A Stratified Rendering Algorithm for Virtual Walkthroughs of Large Environments

by

Rebecca Wen Fei Xiong

B.S., University of California at Berkeley (1993)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 1996

Certified by . . . .
Seth J. Teller
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Chairman, Departmental Committee on Graduate Students

A Stratified Rendering Algorithm for Virtual Walkthroughs of Large Environments

by

Rebecca Wen Fei Xiong

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis investigates the effectiveness of using pre-computed scenes to accelerate on-line culling and rendering during virtual walkthroughs of large, irregular environments.

Maintaining a high frame rate is essential for a realistic virtual walkthrough experience. For a large, irregular outdoor environment where most objects are visible, visibility-based algorithms for reducing the number of polygons to be rendered are often not applicable. Approximation algorithms that simplify individual objects or clusters of objects often require cost-benefit heuristics to select the best representation. With new hardware support for texture mapping, a scheme that approximates parts of the scene as texture maps can be realized.

This thesis presents a new approach to the approximation problem. The novel aspects of the approach are that 1) it maintains, in physical memory, a constant amount of state for scenes of bounded density; 2) substantially reduces the number of polygons to be rendered; 3) requires no cost-benefit calculation; 4) exploits object spatial and temporal coherence; and 5) makes an absolute guarantee of image quality (for a fixed display resolution).

During a pre-processing stage, the space of possible viewpoints are divided into *viewcells.* Then for each viewcell, using *motion parallax,* polygons in the scene are partitioned into *near* and *far* sets. The far polygons are projected onto the sides of a bounding box and stored as texture maps.

During interactive simulation, as the synthetic viewpoint moves, its corresponding viewcell can be determined. Based on the viewcell, *near polygons* are retrieved, culled, and rendered as before. Instead of the original far polygons, only a constant number of *far texture maps* are retrieved and rendered, thus significantly reducing the rendering load in each frame.

This algorithm has been incorporated into a real-time virtual walkthrough system. Performance data collected shows substantial improvement using the Stratified Rendering Algorithm.

Thesis Supervisor: Seth J. Teller
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I would like to thank Professor Seth Teller for supervising this thesis work. Without his insight and guidance during the past two years, this thesis would not have been possible. He has founded a truly terrific graphics group here at MIT.

I am grateful for the opportunity to work with so many excellent colleagues and friends. Thanks to Professor Julie Dorsey for her helpful comments and advices, Eric for helping me work through so many hacks, Cristina and Mike for their bright smiles, Steve and George for being such understanding officemates, Kavita and Satyan for their suggestions. Thanks also to all the other great students in the group. A special thanks to Scott and William for maintaining the system for all of us.

I would like to thank the National Science Foundation for funding my graduate study. Professor Tomaso Poggio has also provided supplementary funding, for which I am grateful.

The support and encouragement of all my friends have helped make the last two years a wonderful experience. I would especially like to thank Kathy, the greatest roommate in the world, and Tony, the best friend I can ask for. Thanks also to Veena, Victor, Susan, Tony C., Janey, Elaine, Helen, Rosanna, Jill, Tom C., Brett, Joe, Nicolas, Mario, Frank, Don, Tom H., James, Ron, and all the other cool people in Ashdown (that includes you, Rama :) .

I would also like to thank my father for introducing me to the exciting world of computers, and my brother Wen for always having an answer for everything even when he is wrong.

Finally, I would like to thank my mother, whose courage has been and will always be an inspiration to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes the Stratified Rendering Algorithm – using pre-computed scenes to accelerate on-line culling and rendering during virtual walkthroughs of irregular environments. It then presents a walkthrough system that incorporates this new algorithm. This chapter first examines existing work on walkthrough systems, then introduces the Stratified Rendering Algorithm.

1.1 Virtual walkthrough systems

Real-time visual simulation of complex environments such as cities can be useful for city planning, embedding of commercial databases, military training, and entertainment. Polygon-based representations of the environment allow for collision detection, re-illumination, and object manipulation. Often, virtual navigation of these complex environments requires a trade-off between realism and speed: either a slow rendering of a realistic scene with more polygons or a quick rendering of a simplistic scene with fewer polygons. A responsive virtual navigation system requires a constant frame rate, so polygons in each frame must be rendered within a pre-determined time period. Yet representing even a medium-sized city can take many polygons. For example, it would take tens of billions of 3D polygons and tens of Gigabytes of texture data to represent Cambridge, Massachusetts (roughly $100km^2$) using an efficient organization such as an adaptive spatial data structure [13]. Rendering such a complex scene at a constant frame rate poses a serious challenge for even the most advanced hardware currently available.

Various solutions have been proposed for satisfying the frame rate criterion without sacrificing scene quality, through applying knowledge about object structure and placement. One common approach uses visibility determination. By pre-processing the model, one can often determine a tightly bound superset of visible polygons that need to be processed during each frame of the walkthrough [14], thus eliminating the rest. For example, if the environment is a building in which floors and walls make natural visibility boundaries, categorizing objects according to these boundaries can significantly reduce the number of polygons that need to be considered in each frame of the walkthrough, resulting in three orders of magnitude speedup during rendering [4]. Using this algorithm, a detailed multi-floor building model can be virtually navigated at interactive rates [15]. But a large, irregular outdoor environment like a city has no obvious visibility boundaries for general viewpoints and thus presents a different problem.

1.2 Approximation algorithms

Another common approach to reducing rendering load of walkthrough systems works well when there is little occlusion, so that many objects in the scene are visible to the viewer. It uses approximations, or levels-of-detail, for objects. The UCLA School of Architecture has built and visualized a model of South Central Los Angeles using this technique. The model consists of collections of texture-mapped boxes, without detailed geometric information [7].

When an object is far from the viewer, its projection consists of a small number of pixels on the resulting image. In this case, one need not render all the detailed polygons comprising the object. Instead, one should render the larger polygon shapes that approximate the object. For example, vertices in the original CAD models can be clustered to produce a simplified version of the model [11].

There has been a great deal of research on getting the most cost-effective levels-of-detail for different objects. One walkthrough system developed at UC Berkeley uses cost-benefit heuristics to select the polygons to be rendered in each frame [3]. The Iris Performer system from Silicon Graphics stores the object model hierarchically. The user can provide several representations of an object with varying levels of detail. Performer then automatically selects a representation based on the location and the speed of the simulated viewpoint [9].

A recent algorithm uses texture maps pasted onto bounding boxes of objects. This can also apply to a group of objects, in which case the texture maps are called textured clusters. This algorithm creates a model hierarchy bottom up by repeatedly clustering together objects and generating the texture map for the new cluster, until all the objects are enclosed by one cluster. Then, each node (object or cluster) is assigned a view-dependent benefit value. Traversing the hierarchy top-down, the representation with the highest accuracy/cost ratio is chosen first, until some representation for each object is chosen and the total rendering time reaches the allowable time for a frame [8].

A major drawback of such object approximation algorithms is that when there are many differently-shaped objects in the environment, automatically calculating and storing accurate but low-cost levels-of-detail for each of the objects can be difficult. Once they have been created, selecting the right one for each object in a frame is still difficult. Ideally, a set of levels-of-detail with the highest overall realism should be rendered in each frame. However, while rendering an object clearly increases the observed realism, the incremental benefit of rendering a level-of-detail for the object is difficult to quantify. At present, the benefit value can only be modeled through comparison with the ideal image, or using size of object on screen, distance from center of screen, or direction of viewpoint, etc. [3, 8]. Once the cost and benefit estimates for each level-of-detail have been calculated, selecting the optimal set translates into an NP-complete knapsack problem, which can be solved only approximately in reasonable time. In addition, current object approximation schemes cannot make any guarantees of the visual quality of the resulting images.

To solve these problems, one recent algorithm uses a hierarchical spatial data structure to store the model geometry, then constructs a simplified representation for contents of each cell in the hierarchy. During interactive walkthrough, a scene is then rendered by traversing the hierarchy: a cell's approximation is drawn if it is sufficiently accurate [2]. A variation on the scheme does not pre-compute the simplified representations, but generates them during the walkthrough, and caches them at each cell for reuse in subsequent frames [12]. While this algorithm avoids many problems common to object approximation, a large number of simplified images still need to be retrieved and rendered during the tree traversal. In addition, because objects need to be split between two or more leaf nodes during division, this algorithm can cause visual artifacts that look like gaps in the split polygons. These artifacts can be eliminated by inflating the node's bounding box to include some overlaps, which can cause slight expansion of distant objects in the rendered image.

1.3 Overview of approach

For the rest of this thesis, an input model representing a city is assumed. Because most cities consist of uniform city blocks with buildings of a limited number of floors, the object density is fairly constant throughout. Therefore, a bounded object density is assumed for the input. It is also assumed that the graphics system used for walkthrough is a multi-processor workstation that supports hardware texture mapping. Multiprocessing allows geometric objects management to be done concurrently with rendering for a higher rendering rate. Expanded texture mapping support in current hardware enables new approaches to interactive graphics.

Instead of using approximations for objects, clusters of objects, or cells of objects, approximations for large portions of the model can be used. The new approach presented in this thesis derives from a simple observation: as the viewpoint changes, *remote objects appear to change relative position more slowly than nearby objects.* In other words, for viewpoints that are close to each other, the rendered appearance of remote objects changes slowly. This temporal coherence can be exploited by pre-computing this appearance, and re-using it for many frames during subsequent walkthroughs.

It is important to note that for a given scene, a human being tends to pay more attention to nearby objects than remote objects. So for user perception, the appearance of near objects is more important than that of remote objects. The new approach exploits this fact by allocating more hardware resources to accurately rendering nearby objects.

1.3.1 Stratified Rendering Algorithm

The Stratified Rendering Algorithm maintains a boundary, or **bounding box**, around the instantaneous synthetic viewpoint, as shown in Figure 1-1. Objects within this bounding box are considered **near** objects. They are rendered in each frame of the walkthrough by a 3D to 2D projection and Z-buffer scan conversion of all the polygons in each object using standard graphics hardware. Objects outside the bounding box are considered **far** objects. These are first projected onto the sides of the bounding box as texture maps in a batch pre-process. Then during the walkthrough, the texture maps are re-projected onto the sides of the bounding box. So in each frame of the walkthrough, only a small, constant number of texture maps need to be projected and rendered, rather than all the far objects. This significantly reduces the rendering load.

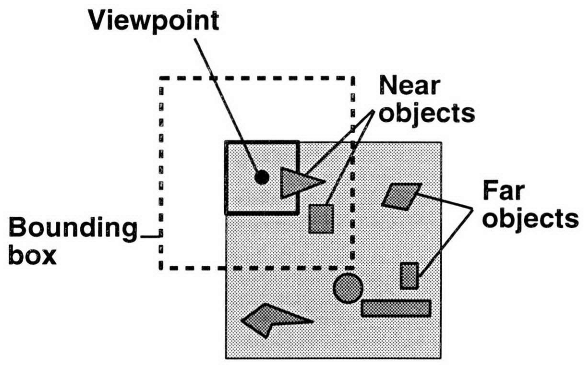


Figure 1-1: A top view of a synthetic viewpoint and its bounding box, which partitions all objects in the environment into near and far objects.

In addition, input models with bounded object density have a bounded number of near objects in each scene. Therefore, a constant amount of state (geometry and texture maps) can be maintained in physical memory during the walkthrough. Because all the far objects are cached as texture maps, cost-benefit calculations for selecting individual objects can be eliminated.

The new approach allows the user to set an absolute bound on angular error between the projection of an actual object and the projection of texture maps containing that object. Thus, the accuracy of rendering during walkthroughs can be guaranteed. Using the Stratified Rendering Algorithm, both speed and realism can be achieved.

1.4 Organization

The thesis is organized as follows. Chapter 2 defines the basic concepts of original viewpoint, bounding box, and viewcell, and derives the relationship among them using motion parallax. Chapter 3 describes the Stratified Rendering Algorithm based on a hierarchical data structure. Chapter 4 describes an extension to the algorithm that uses multiple levels of bounding boxes. Chapter 5 presents an implementation of Stratified Rendering, the CityScape walkthrough system. It supports the usual navigation functions, as well as the selection of different rendering algorithms (for the purpose of collecting performance data). Chapter 6 compares and discusses the performance characteristics of Stratified Rendering versus brute force rendering. Chapter 7 explores possible future extensions to the algorithm and implementation. Finally, Chapter 8 concludes.

Chapter 2

Basic Concepts

This chapter first defines the basic concepts of original viewpoint, viewcell, and bounding box, which form the basis of the Stratified Rendering Algorithm. The angular error bound is also defined. Motion parallax analysis is then used to derive the relationships among them. By setting the angular error bound, the visual quality of images rendered by the new algorithm can be guaranteed.

2.1 Definitions

The Stratified Rendering Algorithm is based on using pre-computed scenes to reduce polygon culling and rendering load during interactive walkthroughs. All possible simulated viewpoints for the walkthrough must be taken into consideration. The space of the possible simulated viewpoints is partitioned into cubic cells, which are called **viewcells**. The center of each viewcell is defined to be the **original viewpoint**. The approximating scenes are projected from these original viewpoints. Each viewcell is enclosed by a **bounding box** that divides the near polygons from the far polygons for all viewpoints in the given viewcell,

The relationship between the viewcell and the bounding box is defined as follows. Given an original viewpoint and a maximum allowable angular error $\boldsymbol{\alpha_{max}}$, the viewcell bounds differing viewpoints such that the object boundary defined by the bounding box remains valid. This means that a projection of far polygons outside the bounding box from the original viewpoint and from any viewpoint within the viewcell should be within α_{max}. The viewcell should be similar to the bounding box, with all six sides parallel to the original bounding box, as shown in Figure 2-1.

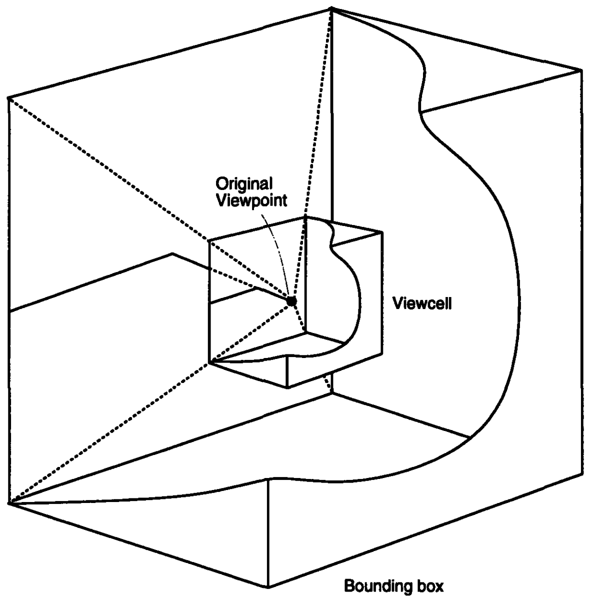


Figure 2-1: Original viewpoint, viewcell and bounding box.

2.2 Analysis of motion parallax

During the walkthrough, instead of rendering all the far objects from the new viewpoint, our algorithm re-uses images projected from the original viewpoint. To do so, one must first determine how much the new viewpoint can be displaced from the original viewpoint before the pre-rendered images become invalid. In other words, one must determine the allowable displacement such that projections of far objects from these two viewpoints are within a given angular error.

To solve this problem, a simple analysis of motion parallax is presented. In Figure 2-2, points P and Q are on the same line when viewed from point V, but are apart when viewed from a new viewpoint, V'.

Let t' be the distance between P and Q ($\overline{PQ}$), t be the distance between Q and V ($\overline{QV}$), and s be the distance between V and V' perpendicular to t ($\overline{VV'}$). Let α be

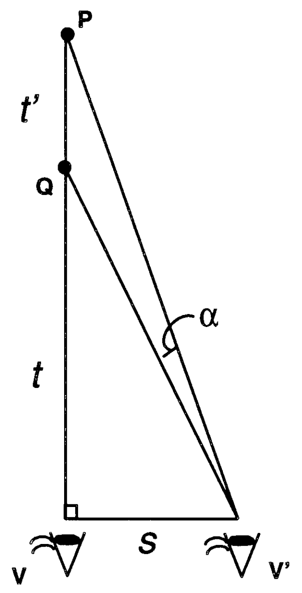


Figure 2-2: The observed angular difference between the two points that are on the same line when viewed from point V depends on distance from point V.

the angular difference in degrees between P and Q when viewed from V' ($\angle PV'Q$). Simple geometry can be used to derive an equation for α in degrees given t, t', and s.

$$\begin{aligned}
\angle V'PV + \angle PVV' + \angle VV'P &= 180^\circ \\
\angle V'PV + 90^\circ + (\angle PV'Q + \angle QV'V) &= 180^\circ \\
\arctan(\frac{s}{t+t'}) + 90^\circ + \alpha + \arctan(\frac{t}{s}) &= 180^\circ \\
\arctan(\frac{s}{t+t'}) + \arctan(\frac{t}{s}) &= 90^\circ - \alpha \qquad (2.1)
\end{aligned}$$

Figure 2-2 shows that when viewed from V, Q represents P exactly. Let Q be an approximation of P when viewed from V'. Q approximates P well when V' is close to V. As V' moves away from V, Q approximates P less while α, the error between P and Q, becomes larger. One can observe several relationships from Equation 2.1: s increases with α and t, and decreases with t'.

2.3 Angular error bound

For a given viewcell, images are computed as follows: from the original viewpoint, all the far objects are projected onto the sides of the bounding box for that viewcell. These images are then used as an approximation for the actual objects during walkthrough. The perceived angular difference between the approximation and the objects must be within α_{max} as defined above. Figure 2-3 shows how motion parallax analysis can be used to derive the equation for viewcell size.

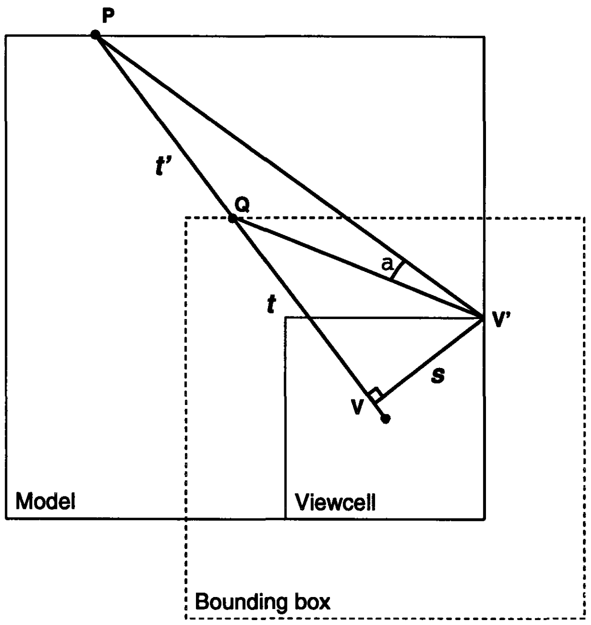


Figure 2-3: Motion parallax applied to viewcells and bounding boxes.

Let P be a point on an object, Q be its approximation on the bounding box, α be the angular error between the two points when viewed from V', a viewpoint other than the original viewpoint. Let V be the perpendicular projection of V' on the line $\overline{PQ}$. The same relationship derived in the previous section holds between the corresponding s, t, t' and α.

By inspection, the largest α comes from using a viewcell at the corner of the model, and placing P at the farthest point perpendicular to the original viewpoint as shown in Figure 2-4. This value must not be greater than α_{max} as defined above.

Let l be the length of the model, b be the length of the bounding box, and v be

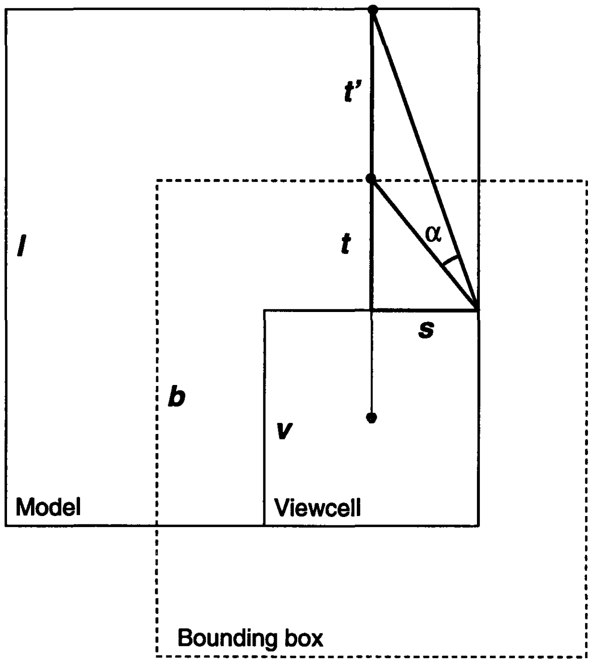


Figure 2-4: The largest α in a given viewcell, bounding box, and input model configuration.

the length of the viewcell. In 3D the four corners of the viewcell facing P give the biggest s. Computing t and t' is straight forward.

$$\begin{aligned} s &= \frac{v}{\sqrt{2}} \\ t &= \frac{b}{2} - \frac{v}{2} \\ t' &= l - v - t \end{aligned}$$

Equation 2.1 can then be applied to derive the correct relationship between α_{max}, and the lengths of viewcell, bounding box, and the model: given these lengths, the maximum angular error α obtained by any viewpoint in the model is less than α_{max}.

$$\begin{aligned} \arctan(\frac{s}{t+t'}) + \arctan(\frac{t}{s}) &= 90^\circ - \alpha \\ \arctan(\frac{\frac{v}{\sqrt{2}}}{l-v}) + \arctan(\frac{\frac{b}{2} - \frac{v}{2}}{\frac{v}{\sqrt{2}}}) &= 90^\circ - \alpha \\ \arctan(\frac{v}{(l-v)\sqrt{2}}) + \arctan(\frac{b-v}{v\sqrt{2}}) &= 90^\circ - \alpha_{max} \qquad (2.2) \end{aligned}$$

The input model size is given while the bounding box size is determined by the available hardware, as will be explained later. From Equation 2.2, it is easy to see that given the sizes of the bounding box and of the model, the viewcell size can be adjusted to fit any reasonable α_{max}. There is only an upper bound on α_{max}, but no lower bound. So the angular error bound can be made as small as needed for accurate rendering.

2.4 Practical results

Equation 2.2 can now be applied to real values to see how viewcells and bounding boxes can be calculated in practice. The correct viewcell size can be computed based on any reasonable values for α_{max}, bounding box length, and model length. As an example, an input model of the size of a college campus and a commercial graphics system are used to derive the viewcell size. A city-sized model is then used to test the analysis when applied to larger scales.

2.4.1 Viewcell size as a function

Figures 2-5, 2-6, and 2-7 show how the viewcell length varies with respect to α_{max}, bounding box length, and model length. Note that multiplying the l, b, and v by a constant does not change the equation.

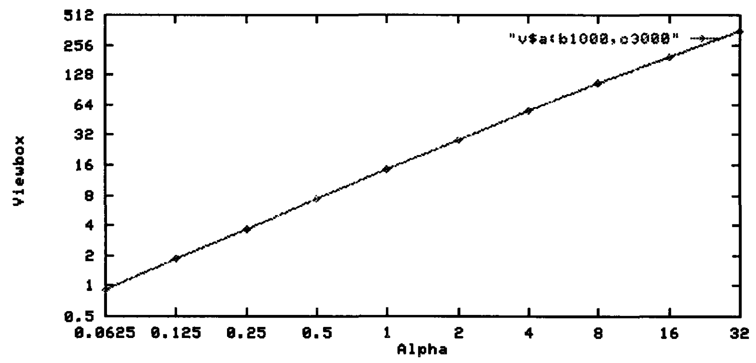


Figure 2-5: A logscale plot of viewcell length as a function of α_{max} in degrees. The bounding box length is 1000 units, and the model length is 3000 units.

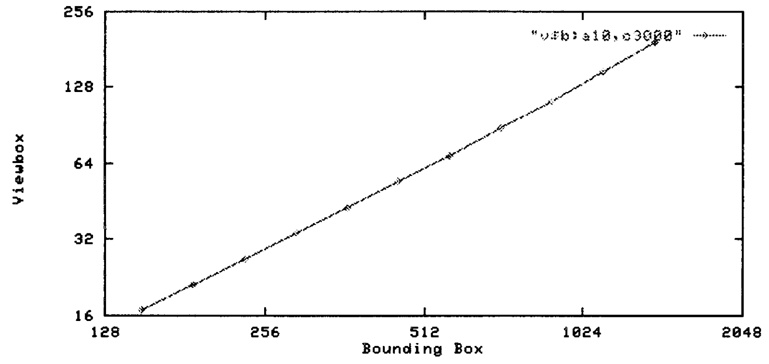


Figure 2-6: A logscale plot of viewcell length as a function of bounding box length. α_{max} is 10°, and the model length is 3000 units.

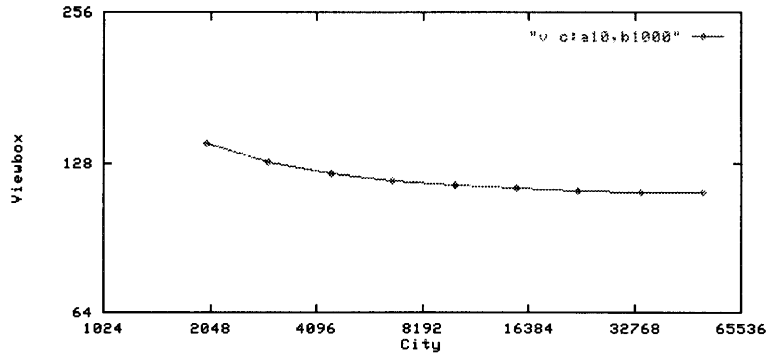


Figure 2-7: A logscale plot of viewcell length as a function of model length. α_{max} is 10°, and the bounding box length is 1000 units.

Triangles/sec	*Pixel Fill, textured*
1.8M	55M to 230M

Table 2.1: Graphics specification for SGI RealityEngine 2

From these plots, one can see that the viewcell length increases almost linearly with respect to α_{max} and bounding box size, and decreases asymptotically with respect to the size of the model.

2.4.2 Bounding box size

Using the new algorithm, in each frame only the near polygons and a constant number of texture maps need to be rendered. The bounding box size should be selected such that on average both the near objects and the texture maps can be rendered in a given frame. The size can be calculated using the rendering speed of the graphics system. Thus, even a complicated scene can be rendered in real-time.

Assuming that the graphics system used for the walkthrough is fully pipelined and capable of rendering on average T textured pixels and P polygons per frame. Let t be the number of pixels on the screen that can potentially be textured, and p the maximum number of near polygons that can be rendered in a frame. The following equation holds:

$$\max(\frac{t}{T}, \frac{p}{P}) = 1$$

This means that either filling textured pixels or rendering near polygons can potentially be the performance bottleneck. However, if the number of textured pixel in a frame is less than T, the hardware filling capacity, then the equation simplifies to the following:

$$\begin{aligned} \frac{p}{P} &= 1 \\ p &= P \end{aligned} \tag{2.3}$$

As an example, data for a Silicon Graphics RealityEngine 2 system shown in Table 2.1 [5] are used to calculate an upper bound on the number of polygons that can be rendered per frame. The analysis below assumes a 60Hz frame rate and a 512 pixel by 512 pixel viewport with on average three quarters of the pixels textured. The horizontal and vertical viewing angles are both 90°. Also, the scene is assumed to contain enough details such that on average there is 1 polygon/m^3.

The total number of textured pixels per frame is $0.75 * 512 * 512 = 197,000$. This is less than the maximum pixel filling capacity of the graphics system, at $55M/60 = 920,000$ pixels per frame. Thus Equation 2.3 can be used. $p = P = 1.8M/60 = 30,0000$ polygons per frame. So the system is capable of rendering 30,000 polygons.

With horizontal and vertical viewing angles at 90°, the view frustum encloses on average $\frac{1}{6}$ of the near polygons. The polygons that lie outside the viewing frustum can be culled away. Thus, only $\frac{1}{6}$ of the polygons in the bounding box need to be rendered on average. Using the value computed for p, the length of the bounding box (b) can be solved for as follows:

$$\begin{aligned}
p &= \text{number of near polygons}/6 \\
p &= \text{volume of bounding box } * \text{polygon density}/6 \\
\text{volume of bounding box} &= \frac{p * 6}{\text{polygon density}} \\
b^3 &= \frac{30,000 * 6}{1} \\
b^3 &= 180,000m^3 \\
b &= 56m
\end{aligned}$$

The length of each side of the bounding box should then be about $56m$.

2.4.3 Sample campus model

The viewcell analysis can now be applied to a sample model. Let the scene model be $1km$ by $1km$ by $10m = 10^7m^3 = 0.01km^3$, about the size of a college campus. There are $10^7 * 1 =$ 10M polygons in the entire scene – many more than what even the fastest graphics system can render in each frame of a 60Hz interactive simulation.

The calculation above shows that a bounding box size of $56m$ should be used. Figure 2-8 shows the trade-off between α_{max} and viewcell size.

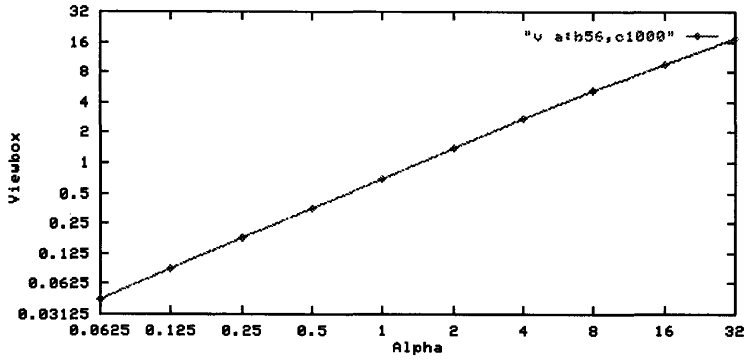


Figure 2-8: A logscale plot of viewcell length as a function of α_{max} in degrees. The bounding box length is $56m$, and the model length is $1000m$.

Choosing an α_{max} of $10°$ gives a viewcell length of $6.35m$. This is a reasonable value for α_{max}, because it is the worse case error bound. For most viewpoints, such as in the middle of the model, or viewpoints close to the original viewpoint, the error is much smaller. The number of viewcells in the scene, n, can now be approximated as follows:

$$\begin{aligned} n &= \frac{\text{model volume}}{\text{viewcell volume}} \\ &= \frac{10^7}{6.35 * 6.35 * 6.35} \\ &= 50,000 \end{aligned}$$

2.4.4 Sample city model

A city like Cambridge, MA with area $100km^2$, can be approximated using a model of $10km$ by $10km$ by $100m$ ($10km^3$). Applying the same analysis as in the previous section, there are $10 * 1000^3 * 1 = 10$B polygons in the entire scene.

Again, $56m$ calculated above is used for the bounding box length. Figure 2-9 shows the trade-off between α_{max} and the viewcell size. Note that the plot is similar to Figure 2-8 because the viewcell size depends asymptotically on the model size.

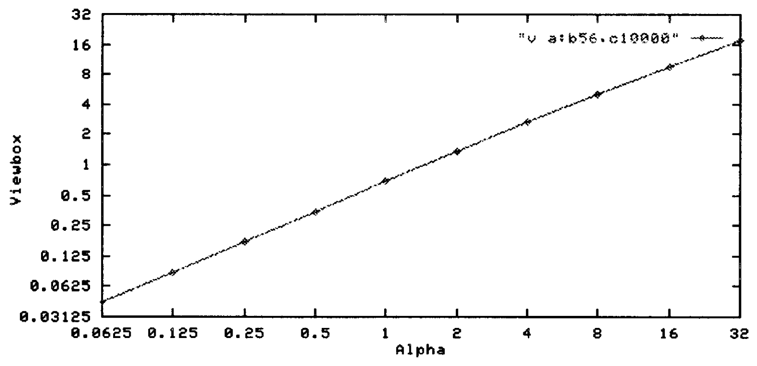


Figure 2-9: A logscale plot of viewcell length as a function of α_{max} in degrees. The bounding box length is $56m$, and the model length is $10000m$.

Setting α_{max} to 10° gives $6.22m$ for the length of the viewcell. It is only slightly less than the result for the sample campus model. There are approximately $4.1 * 10^7$ viewcells.

The analysis above shows that given a reasonable α_{max}, an input model, and the necessary graphics system, the correct bounding box and viewcell sizes can be determined to ensure rendering speed and visual quality.

Chapter 3

Stratified Rendering Algorithm

This chapter presents the Stratified Rendering Algorithm. The input model to the algorithm is a scene description, usually a collection of geometric objects. The algorithm consists of two stages: the pre-processing and the actual walkthrough. For a given viewcell and bounding box configuration, *a given model must be pre-processed exactly once.* The results generated can then be used in all subsequent walkthroughs.

3.1 Pre-processing stage

In this stage, the input model that contains all the objects and all possible simulated viewpoints is partitioned into viewcells. For each viewcell, the bounding box centered at its original viewpoint is determined. As shown in a top view in Figure 3-1, the bounding box classifies objects in the scene into near and far.

The pre-processing stage consists of two steps: space partitioning and texture map generation. When pre-processing is completed, the near objects and far texture maps are associated with each viewcell. The texture map storage and the running time and memory usage of both stages will be analyzed below.

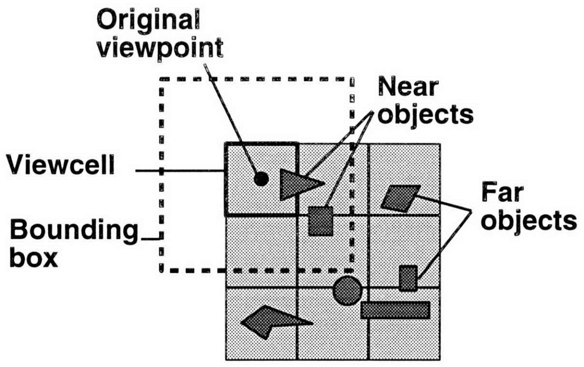


Figure 3-1: A top view of an original viewpoint, its viewcell, and its bounding box, which partitions all objects in the environment into near and far objects.

3.1.1 Space partitioning

A k-d tree structure with $k = 3$ [1] is used to spatially partition the model into viewcells. The k-d tree is a simple hierarchical tree structure that allows flexible spatial divisions at each step of the iteration. It stores all the near objects and far texture maps for each viewcell and allows efficient retrieval of the viewcell corresponding to any viewpoint. For simplicity, k-d tree cells are referred to as kd-cells.

First, all the objects in the model are assigned to the root kd-cell, which contains the entire input model and therefore all the viewcells. Then at each iteration, a kd-cell is divided in half to create two new kd-cells. Each new kd-cell will contain half the viewcells and must store all the necessary near objects for these viewcells. The overall bound for the near objects is called the **object boundary**, as shown in Figure 3-2. This subdivision continues until each leaf kd-cell is no larger than the specified size for a viewcell, and becomes a new viewcell. The object boundary for the leaf kd-cell will then just be the bounding box. So after the subdivision, all the leaf-level k-d tree kd-cells are viewcells, each of which stores the corresponding near objects. The algorithms for subdividing a k-d tree kd-cell and determining the the object associated with the kd-cell are discussed in more detail below.

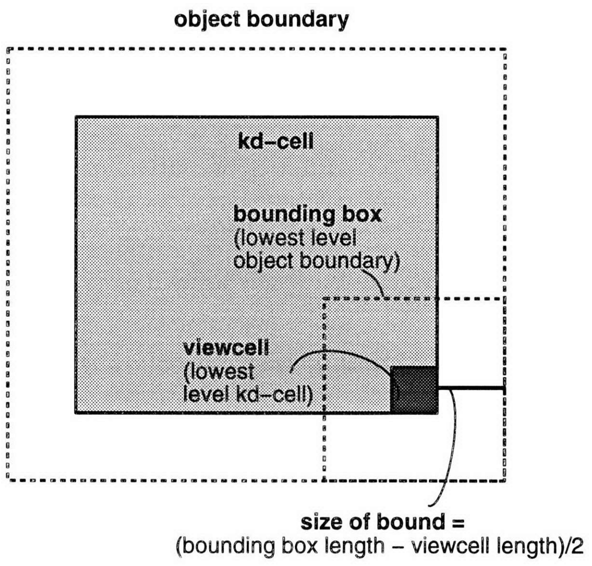


Figure 3-2: When a kd-cell is subdivided, all the objects within object boundary will be associated with the new kd-cell.

Subdivision

The subdivision heuristics are used to decide when to split a kd-cell, and in which dimension and at which point it should be split. When a given kd-cell is larger than the viewcell size, it is divided. To keep the *k*-d tree balanced, subdivision occurs near the middle, at a point that is an integral multiple of the viewcell length.

```
DoSubdivide (cell, whichDim, whichPoint)
   for dimension = x, y, z      ;; go through all three dimensions
      numBoxes = Floor(GetLength(cell,dimension)/viewcellLength)
      ;; compute the number of viewcell lengths in a side of the cell
      if numBoxes >= 1 then
         whichDim = dimension
         ;; finds a dimension with a side longer than viewcell length
         whichPoint = GetMin(cell,dimension)+
                      Min(numBoxes/2,1)*viewcellLength
         ;; subdivide in that dimension near the middle of the cell
         ;; at integral multiple of the viewcell length
         return TRUE
   return FALSE         ;; smaller than a viewcell, no need to split
```

Object association

Each new kd-cell created by subdivision must store all the the near objects for the viewcells it contains. So each kd-cell must store all the objects that lie within its object boundary. As shown in Figure 3-2, this includes all objects in the parent kd-cell within $(\frac{\text{bounding box length - viewcell length}}{2})$ of its side of the partitioning plane. To avoid problems with splitting objects, an object on the boundary of a bounding box is considered a near object for the viewcell and therefore be associated with the viewcell.

Here is the pseudo-code for object association for a new kd-cell:

```
FindNearObj (objectBoundary, nearObjList, parentObjList)
   for each object in parentObjList
   ;; go through objects in the parent kd-cell
      if Intersect(object,objectBoundary)
         Insert(nearObjList,object)
         ;; if the object intersects the object boundary of the new
         ;; cell, it is considered a near object for the new cell.
```

3.1.2 Texture map projection

Each time a viewcell is created during spatial subdivision, its far texture maps can be created from its far objects. For each viewcell, all objects in the model space other than the near objects are projected onto the sides of the bounding box, as shown in Figure 3-3.

Since a city typically consists of mostly buildings of a small number of stories, the overall model is usually much shorter vertically than in other dimensions. The height of the model is often less than the length of the bounding box. Thus, no objects lies outside the bounding box vertically. In these cases, it is unnecessary to create the top or bottom texture maps: only four texture maps need to be created, one for each of the vertical sides of the bounding box.

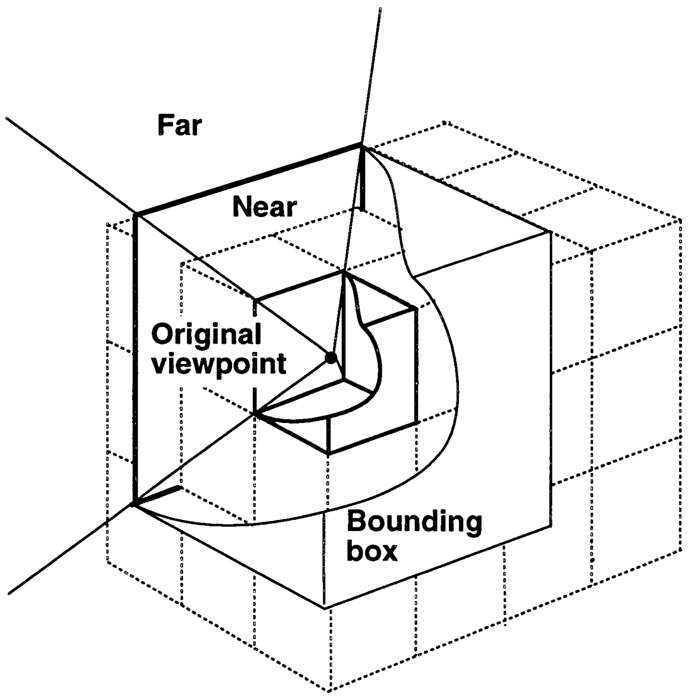


Figure 3-3: A 3D view of a viewcell, its bounding box, and the projection of far objects from the original viewpoint to one side of the bounding box shown in bold.

3.1.3 Running time and memory usage

The time and memory for the pre-processing stage depend on the size of the k-d tree and j, the number of objects in the model. Let n be the total number of viewcells in the scene. Since all the leaf kd-cells become viewcells, there are n leaf cells. The subdivision scheme creates a fairly balanced binary tree, so there are around $2n$ total kd-cells in the tree, and the height h is $\log_2(2n)$. (See Figure 3-4.)

Running time

Note that at the top of the tree, most of the objects in the model are within the object boundary for these kd-cells. But near the bottom of the tree, most of the objects are far objects, so the processing time should be less. As an upper bound, at most j objects need to be processed for each kd-cell to determine which objects are near, but the average number may be much smaller depending on the size of the bounding

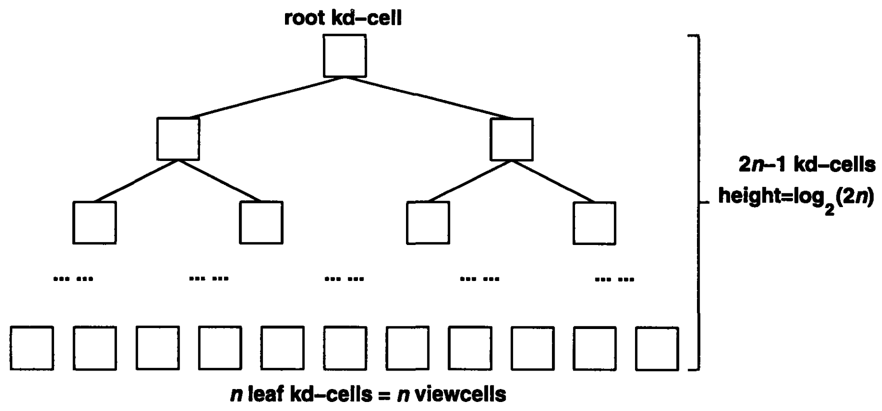


Figure 3-4: A k-d tree with n leaf kd-cells that are viewcells.

box. Far objects need to be projected onto the sides of the bounding box. Because objects far from a kd-cell are also far from all children of the kd-cell, the time needed for generating the list of far objects can also be optimized. For each viewcell, the processing time takes $O(j)$ time. So the overall running time is $O(j * 2n) = O(jn)$.

Much of the running time is spent in creating the texture maps, which can be used as a rough estimate of the overall pre-processing time. This is a reasonable estimate for a multi-processor workstation, where one processor can do the rendering while the others calculate the near objects. For each viewcell, most of the objects in the scene are outside the bounding box. From the original viewpoint, four texture maps of the far objects will be projected. This means approximately j polygons will be rendered. Let P_{sec} be the number of polygons rendered per second, the texture map projection time t_{sec} in seconds can then be estimated:

$$t_{sec} = \frac{jn}{P_{sec}} \tag{3.1}$$

The data from last chapter on RealityEngine 2 indicate that 1.8M polygons can be rendered in each second. The calculation below uses the campus model from Section 2.4.3 with 81,600 viewcells and 10M polygons. Applying Equation 3.1 gives the following:

$$\begin{aligned} t_{sec} &= \frac{10\text{M} * 81,600}{1.8\text{M}} \\ &= 450,000 \text{ seconds} = 130 \text{ hours} \end{aligned}$$

Memory usage

It is impractical and unnecessary to keep all the kd-cells and near objects in memory. Depth-first tree generation will traverse one branch of the tree at a time. When the near objects for both of the new kd-cells have been computed, the near objects for the parent kd-cell can be deleted. So during the generation process, only objects in the ancestor kd-cells need to be stored. The path from root to the leaf kd-cell is at most h. Therefore, at most $O(hj)$ objects need to be stored in virtual memory.

3.1.4 Texture map storage

An obvious concern with the Stratified Rendering Algorithm, especially for large environments with many objects, is that many texture maps must be pre-calculated and stored. For the campus model, almost 81,600 * 4 texture maps need to be generated. (For the viewcells on the outer boundary of the model, there will not be any far objects on the outer sides, so there is no need to create the corresponding far texture maps. The saving depends on the surface area of the model.) Intuitively, however, texture maps will be similar from one viewcell to the next. Therefore, temporal coherence can be exploited by encoding the differences to reduce the amount of data associated with each viewcell boundary.

3.2 Walkthrough stage

Since much of the work has been completed in the pre-processing stage, the computation during the actual walkthrough stage is straightforward. During the walkthrough,

Variables	*Meanings*
p_{text}	number of pixels in each texture map
n_{text}	number of texture maps
n_{poly}	number of near polygons

Table 3.1: Variables for discussing running time and memory use during the Walkthrough stage.

the user's simulated viewpoint moves through the input model. In the beginning of each frame, the viewcell that contains the simulated viewpoint is selected from the k-d tree. The near objects associated with the viewcell are then retrieved and rendered. The four texture maps stored with the viewcell will also be rendered onto the vertical sides of the bounding box.

As the viewpoint moves, the visualization must be updated. If the movement is within the viewcell, then all the far texture maps will stay valid. So these texture maps and the near objects can be simply re-rendered from the new viewpoint. *Only when the viewpoint crosses the viewcell boundary, does the new viewcell need to be constructed and the near objects and far texture maps retrieved.*

The running time and memory usage for the walkthrough stage are now examined. Table 3.1 defines some of the variables used in the following discussion.

3.2.1 Running time

The overall running time consists of two components, the time to render all the near objects and far texture maps for each frame and the time needed to update the near objects and texture maps when the viewpoint enters a new viewcell.

Updating for a new viewcell

For a given k-d tree for n viewcells, height h is approximately $\log_2(2n)$ as calculated above. Retrieval of a viewcell from the tree therefore will take $O(\log(n))$ time. Retrieving all the near objects will take $O(j)$ time. Since the viewcell size is usually

types of traversal	*speed(s)*	*update frequency(s/v)*
walking	$1m/s$	0.2/s
driving	$100km/h$	5.6/s

Table 3.2: Update frequencies for walkthroughs at different speeds.

much smaller than the bounding box size, the lists of near objects are similar from one viewcell to the next. The retrieval of near objects can be accelerated by keeping a delta list of the difference in near objects. When hardware delta-decoding for texture map is supported, texture maps can be retrieved faster.

Let L_{poly} be the average rate of retrieving 1 polygon, and L_{text} be the average rate of loading 1 pixel in a texture map. The time for updating the near objects and far texture maps is then

$$L_{poly}n_{poly} + L_{text}p_{text}n_{text}.$$

Updating needs to be done each time the viewpoint enters a new viewcell. The viewpoint will traverse the largest number of viewcells in a given time period by going perpendicularly across each boundary. Assuming that the user's viewpoint moves at a speed of s, the upper bound on update frequency is then $\frac{s}{v}$.

Recall that calculations in the last chapter give a viewcell length of around $5m$ for the city and campus models. Table 3.2 shows the update frequencies for two different simulated speeds: walking speed of $1m/s$ and driving speed of $100km/hr$.

Combining update frequency and time per update, the total time for updating (t_{update}) can be calculated as follows:

$$\begin{aligned} t_{update} &= \text{(time/update)} * \text{update frequency} \\ t_{update} &= \frac{s}{v}(L_{poly}n_{poly} + L_{text}p_{text}n_{text}) \qquad (3.2) \end{aligned}$$

Rendering time per frame

Note that by doing prefetching based on the velocity and direction of the viewpoint, one can amortize the cost of updating over each frame. Based on the analysis presented in Chapter 2, the correct bounding box size can be selected to ensure that everything can be rendered in a given frame.

3.2.2 Memory usage

Only the near objects and far texture maps need to be kept in physical memory. This is a substantial improvement over the brute force algorithm of keeping all the objects in memory, which is impractical for large models. Furthermore, for a model with a bounded density of objects, *the memory usage during walkthrough will also be bounded.*

Suppose that $m_{overhead}$ bytes of constant overhead need to be stored about the model, K_{poly} bytes of data need to be stored for each polygon, and K_{text} bytes of data need to be stored for each pixel of the texture map. The total amount of memory needed, m_{total}, will then be as follows:

$$m_{total} = m_{overhead} + K_{poly} * p_{poly} + K_{text} * p_{text} * n_{text} \qquad (3.3)$$

Therefore, with the Stratified Rendering Algorithm, both rendering time and memory usage during walkthrough are bounded. These are the primary benefits of using the algorithm. They ensure that even a large model that cannot fit in memory can be rendered in real-time.

Chapter 4

Multi-shell Extension

In the Stratified Rendering Algorithm, when a high frame rate is desired, a small bounding box size must be used so that all the near polygons and far texture maps can be rendered in each frame. When high accuracy is desired, the angular error α_{max} must be small. Because of the near-linear dependency of viewcell size on both the bounding box size and α_{max}, the resulting viewcell size may be quite small. This will result in frequent updates as the synthetic viewpoint moves among viewcells.

The asymptotic dependency of viewcell size on the model size means that the latter has limited direct effect on the size of the viewcell. However, a larger model means there are more viewcells overall. This fact, compounded by a small viewcell size, can potentially cause the following:

- longer pre-processing time
- more memory needed during pre-processing
- more texture maps

Since pre-processing only needs to be done once, pre-processing problems are of lesser concern. On the other hand, texture maps need to be stored for all walk-throughs. Frequent updates during walkthrough can also slow down the traversal rate and are thus also a pressing concern. Ideally, the number of viewcells should be small to reduce update time and texture map storage. These issues will be addressed below by examining a possible extension to the algorithm.

4.1 Shells of bounding boxes

Instead of using just one layer of texture-mapped bounding box, *Stratified* Rendering can be taken to its natural extension: using *shells* of texture-mapped bounding boxes. Shells of viewcells can be used for any given viewpoint, one within the other. Each viewcell will have its corresponding bounding box with its own near and far sides. Intuitively, the outer viewcell box will have a larger bounding box, and will be valid for much wider viewpoint movement. See Figure 4-1.

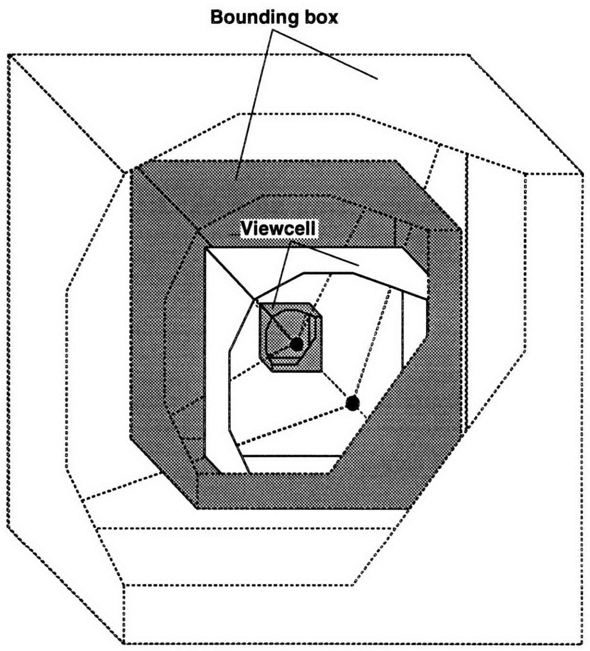


Figure 4-1: Two shells of enclosing bounding boxes: the grey bounding box is associated with the grey viewcell; the white bounding box is associated with the white viewcell.

Now the far objects of one bounding box will include only objects inside the next outer shell of bounding box. This way, the far objects will not be much farther away from the sides of the bounding box. Therefore, the size of a viewcell can be increased.

Let v_i be the length of ith viewcell, b_i be the length of the ith bounding box, and l_i be the largest distance from a point in the ith viewcell to the far objects. $0 <= i < i_{max}$, where i_{max} is the total number of shells. The 0th box is the innermost box. In this context, the original Equation 2.2 for relating lengths of model (l), bounding box (b), and viewcell (v) to α_{max} becomes

$$90 - \alpha_{max} = \arctan(\frac{v_i}{(l_i - v_i)\sqrt{2}}) + \arctan(\frac{b_i - v_i}{v_i\sqrt{2}}). \tag{4.1}$$

Let the overall volume of the model be V_{model}. The total number of viewcells ($n_{viewcell}$) can be calculated.

$$\begin{aligned} n_{viewcell} &= \frac{V_{model}}{{v_0}^3} + \frac{V_{model}}{{v_1}^3} + ... + \frac{V_{model}}{{v_{max-1}}^3} \\ n_{viewcell} &= V_{model} \sum_{i=0}^{i_{max-1}} \frac{1}{{v_i}^3} \qquad (4.2) \end{aligned}$$

4.1.1 Texture map storage

The overall storage depends on the total number of viewcells at all levels. The near objects need only be stored for the innermost level, while the far texture maps must be stored for all viewcells. Let n_i be the number of texture map and p_i be the number of pixels per texture maps at the ith level. Let M_{text} be the average amount of storage needed per pixel in the texture maps. The total amount of storage needed for texture maps, m_{text}, can be calculated as

$$\begin{aligned} m_{text} &= \sum_{i=0}^{i_{max-1}} (\text{no. of } i\text{th level viewcells})(\text{texture map storage/viewcell}) \qquad (4.3) \\ &= V_{model} \sum_{i=0}^{i_{max-1}} \frac{1}{{v_i}^3}(M_{text} n_i p_i) \\ &= V_{model} M_{text} \sum_{i=0}^{i_{max-1}} \frac{1}{{v_i}^3} n_i p_i. \qquad (4.4) \end{aligned}$$

4.1.2 Running time

Notice that using more texture maps per frame increases the amount of main memory and texture memory needed, but does not change the number of textured pixels in an image. By Equation 2.3, the number of near polygons that can be rendered in a frame is still the same. Therefore, the rendering time per frame will not change. The size of the innermost bounding box size can be the same as that of the single layer bounding box calculated previously.

The overall update time for a viewpoint entering new viewcells, however, will be quite different since there will be not just one, but many shells of viewcells for a given viewpoint. There will be a different update frequency for each level of viewcell.

Recall Equation 3.2 for average update time.

$$t_{update} = \frac{s}{v}(L_{poly}n_{poly} + L_{text}p_{text}n_{text})$$

For multiple shells of bounding boxes, the near objects need to be updated for only the 0th shell of viewcell, while the far texture maps need to be updated for all shells. The texture map update time is of more concern since in current hardware, the rendering pipeline must be stopped when updating the texture memory. On multi-processor workstations, the object loading can be done concurrently on a separate processor. The equation for texture update time is as follows:

$$\begin{aligned} t_{text\_update} &= \sum_{i=0}^{i_{max-1}} \frac{s}{{v_i}^3}(L_{text}p_i n_i) \\ t_{text\_update} &= s(L_{text}) \sum_{i=0}^{i_{max-1}} \frac{1}{{v_i}^3} p_i n_i. \end{aligned} \tag{4.5}$$

4.2 Spacing between shells

Given the analysis above, how then should the spacing between shells of bounding boxes be set? What are the trade-offs? Using an outer bounding box to limit how far the objects can be for a given bounding box, the size of the corresponding viewcell can be increased. This decreases the number of viewcells at the innermost level, and the corresponding update time and texture map storage. However, because there will be more shells of bounding box, there will be more corresponding viewcells and therefore more update time and texture map storage for the outer shells.

The goal then is to select the b_i's such that the update time t_{text_update} and texture map storage m_{text} is minimized. Notice that both values are proportional to the term $\sum_{i=0}^{i_{max}-1} \frac{1}{{v_i}^3} p_i$. Assuming that the same number of pixels are stored for each texture map, then p_i is constant. So both values are proportional to $V_{model} \sum_{i=0}^{i_{max}-1} \frac{1}{{v_i}^3}$, the total number of viewcells, as expected. Therefore, the b_i's should be selected to reduce the total number of viewcells.

General case

Given a viewpoint, it must belong to i_{max} viewcells, each with its corresponding bounding box. An ith level viewcell, however, need not belong to a unique $(i+1)$th level viewcell: it can be part of up to eight different $(i+1)$th viewcells. A top view is show in Figure 4-2.

4.2.1 Spacing calculations

The general case allows more freedom in setting the spacing between bounding boxes and viewcell sizes to adapt to α_{max}. However, the ith bounding box may be bounded by up to eight $(i+1)$th bounding boxes. This means up to eight more sets of texture maps need to be stored for each viewcell.

The bound (l_i) for farthest distance from a viewpoint in the ith level viewcell to

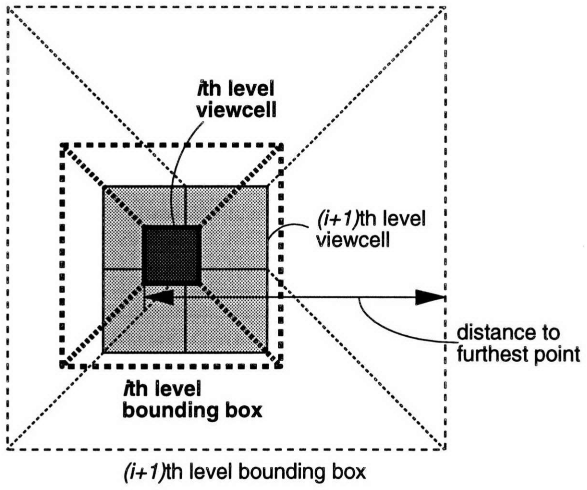


Figure 4-2: A top view of viewcells on adjacent levels and their corresponding bounding boxes.

its far objects is then $v_i + \frac{v_{i+1}}{2} + \frac{b_{i+1}}{2}$. By definition, $l_i < l$. Then Equation 4.1 becomes in this case

$$90 - \alpha_{max} = \begin{cases} \arctan(\frac{v_i}{(\frac{v_{i+1}}{2}+\frac{b_{i+1}}{2})\sqrt{2}}) + \arctan(\frac{b_i - v_i}{v_i\sqrt{2}}) & 0 <= i < i_{max-1} \\ \arctan(\frac{v_i}{(l-v_i)\sqrt{2}}) + \arctan(\frac{b_i - v_i}{v_i\sqrt{2}}) & i = i_{max-1}. \end{cases}$$

Once the bounding box sizes are freely set, the corresponding viewcell size can then be determined. Two different spacing strategies will be examined: linear and exponential. For linear spacing, the bounding box size grows by a constant factor f: $b_{i+1} = f * b_i$. For exponential spacing, the bounding box size grows by an exponential factor f: $b_{i+1} = {b_i}^f$.

Recall that for the campus model from Section 2.4.3, using single level bounding box $b = 56m$, $\alpha_{max} = 10°$, and $l = 1000$ gives 50,000 viewcells of length $6.35m$. For multiple levels of bounding boxes, b_0 is set to $56m$.

Linear spacing

Figure 4-3 shows the resulting number of viewcells using different linear spacing between bounding boxes.

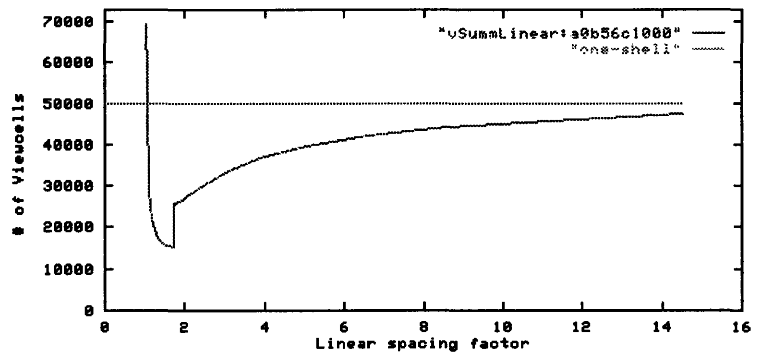


Figure 4-3: The number of viewcells needed for different linear spacing factors compared with the value for the original one-shell algorithm.

When the spacing between bounding boxes are very small, many levels of viewcells are needed. This results in large number of viewcells overall. As the spacing gets wider, fewer levels are needed, so the number of viewcells overall decreases. But when the levels of bounding boxes are too far apart, the far objects within the next level of bounding box become too far. So the benefit from using shells of bounding boxes decreases. The number of viewcells needed gradually approaches that of using only one level of bounding boxes, as shown in Figure 4-3.

The sudden rise in number of viewcells in Figure 4-3 right after $f = 1.72$ comes from the doubling in the number of the lowest-level viewcells as their sizes get too small. (Before this point, the $10m$ vertical dimension can be covered by one viewcell; after this point, 2 viewcells are needed.)

The lowest number of viewcells is 15,000 obtained by setting f to 1.72. Table 4.1 shows the viewcell and bounding box sizes used.

Exponential spacing

By making the inner layers closer together and outer layers farther apart using exponential spacing, fewer levels of viewcells are needed. This can result in smaller number of viewcells overall.

Figure 4-4 shows the resulting number of viewcells using exponential spacing.

Viewcell length	*Bounding box length*	*# of viewcells*
88.0	493.3	144
51.1	286.3	400
29.6	166.2	1156
17.2	96.4	3481
10.0	56.0	10000
Total		15230

Table 4.1: The sizes of viewcells and corresponding bounding boxes for using linear spacing with a factor of 1.72.

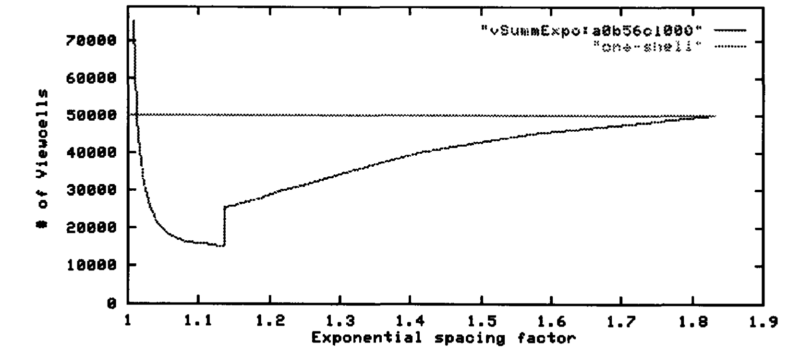


Figure 4-4: The number of viewcells needed for different exponential spacing factors compared with the value for the original one-shell algorithm.

Viewcell size	*Bounding box size*	*# of viewcells*
58.1	368.4	324
30.0	181.1	1156
16.6	97.0	3600
10.0	56.0	10000
Total		15129

Table 4.2: The sizes of viewcells and corresponding bounding boxes for using factor of 1.13 exponential spacing.

Notice that the graph is similar to Figure 4-3 for linear spacing. The lowest number of viewcells is also 15,000 obtained by setting f to 1.13. Exponential spacing does not contribute much. Table 4.2 shows the viewcell and bounding box sizes used. Apparently, the savings from fewer levels are balanced by more viewcells per level.

4.3 Summary

The multi-shell extension reduces the overall number of viewcells to about $\frac{15{,}000}{50{,}000} =$ 30% of the original in the example. But since up to eight sets of texture maps need to be stored per viewcell, the texture map storage does not substantially decrease.

Similarly, because in each ith viewcell, up to eight sets of texture maps need to be updated for the ith level bounding box, depending on the layout of the $(i+1)$th level viewcells, the update frequency is also not substantially decreased.

So this extension may not always improve the algorithm. However, if an input model varies greatly in density, there may not be any far objects between two given levels of bounding boxes. Then there is no need to store the corresponding texture maps or reload them. For models of this type, the multi-shell extension may work well.

Chapter 5

Implementation

The original Stratified Rendering Algorithm presented in Chapter 3 has been incorporated into CityScape, a real-time walkthrough system. The input to the system is a geometric model of a city. The system runs in either the pre-processing mode or the walkthrough mode. In the pre-processing mode, which corresponds to the pre-processing stage of the algorithm, CityScape generates far texture maps. In the walkthrough mode, CityScape uses these far texture maps to allow interactive walkthrough of the input model.

CityScape is implemented using the IRIS Performer from Silicon Graphics. This toolkit is ideal for real-time 3D graphics application because it encompasses the functionality of the graphics library, provides high-level abstraction for easy manipulation, and is tuned for high performance on a multi-processor workstation [10, 6].

5.1 Interface

The user interface of the walkthrough system allows the usual navigational movement: moving forward and backward, turning left and right, ascending, descending, and looking up and down.

From the control panel, the user can set or return to a home position, take a snapshot of a view or a series of views, view from normal navigation viewpoint and from bird's eye viewpoint (for better illustration of the algorithm), as well as toggle collision detection. Different styles of rendering can also be selected: full algorithm, near objects only, far objects only, far texture only, and brute force (render all objects). In addition, the user can select a navigation path through the scene for recording and playback. This allows the same set of scene data to be rendered using several algorithms, so performance data for different rendering algorithms can be collected.

5.2 Design

CityScape consists of three modules:

1. Viewer, which sets up the walkthrough interface
2. ModelMgr, which manages the city model
3. BoxMgr, which manages the *k*d-tree

The main controller is the Viewer, which calls the ModelMgr to create the geometric model. The ModelMgr in turn calls the BoxMgr to create the correct hierarchical data structure, as shown in Figure 5-1. Each of these modules will be examined in turn.

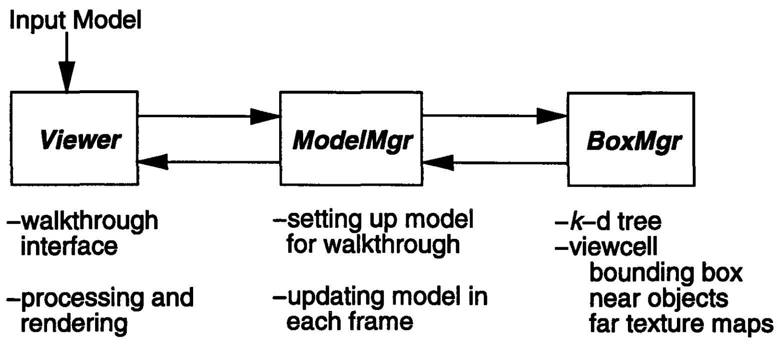


Figure 5-1: CityScape consists of three modules: Viewer, ModelMgr, and BoxMgr.

5.2.1 Viewer

The Viewer is the main controller for CityScape. It manages the application, culling, and drawing processes of the program to take advantage of the multi-processor hardware. It controls the system in either pre-processing mode or walkthrough mode. In the pre-processing mode, the Viewer calls the ModelMgr to create the far texture maps.

In the walkthrough mode, the Viewer calls the ModelMgr to set up the walkthrough model. It then creates and manages the walkthrough interface. In each frame of the walkthrough, as the simulated viewpoint moves, the Viewer calls the ModelMgr to update the model.

The Viewer manages miscellaneous interface operations such as recording/playing back a path. Recording a path consists of writing to a file a sequence of viewpoints and viewing directions. Playing a path consists of retrieving the viewpoints and rendering from them in order.

5.2.2 ModelMgr

The ModelMgr reads in the input file to create the model of geometric objects. It then manages this model. During pre-processing, it calls the BoxMgr to create the texture maps.

In the walkthrough mode, the ModelMgr first calls the BoxMgr to set up the appropriate data structure. During interactive walkthrough, the ModelMgr maintains the correct objects for rendering according to the show style selected. The texture maps are rendered onto the sides of a cube representing the bounding box. In the bird's eye view mode, additional objects may be added to the model: the view frustum object is a wireframe pyramid showing the position and direction of the simulated viewpoint; the viewcell object is a wireframe cube. In the full algorithm bird's eye view mode, for example, the model consists of the near objects, the far texture-mapped bounding box, the view frustum, and the viewcell, as shown in Figure 5-2.

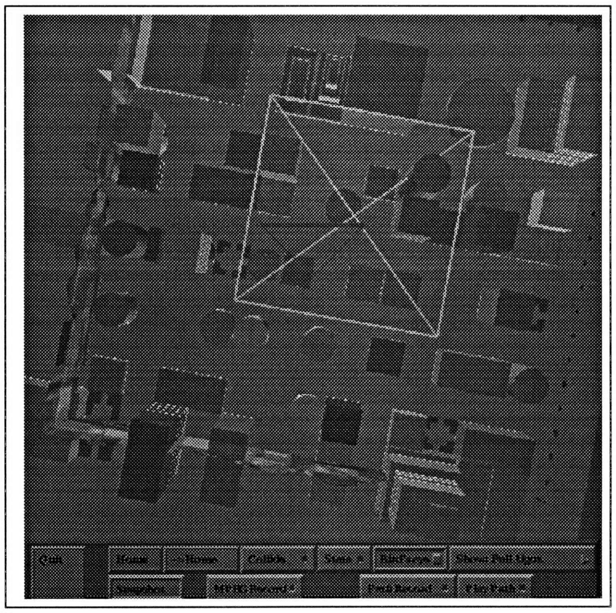


Figure 5-2: A scene using bird's eye view, showing the view frustum in red, viewcell in cyan, and the bounding box using texture maps.

In each frame, as the user's viewpoint moves, the ModelMgr is responsible for checking to see if the viewpoint crosses the viewcell boundary. If so, it calls the BoxMgr to retrieve the new near objects and texture maps.

5.2.3 BoxMgr

The BoxMgr uses the *k*d-tree structure to subdivide the input geometry model into viewcells. In the pre-processing mode, the BoxMgr generates the correct texture maps for each viewcell and writes them to the disk. For each viewcell, the far objects are the difference between the list of all the objects in the model and the list of near objects.

In the walkthrough mode, the ModelMgr will call the BoxMgr to retrieve the near objects and far texture maps for a given viewcell from the *k*d-tree.

5.3 Engineering Decisions

In the walkthrough system, all the near polygons are rendered using hardware lighting effects. Two directional light sources opposite each other at 14° from the horizontal are used.

The available graphics system supports 4MB of texture memory and each pixel occupies 4 bytes. To fit all 4 texture maps for a viewcell into memory, each texture map consists of $\frac{1}{4}$M pixels. The low resolution can cause aliasing effects on the screen. With more texture memory, this problem can be solved.

Chapter 6

Results and Discussion

To test the effectiveness of the Stratified Rendering Algorithm, a series of tests were run on the CityScape virtual walkthrough system using three different rendering algorithms:

1. Brute force: All the objects in the scene were processed and rendered in each frame.

2. Stratified A: Using the Stratified Rendering Algorithm, texture maps were pre-computed using $\alpha_{max} = 7.5°$. During walkthrough, only near objects and a constant number of far texture maps were rendered.

3. Stratified B: Similar to Stratified A, but with $\alpha_{max} = 1°$

All tests were performed on a Silicon Graphics Onyx with four 250MHz R4400 processors, 512MB of memory, one Reality Engine II Graphics Pipeline, and 4MB of texture memory.

The input consisted of a randomly generated cluster of city blocks of $1km * 1km * 67m$ with about 450 buildings and 83,000 polygons. The viewport size and the texture map size were both 512 pixels by 512 pixels. The horizontal and vertical view angles were both 90°. The bounding box length was set to 330m for both Stratified Rendering cases.

Algor.	α_{max}	v	b	*Pre-proc. time*	*Texture map storage*
A	7.5°	$33m$	$330m$	$50min$	4M
B	1°	$5m$	$330m$	14,000min(proj.)	1155M(proj.)

Table 6.1: Relevant information for Stratified A and B

Table 6.1 summarizes the relevant information for the Stratified A and B cases. Note that the pre-processing time and storage for Stratified B were projected values. With texture map compression, the texture map storage and projection time can be improved. A compression rate of 100 to 1 was assumed for Table 6.1.

6.1 Rendering time

The rendering time per frame for different algorithms is plotted in Figures 6-1 and 6-2. A single observer path with 300 frames was used for all runs.

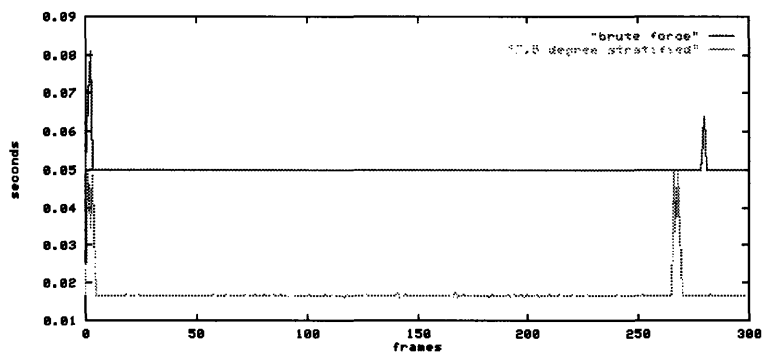


Figure 6-1: Performance data for brute force rendering vs. Stratified Rendering using $\alpha_{max} = 7.5°$.

The brute force rendering case gave an average rendering time of approximately 0.05 seconds/frame. For both Stratified Rendering cases, the average rendering time was approximately 0.016 seconds/frame. This corresponded to frame rates of 20Hz and 60Hz, respectively. Using the Stratified Rendering Algorithm in this case achieved an improvement factor of three.

The initial spikes in all the plots came from loading in a new path. The spike

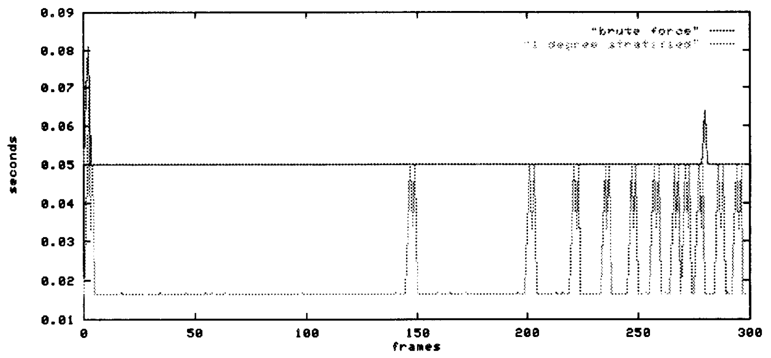


Figure 6-2: Performance data for brute force rendering vs. Stratified Rendering using $\alpha_{max} = 1°$.

around frame 280 for the brute force algorithm came from the change in amount of geometry in the scene.

The spikes in the latter part of the path in Stratified Rendering came from loading texture maps when the viewpoint entered a new viewcell. When α_{max} was 7.5°, the viewcells were larger and the update frequency for new viewcell was smaller. When α_{max} was 1°, the viewcells were smaller and the update frequency was higher. Indeed, Figure 6-1 shows that 1 viewcell boundary was crossed when $\alpha_{max} = 7.5°$, whereas Figure 6-2 shows that 11 viewcell boundaries were crossed for the same path when $\alpha_{max} = 1°$. Furthermore, Figure 6-2 shows that the movement of the viewpoint was accelerating, crossing more viewcell boundaries towards the end.

With more efficient hardware and software for incremental texture decompressing and loading, the time delay for loading texture maps can be reduced. Figure 6-3 shows the ideal performance data with the texture loading spikes removed.

6.2 Image quality

How do the image quality of the different algorithms compare? Figure 6-4 compares a scene rendered using brute force vs. Stratified with $\alpha_{max} = 7.5°$.

As one can see, the difference is strictly within the angular error bound of 7.5°.

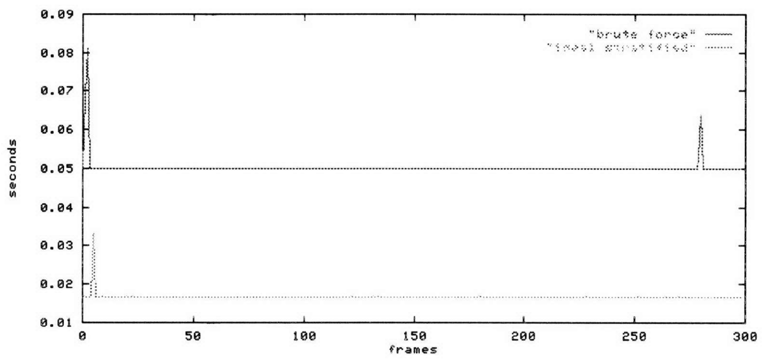
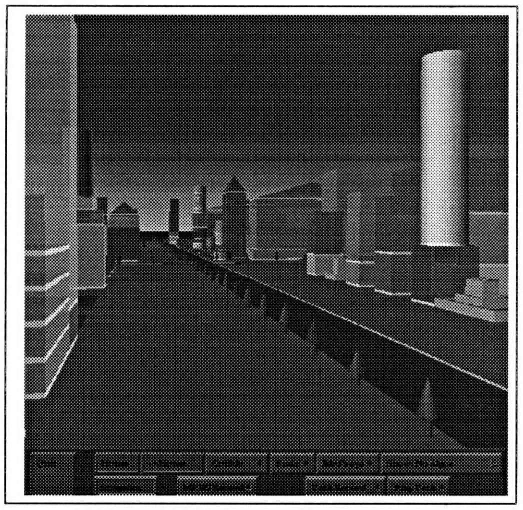
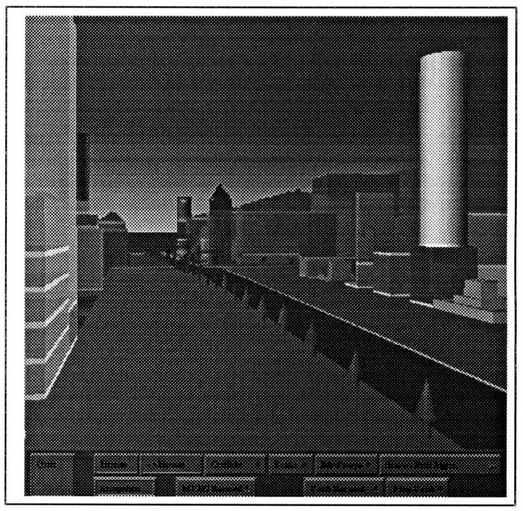


Figure 6-3: Performance data for brute force rendering vs. ideal Stratified.



(a) Brute force



(b) Stratified Rendering with $\alpha_{max} = 7.5°$

Figure 6-4: Image quality of brute force rendering compared with Stratified Rendering.

The near objects that the users tend to pay more attention to are rendered the same, while the far texture map in the background of Figure 6-4(b) causes some slight difference, especially in the alignment of the curbs.

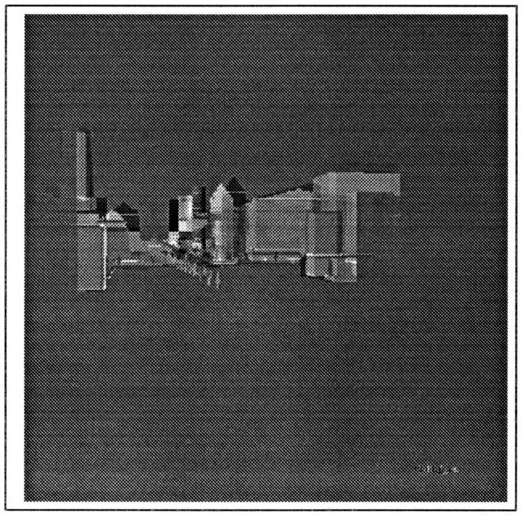


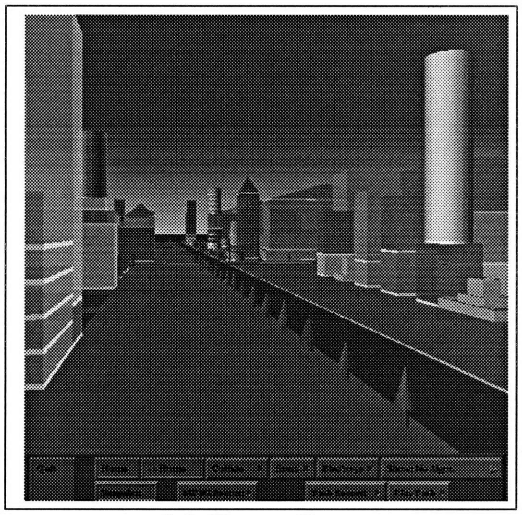
Figure 6-5: Difference between Figures 6-4(a) and (b).

The difference between the two is shown in Figure 6-5. The color values at each pixel are subtracted and added to 128 (neutral grey). The non-grey areas represent differences. Notice that most of the difference is shown in shades of grey. This comes from the slight color differences of the polygons projected from two different viewpoints.

With smaller angular error, one can expect to see even less difference. Figure 6-4 compares a scene rendered using brute force vs. Stratified with $\alpha_{max} = 1°$.

Again, one can observe that the difference is strictly within the angular error bound defined above. Notice that the curb alignment error is reduced for Figure 6-6(b).

The difference between the Figures 6-6(a) and (b) is shown in Figure 6-7. Note that compared to Figure 6-5, both the differences in distances and in colors are reduced because the viewpoint is closer to the original viewpoint.


(a) Brute force

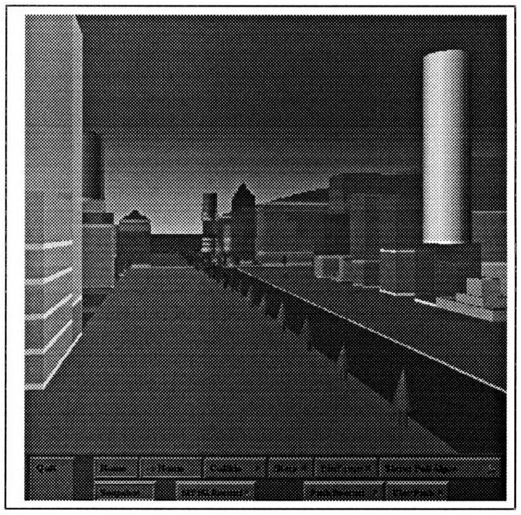

(b) Stratified Rendering with $\alpha_{max} = 1°$

Figure 6-6: Image quality of brute force rendering compared with Stratified.

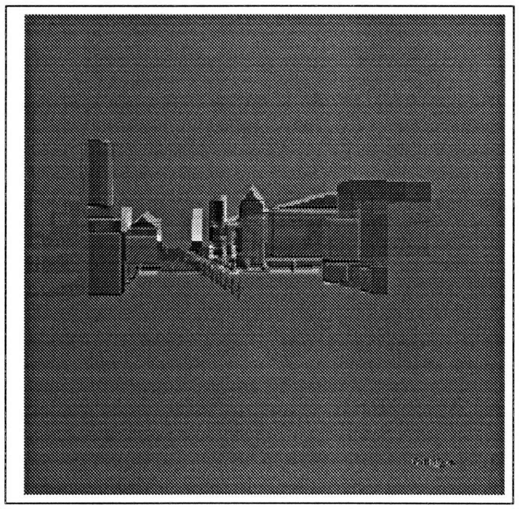


Figure 6-7: Difference between Figures 6-6(a) and (b).

Chapter 7

Future work

Virtual navigation is an area that has seen much recent development because it exposes many of the barriers to achieving truly interactive 3D graphics. The algorithm presented in this thesis goes one step closer to achieving both speed and realism. There remain many interesting future improvements.

7.1 Improvements in texture mapping

Because the new algorithm is based on using texture maps to approximate far objects, improvements in texture mapping hardware or software will translate directly to better performance of the algorithm. Several different aspects of the problem will now be examined.

Texture map compression

It has been mentioned briefly that compressing texture maps will reduce the amount of texture map storage. For viewpoints close to each other, the projected images of far objects will be similar, so delta-encodings of the images can be used. Furthermore, because the underlying geometry is available, and the viewpoints are a fixed distance apart, one should be able to derive a better geometrically-based compression scheme.

Texture map loading

Currently, during the first few frames when the simulated viewpoint crosses a viewcell boundary, the rendering time increases because the drawing process must be stopped to load in the new texture maps from main memory to texture memory. With faster texture mapping hardware, the loading time can be reduced. In addition, if the delta from the decompression scheme can be incrementally loaded in using hardware, the amount of traffic and subsequently the loading time can be substantially decreased.

Some of the work can also be offloaded to other frames, such as by loading a part of the texture map when the drawing process is idle. This can happen when many near objects are culled away because they are not in the viewing frustum. Texture maps for the next viewcell can be pre-fetched depending on the velocity of the simulated viewpoint.

Texture map interpolation

When the viewpoint crosses a viewcell boundary, and the far texture maps are swapped, there will be a sudden jump in the far texture maps. At the boundary of the viewcell, the angular error between the actual far scene and the texture map for each viewcell is at most α_{max} either way. In the worse case, this jump could be close to $2\alpha_{max}$, which will be disconcerting to the user. A better approach would be to smooth out the transition. For example, the two sets of texture maps can be interpolated, such as by gradually changing the delta in the compression scheme.

Note that these three texture mapping issues are inter-related. With further research, a uniform geometrically-based texture map compression-loading-interpolation scheme can be developed that will provide a satisfactory solution to all.

7.2 Near object improvements

One benefit of the new algorithm is that near objects consist of actual polygons that can be manipulated. The current CityScape system uses it for collision detection. It can also be used in many other ways, such as for compositing other actors, etc..

Near Object Management

The size of the bounding box can be enlarged by increasing the number of near objects to be rendered in each frame. For example, active level-of-detail management for the near objects can be used: objects closer to the eye point are rendered in full details while objects further away are rendered using less detail.

Interactivity

The algorithm can be extended to allow the user to interact with the near objects and manipulate them, such as picking objects to see more information. Moving objects will be a more difficult problem since other viewcells' far texture maps may need to be updated to reflect the change.

7.3 Others

Additional topics for further exploration include adaptive viewcell generation that creates smaller viewcells, and therefore more texture maps, for areas with denser polygons. This allows the city model to be more varied.

Saving the normals for buildings when creating the texture maps allows more accurate shading to be computed during navigation. Going one step further, solutions of global illuminations for the far objects can be saved, then interpolated for a given viewpoint. The effect of the near objects on global illumination solutions needs to be studied. With more accurate shading, the visual quality of rendering can be improved, and the overall navigation experience enhanced.

Chapter 8

Conclusion

This thesis has described the Stratified Rendering Algorithm for scene approximation during interactive visualization of large, irregular environments. The algorithm pre-computes a set of far texture maps to approximate far objects. During the walk-through stage, only a small number of near objects and a constant number of far texture maps need to be rendered. This substantially reduces the average rendering load during navigation. In fact, using knowledge of the underlying hardware to determine the number of near objects, one can ensure that a given average frame rate is obtained. Using a given bound on angular error, the visual quality of rendering can also be guaranteed.

An extension to the algorithm that uses shells of bounding boxes is also discussed. The analysis on spacing between the bounding boxes shows that this extension may be useful for certain sparse input models.

The algorithm has been incorporated into CityScape, a virtual navigation system. Tests conducted using CityScape show that compared to the brute force method of rendering all visible objects, Stratified Rendering produces a significantly higher frame rate. At the same time, the visual quality of the resulting images is comparable.

Appendix A

Random City Model Generator

The CityScape Walkthrough System has been tested using random city models created by a simple random model generator. The generator takes the following parameters:

- maximum height of the city
- length of a block
- width of a block
- number of rows and columns of buildings on each block
- number of blocks in either dimensions
- street width and height
- curb width and height
- distance between trees

The output city model consists of a rectangular grid of city blocks. Each of the blocks contains the following items at random locations:

- park of 4 green grass strips
- building of random color, shape, and height
- fountain of random height

Some of the blocks are lined with trees at given separation.

Figures A-1 and A-2 show two models generated using different input parameters.

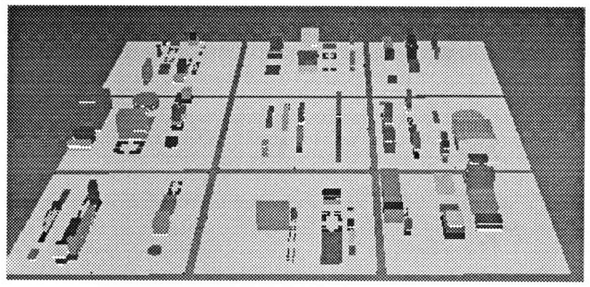


Figure A-1: A sparse model of 9 blocks (up to 5 rows and columns of buildings each).

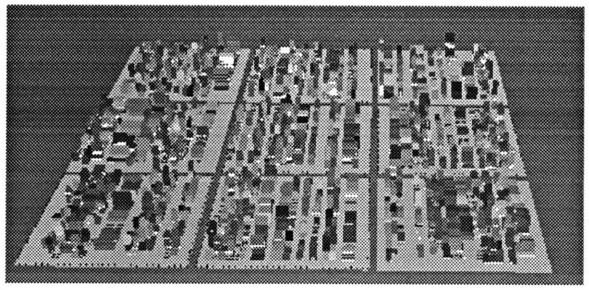


Figure A-2: A dense model of 9 blocks (up to 20 rows and columns of buildings each).

A.1 Algorithm

The output model is generated block by block. Each block can be either a regular city block or a park block. A *park block* consists of one big park with a fountain in the middle. A building block contains park, building, and fountains.

A *building block* is first divided into rows at random spacing, then within each row divided into columns at random spacing. At each rectangular row/column location, if the space is too small, it is left empty. Otherwise, there could be either a park, building, or fountain. A *park* consists of four L-shaped grass strips at each corner. A *fountain* consists of several layers of water and wall. Each inner layer is smaller and taller than the outer layer.

A *building* can be either cylindrical or rectangular, solid or semi-transparent, with either a cone-shaped or pyramid-shaped top. Each building actually consists of multiple *segments*, each with either one-color or alternating-color schemes. A segment can have thin horizontal dividers to separate the floors.

Bibliography

[1] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[2] Bradford Chamberlain, Tony DeRose, Dani Lischinski, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierachy. manuscript, July 1995.

[3] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environment. In *Computer Graphics Proceedings*, volume 26. SIGGRAPH, August 1993.

[4] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Special Issue on Symposium on Interactive 3D Graphics*, pages 11–20. SIGGRAPH, 1992.

[5] Silicon Graphics. Onyx graphics specifications. SGI Web Page, 1996.

[6] IRIS Performer Group. Iris performer 2.0 technical report. Technical report, Silicon Graphics, 1995.

[7] W. Jepson, R. Liggett, and S. Friedman. An environment of real-time urban simulation. In *Symposium on Interactive 3D Graphics*, pages 165–166, 1995.

[8] Paulo W.C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Computer Graphics Proceedings.* SIGGRAPH, August 1995.

[9] Patricia McLendon. *IRIS Performer Programming Guide.* Silicon Graphics, 1992.

[10] John Rohlf and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Computer Graphics Proceedings.* SIGGRAPH, 1994.

[11] Jarek R. Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. Technical report, IBM T.J. Watson Research Center, 1992.

[12] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Computer Graphics Proceedings.* SIGGRAPH, 1996. manuscript.

[13] Seth Teller, Tomas Lozano-Perez, and Alan Edelman. Digitizing Cambridge: Hierarchical acquisition, and real-time visual simulation, of textured 3D geometric models of urban environments at sub-meter resolution, manuscript. August 1995.

[14] Seth J. Teller. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics Proceedings*, volume 25. SIGGRAPH, 1991.

[15] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments.* PhD thesis, University of California at Berkeley, 1992.

23651-78