PREDICTIVE RENDERING

By

Paul Fearing

B. Sc. (Computer Science) Queen's University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

....

....

....

THE UNIVERSITY OF BRITISH COLUMBIA

JANUARY 1996

© PAUL FEARING, 1996

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
201-2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date:

Abstract

Computer graphics has always been concerned with increasing rendering speed. This thesis introduces a new method to reduce the cost of the rendering pipeline. It uses fast, simple transformations to predict scene motion some small number of frames in the future. Scene primitives are grouped into nodes in a "plan tree" according to the predicted future motion of their projected screen coordinates. Different nodes in the tree are rendered at different frame rates. Nodes containing slowly moving or static primitives can be rendered many fewer times than nodes containing quickly moving primitives. The rendered scene subsets are depth composited together to form a final frame.

Predictive rendering draws graphics primitives only when they move. This contrasts with usual rendering methods, where all primitives (moving or not) are drawn each and every frame. Depending on the amount of temporal coherence between frames, predictive rendering reduces the number of primitives sent to the graphics pipeline, allowing dramatic improvements in overall rendering speeds.

In scenes with large amounts of temporal coherence, our implementation achieved speedups ranging between 270% and $1,850\%$ of the normal rendering times. Analysis on several test scenes (varying in polygonal complexity) showed worst-case prediction costs that ranged between 0.05% and 13.5% of the total rendering time. Actual implementation showed that worst case costs varied between $< 1\%$ to 13%, depending on the scene. These costs indicate the penalty that must be paid in scenes with absolutely no temporal coherence.

Table of Contents

List of Tables

List of Figures

Acknowledgments

"This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author, and do not necessarily reflect the views of the National Science Foundation."

The following people and groups should be acknowledged for their help in the direction and development of this thesis and the research behind it. Their contribution was greatly appreciated.

- Funding support was provided by the US National Science Foundation (NSF), National Science and Engineering Research Council (NSERC), the BC Advanced Systems Institute (ASI), the University of British Columbia (UBC), and the Imager Computer Graphics Laboratory.
- Technical support was provided by my supervisor, Alain Fournier, my second faculty reader Kellogg Booth, my student reader Scott Andrews, and assorted Imager lab members.
- Moral and emotional support was provided by my parents, and those Varsity Outdoors Club members that made sure that I didn't get my tan from a computer monitor.
- Physical support was mainly provided by Maxwell House.
- And finally, spiritual support was provided by the glorious physical beauty of Vancouver, and the sunsets and beaches that made most of these ideas possible.

Thanks.

Chapter 1

Introduction

"Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said, gravely, "and go on until you come to the end: then stop."

- Lewis Carroll

Computer graphics has an important and growing impact on many areas of the modern world. It is used in thousands of diverse applications, from visualization and design, to education and entertainment. Common among the myriad of different uses for computer graphics is a desire for rendering speed. Faster rendering gives smoother animations and allow scene databases with even greater complexity. It reduces "thumb-twiddling", freeing the user to spend less time waiting and more time working on the next task.

Computer manufacturers have responded to the "need for speed" with an ever-improving array of machines designed specifically for graphics. These computers make use of specialized hardware, multi-processing and pipelining to increase rendering throughput. Rendering times are reduced primarily by doing the same work, but at a faster rate than before.

One can also improve rendering times by maintaining the rate of work, but reducing the overall quantity. Along the graphics pipeline, there are several opportunities for the hardware to discard or ignore primitives that are deemed non-essential. Reducing the number of primitives the system needs to render at any particular moment improves the overall graphics speed. This becomes especially important as individual primitives have more and more realistic (and costly to render) textures, lighting, and antialiasing.

Unfortunately, the algorithms used inside the graphics pipeline have little or no global information about the complete scene model. This prevents the pipeline hardware and software from making optimizations anywhere beyond the most local level. With only local knowledge, we can only make local optimizations. For example, in a model consisting of many grouped and related polygons, local optimizations can only exclude and test on a polygon-by-polygon basis. Global knowledge allows us to simultaneously exclude large numbers of related polygons over possibly more than one frame. In order to exploit more global optimizations, we must expand the system's understanding of the complete scene model. This implies software culling techniques working to preprocess data that eventually is sent to the graphics pipeline.

Software culling methods have great potential, because they are not restricted by limitations in hardware size or speed. A 20-million polygon database could conceivably be culled to 20 polygons - a million time reduction practically impossible via improvements to the graphics pipeline hardware or microcode. The hardest part of software culling is (obviously) determining which polygons are important and which are not. This process is vastly complicated as objects, lights, and the viewer move over time, changing the relative importance of a scene's components.

The dominant approach so far has been to identify and exploit common or coherent properties in an image, or even sequences of images. Properties generally vary smoothly across areas and time. We can use this coherence to reduce the amount of work required to generate a picture, allowing us to use information about one area to compute the results of other nearby areas with little or no modification. It might cost less to determine the changes to a previous result than to compute everything again from scratch.

In this thesis we will introduce a new method that takes advantage of *temporal coherence* to render polygons only when they move or change. Static polygons are rendered only once. Moving polygons are rendered a number of times proportional to their screen motion. Because slow-moving and static objects only need to be redrawn in a few percent of the total frames, we can potentially achieve huge speedups in many common animation scenarios. The amount of speedup depends on the amount of temporal coherence in an animation. The algorithm produces correctly occluded pictures identical to traditionally rendered images.

In Chapter 2, we provide an overview of the few existing temporal coherence algorithms, concentrating specifically on scan-conversion. Previous work in this area is tentative, at best.

We also describe a few ray tracing temporal coherence algorithms for contrast. In Chapter 3, we introduce the predictive rendering algorithm and describe its requirements, operation, and other considerations. Chapter 4 contains a theoretical analysis of the computational cost of predictive rendering. Prediction costs are compared to total work, giving an indication of the amount of temporal coherence required before speedups are achieved. Chapter 5 describes some design decisions, special implementation concerns, and experimental proofs of the speedups made possible by this new algorithm. Chapter 6 summarizes the algorithm and makes some concluding remarks on its potential. Appendix A provides a list of experimental parameters used during test. Appendix B lists and describes the variables used in this thesis. Finally, the glossary provides definition of the new terms introduced in the text.

Chapter 2

Previous Work

History must not be written with bias, and both sides must be given, even if there is only one side.

- John Betjeman

This chapter provides an overview of previous attempts to exploit temporal coherence in image generation. Temporal coherence takes advantage of consistencies between consecutive frames. If we do a certain amount of work for one picture, we should not have to do the same amount of work for another image that differs only slightly.

Previous attempts to exploit temporal coherence have mostly been concentrated in ray tracing, mainly because the cost per ray-traced frame is so high. Ray tracing methods generally take advantage of the independence of individual pixel samples. Although it is useful to examine the temporal coherence methods used in ray-tracing, we are more interested in temporal coherence for scan-conversion. Scan-conversion is one of the most popular rendering paradigms, and is already very heavily supported (via hardware and software library calls) on most serious specialized graphics systems.

Traditional scan-conversion has always been computationally simple, often fast enough for time-critical applications. High rendering speeds make potential speedups from temporal coherence less critical and thus less researched. But, with the continual improvement of available graphics features (more lights, more textures, more material properties) the average cost per scan-converted primitive is quickly rising. This makes it increasingly worthwhile to research how we can avoid having to deal with a large part of the total primitive database.

We begin by discussing existing methods of temporal coherence for scan-conversion. We will then move on to touch briefly upon ray tracing temporal coherence, to give us a further perspective on the scan-conversion approach. We will ignore radiosity temporal coherence for the purposes of this thesis, because previous work is sufficiently different from scan-conversion to shed no real insight on our problem.

2.1 Scan-Conversion Methods

Temporal coherence methods for scan-conversion have at once been sparsely studied and exhaustively used. Compositing methods have been used in hand-drawn animations as early as 1915 [Halas 59]. Yet, the few fully automatic solutions with real potential for use have only been proposed in the last few years.

2.1.1 Compositing

In traditional cel-based animation, the artist draws hundreds of sequential frames of a scene. Moving objects are varied a tiny bit on each frame, giving the illusion of continuous motion when viewed at fast enough frame rates. Animation artists realized almost immediately that it was too time consuming to redraw the entire scene for each frame. First, constant components of scenes had to be redrawn again and again for each and every frame - a significant amount of extra work. Additionally, the artist was required to exactly duplicate large portions of the previous frame, including brush stroke texturing, precise coloring effects, tiny surface details, etc. Small variations between frames caused unpleasant flickering.

To solve these problems, animation artists quickly moved to transparency-based methods. A scene is separated into layers, and then recomposited during filming. Separation into layers is done by hand, and depends on several factors. The amount of object motion is a prime criteria. The more an object moves, the more efficient it is to place it on its own layer. The background often gets at least one layer, to allow a single, detailed drawing. Unfortunately, the scene cannot be separated totally using only motion as a criteria; it must also be grouped into logical objects that a human animator can draw. A fluttering background leaf and the upper lip of the prime character may need to be updated at the same frame rate, but are unrelated enough to give a human pause when required to draw them as a group. A much more likely grouping is a tree (with fluttering leaves), and the prime character (with an entire moving mouth). Grouping by objects helps the animator, but can increase the amount of work needed, as static parts of the moving group must be redrawn each frame.

Figure 2.1 shows a frame [Halas 90] scanned from *Asterix in Britain* by Pino van Laamsweerde. The simple foreground characters are on a different layer than the complex and carefully shaded and colored background.

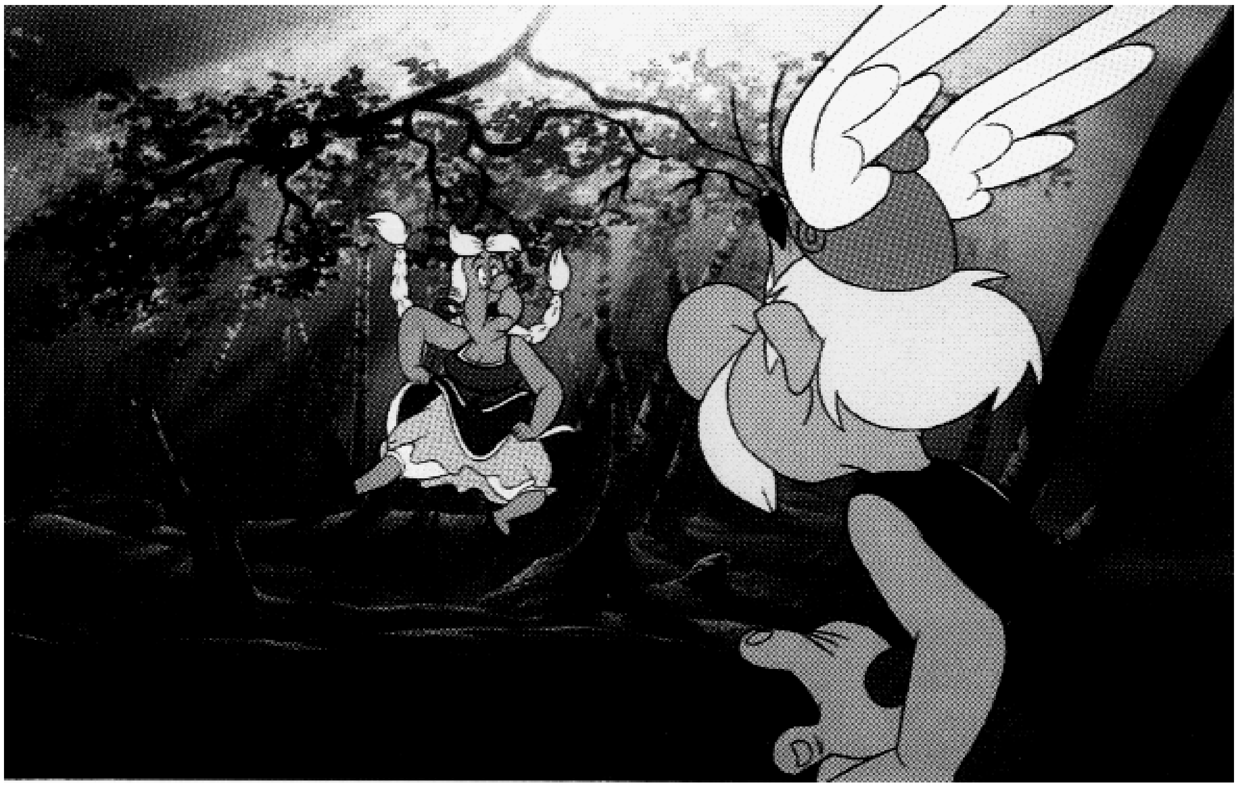


Figure 2.1: Example of a Complex Background and Simple Foreground

This film technique has migrated easily to computer graphics. Image parts are rendered separately and then recomposited, sometimes with the use of a compositing or depth comparison algorithm such as [Porter 84] or [Duff 85]. This can greatly reduce the amount of work to draw static, complex backgrounds. However, just as in cel-based animation, computer compositing requires a great deal of human intervention. Modelled objects must be segregated onto layers by hand, based upon planned motion. The separation into planes is artificial and inefficient. A partially occluded, moving object that suddenly stops must be moved by hand into the static background layer. The layering selection is based upon total scripted knowledge of the future motion. Because a human is required to decide upon the layering, compositing is almost always performed at the object level. It is very hard to determine that, given a certain object's motion, a particular component polygon (potentially one of thousands) does not change between two consecutive frames. This reduces the area of a picture that we can reuse between frames.

To recap, computer compositing has several major flaws. First, there is too much human intervention. An ideal compositing system should automatically place primitives on the correct layers based upon the absolute minimal redraw rates. Secondly, layering should not be based on objects. An algorithm that can predict correct layer placement for entire objects should also be able to predict layer placement for object components. Finally, layering should not need an entire script's worth of advance knowledge about a scene's motion. An ideal compositing system should separate object parts into layers based only on a small amount of future knowledge.

2.1.2 Overlay Buffers

Computer manufacturers have recognized the usefulness of compositing, and have tried to build in hardware and software support. Some systems support block memory compares, where an image (with depth values) is composited with another image (also with depth values). Because the scan-conversion pipeline already involves compositing (submission to the depth buffer on a primitive-by-primitive basis), compositing features can draw heavily on the existing system capabilities.

One particular type of compositing support is an overlay buffer, designed to allow an extremely simple version of compositing. Static scene elements are rendered once and "frozen" into an overlay (underlay) buffer. Subsequent moving objects are then rendered over or under the frozen layer. For example, a flight simulator program might render a complicated instrument panel into the overlay plane. As the scene changes, the program only needs to render the panel dials and needles to update the controls, without having to redraw the static dial labels.

Like multi-layer compositing, the separation into foreground and background must be done by a human with knowledge of the scene's expected motion. Usually, this means that objects do not migrate between the framebuffer and the overlay buffers. As well, depth comparisons are fixed. The overlay buffer will always obscure anything behind it. This prevents consistent occlusion between moving and static objects if the relative depths change. Most overlay planes have a limited number of bits, precluding full RGB images.

2.1.3 Binary Space Partitioning Trees and Precursors

An algorithm developed by Schumacker et al. [Schumacker 69], as described in [Sproull 74], attempts to determine a depth ordering for a scene that is primarily viewpoint independent. Central to the algorithm is the idea that a scene can be segmented into "clusters", where a cluster is a group of faces (polygons) separated by partioning planes. These planes break up the world space into regions, each potentially containing the viewpoint. The division of the world space into regions suggests a priority ordering of the clusters within the various regions. A cluster "A" on one side of a plane can obscure a cluster "B" on another side of the plane, if cluster "A" is on the same side of the partitioning plane as the eyepoint. Region separation results in a binary tree, where internal nodes represent the partitioning planes, and the leaves represent the cluster regions. Each of the leaf nodes contains a priority ordering of all clusters, where priority ordering is determined as if the viewpoint were contained within that particular leaf's region. Within a cluster, faces are assigned a priority order that determines the relative occlusion order, given any viewpoint (as shown in Figure 2.2). Priority orders cannot always be assigned, nor assigned automatically.

Figure 2.2 shows backfaced polygons removed. To actually render the image, the algorithm determines the leaf node containing the viewpoint, and draws the clusters. Clusters can be drawn in order of their priority, where a high priority cluster always obscures a lower priority cluster. Within a cluster, the highest priority face is drawn before lower priority faces.

Schumacker's initial work lead to Fuchs et al. 's binary space-partioning trees [Fuchs 79],

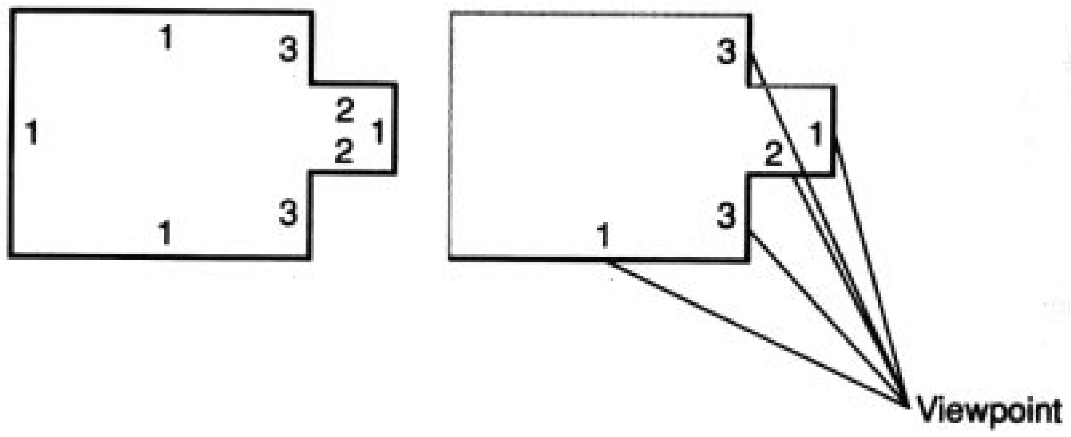


Figure 2.2: (a) Faces in a cluster. (b) Priorities of visible faces. Scanned from Foley 90

which are a generalization of the cluster ordering process. A BSP tree essentially consists of polygons that divide the world space into half-planes.

Because these algorithms perform a good deal of a priori work, they both have some applicability to temporal coherence. Once a BSP tree has been created, the viewpoint can be moved around without requiring a full depth-order recalculation. This amortizes the initial set-up work across a sequence of frames. However, BSP-type algorithms are not quite general enough for our purposes. As soon as a polygon moves, the BSP tree must be readjusted and recomputed. In a large database, this can be a costly process. Non-planar surfaces are not supported. Finally, BSP trees do not take advantage of the depth buffer for visibility computation. This is a problem on those systems that have invested significant hardware and software resources to provide fast depth buffering (i.e. almost all modern high-end graphics hardware).

2.1.4 Visibility Constraints

Hubschman and Zucker [Hubschman 81] attempted to address scan-conversion temporal coherence at a global, algorithmic level. They started by assuming a moving viewpoint circling around a static world. The world consists only of stationary, closed, convex, non-intersecting polyhedra. The goal of their work was to identify areas in the picture that changed between frames, based upon geometric analysis. Two types of constraints were important: changing self-occlusion of a single polyhedra, and changing occlusion between pairs of polyhedra. For a small change in the viewing position, one can use object edge properties to determine which polygons have appeared or disappeared. Partly visible polygons are subdivided into visible and non-visible areas. As the viewer moves around the scene, the visible polygon list is updated and sent to the graphics pipeline.

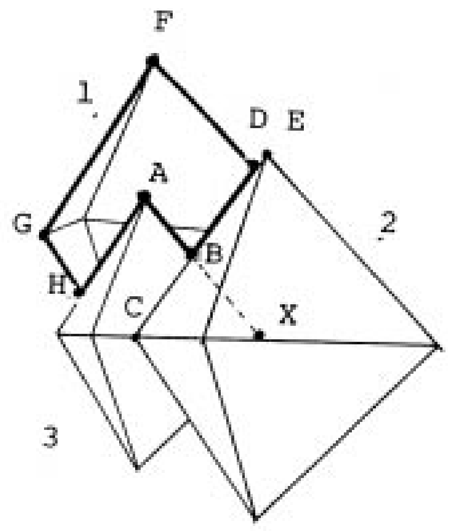


Figure 2.3: Visibility Outline of a Polyhedra, scanned from Hubschman 81

Unfortunately, the initial constraints are far too strict for most applications. Stationary objects preclude animation, while closed, convex, non-intersecting polyhedra prevent many interesting types of object modelling. Even worse, this approach cannot by combined with other types of drawing primitives, preventing a mixed-method solution for the different types of objects in the world.

In addition to the motion and object constraints, the algorithm is almost too complex to implement. It requires a significant amount of extra work to determine visibility, including the projection of edges onto polyhedra. This algorithm also does not use the depth buffer. The limited applicability of this algorithm is supported by the lack of subsequent commercial applications.

2.1.5 Hierarchical Visibility

Greene, Kass and Miller [Greene 93] describe a hierarchical visibility algorithm that uses both object space and image space coherence, with extensions to allow for temporal coherence.

Object space coherence allows a single object-space visibility computation to determine if a group of nearby local objects are visible. For example, deciding that a closed box is not visible in the scene allows one to determine that the contents of the box are also not visible.

Image space coherence allows a single visibility computation to determine if a number of local pixels are visible. By grouping pixel depth values into a hierarchical tree, a higher level depth comparison can determine the occlusion of depth values in descendents. The algorithm is in image space because it operates to the maximal precision of pixels in the frame buffer.

Greene et al. start by assigning the geometry of the scene model to cubes in an octree. The octree encompasses the entire modeling space. Individual polygons are placed in the smallest octree node that can completely contain the primitive. The algorithm starts at the root node of the octree and works recursively down through the root's children. The basic idea is to scan-convert the individual faces of an octree cube, and then compare the faces with the existing depth buffer, computed for a particular viewpoint. If all of the faces of the octree cube are obstructed, then all primitives contained within the cube are also obstructed, and there is no need to submit any contained geometry to the graphics pipeline. If the octree cube is outside of the viewing area, then the entire cube's contents can again be ignored. If some part of the cube's face is visible, then the algorithm draws any immediately enclosed primitives, and then recursively operates on the octree node's children. Figure 2.4 shows the viewing frustrum divided into octree frustrums. The clusters near the apex are areas of high octree density.

Using an octree allows large numbers of polygons to be cut out of the rendering pipeline. If a parent octree node is totally obscured, then all descendent nodes are obscured and do not need to be rendered or processed. This can lead to huge savings in the case where very complex

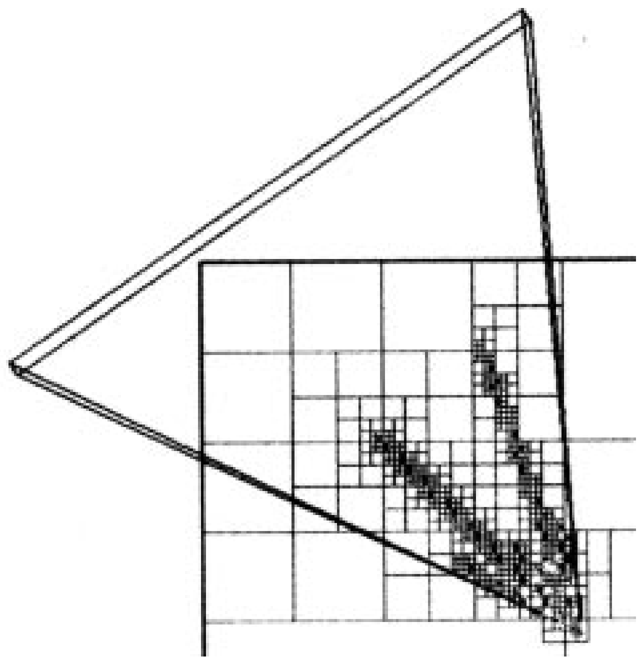


Figure 2.4: Viewing Frustrum and Octree Cubes, scanned from Greene 93

geometry is blocked out by (for example) a wall close to the viewpoint.

In order to determine if an octree cube face is completely occluded, Greene et al. use a form of hierarchical depth buffer. Starting with an initial depth buffer, the algorithm recursively combines four previous depth buffer values into a new (furthest) depth value. This forms a multi-level, multi-resolution depth buffer pyramid - essentially a quad-tree with “max” as the operator that yields a parent from four children. The root of the tree contains the furthest depth value in the entire image.

When drawing a polygon (or octree cube), the algorithm first determines the polygon’s screen bounding box. The depth pyramid is checked to find the node and level that contains enough depth buffer area to cover the entire bounding box of the polygon. If the pyramid’s depth value is closer than than the polygon’s nearest depth value, then the polygon is occluded and can be ignored. If the polygon cannot be culled, then the algorithm descends a level into the hierarchical depth buffer tree. The polygon is chopped to the corresponding subrectangles and then compared with each of the four quadrant depth values. At the limit, individual pixels in the polygon are submitted to the depth buffer hierarchically.

The octree and the hierarchical depth buffer are primarily used for culling out unnecessary polygons from a single frame. However, Greene et al. also claim their algorithm is useful across multiple frames. Temporal coherence can be performed by keeping track of the last frame's visible octree cubes. These are drawn first, with the hope that they will occlude most of the subsequently drawn octree nodes.

This method provides impressive speedups, especially for large models with a lot of depth complexity. It also nicely combines several methods of coherence. Its main strength is in drawing static images, or moving viewpoint images with static objects. However, there are a number of important disadvantages that make this algorithm less than perfect. Even though the idea is fairly simple, there is still a significant amount of extra code required, including two major data structures. The object geometry must be assigned to octrees. Because the octrees are fixed with respect to the model space, there is no guarantee that objects will be distributed nicely throughout the tree. Primitives can also be distributed inefficiently across octree boundaries. The authors use the example of a triangle crossing the root node's center; the triangle must be rendered any time any part of the entire model is visible. This algorithm also requires some additional work to render the octree faces. In the worst case, the extra work results in no savings. In any case, the code required to determine octree assignments adds additional cost and complexity. Greene et al. had trouble implementing the object-space octree, because commercially available platforms did not support (or at least quickly support) calls to determine whether a particular pixel would be visible if it were scan-converted. This made it difficult to submit polygon fragments to the hierarchical depth buffer. Eventually, the authors ended up using an unusual graphics platform, and partial software implementation.

The hierarchical depth buffer also involves some additional costs. The depth buffer tree must be constructed and maintained. Every time the depth buffer changes, depth values must be propagated down throughout the depth buffer tree. For each rendered polygon or octree face, the algorithm must determine both the screen space bounding box node, and the polygon's closest depth. The authors point out that it can be expensive to repeatedly determine the closest depth of a polygon quadrant in order to compare it with a hierarchical depth buffer quadrant. They suggest using the closest depth of the entire polygon, although this now prevents a completely determined visibility test. The polygon can be shown to be completely hidden, or it can end up as undecided, at which point it must be rendered normally.

The authors' concept of temporal coherence works much better for viewpoint changes than for object motion. With only a moving eyepoint, the objects can be statically allocated to octrees. As soon as objects start moving, they can cross octree boundaries, implying that the octree assignment algorithm must be performed for each and every frame. Greene et al. do not give any indication of the costs of this algorithm relative to the total frame drawing cost.

The algorithm first draws the visible octrees from the previous frame, in the hope that they will block out most of the subsequently drawn octrees. However, as soon as objects can move between octrees, the last frame's visible octrees have a much smaller coherence with this frame's visible octrees, especially as a very small object space motion can cause a large change in octree assignment. In fact, an object-space motion so small that it results in no change in projected screen motion can still cause a switch in octree ownership. These flaws prevent the algorithm from taking full advantage of a scene's temporal coherence.

2.2 Ray Tracing Methods

There have been a number of efforts to introduce temporal coherence to ray tracing, mainly because of the high cost per ray-traced frame. Because ray tracing and scan-conversion are fairly different paradigms, the ray tracing temporal coherence algorithms are of limited use to us. Individual ray-traced pixels are affected by secondary effects, where the motion of an off-screen object could potentially change the appearance of all on-screen pixels. Ray tracing allows image refreshing on a pixel-by-pixel basis, greatly reducing the granularity of the coherence. Scan-conversion requires submitting an entire primitive at a time to the graphics pipeline. However, these references have been included to show the relative sparsity of the temporal coherence literature.

2.2.1 Flood Fill and Reprojection

Badt [Badt 88] suggested two different temporal coherence algorithms, one based on image space sampling, and other on "reprojection". The image space algorithm starts by completely computing base frame 1, and sequentially working on adjacent frames. A particular frame k is sampled some number of times. Each sampled pixel is re-traced to compute an exact color. If the recomputed pixel color is different than the corresponding pixel in frame $k - 1$, then the author assumes that "this pixel is just one pixel from an entire region of adjacent pixels that are not the correct color". A flooding algorithm is used to re-cast adjacent rays until recast rays do not change the initial pixel color. The region of adjacent pixels also extends forwards and backwards in time, sampling the selected pixel area until the recast rays do not change between frames. This algorithm can introduce errors if the sampling rate is small. Badt did not implement his image space temporal coherence algorithm.

Badt's second temporal coherence algorithm attempts to speed up ray-traced sequences with a moving viewpoint. On frame 1, the algorithm keeps track of all first level object-space ray intersections. On subsequent frames, the algorithm reprojects the hit locations onto the new viewplane. The new image is computed by averaging the colors of all pixels that trace through the new pixel locations. This method does not guarantee that every pixel in the new frame will have an old pixel that projects to it. Nor does it account for new information at the edges. It also assumes diffuse objects that are completely static.

2.2.2 4D Bounding Volumes

Glassner [Glassner 88] attempts to utilize temporal coherence by creating 4D volumes out of 3D objects and complete knowledge of object motion along a time dimension. These 4D objects are then bounded, in the same way that traditional 3D ray tracing uses bounding volumes to reduce the number of ray intersection tests. These 4D objects are essentially boxes in both time and space. Testing against 4D volumes reduces the number of ray/volume intersection computations for an animation. Bounding in 4D space also reduces the number of bounding box computations from once per frame to once per animation.

2.2.3 Adaptive Temporal Coherence

Chapman [Chapman 90] developed a temporal coherence scheme similar to adaptive antialiasing. Chapman's algorithm renders every k^{th} frame in an animation, checking to ensure that a given pixel's color has not significantly changed since the last rendered frame. If a pixel has changed enough, then it is re-rendered using an interval of $\frac{k}{2}$ frames. The algorithm operates recursively on the two new intervals until $k = 1$. Chapman's algorithm essentially performs a binary search for the frame where a particular pixel changes color. Of course, if k is too large, then fast-moving objects will not be properly noticed or rendered. The algorithm also performs poorly in cases where a very small object motion causes a large difference in pixel colouration, such as when object is textured with a checkerboard pattern. Chapman's method also does not account for cases when a sampled pixel is actually a different object with the same color.

2.2.4 Spatio-Temporal Coherence

Chapman [Chapman 91] produced another ray-tracing algorithm designed to take advantage of temporal coherence. On the initial frame, the algorithm casts a number of primary rays. When a primary ray hits an object, the algorithm then determines the parameterized path of the ray-object intersections over some future time slice. These intersections may wander across (and off) the surface of the intersected object. This eventually leaves a group of intersection-paths for all hit objects. The list of paths is sorted, and subsequent rays are spawned.

The speedup in this method comes from only having to intersect object bounding volumes once per time sequence per ray, rather than once per ray per frame. Drawbacks include a fairly complicated method of representing and determining the intersection paths - the implementers were only able to implement a “translation only” example, where objects could translate through space, without changing orientation.

2.2.5 Voxel Trees

Jevans [Jevans 92] subdivides the object space into a voxel tree. Each voxel keeps track of all intersecting rays cast during the first frame of the animation. When an object moves in space, the enclosing voxels are tagged for update. On the next frame, only rays that passed through a tagged voxel need to be updated (including secondary or higher rays). Jevans' method requires a static camera and a good deal of voxel overhead space.

2.2.6 Frameless Rendering

Bishop [Bishop 94]et al. developed an interesting twist on temporal coherence. Instead of completely redrawing the scene on each pass, the algorithm continually redraws a random sampling of individual pixels. The number of pixels drawn depends on allocated time. When the frame rate is slow, a complete redrawing of the scene each time results in jerky "tearing", as one picture is replaced with another at a relatively low update rate. Continuously updating some of the pixels allows a smoother transition, at the cost of some erroneous blurring. This method is primarily aimed at ray tracing, where pixels are rendered individually. The authors discuss some future possibilities of this approach, including sampling the updated pixels based on object motion.

Chapter 3

Predictive Rendering

You can only predict things after they've happened.

- Eugène Ionesco

In many common animation scenarios, we notice there is a great deal of variation in the amount of motion present. Some scene objects move continually, others are stationary, and still others move at infrequent intervals. Even within a single moving scene element, there may be some areas that remain stationary with respect to the background. No matter how slow or fast the component scene motion is, traditional algorithms always render the complete scene for each and every frame. This is inefficient, as the static and slow moving parts of the scene are redrawn many more times than is absolutely necessary.

We would like to render scene parts only when they move, potentially avoiding a large amount of computational effort. In order to do this, we need a way to segregate objects into groups that can be rendered at different rates. We also need a way to recombine parts drawn at different times back into a complete, consistent image.

In this section, we introduce a new algorithm to reduce the number of polygons rendered on each frame. The algorithm *predicts* which polygons will move for some small set of future frames. These predictions are used to *render* polygons only for a subset of the total frames. The algorithm can be easily extended beyond simple polygons to other, higher levels of graphics primitive.

The main innovation behind this algorithm is the realization that modern rendering hardware is slowly reducing the cost of world-to-screen projection, relative to all the other work required to render a polygon. This realization is supported in Chapter 4, where we compare the relative costs of these operations. We can use a fast, simple world-to-screen projection to predict if and when we need to render fully transformed, clipped, lighted, depth-cued, textured, scan-converted, antialiased, depth buffered polygons. The world-to-screen projection is one of the first parts of the graphics pipeline, and is usually supported by specialized hardware.

We can conceptualize a prediction as drawing a polygon in wireframe. We can use these fast predictions to determine the set of moving polygons for any particular frame. If we look a few frames into the future, we can identify polygons that are stationary over a particular period of time. These polygons only need to be rendered once during that entire frame interval. For each final frame, the single rendered image of the static polygon set is depth composited with any of the (redrawn) moving polygon images that need to be updated at that moment. The final depth composited image contains a correctly occluded picture generated with potentially far fewer polygons sent through the graphics pipeline. If we can throw out even a few of these "full cost" polygons, we can more than pay for the additional prediction costs. Chapter 4 discusses the break-even point of the prediction method.

3.1 Algorithm

This section describes the predictive rendering algorithm. Predictive rendering is composed of three distinct phases. The *grouping phase* decides how to segregate related world objects into sets that can be tracked over time. Grouping objects allows the entire set's motion to be determined with a single prediction. The grouping phase executes once, before any other predictive rendering phase.

We then need to choose a *time slice*, consisting of a number of frames N to predict. Section 3.2.2 describes some of the tradeoffs associated with various sizes of N. During the *prediction phase*, a fast prediction test is performed on each primitive, for each of the N frames. The results of the prediction allow objects to be drawn together depending on their minimal redraw rates. After predicting N frames in advance, the *rendering phase* actually draws the N frames. Objects are rendered at different redraw rates, and depth composited together to form a final correct image. The prediction phase and the rendering phase then alternate on groups of N frames until the animation is complete.

3.1.1 Using the World Database

At the heart of any rendering system is the world database. A user creates this database, compiling objects, lights, material, cameras, textures, etc. to form a 3D model that can be subsequently drawn. The majority of the world database consists of objects, usually constructed out of a few system-supplied graphics primitives, such as spheres, cubes, ellipsoids, and objects of revolution. Objects are often given motion, which can be user-controlled, specified directly, derived via known rules, or even completely random.

We want to predict the future motion of world objects in order to determine when we can avoid unnecessarily redrawing them. Before making any guesses about motion, we must decide how to separate world objects into sets that can be tracked. We define a "prediction primitive", which is the lowest level graphics element that we wish to check for motion. An obvious first choice is the polygon. Choosing this prediction primitive implies that all world objects must be ultimately decomposable into polygons. This restriction is not particularly strenuous; many other common graphics algorithms can only handle polygons. Hardware graphics engines are usually forced to break down curved surface primitives into polygons to take advantage of specialized rendering abilities. Whatever our choice of prediction primitive, if there is a particular object that cannot be broken into primitives, then it is easy to just render it each frame. No speedups are possible for this primitive, but no extra prediction work is required either.

There are number of other possible types of prediction primitives, generally in the form of volume-bounding shapes. The simpler the prediction primitive, the greater the number of predictions, and the higher the overall prediction cost. Simultaneously, as the number of predictions increases, the more we can take advantage of factoring out small subsets of moving elements.

Prediction overhead is proportional to the number of vertices in the prediction primitive.

For example, we can use a block (with eight vertices) as our prediction primitive. To determine if we can avoid drawing all six faces, we only need to check eight vertices. If we represent the same cube with six copies of a four-vertex polygonal prediction primitive, we need to check four vertices for each of six sides. This is a three-fold increase in prediction time, which must be balanced against the ability to factor out individually moving sides.

In general, if a graphics primitive consists of parts that never move in relation to each other, it is usually faster to group them into a new prediction primitive, up until the point where the number of different types of prediction primitives becomes too complex to handle. For the purposes of this thesis, we only use three prediction primitives: the polygon, the block, and the octahedron (or double pyramid). The prediction primitives are shown in Figure 3.1.

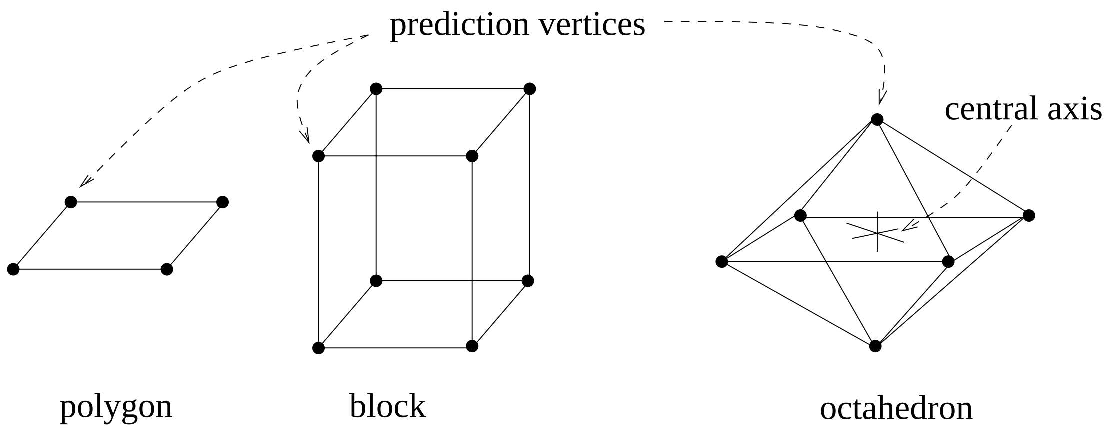


Figure 3.1: Three Types of Prediction Primitives

The block prediction primitive can be used for any six sided, eight vertex world object, such as a cube or block. The block prediction primitive and world primitive have a one-to-one vertex correspondence, i.e., each world vertex is matched with one and only one prediction vertex, and vice versa. Because the prediction and graphical vertices are the same, no extra space is required to keep track of the block prediction primitive locations.

The octahedron is essentially a simplified bounding volume placed around the object. In order for the octahedron prediction primitive to work, the axis lengths must be chosen so that the bounding pyramid completely encloses the object (i.e. the bounding octahedron must actually bound the entire object). This is necessary because prediction is based on the projected difference of the axis vertices. Without this constraint, a static pyramid vertex does not necessarily imply a static object, as shown in Figure 3.2. This ensures that under rotation, the axis vertices always have more projected motion than any internal vertex. Conversely, if the axis lengths are too large compared to object size, a very small rotation will predict object motion based on axis differences, even though the actual object moved very little. This is not erroneous, just inefficient. The octahedron's point of origin is placed on the object's axis of rotation.

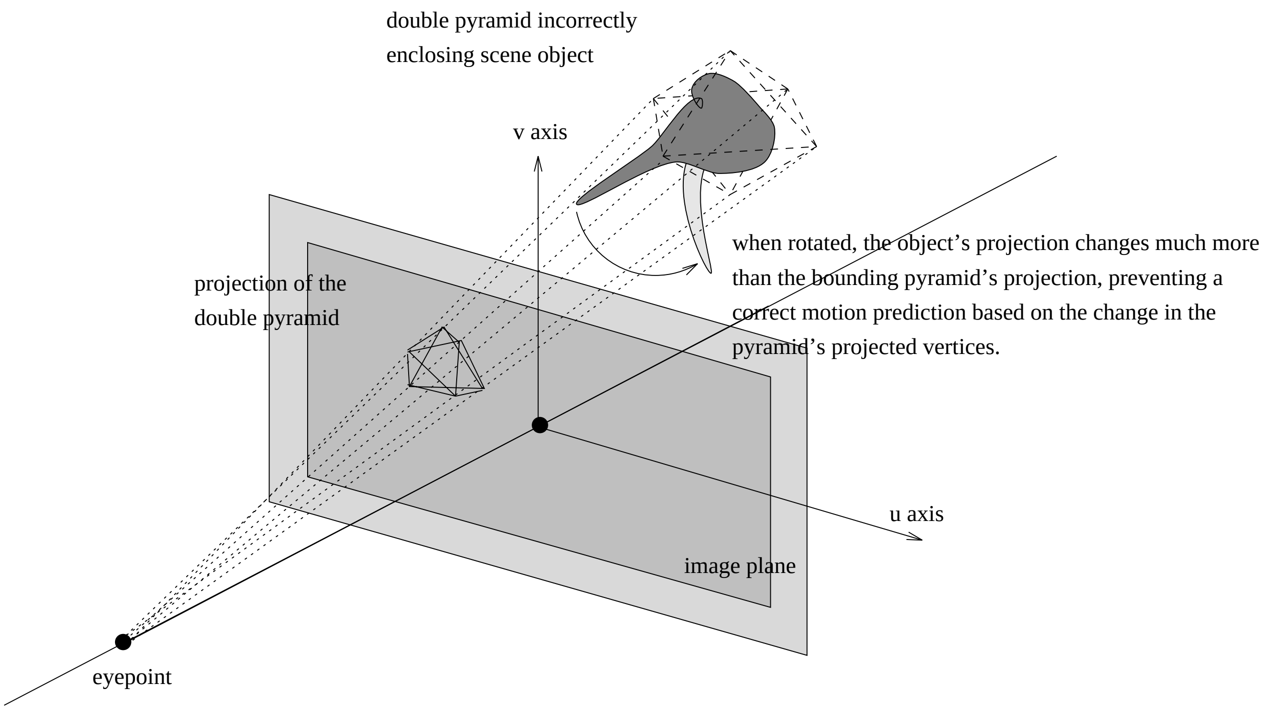

double pyramid incorrectly enclosing scene object
v axis
when rotated, the object's projection changes much more than the bounding pyramid's projection, preventing a correct motion prediction based on the change in the pyramid's projected vertices.
projection of the double pyramid
u axis
image plane
eyepoint

Figure 3.2: Prediction Pyramid with too short an Axis

The octahedron primitive can be used for any grouping of world primitives where all world primitives are guaranteed to remain constant with respect to all other world primitives in the group. Then, the entire group's motion can be predicted using six vertices, instead of the entire set of world primitives. The more complex the world primitive, the more efficient its prediction becomes.

The octahedron primitive is most useful for complex higher order primitives consisting of many, many sides, such as spheres, or snowflakes. It can reduce the prediction overhead considerably. In the case of a static sphere approximated by 400 polygons, our prediction overhead is reduced from 1600 vertex predictions to six vertex predictions. Note that polygon and block objects can also be predicted with an octahedron, but are separated out specially because they are so common. The octahedron does not necessarily need to be aligned with the object axis, nor do the horizontal and vertical dimensions need to be equal.

The octahedron uses only six vertices to bound an object, and is most efficient for objects of approximately the same X, Y, and Z dimensions. We can also use the block primitive, with prediction vertices not necessarily in a one-to-one correspondence with any enclosed world vertices. The extra two vertices require more prediction effort, but may provide a tighter geometrical bounding volume. Because the octahedron is not in a one-to-one correspondance with its bound graphical primitives, we require some extra space to keep track of the prediction vertices.

It is easy to assign world objects to prediction primitives, mostly because they are similar to the (already existing) world primitives created to make model building easier. Table 3.1 shows some example assignments.

We can be fairly haphazard assigning world primitives to prediction primitives, as long as prediction and world vertices are the same. If, for example, we assign a block prediction primitive to a world cube where one side continually moves, we lose some potential speedup (factoring out the non-moving sides), but we do not introduce any errors. The only place where errors can actually be introduced is when the world primitives do not have a one-to-one vertex correspondence with the prediction primitive. The octahedron is the only one of our chosen prediction primitives to do this. If the any of the enclosed vertices move relative to each other while the prediction vertices remain stationary, prediction will (erroneously) decide the object remains static. Thus, it is very important to only bound graphical primitives that will always

Graphical Primitive	**..that is static**	**..that might change size or shape between frames**
point	polygon	polygon
line	polygon	polygon
polygon	polygon ($\leq$ 6 vertices) or octahedron (>6 vertices)	polygon
pyramid	octahedron	many polygons
cube	block or octahedron	block
block	block or octahedron	block
sphere	octahedron	many polygons
torus	octahedron	many polygons
polygon mesh	octahedron	many polygons
hydrogen molecule	octahedron	many polygons

Table 3.1: Example Assignment of Prediction Primitives

remain stationary with respect to each other.

It is possible to separate moving and non-moving parts of the object into different prediction primitives, but we hesitate to do this because it removes some of the prediction effort from the computer and places it on the human. Assigning prediction primitives should be as easy and automatic as possible, taking advantage of static, non-moving groups of polygons only when it is obvious and easy to do so. The whole point of this algorithm is to determine motion automatically. World primitives should not be reassigned new prediction primitives during execution.

Note that any prediction primitive with a one-to-one vertex correspondence can be changed in shape between single steps of a prediction range, and still remain correct. This allows correct prediction even when polygons are mutating. We assume that a one-to-one prediction primitive maintains the same number of vertices throughout. Prediction primitives without a one-to-one correspondence to graphical vertices can only be changed between prediction ranges.

Of course, the most conceptually straightforward way is to assign every world polygon to a prediction primitive polygon. This avoids having to do any work to compute bounding axis lengths. This also may be the only answer if prediction eventually extends to changing material properties, such as a cube with a live video image texture mapped onto each side.

In later sections, we will examine expected and experimental performance increases assuming that we only use the polygon prediction primitive. We show that we can gain speedups even using the simplest type of prediction primitive. We will also compare a few specific examples of more complex prediction primitives.

3.1.2 Prediction Phase

In the prediction phase, the algorithm looks up to N frames into the future. This allows it to decide how often a particular prediction primitive moves. The prediction method is essentially the same for all prediction primitives over all frames.

We begin by assuming we are doing a prediction for a frame f. The algorithm starts by traversing the database by its natural ordering (linear or hierarchical, depending on the ordering of the model), forecasting future motion for each prediction primitive. For any given prediction primitive, we check each of the vertices in fixed order, transforming from object coordinate space into screen space. This transformation consists of several steps. The prediction primitive starts off in object space, where all vertices of the primitive are described relative to a local origin. The primitive vertices must be transformed into the world coordinate system, where all vertices are described relative to a single global origin. We can achieve this by multiplying all vertices by an object-to-world transformation matrix $M_{O \mapsto W}$ as such:

$$M_{O \mapsto W} = \begin{vmatrix} r_{1x} & r_{2x} & r_{3x} & t_x \\ r_{1y} & r_{2y} & r_{3y} & t_y \\ r_{1z} & r_{2z} & r_{3z} & t_z \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{3.1}$$

In a linear traversal of the database, elements r and t describe the rotational and translational motion of the primitive with respect to the world coordinate system. If the database is traversed hierarchically, r and t will instead describe primitive motion relative to the previous coordinate system. $M_{O \mapsto W}$ must be concatenated with the the previous level's $M_{O \mapsto W}$ to arrive at the true world coordinate points. These transformation are all the same (with the omission of a number of steps) as the object-to-screen transformations performed by most graphics pipelines, and are described in greater detail in any introductory textbook on computer graphics [Foley 90] [Hill 90].

Once in the world system, the coordinates must be transformed into the eye coordinate system, where the eyepoint is not restricted to the z axis. A common world-to-eyepoint viewing matrix $M_{W \mapsto E}$ might look like:

$$M_{W \mapsto E} = Rot_z(-azimuth)Rot_x(-inclination)Rot_z(-twist)Trans(0, 0, -distance) \tag{3.2}$$

for a (polar coordinate) eyepoint located $distance$ from the origin, at $(azimuth, inclination)$. Once in the eye space, coordinates must then be projected onto the projection plane. Projection is achieved by multiplying all eye-coordinate vertices by an eye-to-projection plane matrix $M_{E \mapsto P}$ as such:

$$M_{E \mapsto P} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{vmatrix} \tag{3.3}$$

where d is the projection plane normal to the z axis at $z = d$. The x and y components of a homogeneous point must be divided by the 4^{th} vertex element w. Note that there is no need to transform into normalized clipping space, as would normally be done by the graphics pipeline.

Conceptually, we would like to plot the vertices into screen space, by scaling by the window size and number of window pixels. These predicted screen coordinates give the location of a vertex if we were actually bothering to draw completed polygons. However, we can skip a few operations by keeping our vertices on an imaginary projection plane at $z = d$. This avoids having to multiply transformed pixels by a projection plane-to-screen conversion factor, which is a constant over all frames. Note that we have not clipped points anywhere, so that there may be some number of vertices that lie outside the screen space area.

We can reduce the amount of work required for prediction by concatenating the $M_{W \mapsto E}$

and $M_{E \mapsto P}$ matrices into a general transformation matrix $M_{3D \mapsto 2D}$ where

$$M_{3D \mapsto 2D} = M_{W \mapsto E} M_{E \mapsto P} \tag{3.4}$$

Concatenating the two transformation matrices into a single matrix reduces the number of matrix-vector multiplies needed when predicting individual primitive vertices. Because the viewing and projection parameters are constant for a given moment, the concatenation only needs to be performed once per frame. Thus, the predictive coordinates (x, y) of the m^{th} vertex $\vec{V'_m}(p, f)$ of a particular prediction primitive p for frame f are $\vec{V_m}(p, f)$

$$V_m(\vec{p}, f) = \vec{V'_m}(p, f) M_{O \mapsto W} M_{3D \mapsto 2D} \tag{3.5}$$

where

$$x = \frac{\vec{V}_{m,x}(p, f)}{\vec{V}_{m,w}(p, f)} \tag{3.6}$$

and

$$y = \frac{\vec{V}_{m,y}(p, f)}{\vec{V}_{m,w}(p, f)} \tag{3.7}$$

We have described the minimal number of steps required to predict a vertex. Note that many of the matrices described above are already maintained and updated by the graphics pipeline. We only really need to modify the motion matrix $M_{O \mapsto W}$. In Chapter 4, we walk through the viewing pipelines, comparing the number of operations required for the prediction phase and the actual rendering phase. Chapter 4 points out the actual rendering steps that can be avoided during the prediction.

Predicting an individual vertex gives the location of a point in viewing space, at frame f. By looking at how all vertices of a predictive primitive change over time, we can determine if the primitive changes. For any given frame f, a primitive p has one of two states: "static", or "changing", represented by 0 and 1 respectively. A static primitive is defined as one where the difference between all m-component predicted vertices $\vec{V_m}(p, f)$ and $\vec{V_m}(p, f-1)$ are within a specific constant tolerance. We can determine the motion state $S(p, f)$ for an entire prediction primitive p on frame f as such:

$$S(p,f) = \left\{ \begin{array}{ll} 1 & \text{if } (f = 0) \\ & \text{or } \left|\vec{V}_{1,x}(p,f) - \vec{V}_{1,x}(p,f-1)\right| > \delta \\ & \text{or } \left|\vec{V}_{1,y}(p,f) - \vec{V}_{1,y}(p,f-1)\right| > \delta \\ & \text{or } \left|\vec{V}_{2,x}(p,f) - \vec{V}_{2,x}(p,f-1)\right| > \delta \\ & \text{or } \left|\vec{V}_{2,y}(p,f) - \vec{V}_{2,y}(p,f-1)\right| > \delta, etc\ldots) \\ 0 & \text{otherwise} \end{array} \right\} \quad (3.8)$$

Section 3.2.3 discusses an appropriate size for δ, and how it affects the prediction. For the moment, we assume that δ is equivalent to some constant subpixel distance (in the appropriate viewplane, not pixel, units). Practically, we do not need to store the predicted coordinates $\vec{V}_m(p,f)$ for each vertex m over all N. Instead, we keep a single $\vec{V}_m(p,last)$ for frame $f-1$, and use the current $\vec{V}_m(p,f)$ to compute the motion state $S(p,f)$. The value $\vec{V}_m(p,last)$ only needs to be updated when $S(p,f)=1$ (i.e., changing). All vertices of a primitive are updated at the same time. Frames are predicted in increasing order. Even though we can determine on the first vertex if a primitive has changed, we must still predict the motion of all other vertices, for use on the next frame. In the absolute worst case, we need a $\vec{V}_m(p,last)$ for each world vertex, for both x and y components. In better cases, we need a $\vec{V}_m(p,last)$ for each prediction primitive vertex, where the number of predictive vertices $<<$ the number of world vertices.

3.1.3 Clipping

In the prediction scheme described above, we do not do any clipping. Primitives are assigned as static or changing, even if some or all of their components are outside the screen window. At the cost of a little extra work, we can clip out some types of prediction primitives by assigning them a static value if moving vertices are outside the viewing window. This prevents moving (but unseen) primitives from being sent to the graphics pipeline. However, some care must be taken to ensure that clipped prediction primitives truly do not affect the image. First, all vertices must be outside the viewing window. If a single vertex is inside the viewing window, a moving (outside the window) vertex will change the image. Secondly, all vertices must be outside the viewing window on the same side of any of the viewing window boundaries. This ensures that no part of the projected polygon crosses into the viewing window. Figure 3.3 shows two examples where all prediction primitive vertices are outside the viewing window, but are not on the same side of a window edge. When any vertex moves, it affects the image, and cannot be clipped.

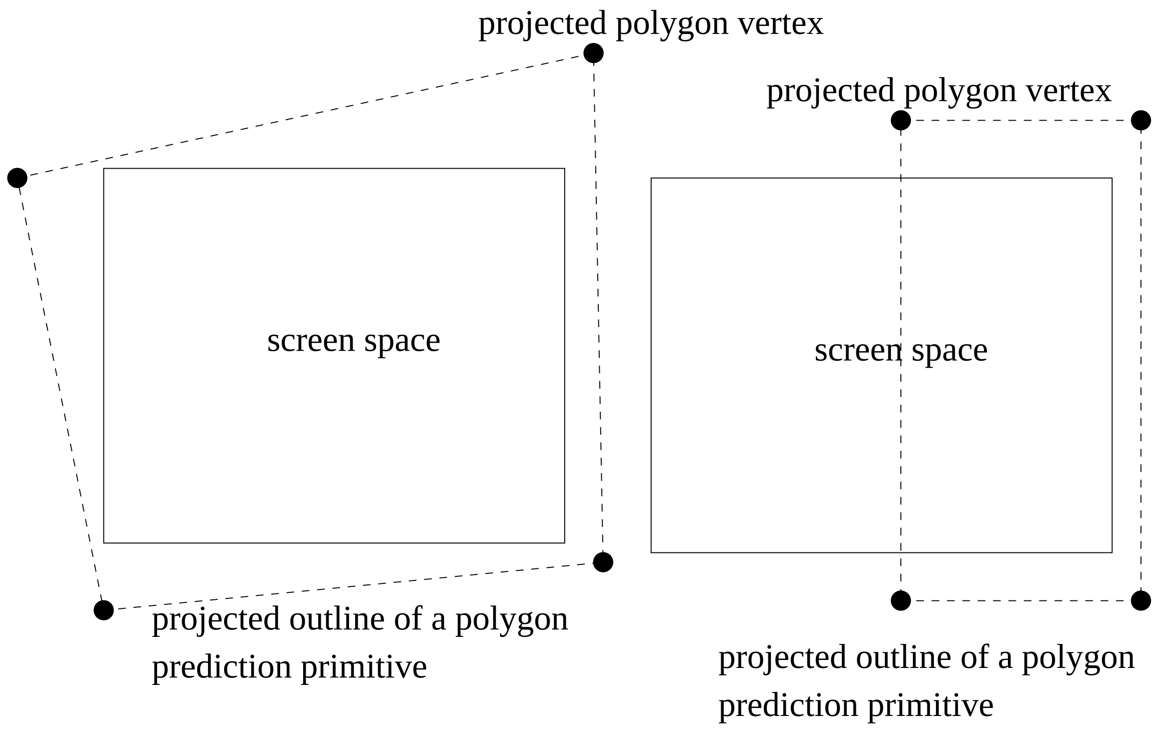

projected polygon vertex
projected polygon vertex
screen space
screen space
projected outline of a polygon prediction primitive
projected outline of a polygon prediction primitive

Figure 3.3: Examples of Incorrect Prediction Primitive Clipping

Clipping ensures that primitives outside the window will be put through the rendering pipeline once, and then not again until at least some part of the primitive wanders into the screen window. We save the rendering pipeline work required to transform the real primitive, and clip it to the viewing volume, for each of the frames the prediction primitive remains outside the window. We can clip out primitives of the octahedron type (i.e. not in a one-to-one correspondance), because we know that the projection of a bounding pyramid will contain the projection of its contents.

It is not totally necessary to clip out the primitives, but it does save future work.

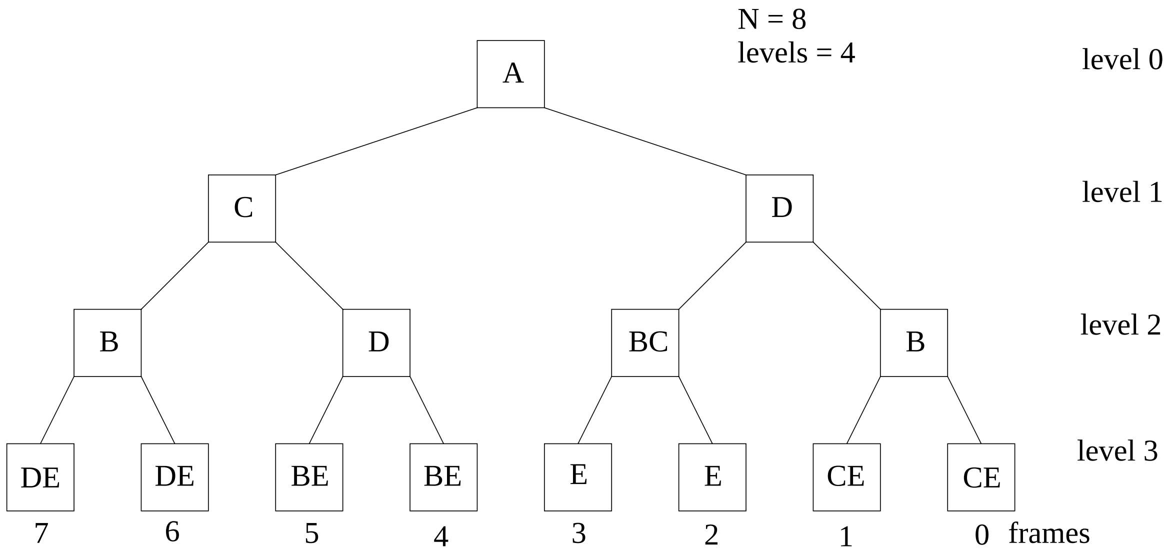
3.1.4 The Plan Tree

After a single prediction pass through the database, we are left with a complete set of states $S(p)$ and predicted positions $\vec{V}(p, last)$ for every prediction primitive. We continue to traverse the database for each frame $[0 \ldots N-1]$ within our future prediction range. At the end of our prediction phase, we are left with a fully computed $S(p, f)$ function, which remembers if primitive p was changing or static on frame f. Practically, each element $S(p)$ can be described as an N-bit data word. The f^{th} bit from the right is set to 0 if there is no change between frame f and frame $f-1$, i.e, $S(p, f) =$ static. If $S(p, f) =$ changing, then bit position f is set to 1. Bit position 0 is always 1. This is exactly a right-to-left concatenation of all N frame states.

We next need to group primitives together, in order to factor them out from some of the frame-by-frame redraws. Our approach will be to construct a binary *plan tree*. The plan tree has $log_2(N)+1$ levels, and N leaves, where each leaf represents a frame from $[0 \ldots N-1]$. Each node contains a record of prediction primitives. A prediction primitive is contained within the lowest level (closest to the root) node l if it remains static for all descendant nodes of l. This means that a prediction primitive p may be contained in multiple nodes in the tree.

This tree provides a plan to factor out primitives moving at similar speeds (as projected on the viewplane). Primitives contained in level 0 do not move during the N frames, and only need to be rendered once (on frame 0). Primitives contained in either of the two level 1 nodes remain static for $\frac{N}{2}$ frames. They need to be rendered on frame 0 and frame $\frac{N}{2}$. Primitives on level $log_2(N)$ must be rendered each frame. Figure 3.4 shows how some example prediction primitives are assigned to nodes in the plan tree. Note that any depth-first traversal of the tree will pass through nodes containing a complete set of a frame's primitives - thus all elements are drawn each frame.

It is important to note that we do not need to construct an actual tree in memory. An individual prediction primitive's location in our conceptual tree can be determined solely by examining the single N-bit word that keeps track of $S(p)$. The idea of a tree is useful because



Prediction Primitive	Prediction	Times Rendered
A	0000 0001	1
B	0110 0101	5
C	0001 0011	4
D	1101 0001	4
E	1111 1111	8

Figure 3.4: Assigning Prediction Primitives to the Plan Tree

we can think of nodes in the tree as the eventual contents of intermediate scratch framebuffers.

To determine the set of primitives that must be rendered on each frame, we traverse the conceptual tree in depth-first order, where each leaf represents a frame. The first time a node is encountered, all contained primitives are rendered into a scratch framebuffer representing a level l. Any subsequent traversals through that node do not cause anything to be drawn. Section 3.1.6 discusses how primitives rendered at different speeds can be combined to produce the correctly occluded result for each and every frame.

We use binary plan trees because they are simple and easy to implement. However, there are many other possible data structure choices for plan tree organization. The most notable of the types of plan trees are k-trees, where $k > 2$. A k-tree has N leaves and $log_k(N) + 1$ levels. Given the same number of leaves, a k-tree with $k \geq 2$ will be flatter than a binary tree, with fewer intermediate nodes. We can use this flatness to our advantage. In some applications, the vast majority of primitives will be separated into two main categories - moving all the time, and static all the time. The root node and the leaf nodes will all contain many prediction primitives, but the internal nodes will be mostly empty. An empty, or nearly empty internal node is inefficient. We are depth compositing an entire framebuffer containing very little actual data. Most importantly, we are using memory for a nearly empty framebuffer. With that extra memory, we could instead expand our prediction range N, and gain potential speedups by factoring out static polygons over a longer time. Figure 3.5 compares $k = 2$ and $k = 4$ for the same fixed N. As k increases, the ratio of leaf to internal nodes increases, reducing depth compositing overhead and factoring ability. A larger k also reduces the sensitivity of changing N, as each slice must be a power of k.

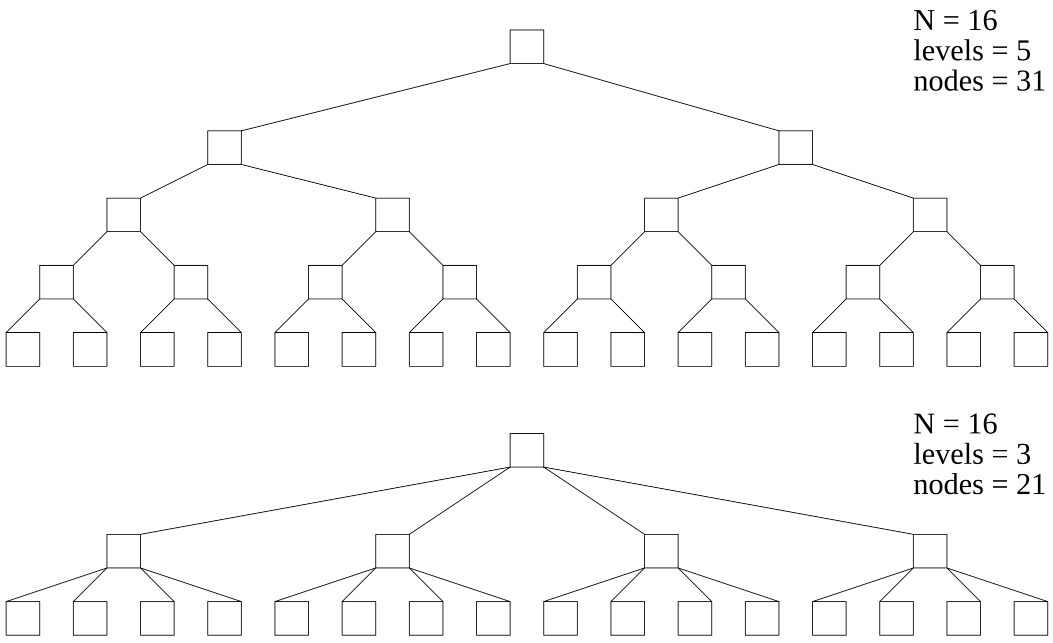


Figure 3.5: Comparison of k-trees: k = 2, 4 with N = 16

In systems with very little frame buffer space, we can increase the order of the plan tree to reduce the amount of scratch space needed. At the limit, choosing a k-tree of size $k = N$ reduces the number of framebuffers needed to two. One framebuffer is used for all primitives that move over any of the N frames, while the other is used for all primitives that are static over the N frames. Considering that one of the framebuffers can be on-screen, we only need one off-screen framebuffer, which is clearly achievable by almost any graphics system of note. Note that this is not similar to an overlay buffer, because both components are correctly and consistantly depth composited.

Of course, by reducing the number of internal nodes, we reduce the opportunity for factoring out slow moving polygons. However, depending on the makeup of the motion, we can improve things by looking a longer distance ahead. In Chapter 4, we experiment with the type of plan tree in order to see how the speed is affected.

One of the problems with fixed k-trees is that they do not take advantage of any natural grouping of the data. For example, a primitive that changes on frames 0, 1, 3, 5 and 7 will be drawn on every frame, as shown in Figure 3.6.

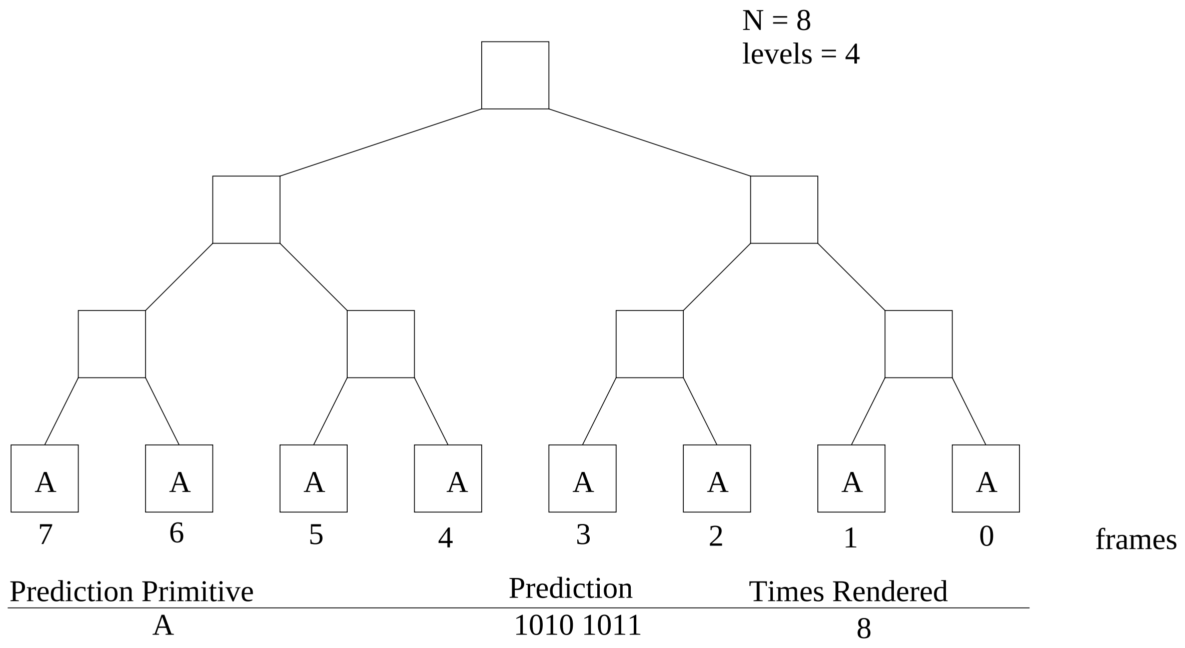


Figure 3.6: Inefficient Allocation to a Binary Plan Tree

One could use a more complicated tree that naturally groups primitives. We avoid this approach because the fixed tree is easy to understand and implement (seeing as no tree is actually required).

3.1.5 The Rendering Phase

After completing the prediction phase for a set of N frames, we move into the rendering phase. Starting at frame 0, we traverse the database looking at $S(p)$ values for each prediction primitive p. The current frame and the $S(p)$ indicate which level the primitive should be drawn on. We can simultaneously determine how many frames we can skip before we have to draw this primitive again. Figure 3.7 gives the decision algorithm. This algorithm handles $S(p)$ equivalences to maximize reuse in the state tree. For example, $S(p) = 10000001$ is treated as if it were $S(p) = 11010001$, factoring out an additional frame at positions 4 and 5.

If the primitive needs to be drawn, we move it to the new location and then draw it into the framebuffer representing level l. If primitive does not need to be drawn, it avoids the rendering pipeline altogether. However, if the primitive is part of a hierarchical structure, we may need to move the primitive so that a subsequent lower primitive (that is drawn) is in the correct place. We can move a non-drawn primitive by manipulating the transformation matrices. The flatter the hierarchical structure, the less overhead required. This is true for predictive rendering in specific, and graphics rendering in general. After depth compositing a single frame, all of the $log_k(N) + 1$ frame buffers are depth composited together to form the result, which can be then displayed.

3.1.6 Compositing

During the prediction phase, we were able to determine future motion of primitives, and subsequently group them into groups of approximately the same projected (apparent) motion. We can draw these groups at different frame rates. However, in order to produce a correctly occluded picture on each frame, we have to combine all of the groups. Every primitive, moving or not, must be included in each occlusion calculation. We do this with a straight depth composite of the multiple framebuffers, using each framebuffer's depth buffer for depth comparison. Compositing provides the proper occlusion for all primitives, on all frames.

This is essentially the same method as used by the graphics pipeline. The graphics pipeline submits rendered graphics primitives one at a time to the depth buffer. Now, each framebuffer

```
int Next_Location(relative_frame, states, *next_frame)
{ /* predict_levels calculated elsewhere */
  if (relative_frame == 0) and (states == 1)
     { /* we test specifically for the case where a p.p. is static for the entire
       frame. These p.p. are drawn on level 0, and skipped for the next N frames. */
     next_frame = N;
     return 0;
     }
  /* we loop through each of the levels we are currently using and check for the
     00000001, 0001, 01, etc. pattern in the right places. */
  for (level = 1; level < predict_levels - 1; level++)
     { /* we need to compare N/2 bit places, then N/4 bit places, then N/8, etc.
          as we descend the levels looking for a match. */
     mask = (0x1 << relative_frame);
     num_bits = N / (0x1 << level);
     not_found_yet = FALSE;
     /* look at an N/2 level if frame is evenly divisible by N/2 (etc) */
     if (relative_frame % num_bits == 0)
        { /* we don't care if the 1st position is a 1 or not - still check to see
             if the rest of the bit positions are 0 */
        for (next_bits = 1; next_bits < num_bits; next_bits++)
            {
            mask = mask << 1;
            if (mask & states) == mask
                { /* we found a 1 where there should be 0. Not this level */
                not_found_yet = TRUE;
                break;
                }
             } /* for */
         /* if we've only found zeros, then it's on this level */
         if (not_found_yet == FALSE)
            {
            *next_frame = relative_frame + num_bits;
             return level;
            }
        } /* if  */
     } /* for level */
   *next_frame = relative_frame + 1; /* no levels matched, goes on the lowest level */
   return predict_levels - 1;
}
```

Figure 3.7: Determining When Next to Draw a Prediction Primitive

can be thought of as a single rendered primitive. Compositing speed is proportional to image precision, and not object precision. It is essentially a straight memory draw from a level m framebuffer into a level $m+1$ framebuffer. Many graphics systems include specific library commands for depth buffer copy-and-compares, including correct handling of any α values. If the leaf framebuffer is the display buffer, we can avoid moving the final depth composited result to a new location.

The scratch framebuffers and depth buffers must be cleared appropriately during the rendering phases. Because some scratch buffers maintain the same picture across several frames, they cannot be cleared on each and every frame. Instead, the number of frame resets is a function of how often a frame is redrawn completely. For a particular time slice, the level 0 buffer will be cleared once, the level 1 buffer will be cleared twice and the level l buffer will be cleared 2^l (or k^l) times. It is also possible to "clean when needed", where framebuffers are cleared at the appropriate times only if they contain some previous drawing.

3.2 Algorithm Considerations

This section discusses some of the additional considerations raised by this algorithm.

3.2.1 Extra Framebuffers

This algorithm uses intermediate scratch framebuffers to hold "factored out" polygons. This extra space puts an implicit limit on the size of N. We can use k-trees to flatten out the plan tree, reducing the number of framebuffers for a given N considerably.

It might be possible to factor out even more coherence (and framebuffers) by using structures more involved than binary trees. For example, we could try to dynamically assign scratch framebuffers to frame intervals based upon the distribution of static predictions. The frame intervals would be chosen based upon the maximal number of polygon-frames that could be factored out for any particular interval.

3.2.2 Prediction Range

The algorithm requires advance knowledge of N future frames in order to predict polygon movement. In many applications, this is not a problem. All motion is known a priori, or can be determined using a simple set of repeatable rules. However, it can be argued that knowing polygon motion N frames in advance is a heavy restriction for animations involving user interaction. In practice, we are helped by the rule that users rarely interact on each frame. We can use the analogy of a car: the driver intervenes every so often, but otherwise the car travels according to its velocity and acceleration. In many cases, the system smoothly extrapolates motion from the last user request. Motion prediction is fairly common in applications with frequent user interaction, such as head tracking for virtual reality [Deering 92]. Besides linear extrapolation, there are higher order filters (such as Kalman filters [Friedmann 92]) that provide probabilistic predictions of future motion.

Predictions are done N frames in advance, but frames are drawn one by one. If the user interrupts and demands an unexpected motion change, we can instantly respond by drawing an entire frame. Thus, the user never gets a frame that does not match the user's input.

N does not need to remain constant over the entire animation. The prediction phase can be changed dynamically according to the frequency of user input. As user interaction increases, we reduce N. This means less predictive work is lost if the user interrupts. As user interaction decreases, we can increase N. The longer the prediction phase, the greater chance there is to exploit polygon coherence over time. This is especially true for objects that remain static over the total animation.

In scenarios without user interaction, we can theoretically choose N to encompass all of the frames. A large N increases coherence and reduces overhead. Practically, we are limited by the amount of framebuffer space available. This depends on window size and system memory. We expect N values to range between [2...64] for 30 frame/sec ($k = 2$) animation. In cases with user interaction, these sizes also minimize the amount of interrupted and lost predictive work.

However, it is not always the case that a larger N factors out more frames. The $S(p)$ for N_1 and N_2 can potentially be different, because the algorithm automatically assumes a redraw at a frame boundary, i.e., every N_1 or N_2 frames. Because predicted movements are computed relative to the last update, motion patterns for moving prediction primitives can be different for different N, and thus updates can also be different. This only applies to moving objects - for stationary objects, a larger N always means less work.

We can also split the object database into objects controlled by the user, and objects not controlled by user. That way, we only do predictive work on objects we know have a well behaved motion. Of course, we only get speedups on those objects, too.

It is conceivable to consider carrying prediction along between time slices. This provides us with more opportunities to factor out static and slow moving objects. If the time slice is small (i.e., $N = 2$), the static background will be still rendered a large number of times, although half as many as before.

3.2.3 Motion Thresholds

There has to be a certain amount of change in the bounding polygon before we consider rendering it again. We can compute a *motion threshold*, denoted by δ and measured in terms of screen pixels. This forms a measure of noticeable change. Practically, we convert the pixel value into a viewplane value in order to do comparisons. Doing thresholding in the viewplane prevents having to continually transform view coordinates into pixel coordinates in order to compare with a pixel threshold.

In general, we set the movement threshold to subpixel values in order to catch all but the tiniest movements. However, the movement threshold can be increased if a certain amount of jumpiness can be tolerated. Thresholds of a pixel or two reduce the rendering work yet still provide reasonably smooth movement.

The movement threshold can be varied between predictive slices, between individual frames, and even between polygons. For example, if only the eyepoint is changed (in a VR application) we could threshold movement depending on the velocity of the change.

Note that the size of δ also somewhat affects the lighting and shading of the scene. Any time a prediction primitive moves, it is affected by the lighting model, and may change in appearance. The change in shading may be insignificant or quite noticeable, depending on the size of the move, the lighting model, the viewing angle, and the type of primitive. With $\delta = 0$, objects that move even the tiniest bit are redrawn, reproducing the lighting correctly. With $\delta > 0$, some slightly moving objects are treated as static. This only approximates the exact object motion and shading, and does not return a completely accurate result. For example, if $\delta < \frac{1}{2}$ pixel, the final position of each projected polygon may be off by up to $\frac{1}{2}$ of a pixel, while the shading will by off by a similar amount. Polygon shading errors are harder to quantify, because lighting changes depend on both world coordinates (to compute the appropriate directions) and on screen coordinates (to interpolate between vertices). It has been our experience that any $\delta > 0$ that nicely approximates motion also avoids any noticeable lighting errors, although this is by no means guaranteed.

If the scene lights or object material properties change, we have to mark all affected polygons as moving. Even though a polygon might remain fixed in space, it could have changed color and thus require re-rendering. For the moment, we will pass over the idea of predictive rendering for lighting changes. We can also simply turn predictive rendering off while the user edits the lighting properties.

3.2.4 Using an Accumulation Buffer

This algorithm is especially useful for doing motion blur with the accumulation buffer [Haeberli 90]. The accumulation buffer averages the results of a sequence of individual frames containing object motion to come up with a single motion-blurred frame. We need to average enough frames so that fastest moving object does not appear "strobed". This means that slower or non-moving objects in the scene are rendered many times more than they need to be. This problem becomes worse the greater the speed differential between the slowest and fastest objects.

With predictive rendering, we only need to render objects proportional to their projected speeds. Faster objects are rendered on each frame, while slower objects might be rendered in only a few frames, or just once.

3.2.5 Prediction Accuracy

Predictive rendering depends upon a fast, accurate prediction of future polygon motion. The algorithm's basic accuracy depends on closely following the steps of the graphics pipeline. We know what a polygon will look like in the future, because we have actually completed the steps to draw it. In this respect, there is no real chance to the prediction.

Prediction depends on knowing future motion. If objects move differently that we expect, we can immediately render frames normally. This is a basic rule for predictive rendering: any time some unexpected change occurs, revert to regular drawing. This is why the worst case cost of predictive rendering is so important. We want to be able to continually perform the prediction (and then throw it away) without much wasted computation.

However, there are several important cases where a prediction cannot be considered accurate. The first (and most obvious) case occurs anytime $\delta > 0$. As δ becomes larger, object motion becomes jerkier. Objects appear stationary for longer than they actually are, and then are suddenly updated. If the worst case ($\delta = 0$) is not computationally cheap enough to still get speedups, then the user is free to twiddle δ to reach a balance of more speed vs. visual error. Note that a $\delta > 0$ will introduce errors in both shape and lighting, as slightly moving polygon will have slightly varying lighting.

The second main case occurs when a user changes any of the properties used to shade and light the polygon. This includes the viewer position, textures, material properties, lighting intensities and light positions. For example, if all objects remain stationary, the motion of a single light will still affect scene appearance, and will force a complete redraw. If the appearance of a polygon is changed during a long stationary period, the new polygon will not be reflected until the object moves, or the time slice ends. This may be adequate if the changes are small, or the time slice is short.

Prediction primitives cannot change shape or content during a time slice, unless this is carefully handled by the prediction code (which will return a non-static result).

3.2.6 Secondary Effects

Compositing methods do not work well when an individual polygon's appearance is affected by other surfaces, i.e., reflections or shadows. This is not a problem specific to predictive rendering. In fact, the general purpose scan-line conversion rendering algorithm also has this defect, mainly because individual primitives are submitted in order to the depth buffer (essentially depth compositing). However, the algorithms designed to address this flaw can usually be extended to work with predictive rendering. Because predictive rendering produces a final image and depth buffer, it does not interfere with any other image precision algorithm that post-processes the image. Algorithms that work on a primitive-by-primitive basis can be extended so that scratch buffers contain intermediate work, which is then joined to form a final result.

Several algorithms have been proposed to add shadows [Appel 68] [Williams 78] [Crow 77]. We look at Williams' [Williams 78] [Reeves 87] shadow buffer algorithm to see how predictive rendering can improve shadow generation. The shadow buffer algorithm uses the final depth buffer and image, so at the very least, shadows can be added using only the final result. If desired, predictive rendering can be used on a per-light basis to create the light's view. This allows for shadow calculation speedups, at the cost of a number of frame buffers proportional to the number of lights. The speedups will not likely be as great, because the light views do not need as much detail as the main view (i.e., textures and all the lighting can be avoided) The number of extra framebuffers required for the lights can be reduced by using k-trees (see Section 3.1.4). In addition to reducing the number of framebuffers, this nicely separates the resultant light view into static and moving components. One sees the potential for an incremental version of Williams' algorithm, where the shadowmap is only recomputed at locations where something has moved in the main viewpoint. If something moves in the light's viewpoint, one could follow pointers to any affected main viewpoint pixels (computed during the first pass), and update them.

Another example is the shadow volume method, first described by Crow [Crow 77], and later modified by others [Bergeron 86] [Chin 89]. Individual polygons are given shadow volumes, that bound the volume of space blocked from a particular light. The shadowing of a particular polygon is a function of the (signed) number of shadows between the eye and a point in space. If we compute shadow volumes for a particular scratch buffer, we can create a master union of all volumes when we depth composite to form the final image. The union of the shadow volumes can then be used to post-process and shadow the image.

3.2.7 Related Hardware

Predictive rendering can be extended to several types of specialized hardware, especially parallel image composition machines such as the PixelFlow [Molnar 92] architecture and Shaw et al. 's [Shaw 89] z-buffer composition hardware. These types of machines split groups of image primitives (from the same frame) among parallel rendering pipelines, compositing the results to form a single image. This type of image composition architecture could be modified to distribute and composite primitives in both time and image-space.

On a smaller scale, we can place prediction and rendering tasks on different CPUs. This reduces lost effort if predictions are thrown away. An ideal predictive rendering machine would contain a separate prediction pipeline, consisting of the first few vertex transformation steps.

3.3 Variations on Predictive Rendering

This section describes three variations on the basic predictive rendering method. These methods can provide further speedups under certain conditions.

3.3.1 Predictive Rendering with On-the-Fly Depth Compositing

We have previously described a method of depth compositing where all of the $log_k(N) + 1$ frame buffers were combined each frame, essentially by a depth-first traversal of the plan tree. We chose this as our method of compositing because it is conceptually easy to understand. However, we can use a different compositing method to reduce the number of depth composites over N frames.

The basic concept is to realize that as soon as any intermediate scratch buffer is completely rendered, it can be composited with all parental scratch buffers. Practically, we would composite an intermediate node's contents with its immediate parent (which in turn has been previously composited with its own parent higher up the tree).

This organization means that individual scratch buffers contain duplicate copies of the same polygon. A level 0 buffer (the root) contains all polygons that never move over all N. A level 1 buffer contains all static polygons, and all polygons that are static over $\frac{N}{2}$ frames. A level

2 buffer contains all polygons static over N, $\frac{N}{2}$ and $\frac{N}{4}$, etc. When drawing a leaf node, this method avoids recompositing two higher parents nodes more than once.

The final result is a reduction in the number of composites from $(N)log_k(N)$ to $N-1$. This should produce a significant improvement in systems where compositing is the main bottleneck.

3.3.2 Predictive Rendering with Subdivision

If object motion is relatively large compared to the predictive periods involved, we can gain even further speedups by using an *adaptive prediction phase*, similar to that described for ray tracing by Chapman [Chapman 90]. In doing so, we assume a large burden of potential error.

Instead of predicting all frames sequentially, we start with the first and last frames. Elements that do not cross the motion threshold are placed on level 0 and removed from consideration. We then subdivide N into two halves, and predict (for each $\frac{N}{2}$) all primitives that have not already been completely predicted. The frame intervals are subdivided recursively until all polygon motion has been isolated to a specific level.

This scheme is very similar to adaptive antialiasing, except it is the temporal domain. If there is little or no change between the internal frames, we can avoid doing predictive work on the vast majority of prediction primitives. However, we can also miss changes that occur below our temporal sampling rate. For example, an object might move from its starting position on frame 0, travel around the screen, and return to its initial position on frame $N-1$. The entire motion will be missed, and subsequently not drawn. Because we are predicting vertex screen motion rather than pixel color, subdivision prediction will properly draw any type of motion where the start and end locations are not the same. The errors caused by insufficient sampling rate are different than those caused by adaptive sampling or Chapman's adaptive temporal coherence [Chapman 90].

This algorithm essentially smooths out jittery local motion in screen space. This smoothing may be acceptable if N is small or polygon motion is large. However, it should be pointed out that this method has a considerable potential for erroneously missing objects, and must be used very carefully.

3.3.3 One-Pass Predictive Rendering

We can improve our prediction speed by combining the prediction and rendering phases. By reducing the number of data traversal passes, we can reduce overhead. We can also use our prediction as the starting point for continuing on to a complete, fully rendered polygon. If this is done right, the prediction cost is contained completely within the rendering cost. This variation requires one framebuffer for each node in the plan tree, so it is really only practical for large memory sizes, large k, or small N.

The algorithm continues prediction as before, keeping track of unchanging runs. When we encounter the last frame, or a frame where a polygon moves, we look back on the polygon's history, and render the polygon into the appropriate frame buffer. This builds a complete depth compositing tree in a single pass, at the cost of many more framebuffers. To draw the frames, we composite the framebuffers in depth-first order. Because all the frames are available, we can avoid having to recomposite all levels on all frames.

Chapter 4

Cost Analysis

So little time, so little to do.

- Oscar Levant

In this chapter, we compare the costs of predictive rendering to the costs of traditional rendering. Contrasting these numbers gives us an idea of how useful predictive rendering will be. If the prediction costs are small compared to the total work required, then factoring out a few static polygons will more than make up for our additional prediction work. If prediction costs are large compared to the total work, then very significant amounts of the picture may need to be stationary before any savings are realized.

We will concentrate on two main architectural paradigms. The first architectural model is the single processor system. This architecture uses a single processor to sequentially compute all components of the rendering process for a single polygon. This does not necessarily imply that a system has a single processing unit - only that there is a single processor working on the complete graphics transformations of a single primitive at any one time. This encompasses architectures where there is a main CPU and a specialized graphics board CPU. Note again that this definition also encompasses multiprocessors containing potentially hundreds of CPUs, as long as each CPU does all the work for a single primitive. Because a single processor must do all the work, avoiding the latter parts of the pipeline can provide important savings. Thus, this type of architecture benefits the most by predictive rendering. The costs of prediction are balanced against the total amount of pipeline work that is avoided. The single processor model encompasses a large class of machines, including the IBM PC, SGI Reality Engine series, and Nintendo and Sega game systems.

The second type of architectural model is the multiprocessor system. Systems of this type contain a series of specialized hardware modules, each designed to perform some small part of the rendering pipeline. At any one time, there are several CPUs working on different parts of the transformation for a single primitive. Multiprocessor systems can potentially gain less benefit from predictive rendering, because the extra prediction costs can only be balanced against the slowest part of the pipeline, and not the entire amount of work done. Avoiding the latter part of the pipeline might not save appreciable work, because later work is executed nearly simultaneously with the initial stages. As we will show, the pipeline bottleneck can still involve enough work to make predictive rendering savings significant. The SGI GTX series is a fairly well-documented example [Akeley 88] of the multiprocessor model.

The cost of predictive rendering depends very heavily on the type of input, i.e., number of primitives, number of vertices, and the natural grouping of related primitives. The total speedup of predictive rendering depends on the amount of temporal coherence present, and the cost of the prediction required to factor out common elements.

Our analysis contains a number of assumptions about rendering costs. We will consistently underestimate the computational effort required for traditional rendering, by simplifying the graphics pipeline and assigning "free" costs to non-trivial operations. For example, we have assumed that the entire traditional rasterization phase can be done for free. It would be possible to develop a model of expected rasterization costs (based on a large number of assumptions), but it is much easier to assume that the whole operation can be done in zero time. Prediction costs are overestimated where described. This makes our prediction-to-rendering cost results extremely conservative. Section 4.1.12 contains an overview of the omitted costs, which are also detailed in discussions of the individual pipeline operations. We have also omitted the cost of several common graphical features, such as texturing, that might be present in many scenarios. This futher hedges our analysis. Section 4.1.13 describes these features further.

We will look at a number of different scenarios, to show how predictive rendering works with different types of input. We assume that all scenarios have no temporal coherence, implying that predictive rendering will cost us extra work. We hope to show that this extra work is small, and can easily be recovered in scenarios with even a little temporal coherence.

The first scenario ("random") consists of a database of 1000 Gouraud shaded polygons, each with four vertices. The polygons have no particular relationship with each other, and can be arbitrarily positioned in space. The scene is lighted with a local viewer, and four local lights. Light strength is attenuated with distance. The effects of a local viewer, local lights and light attenuation are described in Section 4.1.4. The polygon scenario (with different lighting conditions) is described in [Akeley 88], from which we will draw heavily.

The second scenario ("blocks") consists of 1000 blocks, where each block is a fixed, closed object composed of six polygons with four vertices each. This approximates an indoor scene, where the majority of objects are regular and man-made. It can also approximate any voxel or space-filled object. The scene is Gouraud shaded and lighted with a local viewer and four local, attenuated, lights.

The third scenario ("particles") is composed of 1000 static spheres, where each sphere is composed of 144 polygons with four vertices each. This is a fine enough subdivision to avoid noticeable irregularities in silhouette edges. This scenario models particle movement, where the motion of wind or water might be displayed with streams of small spheres. The scene is Gouraud shaded and lighted with a local viewer and four local, attenuated lights.

The final scenario ("molecules") is made up of 1000 fictional, two atom molecules. Each molecule consists of a two atoms joined by a cylindrical connector. Atoms are approximated with 144, four vertex polygons, while the connector uses 12, four vertex polygons. For our example, atoms do not change size, shape, or relative position, and globally rotate around the center of the connector. The molecule scenario is lighted with a local viewer and four local attenuated lights.

Figure 4.1 shows the components of the various scenarios. Examples of all scenes are shown in Chapter 5.

The total number of components in each scenario are summarized in Table 4.2.

Scenario	**Components**	**Polygons/Component**	**Total Polygons**	**Total Vertices**
random	1000	1	1,000	4,000
blocks	1000	6	6,000	24,000
particles	1000	144	144,000	576,000
molecules	1000	300	300,000	1,200,000

Table 4.2: Scenario Polygon and Vertex Count

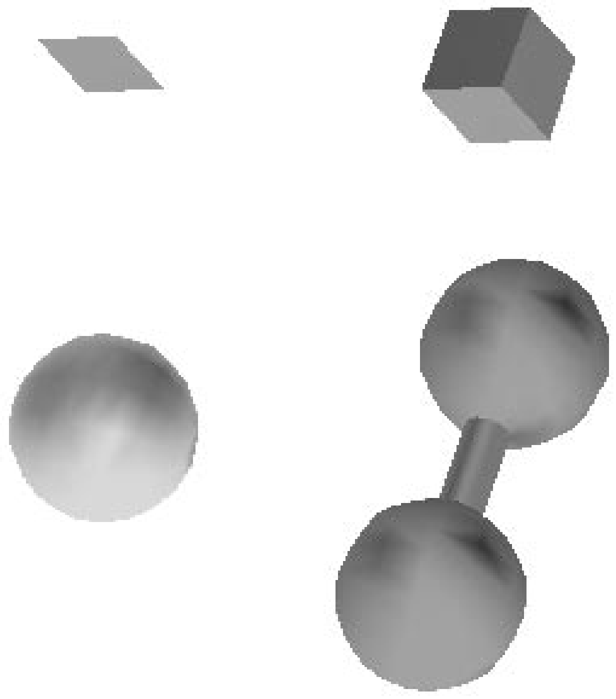


Figure 4.1: Analysis Scenario Components

4.1 Single Processor System

In a single processor system, all steps of the graphics pipeline (for a particular primitive) are carried out by the same processor. If we can avoid having to draw a primitive on a particular frame, we can avoid all the work needed to send that primitive through the pipeline. Figure 4.2 shows the general steps of the single processor graphics pipeline.

4.1.1 Transferring Data to the Pipeline

In an immediate mode graphics system, the object database must be fed into the graphics pipeline for each and every frame. This adds extra cost, but also gives the user programs a great deal of flexibility, allowing easy addition, deletion and modification of graphics objects.

The graphics database consists of vertex and normal information. Each vertex consists of an (x, y, z) floating point value. Some of the vertices and faces require three element (nx, ny, nx)

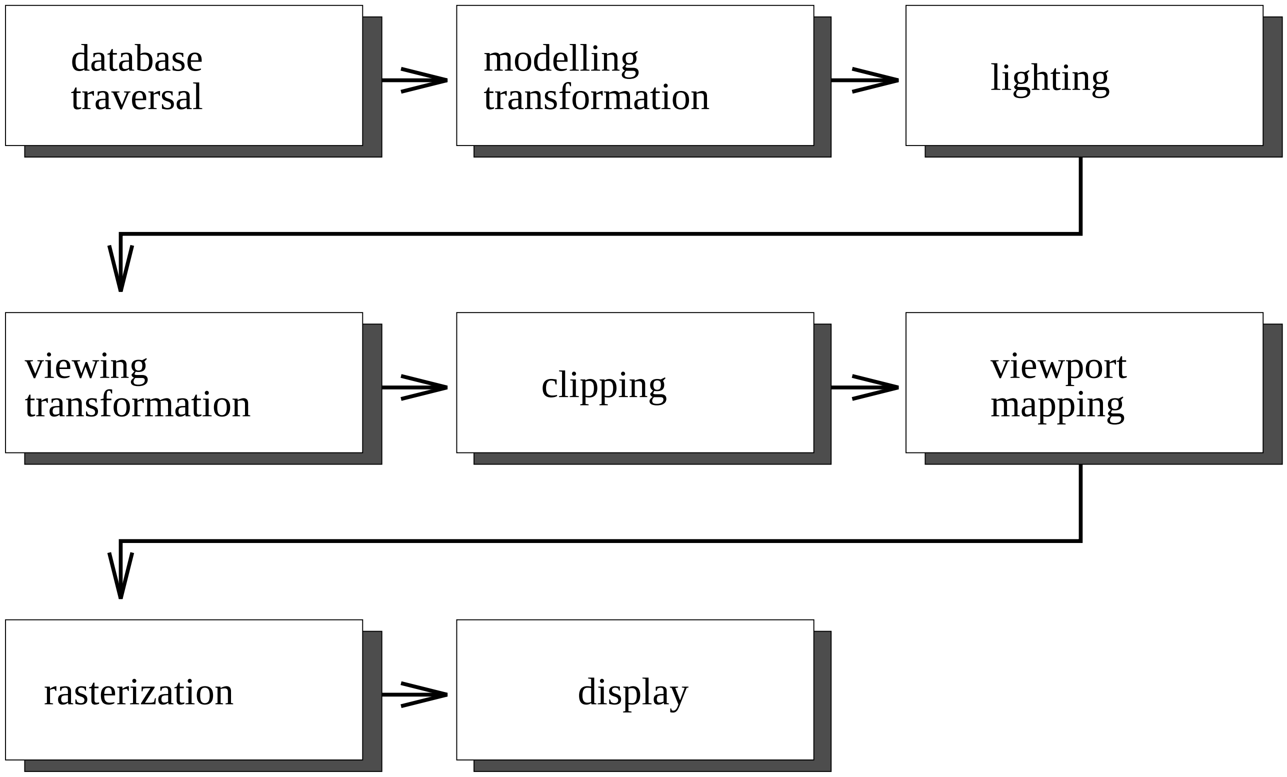


Figure 4.2: Single Processor Graphics Pipeline

normals, used to compute polygon lighting. The "random" scenario can contain non-coplanar polygons, and so requires a normal for every vertex. The "particles" and "molecules" scenarios both contain curved objects approximated by flat surfaces, and so also require a normal for every vertex. The "blocks" scenario can get away with only one normal per side, instead of one normal per vertex.

The data transfer costs for the various scenarios are summarized in Table 4.3, assuming four bytes per float. The data transfer cost is the expense of loading the information into the system. The application program will usually store the object database much more compactly.

The prediction phase also requires transferring data into the graphics pipeline, because we assume that both prediction and rendering are done with essentially the same code. Depending on our particular scenario, we can dramatically reduce the amount of data sent into the prediction pipeline. Ideally, we want to group large numbers of polygons into a single prediction

Scenario	Vertices	Normals	Floats	Data
random	4,000	4,000	24,000	93.8 KB
blocks	24,000	6,000	90,000	351.6 KB
particles	576,000	576,000	3,456,000	13,500 KB
molecules	1,200,000	1,200,000	7,200,000	28,125 KB

Table 4.3: Rendering - Transferring to the Pipeline

primitive. Tracking a single primitive is much cheaper than tracking all of a primitive's components. Luckily, polygons can be grouped naturally to model real-world things, even if only to place six faces together to form a solid object.

The "random" scenario requires a polygon prediction primitive, where each graphics polygon is predicted using all four vertices. The number of rendering and prediction vertices are the same, providing no savings.

Each block in the "blocks" scenario can be predicted using a block prediction primitive. The six sides and 24 vertices of the static block can be predicted with an eight vertex prediction primitive. This reduces the number of prediction vertices to $\frac{1}{3}$ of the rendering vertices. We could also use an octahedron primitive, and reduce the cost even more.

We can gain an even greater data reduction in the "particles" scenario. Each particle can be predicted with an octahedron prediction primitive. This reduces the 144 sides and 576 vertices to six vertices - 1% of the original vertices.

The greatest data reduction can be achieved in the "molecules" scenario. The motion of a molecule, with 300 sides and 1200 vertices, can be predicted with a six vertex octahedron. The prediction requires about 0.5% of the original vertices. Note that the data transfer costs of predictive rendering remain the same even if we dramatically increase the number of polygons used to make up a molecule.

Each prediction primitive vertex requires a three float (x, y, z) value. We assume all points have an implicit homogeneous coordinate $w = 1$. No normals are required for the prediction phase. Table 4.4 shows the data transfer costs for the prediction, again assuming four bytes per float.

Obviously, the more complex the graphical object, the greater the savings we can achieve by

Scenario	**Prediction Primitives**	**Predictive Vertices**	**Floats**	**Data**
random	1,000	4,000	12,000	46.9 KB
blocks	1,000	8,000	24,000	93.8 KB
particles	1,000	6,000	18,000	70.3 KB
molecules	1,000	6,000	18,000	70.3 KB

Table 4.4: Prediction - Transferring to the Pipeline

predicting it with a few vertices. This means that prediction costs will stay constant as objects are given even more detail. One could imagine a single feather, translating and rotating from an axis at the base of the shaft. It would take one prediction primitive to model and track this motion, for cases where the feather was made up of one polygon, one million polygons, or one billion polygons. This data reduction extends to subsequent steps in the graphics pipeline. This is an area where significant savings in the prediction cost can be achieved. The worst case prediction scenario requires 50% of the rendering phase data transfer, while the best case scenario requires 0.25% of the data transfer.

4.1.2 Vertex Transformation

The first actual step of the graphics pipeline is to transform vertices and normals from object coordinate space into world coordinate space. This requires multiplication by a modeling matrix. The modeling matrix is composed (via concatenation) of a current transformation and any previous transformations for objects higher up in a hierarchical structure.

Before transforming individual vertices, we must first generate the transformation matrix. We assume a modelling matrix stack, where new matrices are multiplied with the top stack matrix. In the worst case, polygons are part of a very deep hierarchical structure, where there is a matrix concatenation for every polygon in the database. This is fairly unrealistic, because it assumes that (for example) a block consists of a single side rotated and translated six times. Realistically, there will be many fewer concatenations, depending on the database structure. A very flat structure may have very few concatenations. In a completely flat structure, each modelling matrix requires no concatenations at all. We will look at the worst case and a better case, where each general object (molecule, block, etc.) is part of a tree, and requires a single concatenation. A concatenation takes 64 multiplications and 48 additions. These costs are reflected in the various scenarios as shown in Table 4.5. When we sum up the total costs of rendering, we will assume the concatenation cost is zero. This is the most likely case for the molecules and particles scenario, where each object is probably independent from other objects. Of course, it is easy to think of counter-examples. For example, the position of of a single spark in a fireworks shower might depend on the position of a tree of parent sparks.

Scenario	**Concats**	**Mults**	**Adds**
random (worst)	1,000	64,000	48,000
random (better)	1,000	64,000	48,000
blocks (worst)	6,000	384,000	288,000
blocks (better)	1,000	64,000	48,000
particles (worst)	144,000	9,216,000	6,912,000
particles (better)	1,000	64,000	48,000
molecules (worst)	300,000	19,200,000	14,400,000
molecules (better)	1,000	64,000	48,000

Table 4.5: Rendering - Model Matrix Building

The prediction phase also requires an appropriately concatenated modelling matrix. We assume the same worst case as in the rendering phase, where prediction primitives are organized in a deep hierarchy. We gain considerably by using more complex prediction primitives - concatenation is per prediction primitive rather than per polygon.

It is also possible to take advantage of the concatenation to do some extra work. We can load the multiplied viewing and projection matrices into the very bottom of the matrix stack. As modelling matrices are concatenated onto the stack, they modify the viewing and projection matrices. This essentially builds a single transformation matrix that takes vertices from object coordinates into projected coordinates. The worst and best cases for predictive rendering are now the same, where each modelling matrix requires one concatenation. Table 4.6 gives the costs of building the matrices.

We could use this technique for the rendering phase as well. However, we will avoid it in order to compute lighting attenuation. Light attenuation requires the distance between the light and the vertex, computed in world coordinates. This means that vertices cannot be transformed directly into projected coordinates; two transformations are required, with the

Scenario	**Concats**	**Mults**	**Adds**
random	1,000	64,000	48,000
blocks	1,000	64,000	48,000
particles	1,000	64,000	48,000
molecules	1,000	64,000	48,000

Table 4.6: Prediction - Model Matrix Building

lighting calculation performed as the middle step. Separating the object-to-world from the world-to-projection transformations adds another step, described in Section 4.1.5. If we did not compute light attenuation, we could avoid the step described in Section 4.1.5 by combining it with the transformation matrix in the same way as predictive rendering. Of course, this would increase the best-case concatenation cost - the viewing and projection matrix must always be multiplied with a modelling matrix. Best-case costs would then be the "better" case in Table 4.5.

After building the transformation matrices, we transform the vertices. Transforming a four element vector by a 4×4 matrix requires 16 multiplications and 12 additions. Given that the 4^{th} element of each vector is always one, it is possible to just sum to find the new w, avoiding any unnecessary multiplication by one. This shortcut removes four multiplications, leaving 12 multiplications and 12 additions per vertex transformation. The costs of the rendering phase vertex transformations are shown in Table 4.7.

Scenario	**Polygons**	**Vertices**	**Mults**	**Adds**
random	1,000	4,000	48,000	48,000
blocks	6,000	24,000	288,000	288,000
particles	144,000	576,000	6,912,000	6,912,000
molecules	300,000	1,200,000	14,400,000	14,400,000

Table 4.7: Rendering - Vertex Transformation

Table 4.8 gives the costs for the object-to-projection space transformation for the prediction phase.

Scenario	**Prediction Primitives**	**Prediction Vertices**	**Mults**	**Adds**
random	1,000	4,000	48,000	48,000
blocks	1,000	8,000	96,000	96,000
particles	1,000	6,000	72,000	72,000
molecules	1,000	6,000	72,000	72,000

Table 4.8: Prediction - Vertex Transformation

4.1.3 Normal Transformation

In order to perform lighting calculations, each vertex normal must also be transformed from object coordinate space to world coordinate space. Because normals are not position sensitive, the normal transformation matrix can be shrunk to 3×3 size. A matrix of this size requires nine multiplications and six additions to transform each normal. We assume (rather brashly) that we can compute the normal transformation matrix for free. Normal transformation costs are given in Table 4.9.

Scenario	**Polygons**	**Normals**	**Mults**	**Adds**
random	1,000	4,000	36,000	24,000
blocks	6,000	6,000	54,000	36,000
particles	144,000	576,000	5,184,000	3,456,000
molecules	300,000	1,200,000	10,800,000	7,200,000

Table 4.9: Rendering - Normal Transformation

We make an additional assumption that all normals are of unit length. If this assumption is broken, we require another 22 multiplications and two additions per vertex to normalize. It takes three multiplications and two additions to compute the sum of squares, with an additional eight multiplications for the square root [Akeley 88] and eight multiplications for the inverse of the length. There are three final multiplications to normalize the numerator.

The prediction phase does not need to predict normals, and so no cost is required. If we count the multiplications required in vertex and normal transformation only, we see that predictive rendering costs for our scenarios range between 57% and 0.3% of the rendering costs.

4.1.4 Lighting

Given our assumption of Gouraud shading, we need to compute lighting values for each of the polygon vertices. Lighting values are interpolated across the interior of each final projected polygon.

We have assumed a local viewer, and a number of local lights. A local viewer implies that the viewer is a finite distance from all objects; thus the distances and directions between lights and objects will all vary. An infinite viewer implies that the distance and direction from an object to the light is constant across all the objects.

A fairly common lighting model, found in [Foley 90] and originally developed by [Bui-Tuong 75] is

$$\vec{C}_{object} = \vec{C}_{ambient} + \sum A_{light}\vec{C}_{light}[\vec{C}_{diffuse}(\vec{N} \cdot \vec{L}) + \vec{C}_{specular}(\vec{N} \cdot \vec{H})^n] \tag{4.1}$$

where $\vec{C}_{ambient}$, $\vec{C}_{diffuse}$, $\vec{C}_{specular}$ and n are the ambient, diffuse, specular and "shininess" material properties of individual polygons. A_{light} is an attenuation factor that models light-source falloff. $\vec{C}_{light}$ is the intensity of the light, while $\vec{N}$ is the vertex normal, $\vec{L}$ is the light direction from the vertex, and $\vec{H}$ is the vector halfway between the direction of the light source and the viewer. All material and light properties except for shininess have three components (either RGB or XYZ). This lighting model is supported by many graphics libraries, such as SGI's GL. Indeed, GL supports even more lighting properties, such as emission and spotlighting.

Specular Term

Using local lights and local viewers adds a significant additional lighting burden. When the viewer is local, the halfway vector $\vec{H}$ varies across the polygon, and so the dot product $(\vec{N} \cdot \vec{H})$ must be recomputed for each polygonal vertex, assuming a normal per vertex. The halfway vector $\vec{H}$ can be computed as

$$\vec{H} = \frac{\vec{L} + \vec{V}}{||\vec{L} + \vec{V}||} \tag{4.2}$$

where $\vec{V}$ is the vector from the object to the viewer, and $\vec{L}$ is the vector from the object to the light.

The denominator requires three additions to add the vectors, three multiplications and two additions to compute the sum of squares, and an estimated 16 multiplications to compute the reciprocal square root. The final result requires another three more numerator-denominator multiplications, for a total of 22 multiplications and five additions. We assume $\vec{L}$ and $\vec{V}$ are computed for free.

For comparison, we also tabulate the case with infinitely distant lights and an infinitely distant viewer. When the viewer and light sources are infinite, both $\vec{L}$ and $\vec{V}$ are constant for a given light, and so $\vec{H}$ needs to be computed no more than once per frame. This is computationally negligible.

Computing $(\vec{N}\cdot\vec{H})$ requires three multiplications and two additions. This must be computed once per vertex in both the local and infinite cases. We assume the power of n term can be computed via table lookup. The power product only needs to be computed once per RGB element, and requires an additional three multiplications to account for the $\vec{C}_{specular}$ factor. The total specular cost for a single light, local case is 28 multiplications and seven additions per vertex, while the infinite case is six multiplications and two additions.

Diffuse Term

The diffuse component requires 3 multiplications and 2 additions to compute the $(\vec{N} \cdot \vec{L})$ dot product. This factor must then be multiplied once per RGB component for a total cost of 6 multiplications and 2 additions. The diffuse component must be computed once per normal in both the local and infinite cases.

Attenuation

Energy from a point light source falls off with the inverse square of the distance from the light source. In order to model this energy falloff, light intensity is often modified by an attenuation factor A_{light}. This attenuation factor is not always an inverse square model - it is often computed in ways that are physically inaccurate for a point light source, but more pleasing aesthetically. Attenuation does not apply to infinite light sources. We will use the SGI GL library version of attenuation, where

$$A_{light} = \frac{1.0}{(K_{constant} + K_{linear} * D + K_{squared} * D * D)} \tag{4.3}$$

where D is the distance between the vertex and a light, and $K_{constant}$, K_{linear} and $K_{squared}$ are parameters that model light attenuation dependent and independent of vertex distance.

To make our attenuation model simple (and to hedge our prediction costs even more), we will assume that the $K_{squared}$ term is always zero. To compute the distance D, we require 3 subtractions, 3 multiplications, and 2 additions, with an additional 8 multiplications to compute the square root of the distance sum of squares. We need another addition and multiplication to compute the denominator, 8 more multiplications to compute the reciprocal, and 3 multiplications to factor in the C_{light} product, for a total of 23 multiplications, and 6 additions. The light attenuation cost is per vertex, per attenuated light.

Putting It Together

The ambient term does not require any additional computation. It takes 3 additions to add the specular and diffuse products, followed by 3 multiplications to factor in the C_{light} value.

Multiple Lights

The local case requires a total of 60 multiplications and 18 additions, while the infinite case requires 15 multiplications and 7 additions. The costs per scenario for the local case and infinite cases are shown in Table 4.10.

These costs are for a single light, and do not include any overflow checking, or possible optimizations.

Our test cases call for a scene with four local lights. Table 4.11 shows the finite cases and infinite cases multiplied by four, with an additional 9 additions to per vertex to compute the grand lighted sum.

Of course, the predictive phase does not require any lighting calculations.

Scenario	Polygons	Vertices	Mults	Adds
random (local)	1,000	4,000	240,000	72,000
random (infinite)	1,000	4,000	60,000	28,000
blocks (local)	6,000	24,000	1,440,000	432,000
blocks (infinite)	6,000	24,000	360,000	168,000
particles (local)	144,000	576,000	34,560,000	10,368,000
particles (infinite)	144,000	576,000	8,640,000	4,032,000
molecules (local)	300,000	1,200,000	72,200,000	21,600,000
molecules (infinite)	300,000	1,200,000	18,000,000	8,400,000

Table 4.10: Rendering - Lighting (1 light)

Scenario	Polygons	Vertices	Mults	Adds
random (local)	1,000	4,000	960,000	324,000
random (infinite)	1,000	4,000	240,000	148,000
blocks (local)	6,000	24,000	5,760,000	1,944,000
blocks (infinite)	6,000	24,000	1,440,000	888,000
particles (local)	144,000	576,000	138,240,000	46,656,000
particles (infinite)	144,000	576,000	34,560,000	21,130,000
molecules (local)	300,000	1,200,000	288,000,000	97,200,000
molecules (infinite)	300,000	1,200,000	72,000,000	44,400,000

Table 4.11: Rendering - Lighting (4 lights)

4.1.5 Viewing Transformation

If we use a light model with light source attenuation, we need to transform object space vertices to projection coordinates in two distinct phases. The first phase takes object coordinates into world coordinates, where light-vertex distances can be computed for the attenuation calculation. World coordinate vertices must then be further transformed into projection coordinates. This transformation requires multiplying polygon vertices with a 4×4 viewing matrix composed of an eye matrix and a projection matrix. This would normally require 16 multiplications and 12 additions per vertex. We can take advantage of the sparseness of the viewing matrix to reduce the number of multiplications to 8 and the number of additions to 6, as per [Akeley 88]. The costs of the rendering-phase world-to-projection transformation are shown in Table 4.12.

The prediction phase can transform vertices in a single pass, and so does not require this step.

Scenario	Polygons	Vertices	Mults	Adds
random	1,000	4,000	32,000	24,000
blocks	6,000	24,000	192,000	144,000
particles	144,000	576,000	4,608,000	3,456,000
molecules	300,000	1,200,000	9,600,000	7,200,000

Table 4.12: Rendering - 2nd Phase Vertex Transformation

4.1.6 Clipping

The first few steps of the rendering pipeline can be characterized and measured fairly concretely. Subsequent steps, starting with clipping, depend heavily on particular characteristics of the object database. We are required to make some fairly large assumptions, keeping in mind how easy it is for our assumptions to be violated.

In the clipping phase, primitives are clipped to a 3D viewing volume. Akeley describes a SGI implementation of the Sutherland and Hodgeman [Sutherland 74] clipping algorithm that clip-tests primitives to any one of the six possible bounding planes of the projection volume. Each primitive requires one comparison per vertex per clipping plane. At a cost of 6 comparisons per vertex, the clip-test costs for our sample scenarios are shown in Table 4.13.

Scenario	Polygons	Vertices	Comparisons
random	1,000	4,000	24,000
blocks	6,000	24,000	144,000
particles	144,000	576,000	3,456,000
molecules	300,000	1,200,000	7,200,000

Table 4.13: Rendering - Clipping Comparisons

The clipping costs given above only determine if a primitive needs to be clipped or not; there is additional cost if a primitive must actually be clipped. The cost for clipping is variable, depending on the polygon. Akeley claims that it is common to assume that 10% or fewer of the primitives require actual clip intersection work. If the number of polygons requiring additional work is much greater than this, the total cost can increase dramatically. Clipping can introduce additional vertices. To make things easy, we assume that no polygons need clip work - a very generous allowance, indeed.

Although we have described a clipping algorithm for the predictive rendering process, we choose not to implement it for our analysis.

4.1.7 Projection

After vertices have been clipped in homogeneous space, they must be projected against the viewplane. Projection is accomplished by dividing the x, y and z vertex components by w. This costs 8 multiplications (per vertex) to compute $\frac{1}{w}$, and 3 additional multiplications (per vertex) to project individual components. The rendering costs are shown in Table 4.14, assuming no extra vertices have been introduced in the clipping phase.

Scenario	**Polygons**	**Vertices**	**Multiplications**
random	1,000	4,000	44,000
blocks	6,000	24,000	264,000
particles	144,000	576,000	6,336,000
molecules	300,000	1,200,000	13,200,000

Table 4.14: Viewplane Projection

The predictive rendering phase also requires projection to the viewplane. In the rendering phase, the z value is needed to compute interpolated z values for the depth buffering phase. There is no such requirement for predictive rendering, so projection requires only 10 multiplications per predictive vertex. Costs are shown in Table 4.15.

Scenario	**Prediction Vertices**	**Multiplications**
random	4,000	40,000
blocks	8,000	80,000
particles	6,000	60,000
molecules	6,000	60,000

Table 4.15: Prediction - Viewplane Projection

4.1.8 Viewport Mapping

Before scan converting and doing depth comparison, the projected vertices must be converted into screen space. The x and y coordinates must be multiplied by a scale factor, and added to an offset. Costs are shown in Table 4.16.

Scenario	Polygons	Vertices	Mults	Adds
random	1,000	4,000	8,000	8,000
blocks	6,000	24,000	48,000	48,000
particles	144,000	576,000	1,152,000	1,152,000
molecules	300,000	1,200,000	2,400,000	2,400,000

Table 4.16: Viewport Mapping

The predictive phase keeps all vertices in viewplane coordinates, choosing to instead convert the "motion δ" from screen to world viewplane coordinates, a negligible cost.

At this point, the rendering and predictive pipelines diverge. Up to now, the predictive phase could be accomplished by using a selective subset of the regular pipeline. Now, the prediction phase requires some small computations not currently supported by commercial libraries, while the rendering phases needs significant additional work before it can display polygons. We will complete the additional costs of predictive rendering before finishing the rendering pipeline.

4.1.9 Transferring Data from the Pipeline

The prediction data must be obtained from the pipeline, and compared against last frame's positions. Technically, only the projected x and y components are required for each vertex. The final z component is not used in the prediction. However, it is likely that any mechanism to get at the data will provide the z component as well, giving costs as shown in Table 4.4.

After obtaining the data, each x and y component must be compared with a motion threshold δ. This requires two subtractions, two absolute values, and two comparisons per vertex. We assume an absolute value can be computed with one comparison, giving a total of five comparisons per vertex. Comparison costs are as shown in Table 4.17.

Scenario	Total Predictive Vertices	Subtractions	Comparisons
random	4,000	8,000	20,000
blocks	8,000	16,000	40,000
particles	6,000	12,000	30,000
molecules	6,000	12,000	30,000

Table 4.17: Prediction - Comparing Against Past Vertices

4.1.10 Depth Compositing

There are some additional costs to predictive rendering beyond the actual prediction. During rendering, scratch framebuffers must be depth composited together to form a final result.

For a scenario with a k-tree, and N frame lookahead, we need to depth composite $log_k(N)+1$ individual buffers together. Given that the last buffer is the target, we need $log_k(N)$ composites, where each depth composite operates over a framebuffer of height H and width W. For each pixel composite, we need to read a depth buffer, and potentially write a depth buffer and an RGBA value, for a total of 6 framebuffer operations. Thus, the total depth compositing cost is $6HWlog_k(N)$.

This cost is non-trivial, especially on systems with poor depth buffer access support. However, the cost is fixed. As well, the extra framebuffer accesses required for depth compositing are somewhat mitigated by a reduced number of framebuffer accesses required to rasterize incoming polygons.

We can reduce our depth compositing costs with a few simple modifications. During the rendering phase, we keep track of the number of polygons sent to each scratch buffer. Scratch buffers with no elements do not need to be composited (or cleared) on this frame. This does not always ensure a speedup, but does prevent unnecessary depth compositing. Minimizing the number of levels in the k-tree also helps.

We can also implement on-the-fly depth compositing (described in Section 3.3.1. This compositing method reduces the total number of frame composites from $(N)log_k(N)$ to $N-1$, resulting in a fairly significant improvement in cost.

4.1.11 Rasterization

We now return to the rendering pipeline. The rasterization phase converts clipped and projected polygons into pixel values, and places them in the framebuffer. Rasterization requires sorting polygon vertices by screen coordinates, breaking polygons into chunks, and determining object boundaries by computing the location of edges between vertices. Pixel colors and z values are interpolated between edges of a polygon, and all pixels are submitted to the z buffer.

In the rasterization phase, it becomes much trickier to characterize the computational cost, so in true cavalier fashion we will not even bother.

4.1.12 Single Processor Totals

We can sum up the required totals for our simple scenarios, giving an approximate idea of the cost of predictive rendering vs. the cost of actually rendering. We have assumed a case where concatenations cost nothing, no primitives are clipped, normals are pre-normalized, and $\vec{L}$, $\vec{V}$ and inverse normal matrices are all computed for free. Rasterization costs are also magically avoided, mainly because they are too hard to estimate accurately. Because of the significant underestimation of costs, any of these costs greater than zero improves the attractiveness of predictive rendering.

We do not include depth compositing in the predictive rendering costs, because compositing is not technically prediction; it is used during the actual rendering only if prediction is used. Compositing costs are not related to the scene contents, except as this affects the depth of the compositing tree.

Table 4.18 summarizes the costs for the random scenario. The total prediction cost (for multiplications only) is about 13.5% of the total drawing costs, given our restrictions. Implemented costs should be lower, with actual clipping and rasterization. Because there are the same number of predictive and actual vertices, the transformation costs are nearly the same. Only the lighting makes a significant difference - an important consideration with a multiprocessor architecture.

Table 4.19 summarizes the costs for the block scenario. The total prediction cost (for multiplications only) is about 3.6% of the total rendering costs. The lighting component is less of the total cost, because the number of predictive primitives and actual primitives are now different.

Table 4.20 summarizes the costs for the particle scenario. The total prediction cost (for multiplications only) is about 0.1% of the total rendering costs. There is a significant difference between the number of actual and predictive vertices transformed.

Table 4.21 summarizes the costs for the molecule scenario. The total prediction costs (for multiplications only) range about 0.05% ($\frac{1}{2000}$) of the total rendering costs. In a scenario where

Action	Mults (rend)	Mults (pred)	Add (rend)	Add (pred)	Comps (rend)	Comps (pred)	Data (rend)	Data (pred)
transfer in							93.8	46.9
concatenation		64,000		48,000				
vertex trans	48,000	48,000	48,000	48,000				
normal trans	36,000		24,000					
lighting	960,000		324,000					
vertex trans	32,000		24,000					
clipping					24,000			
projection	44,000	40,000						
viewport	8,000		8,000					
transfer out								46.9
compare				8,000		20,000		
totals	1,128,000	152,000	426,000	104,000	24,000	20,000	93.8	93.8
% of rendering		13.5%		24.4%		83.3%		100%

Table 4.18: Random Scenario - Rendering vs. Prediction

there is no lighting (or normals), the cost of predictive rendering is still about 0.5% of the total cost to render a frame. When predictive rendering costs are this low relative to the complete cost, it pays to run predictive rendering continuously, irregardless of how much frame coherence is expected. Even a tiny bit of frame coherence will recoup the prediction costs.

4.1.13 Analysis of Simplifications and Omissions

We have skipped a number of elements that could easily be added to the pipeline. These additional costs can more than compensate for savings due to code and calculation optimizations that we have skipped over in the analysis. For example, lights supported by the SGI GL library can also have distance squared attenuation and directional spotlighting. Materials can have emission and transparency, as well as being textured and antialiased. Texturing alone adds a very significant resource drain on the pipeline. We also made a number of fairly strong assumptions about costs (listed in the previous section), including quite of a bit of "free" work. By skipping these extra costs, we make the analysis a great deal more conservative, at the expense of reporting better prediction-to-drawing ratios.

This purpose of this section is to show that prediction is inexpensive compared to drawing;

Action	**Mults (rend)**	**Mults (pred)**	**Add (rend)**	**Add (pred)**	**Comps (rend)**	**Comps (pred)**	**Data (rend)**	**Data (pred)**
transfer in							351.6	93.8
concatenation		64,000		48,000				
vertex trans	288,000	96,000	288,000	96,000				
normal trans	54,000		36,000					
lighting	5,760,000		1,944,000					
vertex trans	192,000		144,000					
clipping					144,000			
projection	264,000	80,000						
viewport	48,000		48,000					
transfer out								93.8
compare				16,000		40,000		
totals	6,606,000	240,000	2,460,000	160,000	144,000	40,000	351.6	187.6
% of rendering		3.6%		6.5%		27.8%		53.3%

Table 4.19: Blocks Scenario - Rendering vs. Prediction

thus we can predict almost all of the time. During periods of high frame coherence, the prediction will pay off in big speedups, while other times, the prediction cost will be essentially wasted. The important point is that prediction can be performed as a very small fraction of the total work required.

4.2 Multiprocessor System

Higher-end graphics hardware often uses more than one CPU to speed up computations. One of the most common scenarios is hardware pipelining, where several special-purpose CPUs are strung together in a row. Each CPU works on a single phase of the total transformation. This can speed things up dramatically; as soon as the first primitive has made it through the pipeline, all CPUs have something to work on in parallel. The total cost for the drawing process (ignoring the lag while the pipeline fills) is then the cost of the slowest phase in the pipeline. Predictive rendering cost is now the slowest phase in the transformation, compared with the slowest phase in general rendering. We need to look at the separation of the tasks into CPUs to determine how a particular architecture affects predictive rendering.

In order to give a concrete example, we will look at the Geometry Engine hardware described

Action	**Mults (rend)**	**Mults (pred)**	**Add (rend)**	**Add (pred)**	**Comps (rend)**	**Comps (pred)**	**Data (rend)**	**Data (pred)**
trans in							13,500	70.3
concat		64,000		48,000				
vertex trans	6,912,000	72,000	6,912,000	72,000				
normal trans	5,184,000		3,456,000					
lighting	138,240,000		46,656,000					
vertex trans	4,604,000		3,456,000					
clipping					3,456,000			
projection	6,336,000	60,000						
viewport	1,152,000		1,152,000					
trans out								70.3
compare				12,000		30,000		
totals	162,428,000	196,000	61,632,000	132,000	3,456,000	30,000	13,500	140.6
% of render		0.1%		0.2%		0.9%		1.0%

Table 4.20: Particles Scenario - Rendering vs. Prediction

in [Akeley 88]. The hardware diagram is shown in Figure 4.3, and forms the basis for the SGI GTX series. The GTX is one of SGI's so-called "second-generation" workstations.

The GTX pipeline is divided into four main subsystems: the geometry subsystem, the scan-conversion subsystem, the raster subsystem, and the display subsystem. Of the most interest is the geometry subsystem, which consists of 5 "Geometry Engines", each with a specific task. The five tasks are

1. Transforming vertices, normals, and matrices, maintaining matrix stacks, and normalizing normals.
2. Lighting calculations.
3. Clip testing.
4. Perspective division, and clipping when required.
5. Viewport transformation, color range clamping, and depthcueing calculations.

We can group our single CPU totals by Geometry Engine tasks. The tasks are not equally divided, partially because of our assumptions about clipping and lighting. Predictive rendering

Action	**Mults (rend)**	**Mults (pred)**	**Add (rend)**	**Add (pred)**	**Comps (rend)**	**Comps (pred)**	**Data (rend)**	**Data (pred)**
trans in							28,125	70.3
concat		64,000		48,000				
vertex trans	14,400,000	72,000	14,400,000	72,000				
normal trans	10,800,000		7,200,000					
lighting	288,000,000		97,200,000					
vertex trans	9,600,000		7,200,000					
clipping					7,200,000			
projection	13,200,000	60,000						
viewport	2,400,000		2,400,000					
trans out								70.3
compare				12,000		30,000		
totals	338,400,000	196,000	128,400,000	132,000	7,200,000	30,000	28,125	140.6
% of render		0.05%		0.1%		0.4%		0.55%

Table 4.21: Molecules Scenario - Rendering vs. Prediction

can almost be completed entirely within the first Geometry Engine. We place the predictive rendering comparison function in the viewport Geometry Engine. Table 4.22 shows the new totals.

Action	**Mults (rend)**	**Mults (pred)**	**Add (rend)**	**Add (pred)**	**Comps (rend)**	**Comps (pred)**	**Data (rend)**	**Data (pred)**
trans	116,000	128,000	96,000	96,000			93.8	46.9
lighting	960,000		324,000					
clipping					24,000			
perspective	44,000	40,000						
viewport	8,000		8,000	8,000		24,000		46.9
% of b-neck		17.5%		32.1%		100%		50%

Table 4.22: Multiprocessor Random Scenario - Rendering vs. Prediction

Table 4.23 shows the totals for the molecular scenario.

We ignore the lag caused by the first primitive working its way through the pipeline, and assume that all hardware elements are always busy. Then, the total cost for rendering is the bottleneck of the system. As can be seen, the multiprocessor hardware paradigm bottleneck is in the lighting. We have neglected clipping, texturing and rasterization as potential bottlenecks, although they certainly are good candidates. To compute the ratio of prediction to rendering,

Action	**Mults (rend)**	**Mults (pred)**	**Add (rend)**	**Add (pred)**	**Comps (rend)**	**Comps (pred)**	**Data (rend)**	**Data (pred)**
trans	34,800,000	136,000	28,800,000	120,000			28,125	70.3
lighting	288,000,000		97,200,000					
clipping					7,200,000			
perspective	13,200,000	60,000						
viewport	2,400,000		2,400,000	12,000		36,000		70.3
% of b-neck		0.07%		0.13%		0.5%		0.5%

Table 4.23: Multiprocessor Molecules Scenario - Rendering vs. Prediction

we must compare prediction and rendering bottlenecks. The cost of predictive rendering is still very favorable, mainly because the lighting cost is so great. This implies that predictive rendering should be useful in both single and multiprocessor systems, given our scenarios.

It is interesting to note that the SGI Reality Engine [Akeley 93] hardware, which forms the basis for SGI's first "third-generation" workstation, is based on the single processor model. Each (of potentially many) Geometry Engines works on a single polygon, starting at the initial transformation and ending with the computation of slope and texture information. This means that the new third-generation of SGIs should work well with predictive rendering.

It is important to realize that these cost approximations are abstract; that is, they are computed by looking at a typical graphics pipeline, rather than a specific brand name's pipeline. Individual manufacturers may add little hardware and software twists to the process that change the estimations. For example, we assume that our prediction primitives are not sent needlessly through extra steps, doing null clipping, null lighting, null texturing, etc. How many extra steps we need depends on what sort of access we can get to the pipeline. However, several important points should be clear from this section. First, predictive rendering allows significant savings if we can make single predictions to track objects composed of large groups of polygons. Secondly, predictive rendering is more effective for high-cost, high-work scenes. Finally, predictive rendering and graphics rendering do almost all the same work, so that any modifications or improvements to the pipeline will affect both drawing and prediction equally.

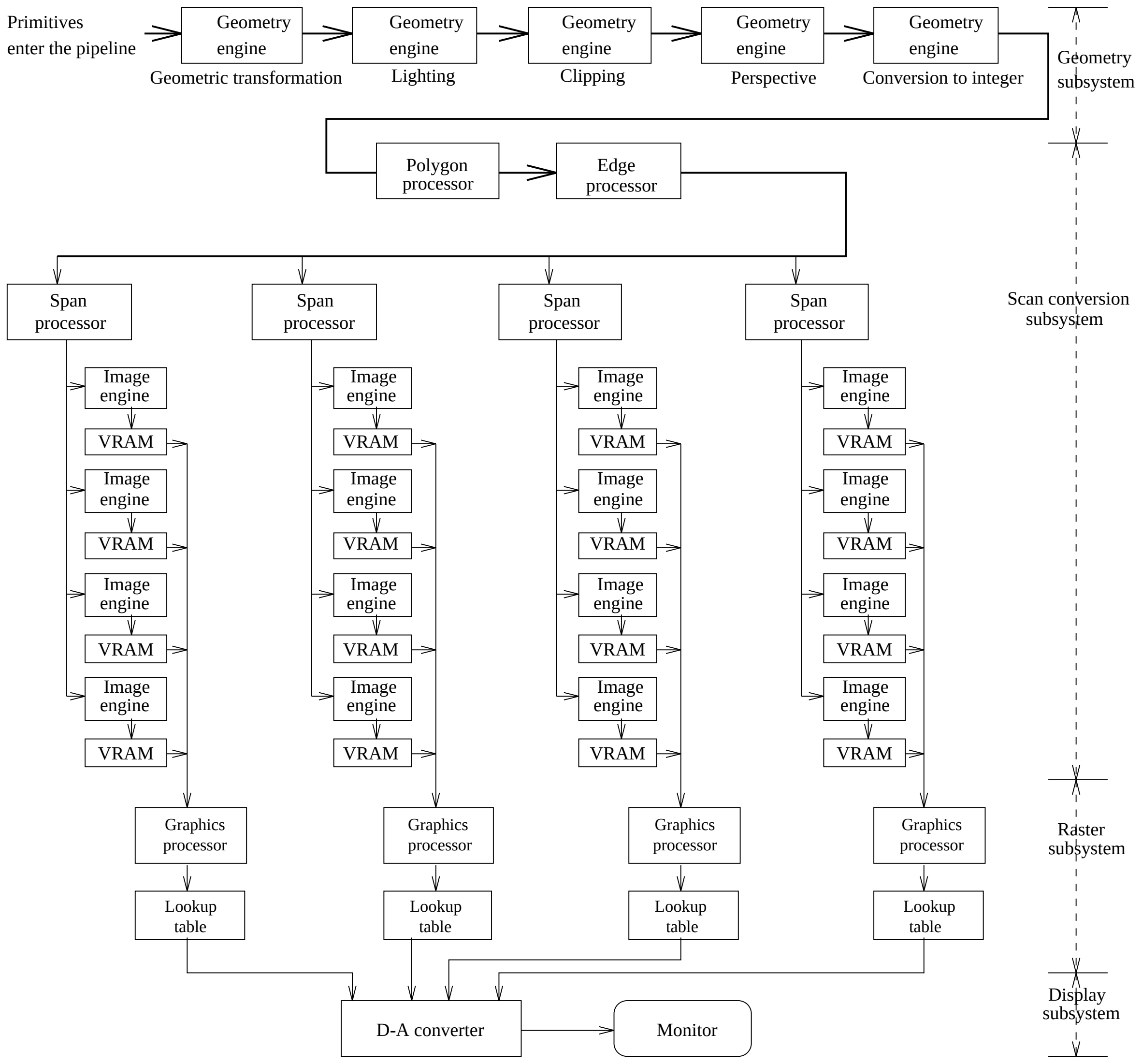


Figure 4.3: SGI GTX System Layout, adapted from Akeley 88 and Foley 90

Chapter 5

Implementation and Results

The essence of science: ask an impertinent question, and you are on the way to a pertinent answer.

- Jacob Bronowski

In order to test the predictive rendering algorithm, we implemented it on SGI workstations, running IRIX 5.3. The GL graphics language provided a good deal of software support.

5.1 Design Decisions

In this section, we describe some of the design decisions and problems that we encountered while implementing predictive rendering. These design choices are specific to our hardware, and its graphics software support. The discussion is included to give other prospective implementers an idea of potential roadblocks when using the SGI system, or other similar machines.

5.1.1 Organization of Scratch Framebuffers

There are several possible options for organizing the scratch framebuffers. The two main competing methods essentially either placed framebuffers in separate windows, or grouped them together into one large window. We eventually decided to open framebuffers as a single, large window and subsequently divide them up through the use of viewports. GL keeps separate context information for each window, including a transformation stack. With multiple windows (and stacks), a hierarchical data structure requires updating more than one matrix stack per primitive. For example, if primitive A is transformed and drawn in framebuffer A, any child primitive B must also be transformed by A's motion before being transformed and drawn into framebuffer B. By keeping all the scratch buffers as a single window, we avoid having to maintain transformation stacks for individual windows. This is cleaner, quicker, and more intuitive.

Using a single large window also reduces other types of overhead. Lights, textures, viewpoints, and materials only need to be defined and changed once, rather than for each scratch buffer.

5.1.2 Visibility of Scratch Framebuffers

We should not need to show the scratch framebuffers at all, except as an intermediate step for understanding. Unfortunately, the SGI's framebuffer organization turns out to be the biggest roadblock to the implementation of predictive rendering. The main problem is that the GL library requires actual screen space for a depth buffered framebuffer. This means that all scratch buffers (and the main window) must physically fit on the screen, without overlapping. It also implies that all scratch buffers must always be open on screen.

Practically, this is an extremely annoying restriction. First, this means that predictive rendering is constrained to window sizes that are small enough to all fit on the screen. Second, N is now implicitly restricted by screen size. We can squeeze a little extra space out by using the front and back buffers as separate scratch buffers, doubling the number of levels we need, but the principal of framebuffer restriction remains. Thirdly, we would like to avoid having the scratch buffers visible. A visible scratch buffer is confusing, detracting, and breaks the abstraction that predictive rendering is almost exactly the same as regular rendering. Even more annoying - if we use both front and back buffers as scratch areas, we have to make sure we don't draw anything else on top of a window that is not redrawn on each frame. Using only the back buffer for scratch protects against this, as long as the scratch buffers are not covered with another doublebuffered window.

There is nothing special about predictive rendering that prevents these problems from being overcome. On some systems (not ours) solutions already exist. Some systems offer additional framebuffer hardware that allows more screens, and thus hidden scratch buffers. Higher-end machines, such as the Reality Engine and Impact, also offer quadbuffering, allowing more framebuffer space. Extra framebuffer space allows scratch buffers to be hidden. On systems that perform depth buffering in software, one can easily expand framebuffer space by expanding memory. Virtual framebuffers can be added almost without limit, and can also be nicely hidden from the user's view. This model matches a great many types of computers.

The only real restriction is on systems where depth buffering is performed by a scarce amount of specialized hardware. Unfortunately, this fits the description of our current implementation hardware. We attempt to get around this by restricting our window sizes to $\frac{1}{4}$ of the screen (about video size). Using one of the scratch framebuffers as our master buffer, this allows up to four levels using single buffers, and seven levels using doublebuffering (assuming the master buffer cannot be doubled up). In order to remove the scratch framebuffers from the screen, we set the overlay planes to block out everything but the master window. This removes most of the unsightly temporary work, while still allowing temporary drawing.

5.1.3 Feedback

One of the big advantages of predictive rendering is that it very closely fits existing graphics pipelines. We initially hoped to use existing hardware to produce the prediction, taking advantage of shortcuts and speedups built into the system.

The GL system provides direct access to the graphics pipeline through a "feedback" mechanism. In feedback mode, transformed vertices sent through the pipeline are placed in a buffer, rather than being drawn on the screen. We can look at this buffer to determine how an initial vertex was transformed. Unfortunately, the feedback mechanism sends each vertex through the complete transformation pipeline - including all the steps that we hoped to avoid, such as lighting and clipping. This makes feedback much slower than we would like. We don't need or want the system to do as much work as it does. It is easy to envision a slight change to feedback, where transformed vertices are sent through a very abbreviated pipeline before being placed into the buffer. The required hardware already exists completely; some flag would be required to tell the system to skip over certain steps.

GL's feedback mechanism clips out boundary points. In addition to being unnecessary prediction work, it causes confusion when determining the past locations of a partially clipped polygon. There is no way to tell if an off-screen vertex has moved or not, because it is always clipped out and not placed in the output buffer. A vertex moving off-screen requires the rest of the on-screen vertices to be redrawn, because the polygon will have changed projected shape. We can artificially expand the size of the clipping window, but the the system performs unneeded clipping tests that always fail to cull anything. If the clipping step is skipped, we can determine if out-of-bounds points move or not, and nicely decide if the rest of the polygon needs redrawing.

The feedback system also requires significant pipelining of primitives - that is, we need to send a large number of vertices into the feedback loop before looking for any results. Separating the prediction input and results by a number of primitives makes the implementation harder. If we had a short-circuited graphical pipeline, we would still have this problem, but the greatly abbreviated number of steps would make it much less severe.

Instead of using feedback, we implemented the transformation steps in software. Prediction is performed by the main CPU, avoiding the graphics pipeline altogether. This is not really very satisfactory, because we cannot use the SGI's high performance graphics hardware. However, it is really the only option without better access to the pipeline.

Thus, all our timing tests operate with the knowledge that we could do a lot better with a few simple modifications that allow us to fully use the graphics pipeline. Unfortunately, it also destroys any chance we have to confirming our theoretical numbers, because we are now comparing graphics CPU cycles vs. main CPU cycles. That is, the prediction is actually done by the main CPU, while the rendering is sent into the pipelined, optimized, graphics subsystem.

5.1.4 Depth Compositing

The SGI supports a number of pixel copying mechanisms, allowing direct draws into both the framebuffer and the depth buffer. Unfortunately, there is no direct way to do an image-to-image copy (with depth compositing) using built-in GL commands. This seems like something of an oversight on the designers part, because of the wide number of uses for image depth compositing.

The best way to actually do a depth composite is to use a GL construct called a "stencil". A stencil is an entire bitplane that keeps track of the result of a previous draw. On the first pass, we draw one image's depth values into the final depth buffer. The stencil is set up to mark every location where the new depth value is closer to the viewer than the old depth value. On the second pass, we draw the color image, where pixels are drawn conditionally depending upon the results of the stencil. This depth compositing method takes only a few lines in GL, and is faster than looping through both images, testing pixels individually.

Unfortunately, we did not implement the depth compositing method described in Section 3.3.1. This would have significantly reduced the overheads caused by depth compositing.

5.1.5 Selection of N

At the start of predictive rendering, we allocate the maximum number of framebuffers allowable for a given window size. Then, depending on the actual N and k the user chooses, we only use some of the available framebuffers. Allocating everything at once prevents dynamically allocating framebuffer space on the fly. This makes implementation slightly easier, but reduces the amount of free memory available during execution. In applications that use a wide range of k it may be better to allocate framebuffers on the fly.

5.2 Experimental Results

In this section, we tested the predictive rendering algorithm in a number of cases, with several goals in mind. First, we wanted to confirm that predictive rendering's actual strengths and weaknesses matched our theoretical musings. Secondly, we wanted to quantify the cost of predictive rendering based upon an actual (although not ideal) hardware and software configuration. Finally, (and perhaps most importantly) we wanted to show how predictive rendering can improve rendering speeds dramatically.

It must be strongly noted beforehand that predictive rendering is ultimately based upon the amount of temporal coherence between frames. We can easily make an artificial example with little or no speedup, just as easily as we can create a million-times improvement. We test our algorithm with several levels of detail and temporal coherence, so that the reader will be left with an impression of the algorithm's potential, if not exact numbers. It is also important to remember that timing results depend on parameters like the number of lights, texturing, attenuation, and material property models. Thus, it is not particularly valid to predict an experimental break-even point for anything but the most specific of cases.

5.2.1 Testing the Analysis Scenarios

We start by testing out the scenarios described in the analysis section. Objects are given a random motion within an enclosing volume. Objects leaving one volume edge "wrap around" to the opposing edge. There are two important end cases. The first end case is the 100% situation with no temporal coherence, simulated by giving all objects a very large motion component. On each step, we expect all or almost all objects to move well beyond the movement threshold. If we compare the timings with and without predictive rendering, we can get an idea of the predictive penalty. This is the slowdown we must tolerate if we are checking for temporal coherence in a scenario where there is none.

The other end case is the 1% case, where all but one percent of the objects remain stationary. Although 1% is essentially arbitrary, it represents the other extreme - a very high temporal coherence. Our timing tests will indicate how much speedup the predictive rendering provides us.

There are a huge number of intermediate cases, with varying numbers of objects moving at various speeds. We will choose three intermediate forms, where 75%, 50% and 25% of the total number of objects remain stationary. The number of objects moving is not really as important as the idea of gradual speedup as temporal coherence improves. Appendix A contains the complete set of experimental conditions for each of the scenarios. Note that these parameters directly affect the possible speedups. Timing tests were run on an unloaded system.

Table 5.24 shows the results for the "random" scenario variants. The "control" case shows the timing cost without predictive rendering. Results are expressed in real-time seconds, and are thus affected by slight load changes. Figure 5.1 shows the random scenario.

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	66s	66s	58s	63s	61s
$N = 2$, $k = 2$	69s	65s	59s	52s	45s
$N = 4$, $k = 2$	67s	61s	51s	41s	31s
$N = 8$, $k = 2$	68s	60s	47s	37s	24s
$N = 16$, $k = 2$	68s	59s	47s	34s	22s
$N = 32$, $k = 2$	69s	60s	46s	34s	22s

Table 5.24: Test Results - Random Scenario Raw Times

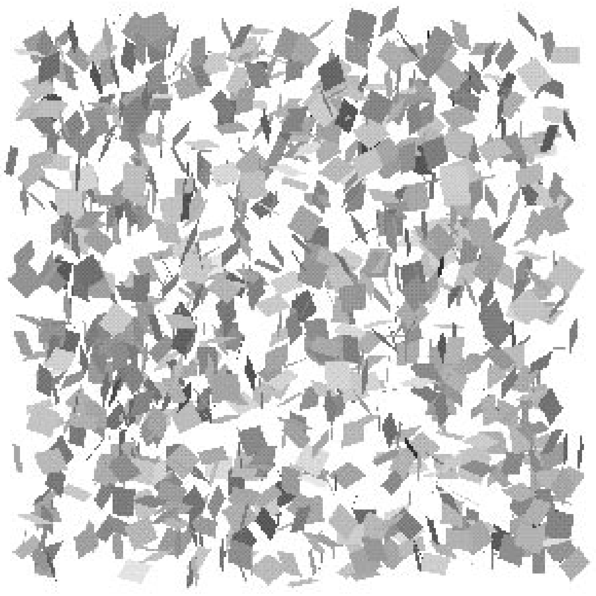


Figure 5.1: Random Scenario

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	1.00	1.00	1.00	1.00	1.00
$N = 2$, $k = 2$	0.95	1.01	0.98	1.21	1.35
$N = 4$, $k = 2$	0.98	1.08	1.13	1.53	1.96
$N = 8$, $k = 2$	0.97	1.10	1.23	1.70	2.54
$N = 16$, $k = 2$	0.97	1.11	1.23	1.85	2.77
$N = 32$, $k = 2$	0.95	1.10	1.26	1.85	2.77

Table 5.25: Test Results - Random Scenario Speedups

These results are perhaps better shown as speedup factors, as shown in Table 5.25. As seen from this Table, predictive rendering allows a best case improvement of 2.77$\times$ faster than before. This is fairly low, given the large amount of temporal coherence. The main reason for this low speedup is the one-to-one correspondence between prediction primitives and actual vertices. This scenario contains the largest amount of prediction work of all the scenarios examined here. Speedups aside, the most important numbers are really the worst cases. In situations where there is absolutely no temporal coherence whatsoever, predictive rendering slows the system down by at most 5%.

We can use the predictive rendering software libraries to keep track of how many polygons avoid the graphics pipeline. We can expresses this as as a percentage of the total number of polygons sent to the pipeline, as shown in Table 5.26. Note that the number of polygons skipped only equals the total percentage of stationary polygons when N is equal to the total number of frames.

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	100.00%	100.00%	100.00%	100.00%	100.00%
$N = 2$, $k = 2$	0.00%	12.50%	25.00%	37.50%	49.50%
$N = 4$, $k = 2$	0.00%	18.75%	37.50%	56.25%	74.25%
$N = 8$, $k = 2$	0.00%	21.75%	43.50%	65.25%	86.13%
$N = 16$, $k = 2$	0.00%	23.25%	46.50%	69.75%	92.07%
$N = 32$, $k = 2$	0.00%	24.00%	48.00%	72.00%	95.04%

Table 5.26: Test Results - Percentage of Random Scenario Polygons Skipped

The random case is the worst of all the test scenarios, because the cost of prediction is the highest. The number of predictive primitives is the same as the number of actual graphical primitives. Even so, we can obtain savings with a small amount of temporal coherence.

Due to the limitations of our implementation, the depth compositing costs are a fairly high proportions of the total cost. We can show this by running the random scenario with a much larger number of polygons (50,000). Now, the depth compositing cost is amortized over a long time frame, and our speeds are increased. Table 5.27 shows the speed-up factors for 50,000 polygons. The more polygons we use, the better the results will be. We have not implemented on-the-fly compositing (described in Section 3.3.1) - a future improvement which should significantly reduce the depth compositing overhead.

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	1.00	1.00	1.00	1.00	1.00
$N = 2$, $k = 2$	0.93	1.09	1.21	1.48	1.56
$N = 4$, $k = 2$	0.93	1.15	1.43	1.96	2.54
$N = 8$, $k = 2$	0.92	1.18	1.54	2.30	3.63
$N = 16$, $k = 2$	0.91	1.19	1.59	2.48	4.46
$N = 32$, $k = 2$	0.89	1.18	1.56	2.49	4.69

Table 5.27: Test Results - Random Scenario Speedups (50,000 Polygons)

Table 5.28 shows the speedup factors for the "blocks" scenario variants. The control time

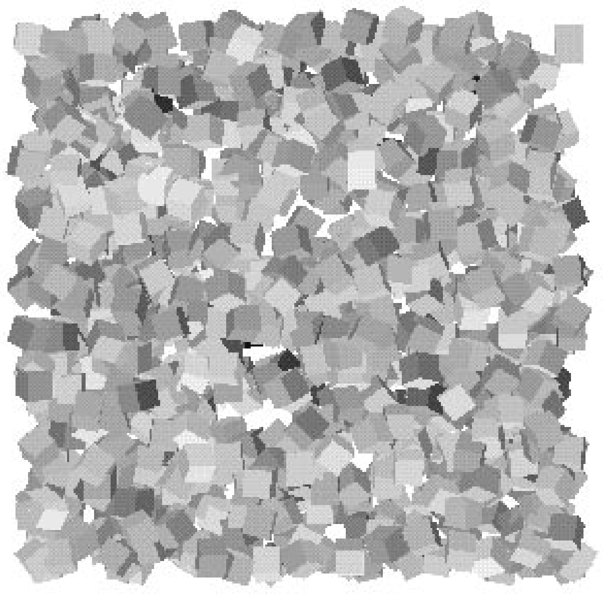


Figure 5.2: Blocks Scenario

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	1.00	1.00	1.00	1.00	1.00
$N = 2$, $k = 2$	0.87	0.96	1.07	1.24	1.46
$N = 4$, $k = 2$	0.87	1.03	1.24	1.60	2.34
$N = 8$, $k = 2$	0.87	1.06	1.34	1.87	3.15
$N = 16$, $k = 2$	0.87	1.04	1.40	2.06	3.88
$N = 32$, $k = 2$	0.87	1.06	1.40	2.06	4.04

Table 5.28: Test Results - Blocks Scenario Speedups

was 101 seconds. Figure 5.2 shows the blocks scenario. The “block” case has a smaller number of predictive primitives, and so performs better than the “random” case. Again, our worst case is not particularly prohibitive compared to the possible best cases. The percentage of polygons skipped is the same as the shown in Table 5.26.

Table 5.29 shows the results for the “particles” scenario. The control times are on the order of 23-24 minutes. Figure 5.3 shows the particles scenario. The “particles” scenario starts to show some extremely large speedups, mainly because a single culled predictive primitive encompasses a non-trivial number of polygons. In the best case, we can draw a 23.6 minute scene in a minute and a half - about 6.5% of the previous time. Table 5.29 shows the results for

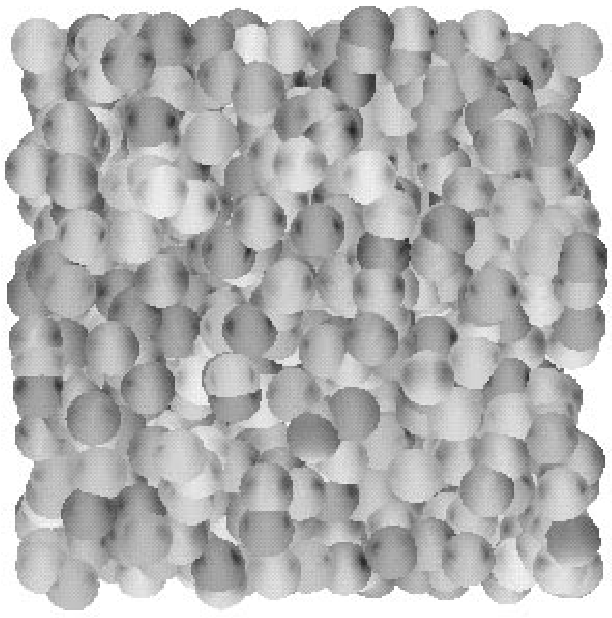


Figure 5.3: Particles Scenario

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
control	1.00	1.00	1.00	1.00	1.00
$N = 2$, $k = 2$	0.97	1.09	1.22	1.54	1.80
$N = 4$, $k = 2$	0.96	1.16	1.45	2.17	3.47
$N = 8$, $k = 2$	1.06	1.20	1.67	2.75	6.45
$N = 16$, $k = 2$	0.95	1.24	1.67	3.13	11.27
$N = 32$, $k = 2$	1.00	1.25	1.72	3.33	15.11

Table 5.29: Test Results - Particles Scenario Speedups

the "particles" scenario. The control times are on the order of 23-24 minutes. Figure 5.3 shows the particles scenario. We can compare the best case particle scenario with the theoretical maximum improvement for $N = 32$, $k = 2$, and the absolute maximal improvement (where N is the total number of frames). Results are shown in Table 5.30.

Table 5.31 shows the results for the "molecules" scenario. Control time was about 45 minutes. Figure 5.4 shows the molecules scenario. The "molecules" scenario shows how predictive rendering can drastically and dramatically improve rendering costs for scenes with large amounts of temporal coherence. The worst case costs are essentially negligible ($< 2\%$) compared to the potential improvements (13% - 1851%). This emphasizes how important it is to

Variant	**100%**	**75%**	**50%**	**25%**	**1%**
$N = 32$, $k = 2$	1.00	1.25	1.72	3.33	15.11
best $N = 32$, $k = 2$	1.00	1.31	1.92	3.57	20.16
absolute	1.00	1.33	2.00	4.00	100.00

Table 5.30: Test Results - Comparision of Particles Scenario Maximal Speedups

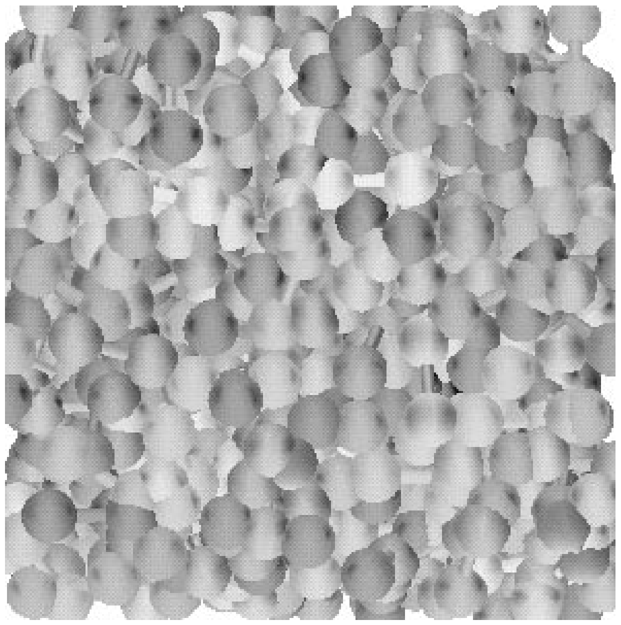


Figure 5.4: Molecules Scenario

group complex objects into simple prediction primitives.

On our target system, the depth compositing is a real bottleneck. Thus, the speedup depends partly on how many depth composites must be done. The number of composites depends largely on our choices of N and k. We perform timing tests for our scenarios using various N and k values. A fixed N and large k means fewer intermediate nodes. This is more efficient for cases where temporal coherence is largely separated into fast moving vs. stationary objects. This basically describes our analysis scenarios, so we expect a large k and large N to be the most efficient. These improvements are not really reflected in these scenarios very well, because the code optimizations avoid depth compositing unused frame-buffers - basically making all choices for k about the same. In this scenario, this means the k timing variations will only reflect other types of overhead.

Variant	100%	75%	50%	25%	1%
control	1.00	1.00	1.00	1.00	1.00
$N = 2, k = 2$	0.98	1.13	1.29	1.76	1.85
$N = 4, k = 2$	0.99	1.25	1.70	2.25	3.69
$N = 8, k = 2$	0.99	1.40	1.74	2.73	6.80
$N = 16, k = 2$	1.06	1.33	1.72	3.31	11.43
$N = 32, k = 2$	1.02	1.29	1.88	3.86	18.51

Table 5.31: Test Results - Molecules Scenario Speedups

5.2.2 Testing a Scene with Changing Motion Thresholds

In this test, we vary the motion threshold δ to give various levels of accuracy and speed. To show off the effect, we slow down the polygon speeds as described in Appendix A. This also demonstrates the effectiveness of varying k. We use the blocks scenario and three different $\delta = 0.01, 0.5, 1.0$. Table 5.32 gives speedup factors for the $\delta = 0.01$ scenario. Table 5.33 gives times for the $\delta = 0.5$ scenario. Table 5.34 gives times for the $\delta = 1.0$ scenario.

control	1.00	1.00	1.00	1.00	1.00
$N = 2, k = 2$	0.87	0.95	1.09	1.24	1.50
$N = 4, k = 2$	0.87	1.02	1.25	1.62	2.31
$N = 8, k = 2$	0.87	1.05	1.36	1.87	3.29
$N = 16, k = 2$	0.87	1.05	1.41	2.06	3.77
$N = 32, k = 2$	0.86	1.05	1.41	2.10	4.25
$N = 4, k = 4$	0.88	1.01	1.27	1.65	2.37
$N = 8, k = 8$	0.88	1.06	1.37	1.90	3.40
$N = 16, k = 16$	0.86	1.06	1.43	2.10	3.92
$N = 32, k = 32$	0.87	1.06	1.43	2.14	4.43

Table 5.32: Test Results - Blocks Scenario - delta = 0.01

When $\delta = 0.01$, the results are almost equivalent to those timed in Table 5.28. The δ is too small to exclude any moving polygons, so all speedups come from reducing the number of static objects drawn. As δ is increased to 0.5, the algorithm avoids drawing a few moving polygons. However, the number of extra polygons skipped is barely enough to balance the extra depth compositing costs now required by non-empty intermediate scratch buffers.

The real speed increase comes when $\delta = 1.0$. Now, a significant number of moving blocks can be occasionally treated as static. This is most noticeable in the 100% case, where the worst

control	1.00	1.00	1.00	1.00	1.00
$N = 2, k = 2$	0.95	1.04	1.14	1.29	1.47
$N = 4, k = 2$	0.92	1.07	1.29	1.60	2.21
$N = 8, k = 2$	0.90	1.06	1.32	1.87	3.00
$N = 16, k = 2$	0.91	1.07	1.34	2.02	3.51
$N = 32, k = 2$	0.90	1.06	1.36	2.06	3.77
$N = 4, k = 4$	0.86	1.02	1.26	1.65	2.37
$N = 8, k = 8$	0.87	1.05	1.36	1.90	3.40
$N = 16, k = 16$	0.87	1.07	1.31	2.10	3.92
$N = 32, k = 32$	0.87	1.07	1.42	2.14	4.43

Table 5.33: Test Results - Blocks Scenario - delta = 0.5

control	1.00	1.00	1.00	1.00	1.00
$N = 2, k = 2$	1.46	1.46	1.48	1.46	1.57
$N = 4, k = 2$	1.31	1.46	1.62	1.83	2.24
$N = 8, k = 2$	1.29	1.48	1.74	2.06	3.15
$N = 16, k = 2$	1.26	1.40	1.68	2.24	3.60
$N = 32, k = 2$	1.23	1.38	1.68	2.34	3.88
$N = 4, k = 4$	0.88	1.04	1.27	1.65	2.34
$N = 8, k = 8$	0.86	1.05	1.38	1.94	3.36
$N = 16, k = 16$	0.87	1.07	1.40	2.10	3.88
$N = 32, k = 32$	0.87	1.06	1.42	2.14	4.20

Table 5.34: Test Results - Blocks Scenario - delta = 1.0

case predictive rendering is much better than the control case.

For the cases where $k \neq 2$, there are no intermediate scratch buffers. This removes any potential for factoring out these slow moving polygons. Increasing N only adds to the additional depth compositing costs, as most moving polygons are only considered static for a frame or two. Because all polygons are updated every N frames, slow moving polygons are thresholded at different rates for different N. Thus, the number of polygons factored out changes depending on N.

Increasing the δ value from $\delta = 0.01$ to $\delta = 1.0$ resulted in a significant improvement in this scenario's speedups. Of special note is the worst case (no temporal coherence), where predictive rendering handily beat out the regular method. Users should decide for themselves if errors of up to a pixel (for up to N frames in duration) are visually allowable in their particular application.

5.3 Future Improvements

Predictive rendering uses temporal coherence to gain speedups. We can combine this with other types of coherence algorithms for even more speed. An obvious possible candidate is the hierarchical depth buffering performed by Greene [Greene 93]. This algorithm performs object space culling, taking advantage of depth coherence to avoid rendering already-occluded objects. The easiest approach is to implement this culling on a level-by-level basis. Because each level remains unchanged, we avoid having to worry about depth coherence changing across frames.

One of the predictive rendering algorithm's main strengths is its similarity to the algorithms in the traditional graphics pipeline. As described in Section 5.1.3, we are currently not taking advantage of this hardware support. With some fairly minor changes to the graphics pipeline, we could perform prediction in hardware. This would reduce our measured prediction timings, which are currently performed only in software.

Our algorithm currently does not perform on-the-fly compositing, as described in Section 3.3.1. This is a fairly simple extension, and would greatly reduce the compositing costs.

As noted in Section 5.1.2, the practical implementation of predictive rendering requires extra framebuffer space. This is currently handled less than satisfactorily for some SGI machines. We expect this problem will go away with newer models of computers.

Our current implementation uses the plan tree to mark how frames change over time. The first element in the plan tree is always "changing", because it always has been redrawn from the previous set of N frames. It is possible to think of the plan tree as marking frame spaces, rather than actual frames. Then, the first element in the plan tree determines if the second frame has changed from the (implicitly always changing) first frame. This produces a slightly more compact notation.

Predictive rendering takes advantage of prediction primitives to group large numbers of related polygons into a single prediction test. The grouping is currently done a priori, and requires some level of intelligence. In our current implementation, there is a significant amount of human intervention to determine prediction primitive groupings that are any more complex than polygons. We could improve the grouping algorithm to make this more automatic. Ideally, an input file of random polygons would produce an optimal predictive primitive grouping.

The predictive state is currently computed by looking at the motion of prediction primitives, and the static or changing state of textures, lights and the viewpoint. At the moment, the motion of the prediction primitives is the prime frame reuse criteria. It is interesting to think of a system that consistently extends prediction to other graphical properties. Such a system would base the static/changing decision on a number of "modules", each responsible for a graphical feature, like texture, lighting, viewpoint, atmospheric properties, etc. For example, we currently decide that a complete redraw is required for any texture pattern modification, of whatever magnitude. One might include a "texture predictor" that tracks a "texture δ" and watches to see if texture U and V coordinates have changed enough since some previous frame. The results of the texture predictor would be integrated with the results of the "viewpoint predictor", which determined if the effect of a viewpoint modification was large enough to be noticeable. Highlights could be tracked with a highlight predictor primitive, based upon light motion in world space. It is likely that (initially, at least) many of these predictors would be very simple: the predictor changes if the predictor's property changes. However, as algorithms are developed to predict and bound new properties, they can easily be added to our prediction paradigm.

Chapter 6

Conclusions

A conclusion is the place where you got tired of thinking.

- Arthur Bloch

This thesis describes a method to improve polygonal rendering rates through the use of predictive rendering. The future behavior of polygons can be quickly predicted by using the first part of the graphics pipeline. Using a polygon's future motion, we can factor out static and slow-moving polygons. These polygons can be rendered less than once per frame. The final result is created by depth compositing polygons rendered at different rates. By avoiding rendering expensive, detailed polygons, we more than compensate for our initial prediction cost.

Predictive rendering is most useful for scenes containing a high-proportion of slow moving or static polygons, or very small polygons that change modestly when viewed from multiple directions. A excellent example is a forest of trees blowing in the wind. Other uses include particle systems, facial animation, or any animation that contains objects moving through a complex static background.

The algorithm is simple to implement, and can be turned on or off as required. It also extends to parallel composition hardware. Predictive rendering is closely linked to the existing graphics pipeline. With only a few minor changes, the actual hardware could be used to perform predictive rendering, greatly improving the speed of our current implementation. Predictive rendering becomes more advantageous as more features (such as displacement mapping, bump mapping, more complex lighting functions, etc.) are added to the polygon rendering phase.

Disadvantages include extra framebuffers, potentially irregular frame rates, and the cost of unnecessary prediction work for rapidly changing databases. Predictive rendering is also less useful for fly-throughs, where the eyepoint can change drastically between frames. We can achieve some savings by reducing the number of times we render slowly moving distant objects. Most close objects will change too much to be culled. However, predictive rendering is still useful. We can turn predictive rendering off for active motion, and then re-activate it when the user stops. Predictive rendering is cheap enough that it causes only a minor slowdown during parts of the animation with no temporal coherence.

Human animators use temporal coherence to drastically reduce the total drawing work-load. We hope that predictive rendering will similarly reduce the computational work-load for graphics workstations. This will allow faster animations and graphical models of even greater complexity.

Appendix A

Experimental Conditions

This section details the experimental conditions for the various timing scenarios. These conditions are important because of the large number of variables involved in temporal coherence speedup. We wish to be frank about what conditions provided what speedups.

A.1 Analysis Scenarios

All four of the analysis scenarios ("random","block","particles" and "molecules") are lighted identically. Each scenario uses a local viewer, with four local, randomly positioned, lights. Lights are randomly attenuated with distance, using SGI's linear model only. Objects are as described in Chapter 4, and are Gouraud shaded with no texturing. Individual material properties are assigned randomly.

Objects are given motion properties randomly chosen from the ranges shown in Table A.35, and are allowed to move within a cube. As objects travel off the edge of the cube, they are transported to an opposite edge. All objects start at a randomly chosen location within the cube. The movement threshold for these scenarios is $\delta = 0.01$ pixel. This non-zero value prevents roundoff errors for otherwise identical results.

All scenarios ran for a total of 100 frames. A full list of properties is provided in Table A.35.

A.2 Delta Scenarios

These three scenarios used the "block" scenario, with three different δ values. A full list of properties is provided in Table A.36.

Property	**Value**
computer	IRIS 4DXG
processor	134 MHZ IP22
objects	1000
frames	100
window size	300 x 300 pixels
cube side	20.0 units
lights	4 random, local
viewer	local
attenuation	linear
num materials	50
X trans	[-0.15 .. 0.15] units
Y trans	[-0.15 .. 0.15] units
Z trans	[-0.15 .. 0.15] units
X rot	[-4.0 .. 4.0] degrees
Y rot	[-4.0 .. 4.0] degrees
Z rot	[-4.0 .. 4.0] degrees
δ	0.01 pixels
particles p.p. radius	0.75 units
molecules p.p. radius	2.5 units

Table A.35: Analysis Scenario Experimental Motion Properties

Property	**Value**
computer	IRIS 4DXG
processor	134 MHZ IP22
objects	1000
frames	100
window size	300 x 300 pixels
cube side	20.0 units
lights	4 random, local
viewer	local
attenuation	linear
num materials	50
X trans	[-0.02 .. 0.02] units
Y trans	[-0.02 .. 0.02] units
Z trans	[-0.02 .. 0.02] units
X rot	[-1.0 .. 1.0] degrees
Y rot	[-1.0 .. 1.0] degrees
Z rot	[-1.0 .. 1.0] degrees
δ	0.01 pixels, 0.5 pixels, 1.0 pixels
particles p.p. radius	0.75 units
molecules p.p. radius	2.5 units

Table A.36: Delta Scenario Experimental Motion Properties

Appendix B

Table of Variables

Name	**Description**
δ	a threshold that decides if a transformed vertex has moved between frames
A_{light}	light attenuation factor
$\vec{C}_{ambient}$	an RGB vector that models ambient lighting
$\vec{C}_{diffuse}$	an RGB vector that models material diffuse reflection properties
$\vec{C}_{light}$	an RGB vector that models light color
$\vec{C}_{object}$	an RGB vector that models object color
$\vec{C}_{specular}$	an RGB vector that models material specular reflection properties
d	distance of the projection plane from the axis
f	a single frame in the range $[0...N-1]$
H	height of a screen, in pixels
$\vec{H}$	the vector halfway between the direction of the light source and the viewer
k	the number of children for each node in the plan tree
$K_{constant}$	constant parameter in the light attenuation equation
K_{linear}	linear parameter in the light attenuation equation
$K_{squared}$	squared parameter in the light attenuation equation
l	a level in the plan tree, where level 0 is the root
$\vec{L}$	direction vector from a light to a vertex
m	the number of prediction vertices in a prediction primitive
$M_{3D\mapsto 2D}$	one-step world-to-eyepoint transformation matrix
$M_{E\mapsto P}$	eye-to-projection plane transformation matrix
$M_{O\mapsto W}$	object-to-world coordinate system transformation matrix
$M_{W\mapsto E}$	world-to-eyepoint coordinate system transformation matrix
n	shininess material property
N	a small sequence of frames predicted into the future
$\vec{N}$	a vector normal to a surface
$S(p)$	the motion state for a primitive primitive p for all N frames
$S(p,f)$	the motion state for a prediction primitive p on frame f
$\vec{V}$	the vector from an object to the viewer
$\vec{V}'_m(p,f)$	vector of XYZ world coordinates for the m-th prediction vertex of prediction primitive p on frame f
$\vec{V}_m(p,f)$	vector of XY screen coordinates for the m-th prediction vertex of prediction primitive p on frame f
$\vec{V}_{m,x}(p,f)$	the X screen coordinate for the m-th prediction vertex of prediction primitive p on frame f
$\vec{V}_{m,y}(p,f)$	the Y screen coordinate for the m-th prediction vertex of prediction primitive p on frame f
$\vec{V}_{m,w}(p,f)$	the fourth coordinate for the m-th prediction vertex of prediction primitive p on frame f
W	width of a screen, in pixels

Table B.37: Description of Variables

Bibliography

[Akeley 88] Akeley, K. and Jermoluk, T. "High-Performance Polygon Rendering". *Computer Graphics (Proc. SIGGRAPH)*, 22(4):239–246, August 1988.

[Akeley 93] Akeley, K. "Reality Engine Graphics". *Computer Graphics (Proc. SIGGRAPH)*, 27:109–116, August 1993.

[Appel 68] Appel, A. "Some Techniques for Shading Machine Renderings of Solids". In *Proceedings of the Spring Joint Computer Conference*, pages 37–45, 1968.

[Badt 88] Badt, S. "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing". *Visual Computer*, 4(3):123–132, September 1988.

[Bergeron 86] Bergeron, P. "A General Version of Crow's Shadow Volumes". *IEEE Computer Graphics and Applications*, 6(9):17–28, September 1986.

[Bishop 94] Bishop, G., Fuchs, H., McMillan, L., and Scher-Zagier, E. "Frameless Rendering: Double Buffering Considered Harmful". *Computer Graphics (Proc. SIGGRAPH)*, 28:175–176, July 1994.

[Bui-Tuong 75] Bui-Tuong, P. "Illumination for Computer Generated Pictures". *Communications of the ACM*, 18(6):311–317, June 1975.

[Chapman 90] Chapman, J., Calvert, T., and Dill, J. "Exploiting Temporal Coherence in Ray Tracing". In *Proceedings of Graphics Interface*, pages 196–204. Canadian Information Processing Society, 1990.

[Chapman 91] Chapman, J., Calvert, T., and Dill, J. "Spatio-Temporal Coherence in Ray Tracing". In *Proceedings of Graphics Interface*, pages 101–108. Canadian Information Processing Society, 1991.

[Chin 89] Chin, N. and Feiner, S. "Near Real-time Shadow Generation Using BSP Trees". *Computer Graphics (Proc. SIGGRAPH)*, 23(3):99–106, July 1989.

[Crow 77] Crow, F. "Shadow Algorithms for Computer Graphics". *Computer Graphics (Proc. SIGGRAPH)*, 11(2):242–247, July 1977.

[Deering 92] Deering, M. "High Resolution Virtual Reality". *Computer Graphics (Proc. SIGGRAPH)*, 26(2):195–202, July 1992.

[Duff 85] Duff, T. "Compositing 3-D Rendered Images". *Computer Graphics (Proc. SIGGRAPH)*, 19(3):41–44, July 1985.

[Foley 90] Foley, J., van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Friedmann 92] Friedmann, M., Starner, T., and Pentland, A. "Device Synchronization Using an Optimal Linear Filter". In *1992 Symposium on Interactive 3D Graphics, Computer Graphics*, volume 26, pages 57–62. ACM, March 1992.

[Fuchs 79] Fuchs, H., Kedem, Z., and Naylor, B. "Predetermining Visibility Priority in 3-D Scenes". *Computer Graphics (Proc. SIGGRAPH)*, 13(2):175–181, August 1979.

[Glassner 88] Glassner, A. "Spacetime Ray Tracing for Animation". *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

[Greene 93] Greene, N., Kass, M., and Miller, G. "Hierarchical Z-Buffer Visibility". *Computer Graphics (Proc. SIGGRAPH)*, 27:231–238, August 1993.

[Haeberli 90] Haeberli, P. and Kurt, A. "The Accumulation Buffer: Hardware Support for High-Quality Rendering". *Computer Graphics (Proc. SIGGRAPH)*, 24(4):309–317, August 1990.

[Halas 59] Halas, J. and Manvell, R. *The Technique of Film Animation.* Communication Arts Books, Hastings House, New York, 1959.

[Halas 90] Halas, J. *The Contemporary Animator.* Focal Press, Boston, MA, 1990.

[Hill 90] Hill, F. *Computer Graphics.* MacMillan Publishing, New York, New York, 1990.

[Hubschman 81] Hubschman, H. and Zucker, S. "Frame-to-Frame Coherence and the Hidden Surface Computation". *Computer Graphics (Proc. SIGGRAPH)*, 15(3):45–54, August 1981.

[Jevans 92] Jevans, D. "Object Space Temporal Coherence for Ray Tracing". In *Proceedings of Graphics Interface*, pages 176–183. Canadian Information Processing Society, 1992.

[Molnar 92] Molnar, S., Eyles, J., and Poulton, J. "PixelFlow: High Speed Rendering Using Image Composition". *Computer Graphics (Proc. SIGGRAPH)*, 26(2):231–240, July 1992.

[Porter 84] Porter, T. and Duff, T. "Compositing Digital Images". *Computer Graphics (Proc. SIGGRAPH)*, 18(3):253–259, July 1984.

[Reeves 87] Reeves, W., Salesin, D., and Cook, R. "Rendering Antialiased Shadows with Depth Maps". *Computer Graphics (Proc. SIGGRAPH)*, 21(4):283–291, July 1987.

[Schumacker 69] Schumacker, R., Brand, R., Gilliland, M., and Sharp, W. "Study for Applying Computer-Generated Images to Visual Simulation". In *AFHRL-TR-69-14*. U.S. Air Force Human Resources Laboratory, 1969.

[Shaw 89] Shaw, C., Green, M., and Schaeffer, J. "Anti-Aliasing Issues in Image Composition". In *Proceedings of Graphics Interface*, pages 113–119. Canadian Information Processing Society, May 1989.

[Sproull 74] Sproull, R., Sutherland, I., and Schumacker, R. "A Characterization of Ten Hidden-Surface Algorithms". *ACM Computing Surveys*, 6(1):1–55, March 1974.

[Sutherland 74] Sutherland, I. and Hodgman, G. "Reentrant Polygon Clipping". *Communications of the ACM*, 17(1):32–42, January 1974.

[Williams 78] Williams, L. "Casting Curved Shadows on Curved Surfaces". *Computer Graphics (Proc. SIGGRAPH)*, 12(3):270–274, August 1978.

Glossary

coherence - Coherence is the amount of similarity between several or many objects that are near to each other in time or space. There are many types of coherence including depth coherence, area coherence, and temporal coherence. Things with high coherence are closely related.

compositing - Combining several individual image layers into a final result image. The term "compositing" is often taken to mean a straight one-over-the-other layering, without occlusion testing between layers. To avoid confusion, we have tried to use "depth compositing" to refer to compositing with occlusion testing.

depth compositing - Combining several individual image layers and their associated depth values, so that the final image contains a consistantly occluded image.

depth buffer - A depth buffer stores the depth values of individual pixels in a rendered scene. As objects are rendered to the framebuffer, the depth buffer performs depth comparisons to determine to object visibility. The depth buffer is also commonly called a Z-buffer.

feedback - In some graphics systems, the graphics library allows intermediate access to the computations contained in the graphics pipeline. This intermediate feedback data can be queried. This allows prediction to be performed using the actual graphics pipeline hardware.

graphics pipeline - The graphics pipeline is the sequence of steps required to transform a single graphics primitive from the user's database into a screen image. It simultaneously refers to the process of transformation, and the special purpose hardware often used to do the work faster.

graphics primitive - A graphics primitive is the general term for low-level geometrical objects supported by the graphical drawing packages. Common graphics primitives include points, lines, and polygons.

grouping phase - In the grouping phase, graphical primitives are grouped together and assigned to prediction primitives. This allows large groups of polygons to be tracked with a single prediction. The grouping phase occurs once, before the animation begins.

k-tree - A k-tree is a plan tree with k branches per node. The most common type of k-tree is a binary tree, with $k = 2$. The larger the value of k, the more efficient predictive rendering is at using scratch framebuffers, and the less effective it is at factoring out motion which is neither always moving nor always stationary.

local viewer - A local viewer is used in the computation of many common lighting models. It assumes that the viewpoint is a finite distance away from the objects being viewed. It is a more stringent and realistic viewing model than the infinite viewer, which assumes that the viewing position is infinitely distant. The infinite model is faster than the finite model, because it allows some lighting model simplifications.

master framebuffer - The master framebuffer is the main viewing window. In a truly seamless predictive rendering system, the user only sees the master framebuffer window, and does not know that other types of framebuffers exist in the background.

motion threshold - A thresholding value, compared against the differences in a vertex's position between sequential frames. A difference larger than the motion threshold δ means that the vertex is considering "moving" this frame. The motion threshold is expressed in pixels, but converted to viewplane coordinates to allow fast comparison in projection space.

multiprocessor system - A multiprocessor system is one where more than one CPU works on a single graphics primitive during its time in the graphics pipeline. Multiprocessor systems usually divide the pipeline into phases, and assign a CPU per phase. Note that this a different definition than is common in much of computer science literature.

plan tree - A plan tree is a conceptual grouping of primitives by amount of predicted motion. The basic plan tree is essentially a binary tree, where objects are placed within the plan tree nodes depending on the amount of predicted motion. Objects that change between frames are placed on the leaves. Objects that never change are placed at the root. Objects that remain stationary for some sequence of frames are placed at an intermediate node as close to the root as possible. Traversing the in-depth first order passed through all objects, for each and every frame. The plan tree is a conceptual grouping - no actual tree needs to be built.

prediction phase - The prediction phase of predictive rendering looks ahead some small number of frames to predict the future motion of groups of polygons (prediction primitives) that all move together. The resulting record of future motion is used to factor out prediction primitives that need to be drawn at the same time. The prediction phase and the rendering phase alternate.

prediction primitive - A prediction primitive is a grouping of some number of polygons that all move along with a single origin. Prediction primitives are either coincident with the graphics polygons, or completely bounding the objects. Predicting the future motion of a prediction primitive implies that the motion of all enclosed polygons can also be predicted.

prediction vertices - Each prediction primitive contains a number of prediction vertices. Each vertex is transformed by the object's future motion, and compared against a previous position.

predictive rendering - Predictive rendering is a new method that reduces rendering times by taking advantage of frame-to-frame temporal coherence.

rendering phase - The rendering phase steps through a small set of frames, one by one. Individual polygons are drawn into scratch buffers according to the results of the rendering phase. After each frame is completed, all scratch buffers are composited together to form a final result. The rendering and prediction phases alternate.

scratch framebuffer - A scratch framebuffer is an extra framebuffer and z-buffer used to hold intermediately rendered results before compositing. Scratch framebuffers are updated at different rates, and do not have to be visible.

single processor system - A single processor system is one where a single CPU works individually on a single graphics primitive for the duration of that primitive's time in the pipeline. Note that this definition includes multiprocessor and massively parallel systems, as long as work is distributed one polygon per processor. Note that this a different definition than is common in much of computer science literature.

time slice - A time slice is the number of frames used to predict the future motion of prediction primitives. User interactions and framebuffer space often make a time slice a fairly small set of frames, but this is not necessarily so. We assume we know object motion over all of the frames in the time slice.

temporal coherence - Temporal coherence is the amount of visual similarity between consecutive frames in an animation sequence. Animations with a lot of temporal coherence contain many adjacent frames that look very similar. Sequences with very little temporal coherence contain adjacent frames that look almost totally different.