cost and accuracy of drawable representations that are stored with each representation in the model are used to select which of these two representations will be assigned to be the initial representation of the node. The representation that has the highest accuracy/cost ratio is selected to be the initial representation and the other, if its accuracy is greater than the selected one, is kept as a backup "better" representation, so that if in the next phase (described below), if there is still enough frame time left we try to improve on the initial choice.

In cases, where it is determined that the benefit of the object falls or surpasses a threshold (cases such as the ones where an AvColorBox or the full detail LOD should be used), then this mechanism is by-passed and no accuracy/cost ratio is examined.

This first phase is implemented as a recursive algorithm and its pseudo-code is given in Figure 6.8.

A clear disadvantage of this algorithm is that its complexity is $O(n)$[5], where n is the number of objects (not graphics primitives) in the model, that is, in the worst case where all

---

[4]We could also assign the benefit to be the sum of all the benefit of the node's children. The approach we adopted insures that the most beneficial object will get the most "attention" first whereas this other approach insures that this attention is given first to the set of objects that is most beneficial.

[5]It runs in time proportional to n.

```
Routine AssignRepCstBenVis(node)
{
  if (node is not visible) return

  AssignRepresentation(node);
  AssignCost(node);
  if (node is a leaf) {
    AssignBenefit(node);
    return;
  }

  node.Benefit = node.CostOfChildren = node.Visibility = 0;
  for each child of node {
    AssignRepCstBenVis(child);
    if (child is visible) {
      node.Visibility = 1;
      node.Benefit = MAX(node.Benefit, child.Benefit);
      node.CostOfChildren += child.Cost;
    }
  }
}
```

Figure 6.8: Pseudo-code for assigning representation, cost, benefit and visibility to the model hierarchy tree.

the objects in the model are visible we need to visit all the leaves of the tree. This problem can be minimized by parallelizing this search since each branch of this tree is independent from each other. A better algorithm that merges this algorithm and the one described below is described in Section 7.5.

6.4.2   Best-First Tree Traversal.

In the second phase of the selection of objects to be rendered, we use the information obtained in the previous phase for each node of the model hierarchy to implement an efficient 'best-first' tree traversal.

To implement this strategy, we make use of a list of meta-objects organized in decreasing order of benefit (computed in the previous phase). We keep accumulating frame time as we select representations to render and whenever the time required to render the children of a node plus the total accumulated time so far exceeds the frame time we insert the representation for the node in the rendering list and move on to the next node.

The algorithm first explores the branches of the tree connected to the most beneficial nodes as follows: Start by inserting the root node in the list and setting the total rendering cost to be the cost of rendering the initial representation associated to the root node. We then process this list until it is empty. We remove the element in the front of the list and discard it if it is not visible.

If the node is a leaf node (containing a conceptual object) and the initial representation assigned is not mandatory, (i.e., the object is neither too close so that the full detail version

88

of the object has to be used nor too far away so that only low cost representations such as the AvColorBox should be used) we check if there is still rendering time left to select a better representation for the object. In the positive case we select to render (insert in the rendering list) the higher accuracy representation for the node and add its rendering time to the total accumulated rendering time.

If the node contains representations for a cluster of objects, we check if instead of rendering the cluster representation we still have time to render all of its children, i.e., the total accumulated time plus the cost of rendering the node's children does not exceed the frame time. In the positive case, we insert each of its visible children in the list ordered by each ones benefit and add their cost to the total accumulated rendering time. Otherwise we insert the cluster's representation into the rendering list.

Note that at each point in this traversal, a complete representation of the scene is stored in the list of meta-objects and whenever there is frame time left to render the children of a node, before adding the cost of the children to the total accumulated cost we subtract the cost of the initial representation for the node.

The pseudo-code for this 'best-first' tree traversal that computes a list of representations at each frame is given in Figure 6.9. The "*" on the pseudo-code indicates where the temporal coherence mechanism described in Section 6.4.3 should be inserted.

This algorithm tends to concentrate most of the frame time on the most "beneficial" objects. While navigating through the model the viewpoint can randomly get close or far away from objects that require most of the frame time. This sometimes causes a seemingly

```
Routine SelectRepresentations(root)
{
  totalCost = root.Cost;

  Insert the root node in ordered list L.
  While (L is not empty) {
        Remove a 'node' from L's head;
        if (node is not visible) continue
        if (node is a leaf) {
          if (current rep. is not mandatory) {
            if (totalCost - node.Cost + betterrep.Cost <= FrameTime) {
              (*)
              Set curr. rep to be the betterrep;
              Add the rep.'s cost to the totalCost;
            }
          }
          Insert object into the rendering list;
          continue;
        }
        costChildren = node.costChildren - node.Cost;
        if (totalCost + costChildren <= FrameTime) {
          totalCost += costChildren;
          for all children of node
            if (child is visible)
              Insert child in L;
        }
        else
          (*)
          Insert cluster into the rendering list;
  }
}
```

Figure 6.9: Pseudo-code for selecting representations to render.

random switch from a cluster representation to the representations of the actual objects (or vice-versa) and back from frame to frame resulting in an undesirable flickering of the two representations. This is minimized by a simple mechanism that explores the temporal coherence that exists from frame to frame and is described in the next Section.

6.4.3   Temporal Coherence

The idea is to discourage switching from representations for parent nodes to representations for children nodes and we keep one counter to count how many times the program decided to switch from clusters to objects and another to count how many times it decided to change from objects to cluster. The actual switching is only allowed if either counter exceeds a pre-fixed threshold. This prevents the switching from one representation to another in one frame and back in the next frame as the viewpoint moves in the model.

This simple temporal coherence mechanism is implemented as a routine that delays the switching from representations of parent nodes to representations of children nodes only. The delayed switching from children representations to cluster representations is not implemented since it would occur in a situation where most of the frame time has already been allocated and this delay would invariably reduce the frame rate.

This routine is part of the representation selection algorithm and should be inserted in the place marked by an asterisk in Figure 6.9 and its pseudo-code is given in Figure 6.10.

```
Routine DelayClusterSwitching(node, dir)
{
  If (dir == 0) { // Changing from cluster to objects.
    If (node was rendered in last frame) {
      Increment cluster to object counter(node.ctoo);
      if (node.ctoo < CTOOTHRESHOLD) {
        Insert node in rendering list;
        return(1); // Delay switching.
      }
      else {
        Mark node as not rendered;
        node.ctoo = 0;
        return(0); // Switch.
      }
    }
    return(0); // Proceed as usual.
  }
  else { // Changing from objects to cluster.
    If (node was rendered in last frame)
      return(0); // Proceed as usual.
    else {
      Increment object to cluster counter(node.otoc);
      if (node.otoc < OTOCTHRESHOLD)
        return(1); // Delay switching.
      else {
        Mark node as rendered;
        node.otoc = 0;
        return(0); // Switch.
      }
    }
  }
}
```

Figure 6.10: Pseudo-code of a simple temporal coherence mechanism to avoid frequent switching from cluster to objects and vice-versa (not implemented).

6.5 Validation of System Design

Up to now we have described how to design a system to achieve a uniform and approximately constant frame rate while displaying the "best" possible image constrained to the available rendering time per frame. However, since the success of our strategy is based on heuristics, i.e., meta-knowledge, we have no scientific guarantee that a system designed in the way we are proposing will achieve its purpose.

Typically, constraint satisfaction problems that use meta-knowledge, to find an approximate solution can be viewed in the way depicted in Figure 6.11. The sets of elements in a space of candidate sets are submitted to a selection process that uses meta-knowledge (heuristics) to select sets of elements, each containing one element from each candidate set, so that each set satisfies the given constraint. These sets are then applied to the problem and the results are observed.

To be sure that the meta-knowledge yields a near optimal result, all possible sets of elements that satisfy the constraint need to be applied to the problem and their error with respect to the ideal result (one that would be obtained if there were no constraints) measured. If we view the error metric as an energy function $E(S_1, S_2, ..., S_n, constraint)$, whose arguments are elements in the space of candidate sets and a constraint, then we can be sure that our meta-knowledge is performing correctly, if and only if for all possible values of our 'constraint' variable, the global minimum of 'E' corresponds to the points chosen by our heuristics. If that is not the case, then by analyzing the behavior of this error function, we can propose changes to our meta-knowledge.
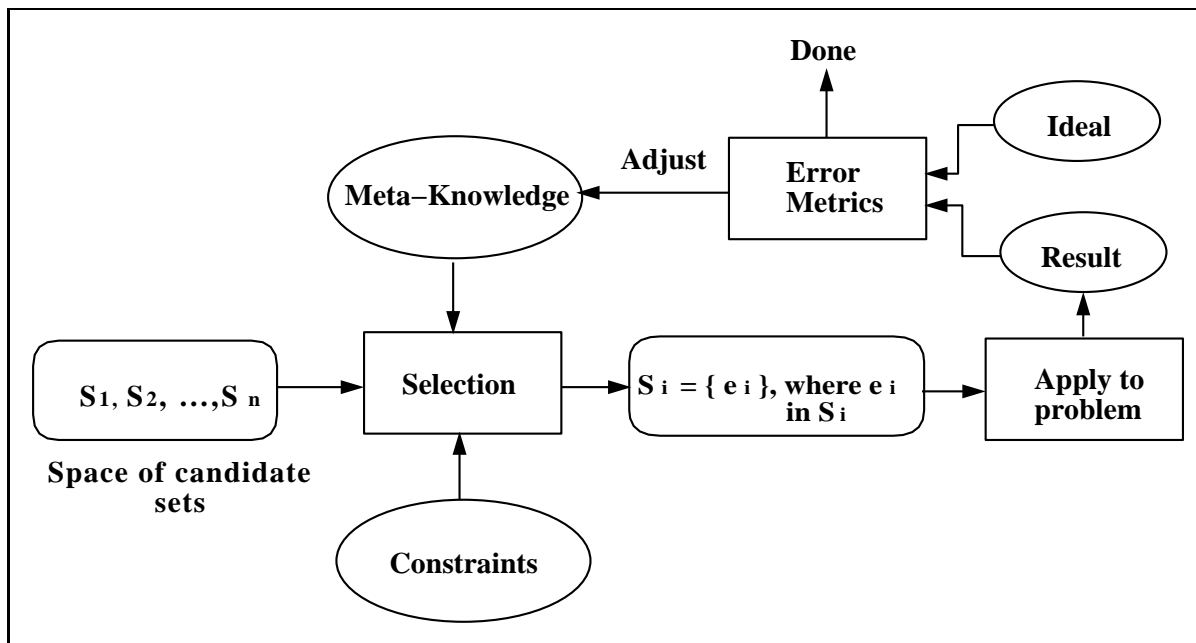
Figure 6.11: Strategy to analyze a constraint-satisfaction system.

In our case, the candidate sets are sets of representations associated to each object in the model. The selected sets are sets of representations for objects (or clusters), that contain one representation for each object (or cluster) for the entire visible scene, subject to a frame time constraint.

To determine if we are achieving our goal, we need to show that our meta-knowledge (heuristic + selection algorithm) is selecting the set of representations, among those that satisfy our frame time constraint, that yields the "best" picture, for all possible frame times.

Furthermore, we need to show that our selection algorithm is able to select representations that when rendered will not exceed the frame time limit. This is just a matter of measuring the rendering time for each frame. The error metric in this case is simply the difference of this time to the desired frame time. In Section 7.4, for a given model and a

pre-determined path we present a graph of the frame versus frame time.

To determine if the best possible image is being achieved at each frame we will need a way of comparing an image with the image that would be obtained if the time per frame were infinite. Again we resort to image processing techniques to determine the error between two images. Since we just want to know which image from a set of images is the most similar to the infinite time image, we can use the Euclidean distance square between two images. This standard similarity measure, also known as correlation[6], is applied to blurred versions of the two images to avoid the difficult correlation of the high frequency components on the two images.

Ideally, to validate our benefit heuristics, for a given scene constrained to a given time 't', we would have to measure the error of all possible selection of impostors that make the rendering of the scene meet 't'. We would need to do this for all possible times, and show that the set of representations selected by the heuristics, yields the global minimum of the error function for each of the times 't'. The ideal proof is not feasible. Instead, since we only have discrete possibilities for impostors, we will use discrete times, and select representations for only two objects. For each time 't', we select representations of the two objects that meet the time constraint and check if the selection made by our benefit heuristic and hierarchy traversal algorithm corresponds to the global minimum of the error function for that particular time. If, for all possible times 't', our selection strategy resulted in the global minimum of the error function, we can be confident about its use.

---

[6]This metric is sensitive to properties of the image such as brightness, and slight changes in shape and size of objects (See [4]).

Time	Selection
t = 6	(3,3)
t = 5	(3,2), (2,3)
t = 4	(3,1), (2,2), (1,3)
t = 3	(3,0), (2,1), (1,2), (0,3)
t = 2	(2,0), (1,1) (0,2)

Table 6.1: Possible set of representations that achieve a particular rendering time.

Our experiment involves a house and a tree object, each of which, can be assigned four different representations, namely, the full detailed object, a texture map (TMRectangle), an average color box (AvColorBox), and no representation, i.e., object is not rendered.

A rendering time is associated to each of these representations and we assumed that representations of the same kind cost the same for both objects. Costs are assigned to these representations in decreasing order of complexity from three (full detail) to zero (no rendering). We then have the configuration shown in Table 6.1.

In our test scene, we have placed the tree close to the viewpoint and the house far away. We are evaluating two components of our benefit heuristic (another simplification), namely, distance to view point and accuracy of representation.

In this situation our algorithm in a first pass would select a representation based on its cost and accuracy and the distance of the object to the viewpoint. The algorithm would select a view-dependent and view-independent representation for each object, and set the one with highest accuracy/cost ratio to be the initial representation.

For the tree, the texture map would be selected and since the house is far away an average color box would be appropriate. In the second pass available time would be first
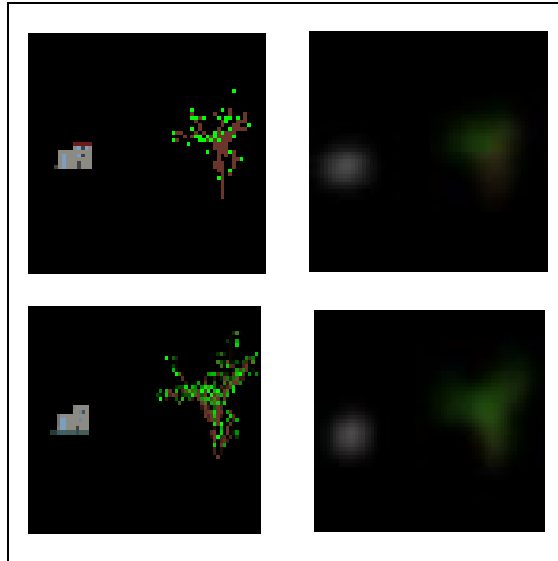
Figure 6.12: Two images and their blurred versions.

given to the tree (since it has higher benefit) and then to the house. Thus, for times six, five, four, and three, the representations selected would be (3,3), (3,2), (3,1), and (2,1), respectively.

For time t = 2, since there is not enough time to render the initially selected representations, the algorithm would render a textured cluster instead. For each of the possible representations, we have computed the correlation between the blurred version of image with the blurred version of the infinite time image. Two of these images with their respective blurred versions are shown in Figure 6.12. The results of this comparisons are shown in Figure 6.13, where a curve for each time on the surface is shown. Examining this surface graph, we see that the points chosen by our benefit heuristic corresponded to points of global minimum on the curves for each time. This is further illustrated by Figure 6.14 which shows correlation versus representation for each time.
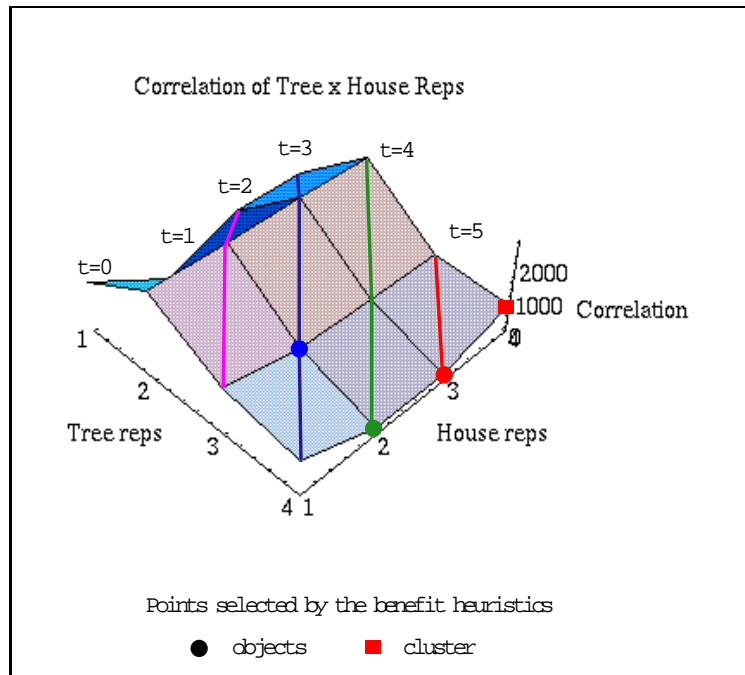
Figure 6.13: Correlation surface of two images. The points selected by our benefit heuristic and selection algorithm correspond to minima of the discrete time curves.

It is interesting to note that the cluster representation for t = 2 is slightly better than the best on the curve. This suggests that in some cases it would be better not to render a given representation than render the cluster version of the group, if the error introduced by the latter is too high. However, if there are too many visible objects that can not be rendered for a given frame time, then large areas on the scene will be blank. This is unlikely to be preferred to a textured cluster representation of the objects in that area.

In general, the results of the comparison of the output of the benefit heuristic with the ideal image needs to be analyzed in a way that would enable us to improve the benefit heuristic used. For instance, since the benefit of an object is computed as an weighted average of all the factors described on Section 4.2, such an analysis could suggest ideal
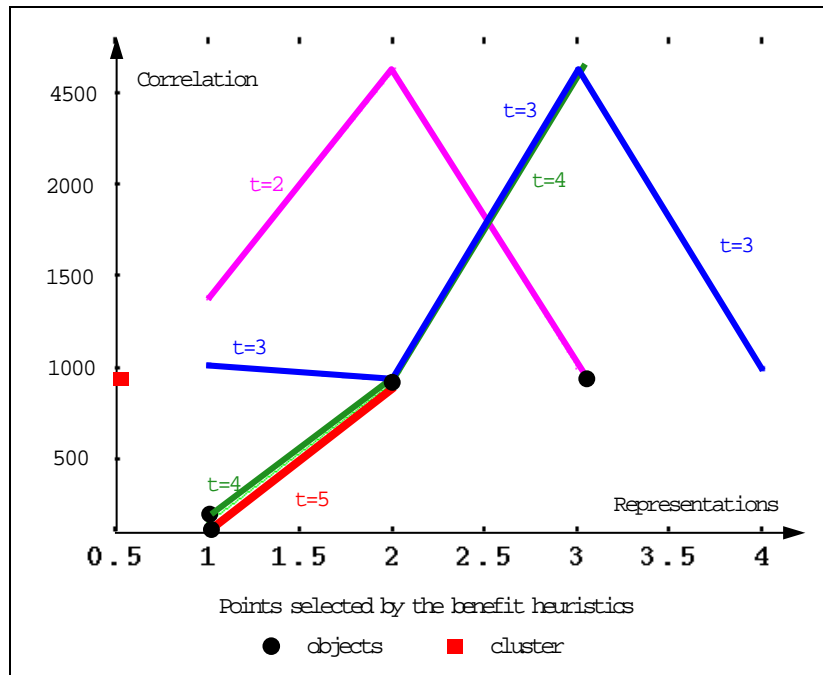
Figure 6.14: Plot of the correlation of two images for four different fixed times.

values for the weights of the components of our per-object benefit heuristic.

An error metric similar to the one used here could also be used to suggest that other factors might influence the benefit of an object. For instance, despite the fact that our benefit heuristic assigns low benefit to objects in the periphery of the field of view, they should have their benefit increased if they are moving since our visual system is able to detect motion even in the periphery of view. A temporal error metric could suggest the addition of a component to the benefit heuristic that would increase the benefit of objects moving in the periphery of view.

Nevertheless, this experiments is an indication that the meta-knowledge embedded in the benefit heuristic and selection algorithm is indeed achieving its purpose.

99

6.6　Summary

In this Chapter we presented the architecture of a visual navigation system that addresses the problem of rendering an environment interactively that can have more unoccluded primitives inside the viewing frustum then the hardware's ability to render them in real-time and our design does not assume that visibility information has been extracted from the model and therefore is particularly useful for outdoor environments.

By extending the concept of having impostor representations for objects to group of objects and finding representations for these groups that take less time to render then the actual objects, we are able to create a hierarchy for the entire model in which each node costs less to render then the sum of its children.

In cases where there are not enough rendering time per frame to render representations for objects, our hierarchy traversal algorithm starts to select textured cluster to render. This prevents large blank areas to appear on the scene.

For a simple scene with two objects, we were able to prove that our benefit heuristic together with our representation selection algorithm achieved our goal of obtaining the best possible image subjected to a fixed rendering time constraint. This increases our confidence that in more complex situations our system will perform well.

7

Implementation Details

This research has resulted in the implementation of three distinct programs. The first program builds the model hierarchy and generates impostor representations for individual objects and textured clusters for groups of objects. The second is a program that reads in the model hierarchy and can display each of its nodes with the purpose of verification and measurement of the rendering cost and accuracy of representations for a variety of viewing directions. The third program does the visual navigation based on files generated by the two previous programs. The formats of these files are presented in Appendix C. The first two programs are executed in a pre-processing phase while visual navigation is done in real-time.

The platform used was a four processor[1] SGI Onyx workstation with a RealityEngineII graphics board. All programs are implemented in C++ and use GL [41] for rendering and implement the framework described in Chapter 5. Both the walkthrough and cost/accuracy

---

[1] Each processor is a 150 MHz MIPS 4400.

measurement programs have an X-Motif user interface described in Appendix B.

7.1   Model Hierarchy Building and Representation Generation

The program that builds the model hierarchy implements the hierarchy building algorithm described in Section 6.3 and generates impostors as described in Section 5.2. It uses two GL windows, one to create impostors and the other merely to illustrate the process of building the model hierarchy, as shown in Figure 7.1. In this Figure, the right window displays the objects/clusters and computes texture maps for each of the sides of their bounding boxes while the left shows octree cells. In this image, the dots represent objects that were not "clustered" yet. The purple square with green dots represents the bounding box of the objects (in green) that completely fit inside it and the "red" band represents groups of objects already "clustered".

Objects are orthographically projected in six directions (perpendicular to the faces of the object's bounding box) and surrounded by a square that delimits the object's image which is then grabbed as a texture map. The resolution of this image can be adjusted by pressing the up and down arrow keys. Since texture map memory is a limited resource, we usually decrease the resolution of images that do not present "too many" details and increase the resolution of those which do. After the entire hierarchy is built, each of its levels is displayed and saved in files for use by the other two programs.
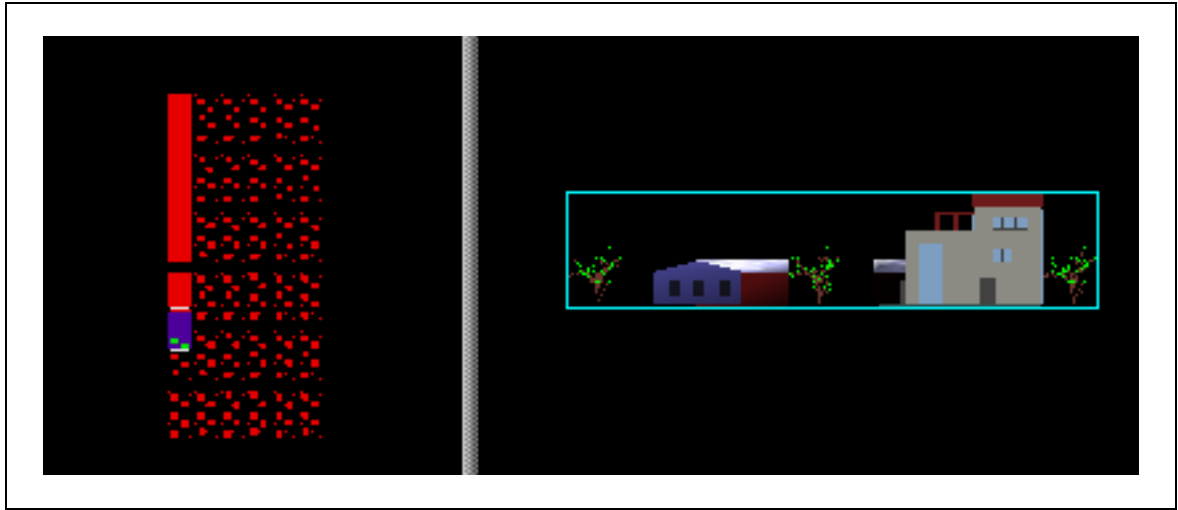
Figure 7.1: Model hierarchy building and representation generation.

7.2 Model Hierarchy Visualization and Measurement

The purpose of this program is three-fold. It is designed as a verification program to visualize the octree (model hierarchy) and the representations for all its nodes. By clicking arrow buttons on an X-Motif interface we can display on the graphics window any representation of any node in the hierarchy at any viewing direction.

This program is also used to measure the rendering cost of impostors. In Section 3.4, we were able to come up with a formula which approximates the cost of rendering an object composed of triangles. In general however, objects can be composed of many polygons, each of which can have a number of vertices. Furthermore, these polygons, can be simple, convex, concave, and in general non-planar[2] and will first have to be broken down to triangles before

---

[2]A polygon is: simple, if edges do not cross each other; convex, if a line connecting two interior points is also in the interior of the polygon; concave: if it is not convex; non-planar, if its vertices do not lie on the same plane.

being sent to the graphics pipeline. Our initial heuristic did not prove to be very effective. Therefore, and in the context of Section 6.5, we have decided to avoid dealing with the issues mentioned above with respect to arbitrary polygons by changing our cost heuristic (meta-knowledge). Instead of computing rendering cost, we measure it for each representation for each object/cluster and store these costs in a file to be read by the walkthrough program.

Finally, this program serves the purpose of measuring the accuracy of representations according to different viewing directions by performing the sequence of image processing operators described in Section 4.3.

To measure the cost of a representation, we first select the appropriate meta object and the desired representation by pressing up and down arrow buttons on the control panel. The appropriate meta object representation appears on the graphics window. We can then position the representation using the mouse, specify the number of times it will be drawn[3] and start the time measurement. The cost[4] in miliseconds is then displayed in the appropriate field.

To measure the accuracy of representations, we put the the program in "accuracy" measurement mode. When in "accuracy" mode, the graphics window is divided in two, one for displaying the image of the actual object(s) and the other for displaying the impostor image that we want to compare against. We select the meta object and impostor representation

---

[3]This is to prevent our measures to be dependent of the state of the machine during the measurements and the workstation's clock resolution.

[4]Using this cost measurement program we realize that the raster stage of the machine we are using (RealityEngine) is very fast. The time difference between rendering an object at a position where it has the highest image size and in a position that it occupies just one or two pixels was around 0.2 miliseconds for all the objects we have used.

and the viewing direction from which the meta object will be rendered. Currently we use five directions, four of which are perpendicular to the lateral faces of the meta object's bounding box and one perpendicular to its top face. When ready, we start the sequence of image processing techniques explained in Section 4.3 in both windows. A similarity value is output as a result of this process. Both cost and accuracy measurements can automatically be made for the entire model.

7.3 Visual Navigation

The walkthrough program implements the framework described in Chapter 5 and the traversal algorithms described in Section 6.4. The benefit of an object is computed as an weighted average of its distance to the viewpoint, distance to the center of the screen, and semantics.

A smooth animation is achieved by selecting and rendering representations independently in distinct processors and smoothly switching between two drawable representations as explained below.

7.3.1 Multiprocessing

Parallelization is achieved by keeping two variables for the viewpoint and two chains of meta-objects that can be selected by a binary *'thread'* variable and spawning a separate process to compute the meta-objects to be rendered while using the parent process to actually hold the graphics pipeline. At each frame, we fill the chain selected by *'thread'*

```
Routine Render()
{
   While not end of simulation {
           Update viewpoint;
           Get viewing matrix;
           V(ds);
           DrawModel(thread);
           P(cs);
           thread = thread;
   }
}
```

Figure 7.2: Pseudo-code for the rendering process.

and render the chain selected by $\overline{thread}$. At the end, we synchronize the two processes and make $thread = \overline{thread}$ and start over. The pseudo-code for the rendering process is given in Figure 7.2 and that of the selection process in Figure 7.3.

Synchronization primitives like semaphores or barriers (Irix specific) were used to achieve the synchronism between these two processes in a consumer/producer [48] fashion (the selection process produces a rendering list which is then consumed by the rendering process).

In these algorithms two semaphores are used: 'ds' and 'cs', the draw and compute semaphores in a shared memory environment. If desired, prior to the execution of each algorithm we can specify the processor on which each process will run.

If a higher degree of parallelism is desired and extra processors are available in the system, then different branches of the model hierarchy can be traversed by different processors runing the 'AssignRepCstBenVis()' routine.

It also is important to mention that when updating the viewpoint, we are actually

106

```
Routine Select()
{
  P(ds);
  AssignRepCstBenVis(root);
  SelectRepresentations(root, thread);
  V(cs);
}
```

Figure 7.3: Pseudo-code for the representation selection process.

getting the viewpoint for the next frame and copying the viewpoint used by the selection process in the previous frame to draw the current frame. The viewing matrix, which needs to be read directly from the hardware, is obtained by the rendering process (which holds the graphics pipeline) and subsequently used (via shared memory) to compute visibility, benefit and initial representations by the selection process.

7.3.2 Representation Switching

The 'DrawModel' routine in Figure 7.2 is a simple loop that for all selected objects it issues draw commands and therefore prevents the graphics pipeline from becoming idle, thus achieving optimal performance. To minimize the effects of switching drawable representations from frame to frame ("popping") we use alpha blending to smoothly fade the new representation 'in' while fading the old representation 'out'.

By specifying an alpha value, the hardware blends the incoming source color (of the primitive being drawn) with the destination color (already in the frame buffer) to obtain a new pixel color $= \alpha * source_{rgb} + (1-\alpha)dest_{rgb}$, when the objects are drawn in a back-to-front

order, i.e., objects far from the viewpoint are drawn first.

This fading in/out of drawable representations is accomplished as follows: we start by checking if the meta-object is currently in the process of changing representations. In the negative case, if the current representation is different from the one that was last drawn we mark the object as changing representations, marked the 'in' and 'out' representations, set a value for 'α' and call a blend function to draw the 'in' representation with 'α' and the out representation with (1-α). Otherwise we render the current representation as usual. In the positive case, we check the number of times the in/out representations have been rendered against a threshold, and if it has not been attained we blend the representations. Otherwise, we mark the object as not changing representations and set the last representation to be the current one. The pseudo-code for fading two representations in and out is given in Figure 7.4.

7.3.3   Visibility Determination

Visibility checking is a critical point in our representation selection algorithm and we determine if bounding boxes enclosing objects in the model intersect the viewing frustum.

A simple way of checking if a point in space is inside the viewing frustum (VF) is to check if it is in the intersection of all half spaces of the six planes that determine the VF as follows:

1. Get the coordinates of the eight points that compose the VF and the point in eye-coordinates.

108

```
Routine FadeReps(node)
{
  if (node is not switching reps.) {
    if (last rend. rep ≠ current rep) {
      Mark node as switching reps;
      Mark the 'in' and 'out' representations.
      Zero rep. counter(node.repcount);
      BlendReps();
    }
    else {
      Set last rep. to current rep;
      Draw as usual;
    }
  }
  else {
    if (node.repcount > REPTHRESHOLD) {
      Set last rep. to current rep;
      Mark node as NOT switching reps;
    }
    BlendReps();
  }
}
```

Figure 7.4: Pseudo-code for blending two representations of an object.

2. Using the right-hand rule compute the outward facing normals for each of the VF's face. This involves a cross-product computation.

3. For each normal compute the dot product of the point in eye coordinates and the normal. If all the dot products are positive then the point is inside the VF.

Using this algorithm, to check if a box intersects the VF, in the worst case we will have to compute: 8 vertices in eye-coordinates, 6 dot products for each of the 8 vertices, i.e., 48 dot products. This would be too expensive to compute. Another way of doing the visibility test would be to compute the intersection between the bounding box of the object in eye-coordinates with the viewing frustum in the following way:

1. Get the object's bounding box in eye-coordinates.

2. Compute the intersection between the bounding box and a box formed by extending the slice of the viewing frustum corresponding to the farthest z-value of this box to its nearest z-value.

This would be a simple test if there was a fast way to compute the object's bounding box in eye-coordinates. Therefore, we do not actually compute the objects bounding box in eye-coordinates. Instead, we compute the bounding box of the bounding box of the object in eye-coordinates.

Although this visibility test is not precise it is much faster to compute than the previous one since it involves the computation of 8 vertices in eye-coordinates and a few range
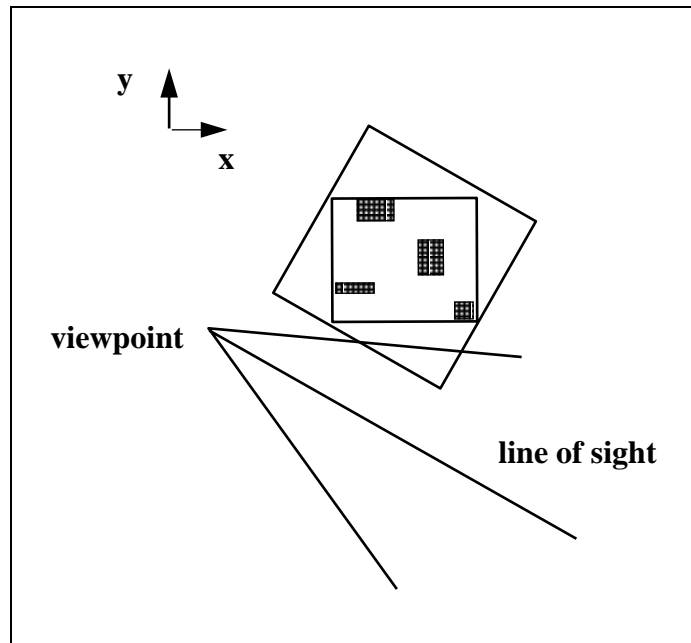
Figure 7.5: Checking the visibility of a set of objects against the viewing frustum. The test returns true although no object in the cluster is inside the viewing frustum.

comparisons. However, the visibility test can return true even though no object inside the cluster is also inside the viewing frustum as shown in Figure 7.5. This problem is even worst for large clusters. Therefore in the first pass of the model hierarchy traversal we determine the visibility of each object in the model, i.e., if it is inside the viewing frustum, and make a cluster node visible if and only if at least one of the objects that it subsumes is inside the viewing frustum.

7.3.4 C++ Implementation

The object-oriented design is implemented through C++ classes. The main abstraction, the meta object, is the parent class from which the conceptual object and cluster classes

inherit data and a virtual draw function.

This meta object contains a visibility flag and the data required to implement the temporal coherence and representation switching mechanism described in Sections 6.4.3 and 7.3.2, respectively. The 'drawable representations' component of the meta object class in Figure 6.1, is actually a handle [34], i.e. a pointer to an array of drawable representations class that contains pointers to instances of a 'Drawable' class. A 'current representation' field of the meta object indicates which representation in the array will be rendered at each frame.

In addition to the data inherited from the meta object class, conceptual objects have associated to them a rotation angle and a translation vector so that instances of the object can be replicated in the environment. The cluster class, besides inheriting the data from the parent class also contains an array of pointers to meta objects.

The array of drawable representation class which is pointed to by the meta object class contains, besides the array of pointers to drawables, information such as the number of view-dependent and view independent representations and an array that tells which view dependent representation is suitable for a particular viewpoint region.

The 'Drawable' class is the main abstraction from which all the other drawable representations depicted in Figure 5.2 are derived. It defines a virtual draw function that is redefined for each specific drawable class.

To draw one representation on the screen, a representation is selected and its position in the array of representations is stored in the current representation field of the meta object,

with a subsequent call to the virtual draw function.

7.4   Performance

Our test model had around 1.6 million polygons and during the tests the texture maps generated by the hierarchy building program were constrained to the available texture memory of one megatexels by selecting appropriate octree cell sizes (for an octree with four levels) and adjusting the resolution of the textured representations for objects and clusters.

For this model we were able to keep a real frame rate of around 16 frames per second for a target frame rate of 30 frames per second throughout the simulation without too much degradation in image quality. Figure 7.6 shows the image seen by the observer (top) and a top view of the same scene showing where clusters are being displayed (bottom).

The discrepancy between target and real frame rate is due to the fact that in a multi-processing system it is almost impossible to achieve a fixed frame rate since it is subjected to process and graphics context switching due to random interrupts by other processes. These switchings can be minimized by using the mechanisms described in Section 3.3 but since this would disturb other users in the system, none of the measures described were attempted (See Appendix A for other considerations about maintaining a fixed frame rate.).

The performance of the system for our test path is illustrated in Figure 7.7. When measuring the user time, the graph on the right shows that our estimation of cost and rendering algorithm are achieving the goal of keeping a uniform and high frame rate. When

113
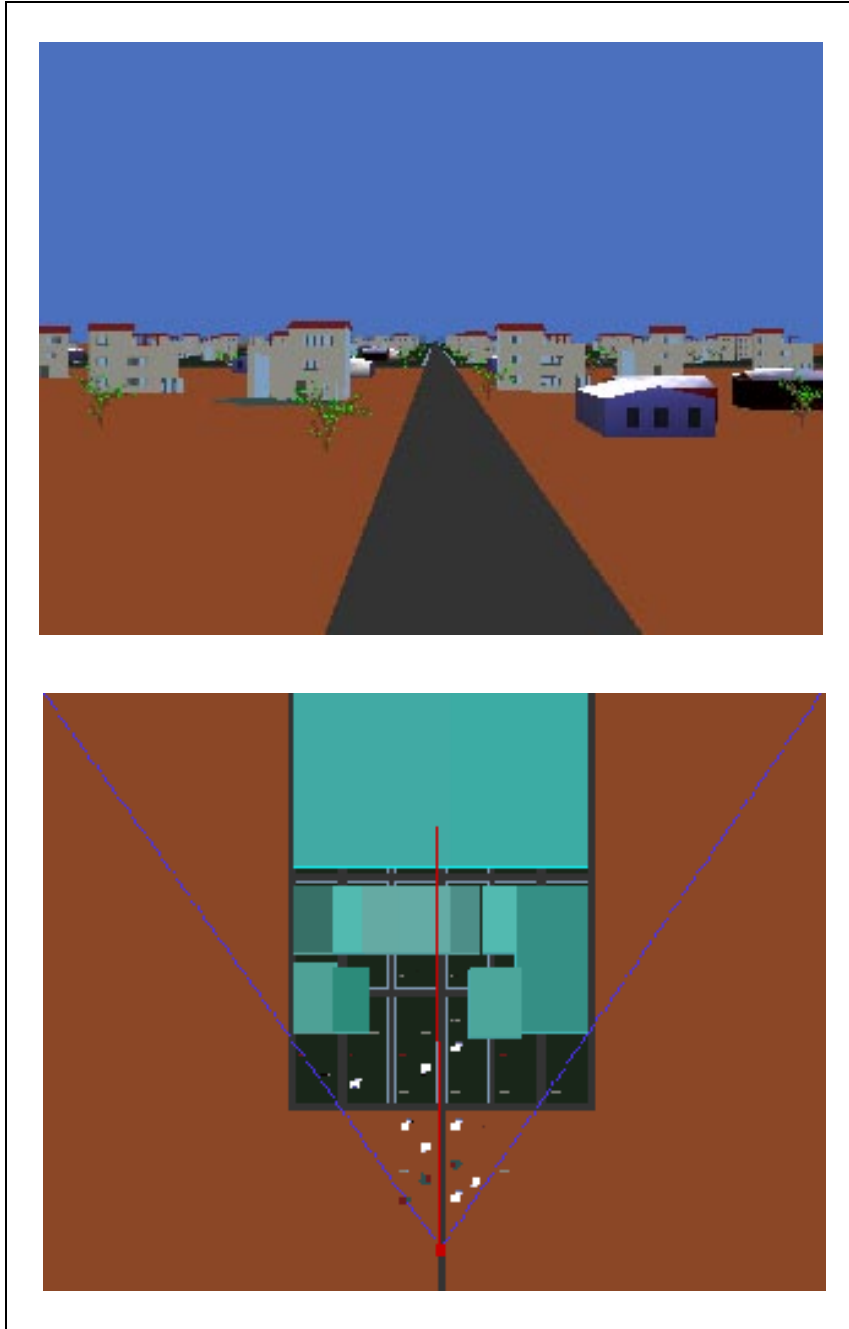
Figure 7.6: Image seen by the observer (top) and top view of the same scene (bottom) showing where clusters are being displayed.
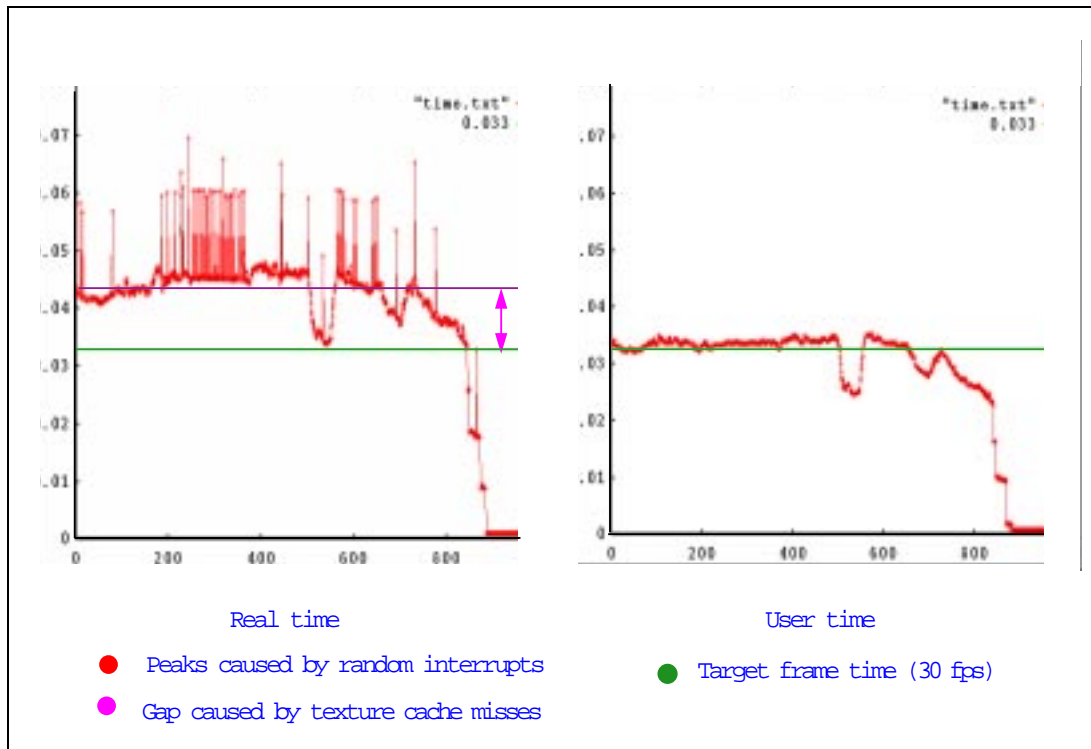
Figure 7.7: Plot of frame versus frame time. Real time (left) and user time (right) with and without smooth LOD switching using transparency blending, respectively.

the real time is measured in the left picture, spikes appeared due to either random interrupts or by the resolution of the clock used (100 milliseconds). We also note that the entire curve is shifted by an almost constant amount with respect to the target frame time of 1/30 of a second. This is happening because of the mechanism used to smoothly change between two representations, since two representations need to be rendered simultaneously. We minimize this effect by decreasing the estimated frame time by moving a slider on the control panel of our X-Motif interface and by adjusting the number of frames during which the transition of representations takes place (the threshold in Figure 7.4).

When our system ran without clusters, and using only the full detail LODs and view

frustum culling, we had a frame rate of around 1 frame per second for certain viewpoints in our test path. We have not tested the navigation of this environment with a toolkit like Performer [37].

7.5 Limitations

One phenomenon missing is the illumination of the environment. The illumination of a complex environment can be pre-precomputed using the radiosity method in a view-independent fashion and the shading attributes of objects and clusters would need to be incorporated to their representations. Instancing of objects would not be practical since two identical objects in different locations in the model would have different shading attributes. These attributes would make the two identical objects different, that is, they would have different representations that would need to be stored separately.

In order to guarantee that objects close to the view point or more generally speaking, with high benefit values, are displayed at their full resolution, the rendering complexity of single 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 would have to be represented by impostors even if the observer comes very close to them.

In our current implementation, each object/cluster has 5 texture maps, associated with each face of its bounding box. If the viewpoint is above the object, then the top texture map

will be displayed. This looks wrong except for viewpoints right above the object/cluster. This problem could be minimized if we had 4 texture maps that would be pasted onto diagonal slices of the bounding box of each object/cluster. Ideally, we would need at least one texture map for every sample point in the hemisphere in Figure 4.3. If the model does not use instancing of objects very often then too many texture maps may be need.

The basic limitation here, is the number of texture maps that can be used to represent objects and clusters. The more texture maps representing an object or cluster the better are the chances of getting the correct representation during the walkthrough depending on the viewpoint. In the case of objects, we can measure the accuracy of the texture mapped representation and render a view-independent version of the object (LOD) in cases where the accuracy/cost ratio for the texture map is relatively low compared to the accuracy/cost ratio of the LOD. For clusters however, although this image error will also be minimized for larger texture memory sizes, the best we can do for now is to try to estimate the image error of a cluster represented by a textured cluster is, i.e., how accurate the texture map represents the objects in the cluster.

Consider the two dimensional case of a cluster containing two objects. Here, the image error is considered to be the disparity between the projection on the view plane of an object in the cluster and its corresponding image on the textured cluster. In the case where the line of sight is pointing straight to the textured cluster, the situation that yields the minimum image error is when the two objects are aligned and the viewpoint is far away from the center of the cluster. The worst case, occurs when the viewpoint is close to the textured

117
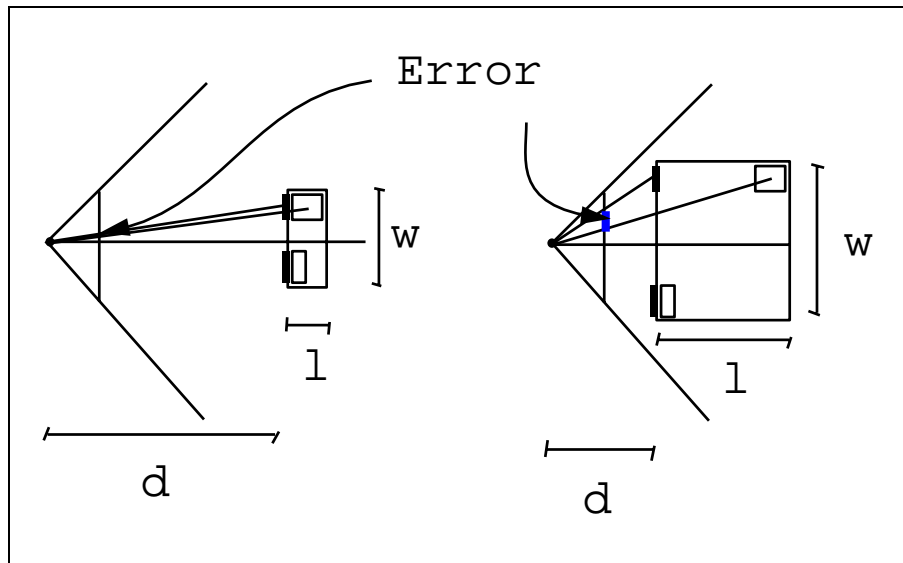
Figure 7.8: The best and worst case image error for a cluster with two objects. Left best case, error is approximately zero. Right, error increases as distance to cluster decreases and cluster sizeincreases.

cluster and the cluster bounding box is large. This is illustrated in Figure 7.8.

Using simple geometry we calculate this error to be: $\frac{nwl}{2d(d+l)}$, where n is the distance to the near clipping plane and the other variables are as in Figure 7.8.

The image error therefore increases as the dimensions of the cluster, l and w (and also height in 3D) increase and its distance to the viewpoint decreases. Therefore, to keep the image error to a minimum the clusters need to be displayed far from the viewpoint and need to be small. This requires the complexity of the objects near the viewpoint to be low and the model hierarchy octree to have several levels depending on the size of the entire model. While the former is not always possible since it depends on the model (unless we change the model), the latter increases the number of nodes of the octree and therefore the amount of texture memory used.

Theoretically we can have as many images in memory as our virtual memory size, but in practice, a texture map needs to be in a texture cache to be rendered. As the size of texture memory increases these problems tend to be minimized.

Another limitation of our current implementation is that our benefit calculation requires that a cluster know something about the benefits of its children, so all primitives are visited once per frame in the first phase of the model traversal, and our program is thus $O(n)$, where n is the number of objects.

A more efficient approach would combine the two phases of the model hierarchy traversal described in Section 6.4 into a single incremental approximation algorithm that would take advantage of the coherence of the per-object benefit that exists from frame to frame of a walkthrough. The basic idea is that if a given object is the most beneficial of its cluster in one frame, it is likely that it will continue to be the most beneficial one (or close to) in the next frame, and we can at each frame recompute only the benefit of the most beneficial object in each cluster.

The computation of benefits as in the first phase (Section 6.4.1) of the model traversal would be incorporated to the representation selection as in the second phase (Section 6.4.2) of the model traversal in the following way. Initially, the algorithm described in Section 6.4.1 would be applied only once to the entire model and each cluster would store a pointer to the most beneficial object that it subsumes. In the subsequent frames and for each frame, starting at the root node we recompute the benefits of only those most beneficial objects associated to the clusters directly descending from the root node and insert the nodes into

119

a list in decreasing order of benefit. At this time we would also compute the visibility of the node, an initial representation to render as before and its cost. We would then proceed selecting representations to render as in the second phase (Section 6.4.2) until we reach the user specified frame time. We remove the first node in the ordered list and repeat the process applied to the root node to this node and recompute the benefit of only those most beneficial objects associated to the clusters directly descending from the node. We then insert the nodes children in the list in decreasing order of the newly computed benefits. We also compute the maximum of the benefit of the node's children and if this is greater than the benefit associated to the object that the node points to we make the node's pointer point to the new most beneficial object in the cluster and propagate this change up until the benefit stored in a node's ancestor is already greater then this propagated benefit. This up propagation will obviously require that each child know who its parent is.

This new algorithm is still essentially a best-first search that examines the most promising paths of the model hierarchy first and spend the most time evaluating benefit, visibility, initial representation and cost for only those nodes in this path. The only difference is that instead of visiting all the nodes in the model hierarchy at each frame, we visit only the nodes that belong to clusters where the most beneficial objects belong. This seems very reasonable since if there is not enough rendering time to render the children of a node and therefore none of its children will be rendered, it is a waste of time to determine the benefit of each one of its children. As the user moves through the model and since at least one benefit is recomputed for each cluster, clusters that had low rendering priority in the past

may have their priority increased (and vice-versa) and will be explored more thoroughly. We would also apply the algorithm in Section 6.4.1 whenever a branch of the model hierarchy becomes newly visible.

As the algorithm in phase two, this algorithm will have its worst case $O(n)$ if the allowed frame time is long enough to render all the objects in the model at their lowest cost representation. Since we are assuming that in a large environment this will not be the case, in practice the algorithm should have an average constant complexity $O(1)$. The complexity of this algorithm is actually connected to the frame time allowed per frame and not to the number of objects in the model.

7.6   Summary

In this Chapter we presented the three major programs that compose our implementation.

The hierarchy building program creates an octree and view-dependent representations with the help of the graphics hardware. This tree and all the representations associated to its nodes can be visualized by a second program. This program is also used to measure accuracy of representations for several viewing directions and to measure the rendering time for each representation.

To measure accuracy of representations we used the image processing techniques in Section 4.3, although we realize that much more powerful techniques and methodologies are

needed.

The cost of each one of these representations is measured and stored in tables to be input by the visual navigation program. This, instead of the formula developed in Chapter 3, is used to avoid the approximations that this formula entails as well as the complexity introduced by arbitrary polygons.

We have shown that for a model containing around 1.5 million polygons we were able to achieve a uniform frame rate without too much degradation of image quality through out our test path.

Although the system performed well for the test path, there are limitations that need to be kept in mind. The most important ones are the fact that this system heavily uses texture mapping memory (a limited resource) and that the textured clusters may look wrong when viewed from certain viewpoints. This image error decreases when clusters are small relative to their distance to the viewpoint.

8

Conclusion

This Thesis presents a novel way of viewing objects and clusters of objects in the context of visual navigation systems. The framework developed is based on the idea that an object or cluster of objects can have many different representations, including view-dependent ones. If levels-of-detail representations are not provided then we can still achieve a reasonably good visual performance by using the automatically generated impostors described in Section 5. Even if impostors for clusters are not used, the octree algorithm together with the impostors for objects can be used to cull the model against the viewing frustum and to improve the interactivity of any visual navigation system, respectively.

The utilization of these impostors for clusters, however, is what makes our system useful for navigation of models that have more unoccluded primitives inside the viewing frustum then the hardware's ability to render them in real-time. Low cost impostors for clusters are rendered instead of the objects they represent when the frame time is running out. As far as we know this work presents a first attempt to solve the problem of navigating through

an environment with too many visible primitives.

However, as we pointed out in the limitations Section 7.5, this gain in interactivity is not introduced without a penalty in the quality of the image displayed, since the textured clusters can look wrong depending on the viewpoint. Far away small clusters introduce less error than larger ones closer to the viewpoint.

We believe that this system can be improved and these are our suggestions for future work.

- Enhance the flexibility of the system by adding new impostor types to the current set of automatically generated impostors. For instance, in situations where the object is far away, an object could be represented by a single point with the appropriate color.

- If we recall the formalization of the problem in Section 6.1, we see that the cost requirement for representations in an intermediate node is just that its most costly representation cost less than the sum of the lowest cost representation of its children. The representations for clusters do not actually need to be textured clusters. They can be anything, for instance, a triangular mesh containing all the objects in the cluster. Besides the fact that triangular meshes are much faster to render than conventional polygons, techniques such as those described in Section 2.2.1 could be used to simplify this mesh to satisfy our time requirement. In general, we suggest research on ways of obtaining simplified representations for groups of objects.

- Selecting a good representation to display at any given point in the simulation is

critical for the performance of this system. This selection is dependent on reliable ways of determining how an impostor is similar to the ideal full detail object. We believe that a more in-depth look into the vision literature can suggest better ways of comparing two images than what has been presented here.

- For commercial systems, investigate better benefit heuristics that take into account not only the characteristics of objects and representations as mentioned in Section 4.2 but also the surrounding environment as suggested in Section 4.4.

- Investigate better error metrics for images in which clusters are used so that we can guarantee a given image quality at each given frame of the simulation.

Other suggestions involve implementing the algorithm that combines the two phases of our tree traversal as described in Section 7.5 and devising new ways of placing textured clusters. Instead of simply pasting the texture maps onto faces of the object/cluster bounding box that they represent, define slabs at every $\Delta\theta$ degrees so as to obtain the convex hull of the cluster of objects, and paste the texture maps onto these slabs. This would be very helpful for view angles other then those perpendicular to the texture map. In some cases it will even reduce the image error associated to texture maps pasted onto a face of the object/cluster 's bounding box.

A

Frame Rate Estimation

Programs that attempt to maintain a user specified frame rate need to render each frame in a time less then the frame time ($\frac{1}{framerate}$) and in multiples of the screen refresh rate ($\frac{1}{60}$ for a 60Hz refresh rate) since a slight increase in the rendering time of each frame will result in a drastic decrease of the actual frame rate, specially for high frame rates (above 30 fps), since each back-to-front buffer swap will have to wait for a screen refresh to draw the front buffer on the screen.

For instance, if the screen refresh rate is 60Hz, the desired frame rate is 60 and each frame is taking $\frac{1}{60}$ + just a small fraction of $\frac{1}{60}$, the actual frame rate will be 30 fps. For the same small fraction excess and a desired frame rate of 30 fps the actual frame rate will be 20 fps, for 10 fps the resulting frame rate will be eight fps and so on.

In general the actual frame rate can be approximated by assuming that each frame is drawn with the same amount of time (ft). The number of retraces per frame (rpf) is: $rpf = \lceil ft \times srr \rceil$, where srr is the screen refresh rate in Hz and ft is measured in seconds. The approximate frame rate (fps) is then: $fps = \lfloor \frac{srr}{rpf} \rfloor$. With this formula an upper bound on the frame rate can be calculated.

In practice however, many factors influence the computation of the time to draw a single frame, including state of the system and whether the graphics pipeline is flushed before the

126

time measurement is taken. Therefore, to measure the frame rate a one second timer can be set up and each time a 'swapbuffer' is issued a frame counter can be incremented. When the timer goes off, the counter is display and then zeroed.

B

X-Motif User Interface

Both the program for visualization and measurement of the model hierarchy, and visual navigation have user interfaces written in X-Motif. This facilitates parameter changes prior to and during the simulation for evaluation purposes.

B.1 Model Hierarchy Visualization and Measurement User Interface.

This program's GUI provides buttons and fields that are used to select meta objects, representations and viewing directions so that we can measure their rendering cost and compare representations for clusters and objects against their actual geometry. A picture that shows this interface is shown in Figure B.1.

The up and down arrow keys in the control panel are used to select the desired meta-objects and representations. The names of the meta-object and the type of representation selected is shown on the panel while it is displayed on the graphics window.

The '# of redraws' field is used to input the desired number of times that we want to render a representation in order to get an average time.

The mouse is used to position the representation and the 'time' button to start measuring
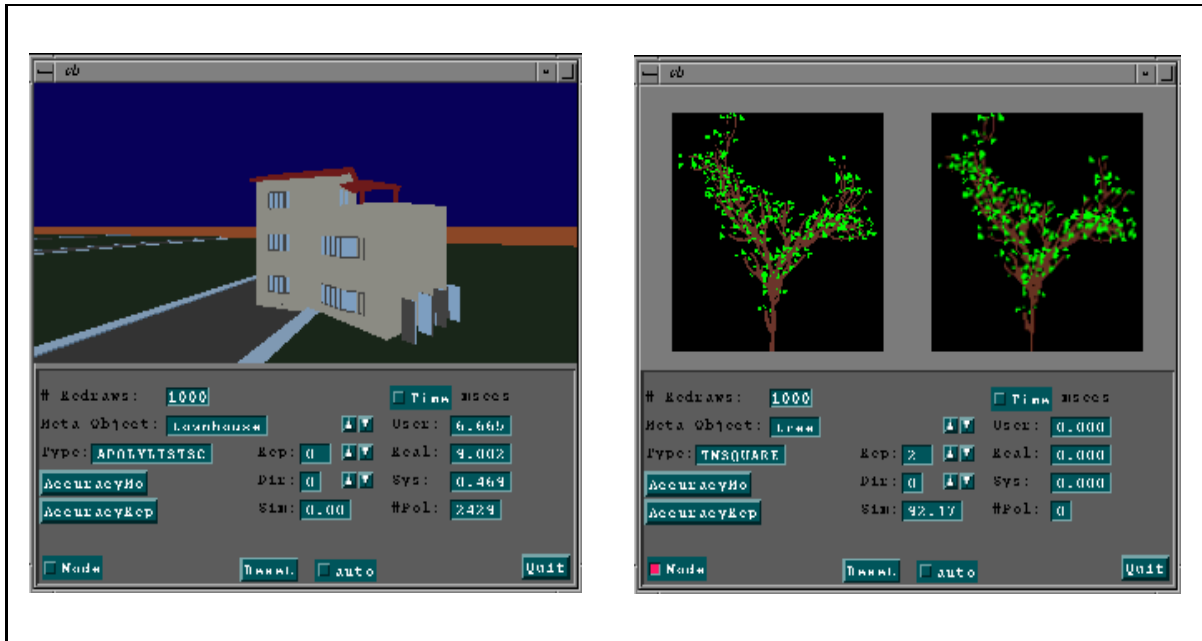
Figure B.1: X-Motif interface of the cost (left) and accuracy (right) measurement program.

time. The user, system and real times in milliseconds are given in the fields below the 'time'
button together with the number of polygons drawn.

The 'accuracy' button is used to put the program in 'accuracy' mode (The default mode
is 'time' measurement mode). In 'accuracy' mode, the graphics window is divided in two
to display the original and impostor images. The arrow keys are used to select the desired
meta-object representation and viewing direction.

The 'AccuracyRep' button is used to initiate the comparison. The similarity of the two
images appearing in the 'Sim' field indicates the estimated accuracy in percent.

The 'auto' button is used to automatically measure cost and accuracy for the entire
model hierarchy depending on the mode the program is in.
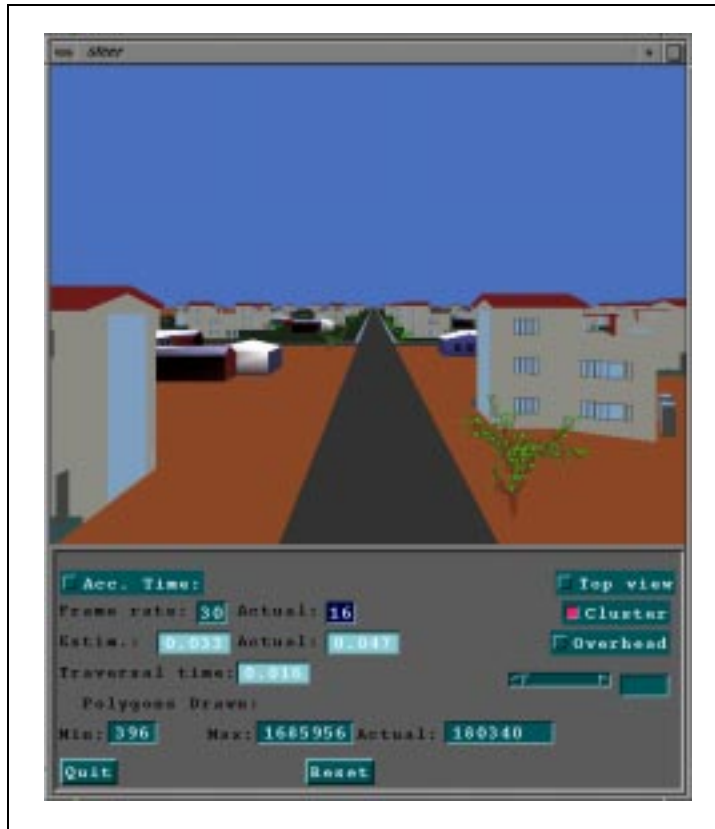
Figure B.2: X-Motif interface of the walkthrough program.

B.2 Visual Navigation User Interface

This program also like the previous one has an X-Motif interface with a GL graphics window and a control panel with buttons, fields and a slider to control parameters associated with the visual navigation as shown in Figure B.2.

By using the mouse the user navigates inside the environment in the following way: Whenever, the mouse pointer enters the graphics window it is positioned in its center. Moving the mouse pointer up and down, left and right causes the virtual user to move forward/backwards and turn left/right, respectively, with a speed determined by the distance

130

of the mouse pointer to the middle of the screen. By pressing the left/right mouse button the user goes up/down and the middle mouse stops the navigation by placing the mouse pointer in the middle of the graphics window.

The user can also determine the quality of the image being displayed and therefore the system response to mouse movements by either changing the frame rate or moving a slider. The initial frame rate can be changed by simply typing a two digit number (less then 60, the screen refresh rate) on the frame field. Moving the slider we can subtract a percentage of the current frame rate. This is useful to fine tune the application by accounting for its overhead which had not been accounted for when the cost of the representations were measured.

Three fields show the timings associated with the navigation. The first shows the estimate time that the list of meta objects that are sent to the graphics pipeline will take to get rendered, the second shows the amount of time it actually took and the third the time spent in the model hierarchy traversal. Three other fields display the maximum, minimum and actual number of polygons, i.e., the number of polygons if no impostor representations (or LOD) were used, the number of polygons if all the simplest impostors were used, and the actual number of polygons that were selected and drawn per second.

By pressing the 'Cluster' button we can activate/deactivate the use of the model hierarchy and therefore we can compare the performance and characteristics of using our approach as opposed to having a simple flat structure that culls the object against the viewing frustum. A top view of the model with the meta objects being culled against the

viewing frustum can be seen by pressing the 'Top view' button.

Some of these features can also be toggled by pressing the appropriate key when the cursor is inside the GL window. By pressing the 'R' key we can record a walkthrough to a file that can be played with the press of the 'P'lay key. This feature is useful to analyze the behavior of the program and for recording videos of the navigation.

By pressing the 'B'egin time and 'E'nd time keys we can record on a file ("time.txt") the real time the machine took to render each frame either in playback or while under user control. This file can be easily plotted with a tool like Gnuplot or Mathematica.

C

File Formats

The programs described in the implementation Chapter 7 access the files described below.

C.1 Model Hierarchy Building and Representation Generation

This program reads/writes the files whose names and formats are given in table C.1.

File "model.obj" contains the conceptual objects of the model, like house, tree, car and so on. Upon reading an object name it loads the appropriate file containing the geometry of the object.

File "model.dsc" contains the description of the environment. This tells the program where each conceptual object has to be place in world coordinates. Each object name is followed by a translation point and rotation angle.

The "mhierarchy" file contains the model hierarchy. Each line represents either a cluster or a conceptual object, represented by the letters 'O' or 'C', followed by its bounding box. A cluster's description is followed by the number of children it contains. An object is followed

File name	I/O	Format
model.obj	I	house1 tree8 car3 ⋮
model.dsc	I	house1 x y z α tree8 x y z α car3 x y z α ⋮
mhierarchy	O	C minx miny minz maxx maxy maxz nc ⋮ O minx miny minz maxx maxy maxz index x y z α ⋮
mhierarchy.box	O	x y z ⋮ r g b ⋮
mhierarchy.tmap	O	ntex nx ny . . . nx ny . . . ntex ⋮

Table C.1: Files accessed by the model hierarchy building program.

by an index into a table of objects (file "model.obj") followed by its position and orientation in space.

"mhierarchy.box", contains the description of AvColorBoxes, i.e., eight points and six colors, for the clusters in the model. Similar files contain the bounding boxes for each of the objects in the model ("house.box", "tree.box" and so on.).

"mhierarchy.tmap" is a binary file that contains the texture maps (TMRectangle) for all clusters in the model hierarchy. The number of texture per cluster 'ntex' and the dimension for each texture map 'nx' and 'ny' are specified. Similar files contain the texture maps for objects (house.tmap, tree.tmap and so on.).

C.2   Model Hierarchy Visualization and Measurement

This program loads in the output files described in table C.1 and computes the cost and accuracy of each object/cluster representation in the model. It also loads in the file "model.obj", with the appropriate representations added, as shown in Table C.2.

In this new format the name of the conceptual object is given along with the number of representations of a specific representation type. In this particular example, we have the object 'house1' being represented by a total of 10 representations, namely, three lists of triangles (usually corresponding to three levels-of-detail), one box with average colors, five texture maps, and one texture mapped box (since this is a box-like house). Among the representations for 'tree8' (a pine tree) we have included a rotating texture map, since it

135

Format
house1 3 TriList
house1 1 AvColorBox
house1 5 TMRectangle
house1 1 TMBox
tree8 1 TriList
tree8 1 AvColorBox
tree8 5 RotTMRectangle
⋮

Table C.2: New "model.obj" file containing objects and their multiple representations.

is a symmetrical object. Also note that only one representation of 'tree8' was provided by the database modeler but we still have other low cost representations that we can use.

When in accuracy mode, this program outputs a file in the format described in Table C.3. In this table, the files with extension ".acc" contain indices to the array of representations for each meta object, for each direction that the program compares impostors with actual objects (in this case five directions). For example, consider file "objects.acc". The first line of this file says that the best view-dependent representation for object 'house1' when the viewpoint is in viewing region three is representation 5. Files with extension ".cst" contain the measured rendering costs for each representation associated to a meta object.

C.3 Visual Navigation

This program inputs a control file containing simulation parameters such as, frame rate, forward and turning speed, viewing parameters (such as viewport dimensions and viewpoint).

File name	Format
Objects.acc	house1 4 7 5 6 9
	tree8 2 5 3 4 8
	$\vdots$
Cluster.acc	cluster1 1 4 2 3 5
	cluster2 1 4 2 3 5
	$\vdots$
Objects.cst	house1 c1 c2 ... c10
	tree8 c1 c2 ... c7
	$\vdots$
Cluster.cst	cluster1 c1 c2 ... c6
	cluster2 c1 c2 ... c6
	$\vdots$

Table C.3: Files output by the cost/accuracy measurement program.

It then reads "model.obj" (in Table C.2), the geometric models, the files that describe the model hierarchy and the representations associated to meta-objects (the output files in Table C.1), and the files that describe cost and accuracy of representations (in Table C.3).

Bibliography

[1] John M. Airey, John H. Rohlf, and Jr Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive Computer Graphics)*, pages 41–50, 1990.

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

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

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

[5] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, pages 542–554, October 1976.

[6] S. Bryson and S. K. Feiner. Research frontiers in virtual reality. *Computer Graphics*, pages 473–474, 1994. Siggraph Panel.

[7] Kenneth Chiu and Peter Shirley. Rendering, complexity and perception. *Eurographics' 94*, June 94.

[8] Jim Clark. A telecomputer. *Computer Graphics*, pages 19–23, 1992.

[9] Michael F. Cohen and Donald P. Greenberg. The hemi-cube a radiosity solution for complex environments. *Computer Graphics*, pages 31–40, July 1985.

[10] C. Cruz-Neira, D.J. Sandin, and T.A. Defanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Computer Graphics*, pages 135–142, 1993.

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

[12] Michael F. Deering and Scott R. Nelson. Leo: A system for cost effective 3D shaded graphics. *Computer Graphics*, pages 101,108, July 1993.

[13] Martin A. Fischler and Oscar Firschein, editors. *Readings in Computer Vision: Issues, problems, principles and paradigms*. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[14] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

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

[16] Thomas A. Funkhouser, Carlo H. Sequin, and Seth Teller. Management of large amounts of data in interactive building walkthroughs. *1990 Symposium on Interactive 3D Graphics, Computer Graphics*, pages 11–20, 1992.

[17] Branko J. Gerovac. Implications of merging digital television, communications and computing. *Computer Graphics*, pages 393–394, 1992. Siggraph Panel.

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

[19] Roy Hall, Mimi Bussan, Priamos Georgiades, and Donald P. Greenberg. A testbed for architectural modelling. *Eurographics' 91*, pages 47–57, 91.

[20] Chandlee B. Harrell and Farhad Fouladi. Graphics rendering architecture for a high performance desktop workstation. *Computer Graphics*, pages 93–100, 1993.

[21] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1994.

[22] Hughes Hopper, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetsle. Surface reconstruction from unorganized points. *Computer Graphics*, pages 71–78, 1992.

140

[23] Hughes Hopper, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetsle. Mesh optimization. *Computer Graphics*, pages 19–26, 1993.

[24] Silicon Graphics Inc. *React In Irix: A description of real-time capabilities of Irix v5.3 on Onyx/Challenge multiprocessor systems.*, 1994.

[25] Taligent Inc. *Taligent Guide to Designing Programs: Well-mannered object oriented design in C++*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[26] Wayne E. Carlson James R. Kent and Richard E. Parent. Shape transformation for polyhedral objects. *Computer Graphics*, pages 47–54, 1992.

[27] Ralph E. Johnson and Brian Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, August 1991.

[28] Stanley B. Lippman. *C++ Primer*. Addison-Wesley Publishing Company, Reading, MA, second edition, 1993.

[29] Paulo W. C. Maciel. Interactive rendering of complex 3D environments with pipelined graphics architectures. *Technical Report TR403*, May 1994.

[30] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. *To appear in the 1995 Symposium on Interactive 3D Graphics, Computer Graphics*, 1995.

[31] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Publishing Company, Reading, MA, 1993.

[32] A. Mitchel. Determinants of immersivity in virtual reality: Graphics vs. action. *Computer Graphics*, page 496, 1994. Siggraph Panel.

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

[34] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley Publishing Company, Reading, MA, 1993.

[35] Karol Myszkowski and Tosiyasy L. Kunii. Texture mapping as as alternative for meshing during walkthrough animation. *Eurographics' 94*, June 94.

[36] Michael Kass Ned Greene and Gavin Miller. Hierarchical z-buffer visibility. *Computer Graphics*, pages 231–238, 1993.

[37] John Rohlf and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Computer Graphics*, pages 381–394, July 1994.

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

[39] B. Schneider, P. Borrel, J. Menon, J. Mittelman, and J. Rossignac. Brush as a walkthrough system for architectural models. *Eurographics' 94*, June 94.

[40] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics*, pages 249–252, July 1992.

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

[42] Silicon Graphics, Inc. *Graphics Library Programming Tools and Techniques*, 1992.

[43] Silicon Graphics, Inc. *IRIS Power C User's Guide*, 1993.

[44] Silicon Graphics, Inc. *IRIX System Programming Guide*, 1993.

[45] Paul S. Strauss. A realistic model for computer animators. *Computer Graphics and Applications*, pages 56–64, November 1990.

[46] Bjarne Stroustrup. *C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, second edition, 1993.

[47] Bjarne Stroustrup and Margaret A. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, MA, 1991.

[48] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ, 1987.

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

[50] Allan Watt and Mark Watt. *Computer Graphics Animation and Rendering Techniques*. Addison-Wesley, first edition, 1992.

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