Viper: A Quasi-Real-Time Virtual-Environment Application

Richard L. Holloway

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

When creating a virtual-environment (VE) application, there is often a conflict between the complexity of the virtual environment and the update rate of the system. Typically, more complex environments are more useful and interesting, but may require more rendering work than the graphics system can perform at interactive rates. The *viper* application gives improved performance in a VE visualization application by allowing the user to trade off image quality for interactive performance. The user can specify a given update rate and *viper* will attempt to match that rate by simplifying the scene. The user can specify that he wants a full-complexity image (with a correspondingly slower update rate) at any time.

1. Introduction

Virtual-environments systems often use head-mounted displays (HMDs) in order to either immerse the user in a virtual world or superimpose virtual objects into the real environment. In both cases, the system needs to respond almost instantaneously to user movements, or the illusion will be degraded. The delay between the user's movement (e.g., a head rotation) and the update of the image on the HMD is known as *latency* or *lag*, and it has been shown to have detrimental effects on operator performance in numerous studies.

For complicated scenes or objects, image generation is often the largest contributor to the system latency. If the virtual objects contain tens of thousands of polygons, the graphics engine is often incapable of rendering stereo images at interactive rates. *Viper* is a VE application that allows the user to move around complex models in real time by simplifying the scene until the requested update rate is met or until the user demands a full-complexity view. The premise is that for some types of visualization (as opposed to simulation), much of the user's time in the virtual environment is spent getting into position in order to examine some virtual object or objects, and during this navigation time, full-complexity models are not necessary. Thus, the application may be able to reduce the complexity of the virtual objects in order to give good response while the user is moving, and then, once in position, the user can specify (via a button press or command) that the system should use the full-complexity, slower version of the model.
2. Prior Work

The idea of trading off image quality for response time is not new. [Bergman et al. 86] used a technique called adaptive refinement for manipulating complex datasets on a workstation. Here, the user is moving the viewpoint with an input device such as a joystick and viewing it on an ordinary CRT. Given this sort of system, it is possible to implement the system such that while the user is moving the joystick, simple images will be drawn in real time; when the user stops moving the joystick, successively better versions of the image will be drawn until either the user moves the joystick again, or the best-quality image has been generated.

The differences between viper and the adaptive refinement approach is that their focus was on using idle workstation cycles and not on guaranteeing a specified level of interactivity. While this technique is useful for improving interactive performance, it does nothing to guarantee performance. There is no notion of real time in this system, only a sequence of steps to be performed until they are finished or interrupted. The minimal image consisted of 10% of the image’s vertices displayed at whatever update rate the graphics engine could muster (which for the 8,029-vertex test case they used was about 1 Hz on a Masscomp workstation).

Also, one of the key assumptions of this work is that there is some way to tell when the user has paused to see a particular view. With mice and joysticks, one can set a “movement delta”- a minimum amount that the joystick value has to change before it will be detected as true movement rather than noise on the A/D channel. With a head-mounted display, the user's head is always moving slightly, so there is no real opportunity for improving the image while he holds still.

Another alternative that is commonly used in simulators is the "level of detail" approach [Clark 76]. A hierarchy of object descriptions of increasing detail is kept for each object and the system dynamically switches between them depending on the viewing parameters. While recent advances in retiling algorithms [Turk 92, Schroeder et al. 92] make this an attractive option for systems where realism or scene integrity is important, it is often impractical to generate many datasets of varying complexity for a model (such as a medical dataset) that will only be viewed a few times in a visualization system.

[Bergman et al.] gives a more complete listing of related work.

3. System Description

3.1 Overview

The basic idea for viper is to use image database management and a notion of real time to achieve a specified update rate. If the system is not meeting the requested update rate, it will throw out some of the polygons in the scene in order to reduce the workload on the graphics engine. If the system is doing better than the requested rate, it can add polygons back to the scene. The object is to remove polygons in such a way that the objects are still recognizable (although they may appear to be made of a see-through mesh) while reducing the load on the graphics system until the required update rate can be achieved.
Full-quality images can be requested by the user by pressing a “slow-motion button” on the hand-held input device: When the user presses this button, full-quality images are generated at whatever (slower) rate this requires. Releasing the button speeds things up again to the requested update rate.

Viper is built on top of a non-real-time version of Unix, so there is no low-level operating-system support for guaranteeing that deadlines will be met. Thus, viper uses an adaptive scheme to approximate real-time performance (hence the term “quasi-real-time”). The program monitors its performance on the previous frame and uses knowledge about that frame to decide how much to attempt for the current frame. Since the system is non-preemptive, there is clearly a risk of failure— the system load could change in unpredictable ways so that the current frame’s decision is always wrong. Without any constraints on the external environment, there is no way to avoid this possibility. In practice, however, viper is able to deliver the requested frame rate about 90% of the time.

3.1 Hardware

The image generation for viper is done by Pixel-Planes 5, a highly parallel raster graphics engine developed at UNC. Even though Pixel-Planes 5 can render over two million triangles per second, medical datasets with tens of thousands of polygons concentrated in a small area of the screen can overload the system’s renderers and bring its update rate down below interactive rates.

The host for Pixel-Planes 5 is a Sun 4 running SunOS release 4.1.1. The host receives the tracker data and computes the viewing matrices based on the latest tracker information and sends it to Pixel-Planes. It also handles the timing calculations for determining the current update rate.

The HMD is a VPL Eyephone Model 2, which is opaque and has wide field-of-view optics for spreading the images out over a wide area.

Tracking is done by a Polhemus 3SPACE magnetic tracker, whose latency is negligible when the graphics system is overloaded. Our 3SPACE runs at 30 Hz when tracking head and hand, with a latency of about 100 ms [Mine 92].

The hand-held input device is a hollowed-out billiard ball with a 3SPACE sensor mounted inside and two buttons mounted on the outside.

3.2 System Software

Viper is built on top of a suite of libraries developed at UNC for creating virtual environments [Holloway et al. 91]. There are libraries for communicating with the tracker, A/D devices, and for building and changing the virtual world.
The graphics library used in this system is PPHIGS, a local implementation of the PHIGS+ standard. Since PHIGS is a display-list oriented standard, virtual objects are stored in a display list which is traversed at run-time. Objects can be described in terms of hierarchies of structures, which are made of primitives such as polygons, lines, color changes, etc., and may include calls to execute other structures or even conditional returns, which allow early exit of a structure during traversal. This last feature is used extensively by this system for reducing the load on the graphics engine and is discussed more in the next section.

3.3 Seg: The Database Segmenter

The first task to be solved was how to tell the graphics system which primitives to render or leave out, since in normal operation everything is displayed if it is visible. With PPHIGS, the most efficient solution turned out to be to preprocess the archive files containing the structures and divide them into segments bounded by conditional return calls¹. The PPHIGS conditional_return call allows the display code to stop processing a structure based on a global system flag. An example follows.

```plaintext
structure example {
    polygon { <coords> };
    polygon { <coords> };
    polygon { <coords> };

    cond_return flag_value1
    polygon { <coords> };
    polygon { <coords> };
    polygon { <coords> };

    cond_return flag_value2
    polygon { <coords> };
    polygon { <coords> };
    polygon { <coords> };

    .
    .

};
```

When the example structure is traversed by the display code, each flag_value is logically ORed with a user-settable system flag, and the return is executed if the result is non-zero. Since there are 32 bits in this system flag, each structure can have 32 conditional returns inserted into it, allowing the display code to break out of it at any of these 32 points. This strategy allows the display code to go as deep into each structure as time permits while keeping the overhead low.

¹ While the conditional_return primitive may be specific to PPHIGS, implementation using other standards (such as OpenGL) should be possible using appropriate data structures for the segments and application-level control of which segments are displayed.
Thus, *seg’s* job is to parse the archive file and insert the conditional return statements. This is not as straightforward as it may seem, since structures can contain calls to other structures, instances of transformation matrices, color changes, and other such qualitatively important information. We really only want a quick exit at a point when all of the primitives after that point are polygons with the same color, transformation matrix, etc. (such groups of polygons will be referred to as *spans*). In addition, since polygons are usually grouped in a coherent fashion (i.e., polygon $i$ is physically near polygon $i+1$), we cannot just insert conditional returns into the list as is, or entire chunks of the object will just disappear altogether when the conditional returns are executed. If most of the polygons are small and are roughly equally important (as in the case of a medical dataset), the easiest way to resolve the problem is to evenly distribute the polygons from the span into the new segments; i.e., each segment should contain some polygons from each part of the span. This is what *seg* does, and it guarantees an even distribution of polygons in all segments.

The new segments are output into a new archive file in a form suitable for *viper*, whose task now is just to display the objects and determine how many segments should be displayed for each frame.

### 3.3 Viper

The *viper* application was derived from an existing program called *vixen* which was developed at UNC for flying around, grabbing, and scaling virtual objects in a virtual environment. *Vixen* has no way of guaranteeing interactive performance: If you load in a huge object, you may be forced to work in a virtual environment at one update per second. For that reason, the name *viper* stands for "Vixen with Interactive PERformance".

*Viper* takes *seg’s* output and puts it into a virtual environment for the user. Only the structures processed by *seg* have polygons removed from them at run-time; objects representing the environment and the user's hand are always displayed at full complexity. This amounts to a crude sort of priority scheme: Certain objects, such as those essential for interaction, are high priority and are therefore not affected by the polygon-reduction mechanism, and the other objects in the scene are treated as lower priority and are therefore subject to having polygons removed. Clearly, this scheme could be extended to more priority levels, an idea which is discussed in section 6.
The main function for *viper* looks something like this:

```c
main()
{

init();
read_archive_file();
create_world();

while ( ! done )
{
frameRate = calcFrameRate();

adjustment = evaluateLastFrame(frameRate, DesiredFrameRate);
adjustDisplayList(adjustment);
read_tracker();
handle_button_events();
update_displays();
}
}
```

*Viper* reads in the specified archive files (which are assumed to have been preprocessed by *seg*), adds the files’ structures to the display list, creates the virtual environment, and then enters the main interactive loop. In the loop, it calculates the frame rate for the previous frame, compares it to the desired frame rate, and then calculates an adjustment to the *level* (the number of segments displayed) based on the difference. Once the display list is adjusted, it does all of the usual things for a VE application: it reads the tracker, updates the head and hand positions in the environment, handles any button presses or keyboard commands, and finally, updates the displays (one for each eye).

The user interface is fairly simple: When the user runs the program, he can specify a frame rate that the system should try to achieve; he then enters the virtual environment and uses the left button on the hand-held input device for flying, and the right button for "slow motion".
4. Results

4.1 Overview

Overall, the results were good: The system met the requested rate of performance under most circumstances, and the simplified objects were recognizable and not overly distracting. Medical datasets were used for most of the study (see notes below on man-made objects). In general, when the user enters an environment with a large number of polygons (here over 10,000) after requesting an update rate of over 10 Hz, the system immediately begins reducing the level until the rate is met. Thus the first thing one notices at startup is that the virtual objects seem to be disintegrating, which is not as unsettling as it sounds (see section 4.2). Once the specified update rate is reached, the system typically falls into oscillation between two levels corresponding to frame rates just above and just below the desired rate, which means that a certain subset of the objects polygons flicker. This could easily be suppressed if it were found to be distracting, although a certain amount of flickering is inevitable for a dynamic session where the level must be changed frequently. Once the objects are simplified, it is quite easy to move to a good position for viewing an object and then press the slow-motion button to get a full-quality view of it. The time to change from the degraded object to the full object (or vice versa) is usually on the order of one second.

I collected data from several sample runs with the system to evaluate its performance. I tried both static (no appreciable head motion, and thus no major image complexity changes) and dynamic (significant load changes) sessions, both with and without the use of the slow motion button. I also varied the load from light loads (a few hundred polygons) to heavy loads (tens of thousands of polygons). The graphs and their explanations are at the end of this section; the results are summarized below.

4.2 Qualitative Observations

One of the most surprising and encouraging results was that for medical data sets, missing polygons actually seemed to enhance the perception of their shape. For organs such as lungs, the smooth shading and low resolution of current commercial HMDs makes it difficult to judge the shape of the object's surface. When a large fraction of the polygons were removed in order to simplify the image, the organs appeared as a loose mesh of polygons which actually seemed to bring out the shape characteristics better than the complete model. Granted, this is only a limited set of informal observations, but this result is promising and further study is encouraged. At the very least, the objects were still recognizable and useful in their degraded form.

A counterexample to this phenomenon, however, is the case of man-made objects, or any objects with large polygons. The display of a complex space-station model with viper was significantly less effective than that of the medical datasets. When large polygons disappear, it is disorienting, and when they flicker, it is quite distracting. Successful display of such objects would require either a level-of-detail approach or a priority scheme based on polygon size (and probably location).
4.3 Lessons Learned

- The overhead of trying to make an application perform better may actually make it perform worse: The first version of seg divided large structures into hundreds of small structures— the overhead for the new structures overwhelmed any performance increase due to fewer polygons.

- Many Unix systems have little or no support for real-time operations, including finding out what time it is with any accuracy. The gettimeofday call on the Sun 4 has a 10 millisecond granularity, which introduces significant error into the frame-rate calculations in viper.

- The system worked even though no adjustments were made on the host processing end— all of the slack was taken up by varying the load on the graphics subsystem.

- Trying to adjust the load as quickly as possible did not work. In other words, if the system suddenly became heavily loaded, dropping a bunch of polygons all at once instead of doing it gradually failed. It failed because for transient variations, the system would add or subtract many polygons very suddenly, sending it into oscillation and producing large, abrupt image changes. Since there were only 32 levels per structure, dropping or adding them one at a time was reasonable.

4.4 General Explanations of Figures

- The graphs display both level information (dotted line) and frame rate information (solid line). The horizontal axis gives the frame count. The vertical indicates both frame rate and level. Specific data on the session is given underneath the graph.

- The program initially starts out with all 32 levels displayed, then adjusts the number dynamically. When the “slow motion” button is pressed, the number of levels is set to 32 for as long as the button is held, which is shown by the dotted line going abruptly up to 32 (except under light loads, where it is already at or near 32) and staying flat.

- The frame rate is initially estimated to be approximately 20, so the frame rate curve always starts there.

- Within the normal system oscillations, there are occasional peaks and valleys caused by changes in the Unix load, which further exercises the system.

- The granularity of the system clock on the Sun 4 causes discretization of the frame rate values following 10 millisecond intervals: 10 milliseconds -> 100 Hz, 20 ms -> 50 Hz, 30 ms -> 33 Hz, 40 ms -> 25 Hz, etc.

- The tracker data is buffered by the serial driver and comes in chunks of around 35 bytes, which leads to some tracker updates seeming to occur very quickly while others take longer than average. Frame-rate values of more than 30 Hz are caused by the granularity of the system clock and the tracker-data buffering.

The figures themselves are after the bibliography.
5. Discussion

Many current VE systems overload the graphics system so much that users are forced to slog through virtual worlds that seem as if they are submerged in motor oil. While this may be unavoidable in some cases, many VE applications could profit from techniques aimed at increasing interactivity at the expense of image quality. The author’s opinion is that the demands on processors will continue to grow at least as quickly as the technology improves, so the problem of having more to display than is possible in 1/20th of a second will not go away any time soon.

While the results for this system are encouraging, this approach is clearly not a general solution: Any application that needs to do simulation is unlikely to be able to simply remove polygons from the scene. Other approaches, such as level-of-detail methods, are more appropriate for such applications. This method is most applicable when the objective of the application is not so much to present an illusion of another world, but to allow a 3D visualization of some number of objects.

6. Future Work

There are a number of possibilities for improving and generalizing this work. A true real-time operating system would remove the glitches that occasionally cause the entire system to pause and would enable the use of real deadlines. Priority schemes could be used to ensure that the most visually important polygons are kept in the scene at all times. Priorities could be assigned based on polygon size, object type, or could even be assigned by the user or system builder. For systems that can afford to create and store multiple models at different levels of detail and switch between them at run-time, level-of-detail approaches could be implemented.

9. Acknowledgments

I would like to thank Mark Mine, Russ Taylor, Gary Bishop, and the Pixel-Planes group for invaluable technical assistance, Natalie Sipes for help with document preparation and distribution, and Kevin Jeffay for his help on the real-time aspects of this work.

This research was supported by the following grants: DARPA #DAEA 18-90-C-0044, NSF Cooperative Agreement #ASC-8920219, and DARPA: “Science and Technology Center for Computer Graphics and Scientific Visualization”, ONR #N00014-86-K-0680, and NIH #5-R24-RR-02170.
10. Bibliography


