Low Latency Rendering on Pixel-Planes 5

Jonathan Cohen
Marc Olano
University of North Carolina at Chapel Hill

Abstract

In this paper we present a rendering system called Slats, which renders triangle data sets with only a single field time (16.7 ms) of latency in the rendering pipeline. We measure this rendering latency from the time Slats begins transforming the data set into screen coordinates (using the most recent tracking information) to the time the display devices begin to scan the pixel colors from the frame buffers onto the screens. The Slats pipeline runs on Pixel-Planes 5, though we hope that the ideas we present and the compromises we make will be useful to designers of low latency pipelines on other platforms.

We begin with an introduction to the latency problem, including some discussion of future see-through head-mounted display applications requiring low latency rendering. Next, we present the previous work in low latency rendering. We then present a brief, but necessary, hardware overview, leading into a discussion of the rendering pipelines running on this hardware. We describe and analyze both the standard Rendering Control pipeline and the Slats low latency pipeline. We follow this with a section describing some problems we encountered and conclude by mentioning future work.

Introduction

Computer graphics, at first a static medium for visualizing large volumes of data, is becoming more and more a medium for dynamic interactions. Highly parallel, specialized graphics architectures have empowered graphics platforms with the ability not only to display various views of virtual objects or worlds, but to do so quickly enough to convey the sensation of motion. Our visual systems build on our knowledge of an object according to the way it moves, or appears to move as we change our view of it.

Performance of these current graphics systems is commonly measured in terms of the number of triangles rendered per second or in terms of the number of complete frames rendered per second. While these measures are the most common, they don't tell the whole story.

Latency, which measures the start to finish time of an operation such as drawing a single frame, is an often neglected measure of graphics performance. For some current modes of interaction, like manipulating a 3D object with a joystick, this measure of responsiveness may not be important. But for emerging modes of "natural" interaction, latency is a critical measure.

Head-mounted displays (HMDs) attempt such a "natural" mode of interaction. The user's head position and orientation are tracked to determine what viewpoint to use in generating a stereo pair of images for the eye displays. The higher the total latency in such a visualization system, the more the world seems to lag behind the user's head motions. The effect of this lag is a high viscosity world.

The effect of latency is even more noticeable with see-through HMDs. Such displays superimpose computer generated objects on the user's view of the physical world. The lag becomes obvious in this situation because the real world moves without lag, while the virtual objects shift in position during the lag time, catching up to their proper positions when the user stops moving. This "swimming" of the virtual objects not only detracts from the desired illusion of the objects' physical presence, but also hinders any effort to use this technology for real applications.

Future applications of see-through HMDs will augment the real world with computer generated images to enable us to see otherwise invisible data and to take action based on this additional visual information. For instance, this augmented reality could help an architect to visualize building modifications before finalizing the modification plan. Or it could present 3D instructions to guide the performance of "complex 3D tasks" [FEIN93], such as repairs to a photocopy machine or even a jet engine. This technology could even grant computer simulated "x-ray vision" to doctors, allowing them to see in 3D the fetus inside a pregnant woman or the tumor inside a cancer patient. Current research into the use of see-through HMDs by obstetricians to visualize 3D ultrasound data indicates the need for lower latency visualization systems [BAJU92]. The use of see-through HMDs for assisting surgical procedures is unthinkable until we make significant advances in the area of low latency graphics systems.

A possible solution to this lag problem is to use video techniques to cause the user's view of the real world to lag in synchronization with the virtual world. While this solution may seem quick and easy, it doesn't really do anything to fix the problem. You can't use your hands to interact naturally with a lagging world.

Another solution to the latency problem is to predict where the user's head will be when the image is finally displayed. This technique, called predictive tracking, involves using both recent tracking data and accurate knowledge of the system's total latency to make a best guess at the position and orientation of the user's head when the image is displayed inside the HMD.

Predictive tracking alone, however, will not eradicate the latency problem, because a large total latency in the tracking and image generation systems forces the prediction program to predict far into the future, undermining its efforts to make accurate predictions. However, if we strive to reduce the total latency, we can hope to hide the effects of the remaining latency with predictive tracking techniques.

We have designed a rendering pipeline called Slats to be part of such a minimal latency system. The rendering latency of Slats is only one field time (16.7 ms),

making it ideal for this purpose. We measure this rendering latency from the time Slats begins transforming the data set into screen coordinates (using the most recent tracking information) to the time the display devices begin to scan the pixel colors from the frame buffers onto the screens.

Previous Work

The minimization of latency is a relatively new area of research. While most graphics system designers probably consider effects of their architectural choices on latency, it is not common to reduce latency at the expense of any significant throughput. But as people begin to write applications that use graphics to augment reality rather than supersede it, latency will become much more important.

Matthew Regan and Ronald Pose are currently building the prototype for a hardware architecture designed to work with virtual reality systems [REGA93]. They eliminate latency for head rotations by rendering a scene on the six faces of a cube. The pixels are scanned out to the display using a beam racing technique instead of being read from a frame buffer. As a pixel is needed for display, appropriate memory locations from the rendered cube faces are read. A head rotation simply alters which memory is accessed, and thus contributes nothing to the latency. The lookup process can take into account distortion functions with no additional penalty, and simple anti-aliasing is performed during the beam racing process. Of course, a real user of an HMD always translates his head in space in addition to rotating it. Regan and Pose use object-space subdivision and image composition to allow objects to be prioritized and re-rendered as necessary to accommodate translations of the user's head. The image may not always be correct if the rendering hardware cannot keep up, but the most important objects, which include the closest ones, should be rendered in time to keep their positions accurate. Distant objects, which change very little as a result of head translations, may not be re-rendered in time, but should affect the user's perception of the image much less than the closer objects.

The HMD research group at the University of North Carolina is attempting to reduce latency on currently existing hardware. This is part of a multiple-front attack on the more general problem of registering real objects with computer generated ones [BISH93]. Latency is one of several factors we need to deal with to reduce the "swimming" of virtual objects in the real world space.

Mine has measured the total latency of our graphics systems when configured with various tracking systems [MINE93a]. He measures the latency at each stage of the pipelined system. These sorts of detailed measurements are necessary for finding the largest sources of delay as well as for knowing how far into the future we need to predict the user's head location to eliminate latency.

Mine and Bishop have also presented a technique called just-in-time pixels, which deals with the placement of pixels on a scan-line display as a problem of temporal aliasing [MINE93b]. Although the display may take many milliseconds to refresh, the image we see on the display typically represents only a single instant in time. When we see an object in motion on the display, it appears distorted because we see the higher scan lines before we see the lower ones, making it seem as if the lower part of the object lags behind the upper part. Avoidance of this distortion entails generating every pixel the way it should appear at the exact time of its display. They also suggest approximating this by applying the proper transformation for every scan-line rather than for every pixel.

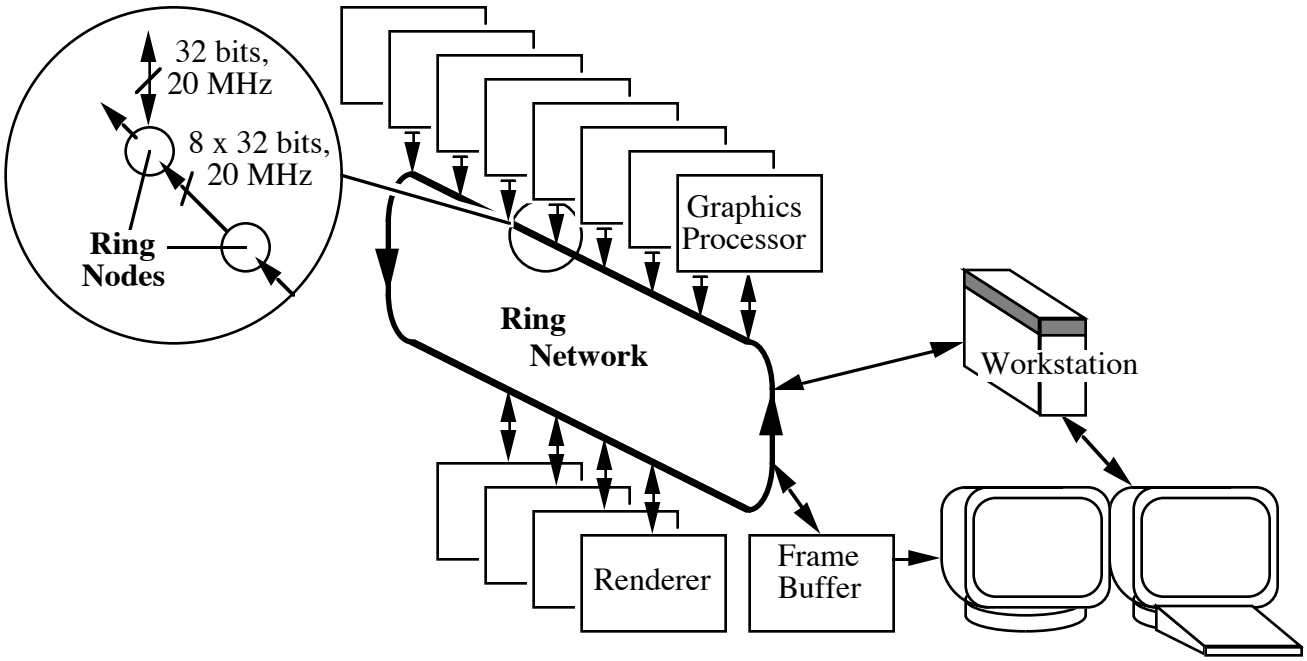
The Slats system we will present is much more practical than just-in-time pixels, but we can also see a theoretical foundation derived from that of just-in-time pixels. It approximates just-in-time pixels by using a new transformation for every field of the NTSC display rather than for every pixel or every scan line. However, this is merely an after-the-fact observation, because Slats was not developed with just-in-time pixels in mind.

Before going into the details of Slats, we will present some background on the hardware used and the rendering pipeline typically used on this hardware.

Hardware

We have implemented Slats on Pixel-Planes 5 (Pxpl5) [FUCH89]. This is not to say that our approach does not apply to more readily available architectures, but using Pxpl5 gave us total control over the graphics software, which was all developed in-house. Because our goal was to achieve lower latency by modifying the rendering pipeline, such low-level control was necessary. Before describing the standard rendering pipeline on Pxpl5 and our new rendering pipeline, we need to present some background on this hardware platform.

Pixel-Planes 5 uses parallelism at both the transformation and rasterization stages of the rendering process. Primitives, such as triangles, are typically generated on a host workstation and sent via a ring network to a set of graphics processors, where they are stored in local display lists. The graphics processors traverse these display lists, transforming the primitives from their object-based coordinate systems to a screen-based coordinate system and generating appropriate rendering commands. The graphics processors then send these commands over the ring to the renderers, which perform rasterization and shading. Finally, the renderers send the resulting pixel values to a frame buffer, which is synchronized with a video display for output.



Ring Network
Pixel-Planes 5's ring network connects the host interface, graphics processors, renderers, and frame buffers. It provides 8 communications channels, each capable of handling 20 megawords/sec. This total of 160 megawords/sec is necessary to support the simultaneous transmission of rendering commands, pixel values, and synchronization messages while rendering at rates of 1M primitives per second. This communications bandwidth is an important consideration when determining the feasibility of a new graphics pipeline.

Graphics Processors
The graphics processors (GPs) are general purpose 64-bit Intel i860 processors, each with 8 MB of DRAM. These processors run at 40 MHz. They function as a MIMD set of nodes with some synchronization to control their states with respect to that of the overall graphics pipeline. The GPs typically perform some transformations on graphics primitives. Each works independently on the set of primitives it has stored in its RAM. We will discuss the interaction of the GPs with the rest of the system more thoroughly in the section describing the rendering pipeline.

Renderers
The customized renderers give Pixel-Planes its name. Each renderer is a SIMD array of 128x128 single-bit processors, with each processor corresponding to a particular pixel on the screen. A renderer rasterizes primitives by evaluating linear expressions simultaneously for every pixel in the array. Some of these expressions represent lines, while others represent planes.

For example, a GP might send the following commands to a renderer to rasterize a triangle and maintain the renderer's Z-buffer for some screen region. First, enable all the pixels. Next, evaluate a linear equation we call an "edge expression" (though it actually represents the entire line that includes some edge of the triangle). Disable all the pixels sitting on the wrong side of the line, leaving all pixels on the correct side (which includes the inside of the triangle) enabled. Evaluate two more edge expressions, leaving only the pixels inside the triangle enabled. Now evaluate the Z-plane equation, which the GP determines from the Z coordinates of the triangle's vertices. Next, compare the newly interpolated Z value at each pixel with the Z value of the closest primitive at that pixel so far. If the new Z value is farther away from the viewer than the old one, the pixel disables itself. For each pixel that is still enabled, the new Z value replaces the old, and the pixel remains enabled for the interpolation of normal planes and color planes.

The color, normal, and Z value computed are saved for the pixels in that primitive. Any further shading calculations that make use of these values are deferred until the end of the frame, when the renderers have finished executing the commands for all the primitives. This deferred shading technique allows the shading to be done once per pixel instead of once per pixel per primitive [TEBB92]. When the renderer completes the shading for the region, it sends the final color values for that region to a frame buffer.

Frame Buffers

Pixel-Planes 5 can operate with both high-resolution and NTSC-compatible frame buffers built from video RAMs. A high-resolution frame buffer contains data for a 1280 x 1024 array of screen pixels. However, HMDs use NTSC video, so we will not consider the high-resolution frame buffers further.

An NTSC frame buffer contains data for a 640 x 480 resolution display, with 24 bits of color data at each pixel. Each NTSC frame buffer is considered a double buffer because there are two separate buffers representing the same set of pixels. This allows the graphics software to write pixels into one buffer without affecting the currently-displayed buffer. When the software is done writing some data, and a vertical retrace occurs on the display controller, the software swaps the two buffers, allowing the new image to be scanned out.

Each of these two buffers in an NTSC frame buffer contains two fields -- one containing the odd lines and one containing the even lines. Dividing a frame into fields is known as interlaced display; all the odd lines are scanned out after one retrace, then the even lines are scanned out after the next retrace. These retraces occur at 60 Hz.

One additional complication is that we deal with dual frame buffer stereo, using an NTSC frame buffer for each eye. This means we can work with up to eight

fields of data. The display controllers for the left and right eye displays are synchronized, so the vertical retrace comes at the same time for both eyes.

Rendering Pipelines

Now that we have covered the basic Pixel-Planes 5 hardware, we are ready to consider the rendering process. Rendering is typically broken down into several stages that can be pipelined to achieve higher throughput on machines with enough hardware to execute the various pipeline stages at the same time. We will begin by presenting the standard Pxpl5 rendering pipeline and analyzing its latency. Then we will present the Slats rendering pipeline, pointing out fundamental differences from the standard pipeline.

Rendering Control Pipeline

Description

The standard Pxpl5 rendering pipeline is controlled by a software layer called Rendering Control (RC) [ELLS91]. RC assigns one GP the role of master GP (MGP), while the others serve as slave GPs (SGPs). The MGP synchronizes the host, SGPs, renderers, and frame buffers. The pipeline it controls comprises the following stages:

1) SEND - Send editing commands from the host to the SGPs
2) EDIT - Execute editing commands
3) TRANSFORM - Transform primitives
4) RENDER - Render primitives
5) COPY - Copy pixels to the frame buffer

See figure 1 for an example of this pipeline.

During the SEND stage, an application running on the host makes some calls to set attributes (such as color), create and edit primitives (such as triangles or spheres), and set transformations, including the viewing transformation. These commands are stored in a display list, which the host distributes to the appropriate SGPs. The host typically distributes primitives among the SGPs in a round-robin fashion to promote even load balancing and broadcasts attributes and transformations to all the SGPs. Typical applications send few editing commands from frame to frame because of the limited communications bandwidth out of the host workstation.

At some point, the application decides that it is through sending editing commands and requests that the MGP begin the new frame. This request marks the end of the SEND stage. The host may not begin the SEND stage for the next frame until the EDIT stage from the current frame finishes.

Before the EDIT stage can begin, several conditions must be met:

1) The SEND stage from the current frame is finished.
2) The TRANSFORM stage from the previous frame is finished.
3) The RENDER stage from the 2nd previous frame is finished.
4) The COPY stage from the 3rd previous frame is finished.

When the MGP learns that the above conditions are met, it initiates the EDIT stage by telling the SGPs to begin applying to their local display lists the editing commands they have received from the host. For many typical applications, this stage is brief because display lists remain largely unchanged from frame to frame.

Each SGP can start its TRANSFORM stage as soon as it finishes applying its editing commands. The 2nd previous RENDER stage must also be finished, but this condition is guaranteed to be met before the EDIT stage can begin. So we can see that the start of the TRANSFORM stage requires no synchronization by the MGP.

During the TRANSFORM stage, each primitive is first transformed and then "binitized." The SGP applies some linear transformations to each primitive to express the primitive in terms of a screen coordinate system rather than an object coordinate system. It uses the screen coordinates to determine which screen regions each primitive may lie in and to generate renderer commands. As mentioned in the section about the renderers, these commands typically include "edge" equations and plane equations for adding primitives to a Z-buffer. These commands are stored in "bins," each of which holds commands for one 128x128 screen region.

Each SGP actually maintains two bin contexts, each of which has one bin per screen region. The TRANSFORM stage uses one context for binitizing the current frame and the other context for storing the previous frame's commands until they are sent to the renderers. For primitives falling into multiple regions, the SGP places commands (or, rather, pointers to these commands) in every applicable bin of the current frame's bin context. When all the primitives have been transformed and their commands have been generated and binitized, the SGP announces the completion of its TRANSFORM stage to the MGP.

When the MGP learns that all the SGPs have finished their TRANSFORM stages, it initiates the RENDER stage by assigning a screen region to every renderer and distributing renderer tokens. The MGP gives highest priority to the screen regions with the most commands waiting in SGP bins by assigning these regions first. The MGP then sends the token for each assigned region to a different SGP, which receives the token, sends out the bin for the token's region to the token's renderer, and forwards the token to the next SGP on the ring. The next SGP follows the same procedure. The last SGP to receive the token sends not only its

binitized commands, but also the end of frame commands for that region, which typically include some deferred shading commands. This SGP then returns the token to the MGP, which assigns the associated renderer to the next unrendered region. If there are no more unrendered regions, the RENDER stage for this frame is complete.

The COPY stage is loosely synchronized with the RENDER stage. Copying a region to the frame buffer does not require that the entire RENDER stage be completed. As long as a renderer has finished rendering a region and the frame buffer is available, the renderer may copy the region to the frame buffer. The frame buffer is considered available when half of the double buffer is being scanned out to the display, while the other half contains data that were scanned out earlier. The renderers can safely write to this old half. If the buffer is available when the MGP first sends out a rendering token, it sets a flag allowing the last SGP to include the copy to frame buffer command in its end of frame commands. If not, the MGP is responsible for issuing the command for that region.

For an interlaced stereo configuration, Pxpl5 uses two interlaced frame buffers (one for each eye). Rendering Control renders the stereo pair by putting first the transformation for the left eye, then the transformation for the right eye, into the pipeline. It considers the pair as one frame, so all editing commands are performed before rendering the left screen, and only the viewing transformation is changed before rendering the right screen. The SGPs and renderers know which viewing transformation, bin context, and frame buffer to use for each eye's pixel data. See figure 2 for an example of this dual frame buffer stereo pipeline.

For discussion of this configuration, we need to modify our descriptions of the conditions to start each pipeline stage. To begin the next SEND stage on the host, the EDIT stage from the previous *frame* must be finished, just as in a single frame buffer configuration. To begin the next EDIT stage, the following conditions must be met:

1) The SEND stage from the current *frame* is finished.
2) The TRANSFORM stage from the previous *frame* is finished.
3) The RENDER stage from the 2nd previous *eye* is finished.
4) The COPY stage from the 3rd previous *eye* is finished.

To begin the next TRANSFORM stage, the following conditions must be met:

1) The EDIT stage from the current *frame* is finished.
2) The RENDER stage from the 2nd previous *eye* is finished.
3) The COPY stage from the 3rd previous *eye* is finished.

Latency analysis

Now we are ready to analyze the latency of this Rendering Control pipeline. For this analysis we assume that the first stage, the sending of editing commands, is negligible. We measure latency from the start of the EDIT stage to the end of the COPY stage.

The minimum latency situation occurs when the TRANSFORM and RENDER stages are trivial (few primitives). In this case, the bottleneck is the time to copy pixel data from the renderers to the frame buffers (see figure 3). It takes about 780 μs for a renderer to transfer one region of pixel data to a frame buffer. Each eye display has 20 regions, so the total copy time is 780 μs/region * 20 regions/eye * 2 eyes = 31.2 ms. Because each odd or even field takes 1/60 second (about 16.7 ms), that comes out to almost one field time per eye in copy time alone. Assuming the TRANSFORM and RENDER stages each take half a field or less, they will get farther and farther ahead of the COPY stages. Because the pipeline can maintain only a limited number of bin contexts at each stage, this results in pipeline bubbles. The transforming and rendering will each be stalled for half a field, giving us a rendering latency of 3 fields (50 ms) from the start of the TRANSFORM stage to the end of the COPY stage.

To measure an actual minimum latency condition, we ran a small application on a small Pxpl5 configured with 13 GPs, 5 renderers, and 2 NTSC frame buffers. We rendered a scene with 12 triangles (1 for each SGP), each covering the entire screen. This forced every GP to communicate with every renderer, as it would in a typical scene. The application ran at 30 frames/second. The mean time from the beginning of the TRANSFORM stage to the swapping of the double buffer was 56.5 ms (about 3.4 fields). This is slightly longer than we predicted in the preceding paragraph. This may be due in part to varying degrees of synchronization between the COPY stage and the vertical retrace on the frame buffer. It may also reflect the fact that the overhead for communication between the GPs and renderers becomes significant for scenes with few primitives.

As more primitives are added to the display list, the TRANSFORM and RENDER stages take longer, and the latency increases. We cannot determine a maximum latency by analyzing a pipeline diagram. Such an analysis shows us only that the latency can increase without bound. The maximum latency in practice occurs when the application reaches a limit to the length of the pipeline stages it can handle, which is determined by the maximum number of primitives that can be stored in each GP's display list and bins. Adding transformations as well as primitives to the display list can also increase the length of the TRANSFORM stage, but we are currently considering only data sets in which primitives vastly outnumber transformations in the display list.

We measured the maximum latency using the same application as we used to measure the minimum latency, but we dramatically increased the number of

triangles. Our application could send up to 7700 full-screen triangles without overloading the bin memory on the GPs. This number does not reflect the maximum number in a typical application, however. Although only 7700 triangles are transformed, full-screen triangles are a pathological case for the RENDER stage. Triangles in typical applications seem to affect only 1.3 regions (per eye) on the average, but these triangles affect 20 regions each, using over 15 times the bin memory used by more typical triangles. The application ran at 2 frames/second with a mean frame time of 293 ms, or about 17.5 field times. Of course, the latency is less of an issue at this point than the unacceptable frame rate.

Neither of these analyses is really appropriate for typical applications. In fact, the pipeline diagram for a typical HMD application does not have a steady state. The number of primitives per region depends on the viewing transformation, which changes from frame to frame. To make some intermediate analysis of the pipeline, we will assume that the application tends to use the pipeline hardware efficiently (i.e., GPs and renderers are always working). If the TRANSFORM, RENDER, and COPY stages all have equal length and the frame rate is 30 Hz, the latency may be as large as 4 fields (about 67 ms), depending on how far into the RENDER stage the COPY stage is begun (see figure 4). If the TRANSFORM and RENDER stages are of equal length, but longer than the COPY stage, the latency should be 3 times the length of the TRANSFORM stage plus from 1 to 2 times the length of the COPY stage.

To measure the latencies of some typical applications, we have instrumented a fly-through model viewer to make the same latency measurements as our previously mentioned mini-application. The model viewer allows a user in an HMD to interact with a 3D model by walking and flying through the model, grabbing the model, and scaling the size of the model. The following table summarizes the results:

Model	Primitives	Mean Latency	Std. Deviation
Mig	417 triangles	4.61 fields	0.589 fields
Molecules	2,620 triangles, 1,219 spheres	4.47 fields	0.502 fields
head	59,592 triangles	8.13 fields	2.48 fields

The rendering latency for these interaction sequences varies not only with the number of primitives in the scene, but also with the arrangement of the primitives and the path of the user's head. The mean latency and the standard deviation are coarse measures that can change a great deal from session to session.

Slats Rendering Pipeline

Description

Slats is a single field-time latency rendering system that runs on Pxpl5 in place of the standard Rendering Control system. The current prototype uses 1 GP, 5 renderers, and 2 NTSC frame buffers. The goal of this system is to serve as an example of low latency rendering and to investigate the benefits of rendering with only 1 field time of latency. We did not optimize the system for the greatest possible throughput but instead focused on achieving the proposed latency goal and ensuring the correctness of the displayed image. Further optimization might improve throughput dramatically, but would also complicate the correctness problem by forcing us to trade latency for correctness in certain worst cases.

The Slats pipeline comprises the following stages:

1) SEND - Send viewing transformations for both eyes from host to GP
2) TRANSFORM - Transform and binitize triangles
3) RENDER - Render triangles
4) COPY - Copy pixels to the frame buffer
See figure 5 for an example of this pipeline

The SEND stage is a simplified version of RC's SEND stage. Because this is a prototype system, the host sends the primitives only once during initialization and cannot edit them during execution. So the only data the host sends through the pipeline are the two viewing transformations converting from world coordinates to left and right screen coordinates. This simplification has little effect on our comparison with RC because we assumed that few editing commands were sent under the RC system anyway.

Unlike Rendering Control, the SEND stage never requests that a new frame be drawn; the host just acquires tracking data, calculates the viewing transformations, and sends these transformations to the GP as quickly and as often as possible. Though we have placed this SEND stage in the list of pipeline stages for comparison with Rendering Control, it actually does not qualify as a pipeline stage; it occurs asynchronously with respect to the true rendering pipeline. For Slats to function as intended, new tracking data must arrive at the host 60 times a second. If delays occur in receiving tracking data, Slats reuses the most recent tracking data until new data arrives. Such delays effectively increase the total latency of the system, so they must be infrequent if we are to observe the system's most responsive behavior.

The TRANSFORM stage uses only one GP. This minimizes communication between the GPs and contention for the renderers. A major difference between Slats's and RC's TRANSFORM stages is that Slats considers both eyes simultaneously, making only one pass over the primitives, whereas RC considers them sequentially, making one pass for each eye (we will see later how this accelerates the COPY stage). The GP employs the same transformation method

as do the SGPs in RC, but the binitization differs. In the current implementation, the GP puts all the renderer commands for drawing triangles into a single bin. Every region renders this single set of triangle commands. If we assume that a typical triangle appears in only 1 or 2 regions, we see that rendering the triangle in all 20 regions uses over 10 times the necessary bandwidth to send the commands and 10 times the necessary rendering time. This waste is justified for the moment because it ensures that the latency depends only on the number of on-screen triangles and not their arrangement. This makes it easy to determine the maximum data set size that Slats is guaranteed to render within a single field time. For this prototype, that data set size is 140 triangles. Using more triangles will prevent the current Slats prototype from correctly rendering the scene when all the triangles are on-screen.

The RENDER stage differs somewhat from RC's RENDER stage, though both eventually generate pixels for the frame buffers. The most important difference is that the assignment of renderers to screen regions is static. Each of the 5 renderers maintains 4 regions -- a top and bottom region in each of the 2 eyes. (Note that when we consider the odd and even fields separately, each eye has only 10 regions of pixel data, so 5 renderers can render the 20 total regions of a particular field.) This static binding of regions to renderers allows the GP to start sending renderer commands before it has finished transforming the primitives. In fact, the GP sends commands to all the regions whenever it has enough data to fill a ring message of maximum size. The use of maximum-sized messages minimizes the message overhead.

One consequence of the static binding and maintenance of 4 region contexts per renderer is that the 4 regions must share the 208 bits of pixel memory. Because of this restriction, Slats does not perform Phong shading, which requires an interpolated normal at every pixel. It currently uses color interpolation on vertex colors precomputed using a radiosity model. These vertex colors could also be computed on GP by evaluating a lighting model at each vertex.

The COPY stage also is organized differently from the COPY stage of RC. Because RC does each eye sequentially, it can copy to only one frame buffer at a time. This wastes ring bandwidth, because each frame buffer has its own ring port, making it feasible to copy to the left and right frame buffers simultaneously. As soon as the Slats GP is done transforming and sending triangle commands to the frame buffers, it can send the copy commands. Because the two frame buffers can receive data simultaneously, the 20 regions can be transmitted in the time it takes to transmit 11 regions sequentially. Here is the reason for the 11 transmission times (rather than 10). If all the renderers send regions alternately to the left frame buffer then the right frame buffer, there is one copy time during which no data is sent to the right frame buffer as the transmissions begin and one copy time during which no data is sent to the left frame buffer as the transmissions end. The problem can be eliminated only by having the GP synchronize every region copy to ensure a particular order. The

time taken by this synchronization would severely hinder the GP's transformation capability.

Because the time required to transmit all the regions is less than half a field time, we can make an optimization to the COPY stage. To guarantee proper image display, the COPY stage in its original form must finish copying all 20 regions before the occurrence of the vertical retrace . We can optimize this because copying half of the regions takes only one quarter of the field time. This allows us to copy the top regions before the retrace and the bottom regions immediately after the retrace (see figure 6). The GP needs to synchronize these two halves of the COPY stage to ensure that all of the top regions are copied before any of the bottom regions are copied. By allowing Slats to copy only 10 regions before the vertical retrace, this optimization gives the TRANSFORM and RENDER stages more time to do their work and is responsible for raising the maximum number of triangles of the prototype system from 110 to 140.

Latency Analysis

Analysis of this pipeline in terms of latency is, by design, trivial. The rendering latency, from the beginning of transformation to the beginning of scanout, is one field time, or 16.7 ms. The frame rate is also trivial: 60 frames/sec, the same as the NTSC monitor. One way to look at the throughput is to say we draw 140 triangles x 2 eyes x 60 Hz = 16,800 triangles per second. This measurement stands on rather shaky ground, however, and really pushes the limits of the definition of triangles per second. Though we really draw the triangles 60 times every second, we draw them at only half the resolution of the display device. So the transformation occurs 60 times per second, but full screen rasterization really occurs only 30 times per second.

Problems Encountered

While we were testing Slats, many problems came up that hindered our testing the effectiveness of single field time rendering for see-through HMD applications. These problems fall into the categories of tracking problems and display problems. If tracking transport time is not properly managed, it can dwarf the time savings we have made by reducing rendering latency. Also, static registration problems, caused by magnetic and optical distortions, can cause objects to be incorrectly positioned in the virtual world even when the viewer remains perfectly still. This makes the reduction of dynamic registration problems, such as latency, harder to detect, because a virtual object's registration with the real world changes as the viewer moves, even on a system with zero latency.

Tracking Problems

We have had 4 tracking systems available for research in our department recently: a Polhemus 3-space, a Polhemus Fastrak, an Ascension Flock of Birds,

and an optical ceiling tracker [WANG90]. Both of the Polhemus trackers have magnetic fields with easily noticeable distortions, causing a plane of movement to appear bowl-shaped. Our lab environment, which has metal in the floors and ceiling, is far from ideal, and contributes to this effect. The Ascension and optical ceiling trackers gave us more reliable tracking data, but could not quite match the Fastrak's update rate.

The effect of a distorted magnetic field is dramatic in a see-through HMD. Because the tracker measures the head's position and orientation inaccurately, the computer generated objects are not properly registered with the real world. Even if we could reduce the system's total latency to zero, the objects would appear to move as the user's head moved. This makes it harder to distinguish between object motion caused by latency and object motion caused by poor registration.

The other tracking problem is not related to the actual tracking hardware, but to the method of transporting the data. We currently get tracking data (except for optical ceiling data) by way of a Unix serial port on the host workstation. Unfortunately, the Unix kernel polls the serial port only at 30 ms intervals to see whether new data have arrived. This adds almost 2 field times to the total latency, which already includes tracker, rendering, and display latencies. As a result, tracking data arrive sporadically on the host, obscuring the effects of lower rendering latency. This transport delay affects the standard RC pipeline much less noticeably than it does the Slats pipeline, because this delay is a smaller percentage of the RC pipeline's latency than it is of the Slats pipeline's latency.

To remove this unnecessary latency, we applied a temporary hack to the Unix kernel. This hack modifies the kernel to cause an interrupt on the receipt of every character at the serial port. This worked with the Polhemus trackers, but not with the Ascension, which caused the workstation to crash when run with the kernel modification.

Display Problems

The other class of problems relates to the available see-through HMDs. Our lab has two types of see-through HMD: video see-through and optical see-through. The video see-through uses a camera mounted on an HMD to capture images of the real world. Because we have no depth information about the world, we merge the video images with the graphics by using color chroma-keying. The computer renders the image with a particular background color, and all the pixels of that color are later replaced with pixels from the video. This overlaying of computer generated objects onto video images without proper handling of occlusion is one major problem with our current video see-through.

Another problem with the video see-through HMD is that it is difficult, if not impossible, to get the perspective right from two camera images. It might be

possible to get a proper perspective arrangement, but not the one lined up in front of the user's eyes. The visual result may still be convincing, though, much like looking through a periscope. Because of the difficult nature of this problem, we still use only one camera with our video see-through HMD.

A third problem with the video see-through is distortion created by the camera optics. This results in static registration problems much like those from the magnetic field distortions.

Our other see-through HMD is an optical see-through system, which projects the images through a series of lenses onto circular, half-silvered mirrors. While wearing this HMD, you can see the real world as you see it normally. There is no latency involved, you can see your hands just where they should be, and you even have your normal peripheral vision. The half-silvered mirrors have the same occlusion problem as the video see-through, because we still have no depth map of the real world. In addition, the mirrors make up only a small portion of your field of view, so keeping the computer generated objects where you can see them is difficult. Even small amounts of total latency can drive the objects past the edges of the optics, making it hard to know how bad the problem is.

Our current optical see-through HMD also has registration problems. This is due partly to lens distortion but largely to the fact that the head unit is not very sturdy and can bend when subjected to the slightest pressure.

The video see-through has certain advantages. The hardware already exists in pieces, and mounting a camera on an HMD is relatively easy (calibration, on the other hand, is hard). Because the user looks only at the displays in the HMD, the computer can control all the user's visual input. Also, much like the non-see-through HMDs, the use of video input instead of direct sensory input detaches us slightly from the real world, making the computer generated additions easier to accept. On the other hand, this does not advance the cause of making them seem truly real.

The optical see-through is currently more limited than the video see-through in its ability to put objects anywhere in your field of view, but it allows you to look at and touch the real world in the most natural way.

Future Work

Slats still has plenty of room for improvement. We can enable Slats to handle larger data sets by performing the binitization step correctly, sending triangles only to the necessary regions. Using larger data sets with such a system may cause it to run longer than one field time for some viewpoints. We think we can use the available frame buffer bandwidth to copy the most recent complete field into a backup field in case we miss a vertical retrace. Of course, for the system to

render with a single field time of latency, such misses must be rare. For this more complex system, the acceptability of a data set will depend not only on the number of triangles it contains, but also their arrangement.

We might also increase the throughput of Slats by adding multiple GPs. This would add a problem of contention for access to the renderers, but we could presumably increase our throughput substantially, because the renderers can handle many more primitives than we can generate with such a short TRANSFORM stage, even if we use many GPs.

Most importantly, we need to come up with new approaches to achieving low latency. We strongly encourage other researchers in industry and academia to investigate ways of tackling the latency problem. These will include new ways of using our current hardware as well as ways to design future hardware with latency in mind. Low latency rendering, when coupled with predictive tracking, should help us to achieve zero latency systems in the future and to create more natural virtual environments.

References

BAJU92 Bajura, M., H. Fuchs, R. Ohbuchi, "Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery within the Patient," *SIGGRAPH '92*, Vol. 26, No. 2, pp. 203-210.

BISH93 Bishop, G., "No Swimming," University of North Carolina Department of Computer Science, HMD Research Project Goals.

ELLS91 Ellsworth, D., "Pixel-Planes 5 Rendering Control", University of North Carolina Department of Computer Science Software Documentation.

FEIN93 Feiner, S., B. MacIntyre, and D. Seligmann, "Knowledge-based Augmented Reality," *Communications of the ACM,* Vol. 36, No. 7, pp. 52-62.

FUCH89 Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *SIGGRAPH '89*, Vol. 23, No. 3, pp. 79-88.

MINE93a Mine, M., "Characterization of End-to-End Delays in Head-Mounted Display Systems," University of North Carolina Department of Computer Science Technical Report TR93-001.

MINE93b Mine, M., and G. Bishop, "Just-In-Time Pixels," University of North Carolina Department of Computer Science Technical Report TR93-005.

REGA93 Regan, M., and R. Pose, "An Interactive Graphics Display Architecture," Department of Computer Science, Monash University, Clayton, Victoria, Australia.

TEBB92 Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading," University of North Carolina Department of Computer Science Technical Report TR92-034.

WANG90 Wang, J. F., R. Azuma, G. Bishop, V. Chi, J. Eyles, and H. Fuchs, "A Demonstrated Optical Tracker with Scalable Work Area for Head-Mounted Display Systems," *Proceedings of 1992 Symposium on Interactive 3D Graphics*, March 1992, pp. 43-52.

Figures

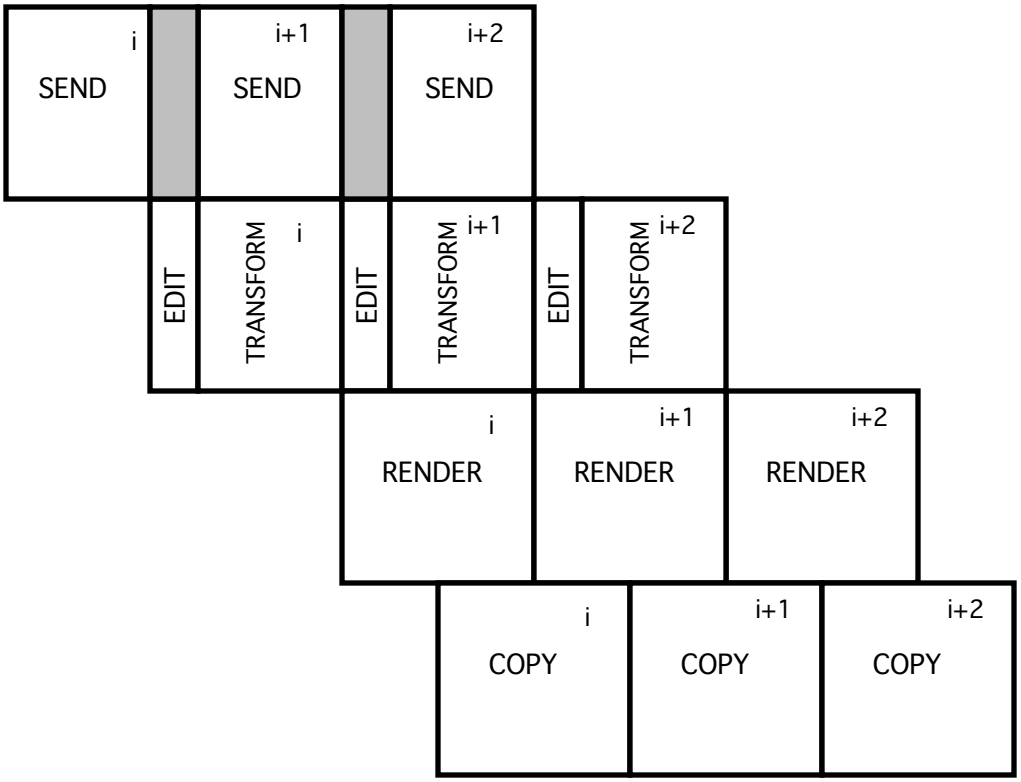


Figure 1: Rendering Control pipeline for frames i, i+1, and i+2

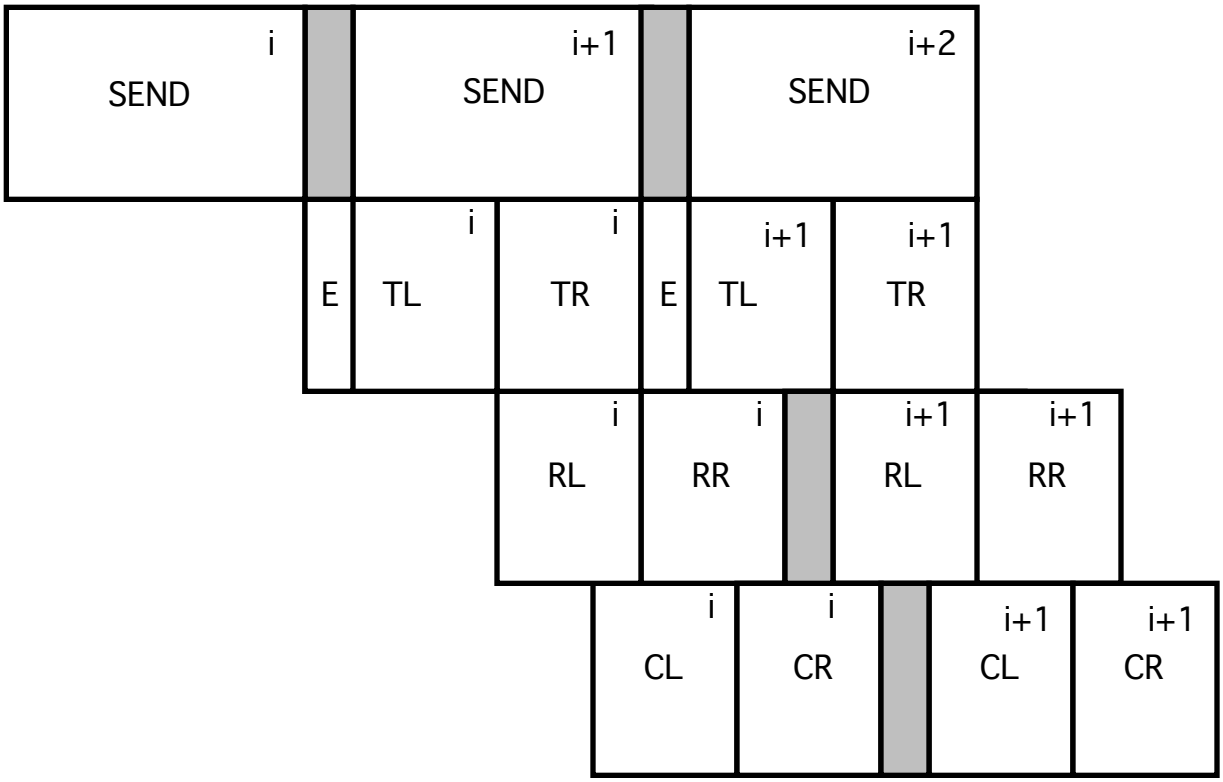


Figure 2: Rendering Control pipeline using dual frame buffer stereo
E=EDIT, TL=TRANSFORM left, TR=TRANSFORM right, RL=RENDER left, RR=RENDER right, CL=COPY left, CR=COPY right

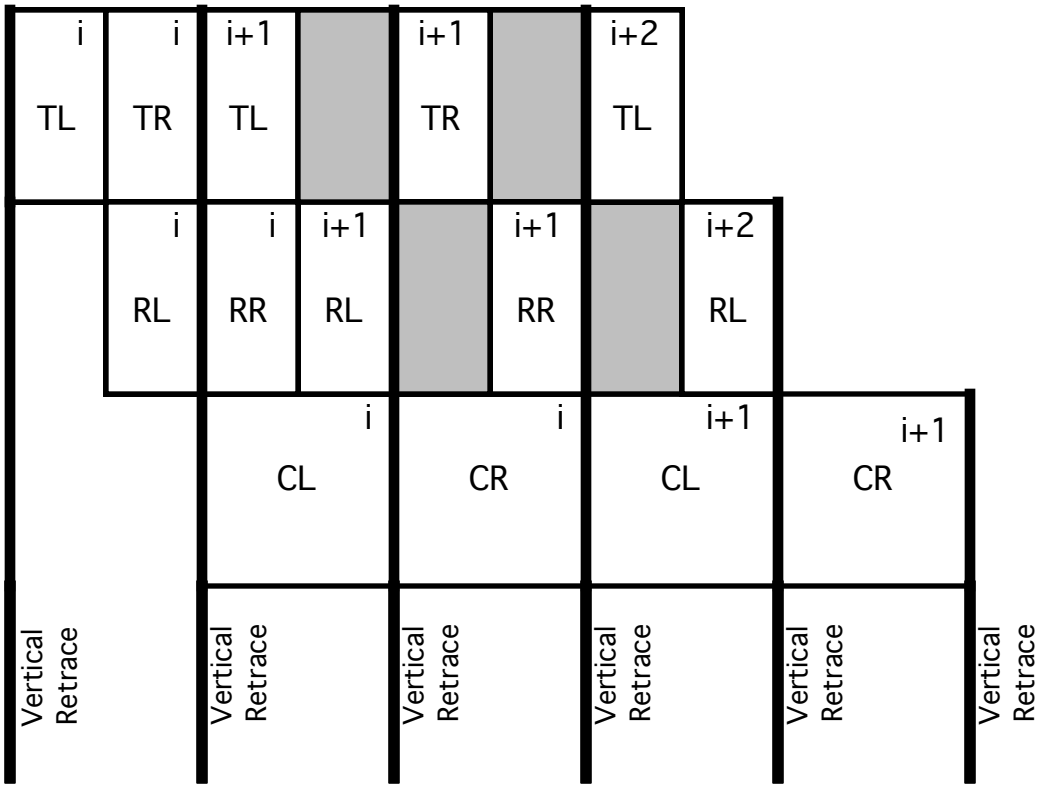


Figure 3: Rendering Control pipeline using dual frame buffer stereo for a minimum latency application
TL=TRANSFORM left, TR=TRANSFORM right,
RL=RENDER left, RR=RENDER right,
CL=COPY left, CR=COPY right

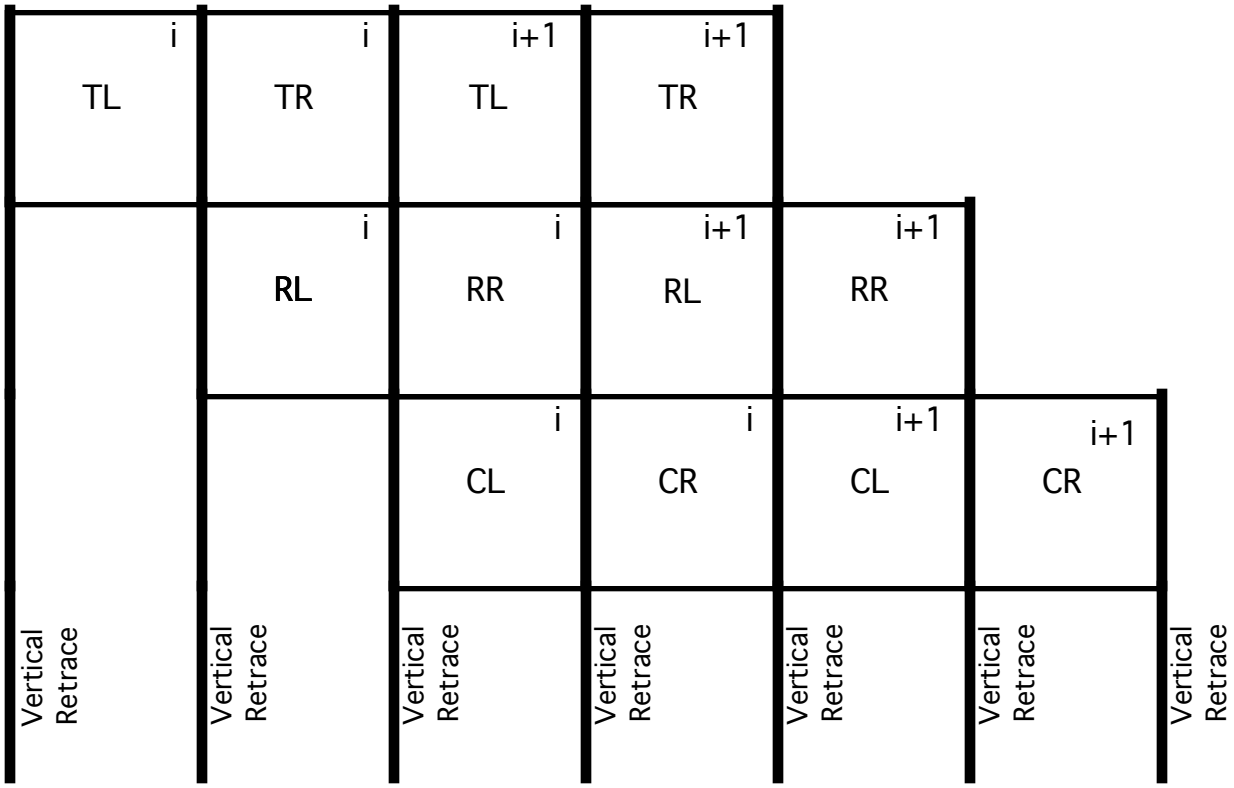


Figure 4: Rendering Control pipeline using dual frame buffer stereo with minimum latency for 100% utilization of hardware
TL=TRANSFORM left, TR=TRANSFORM right,
RL=RENDER left, RR=RENDER right,
CL=COPY left, CR=COPY right

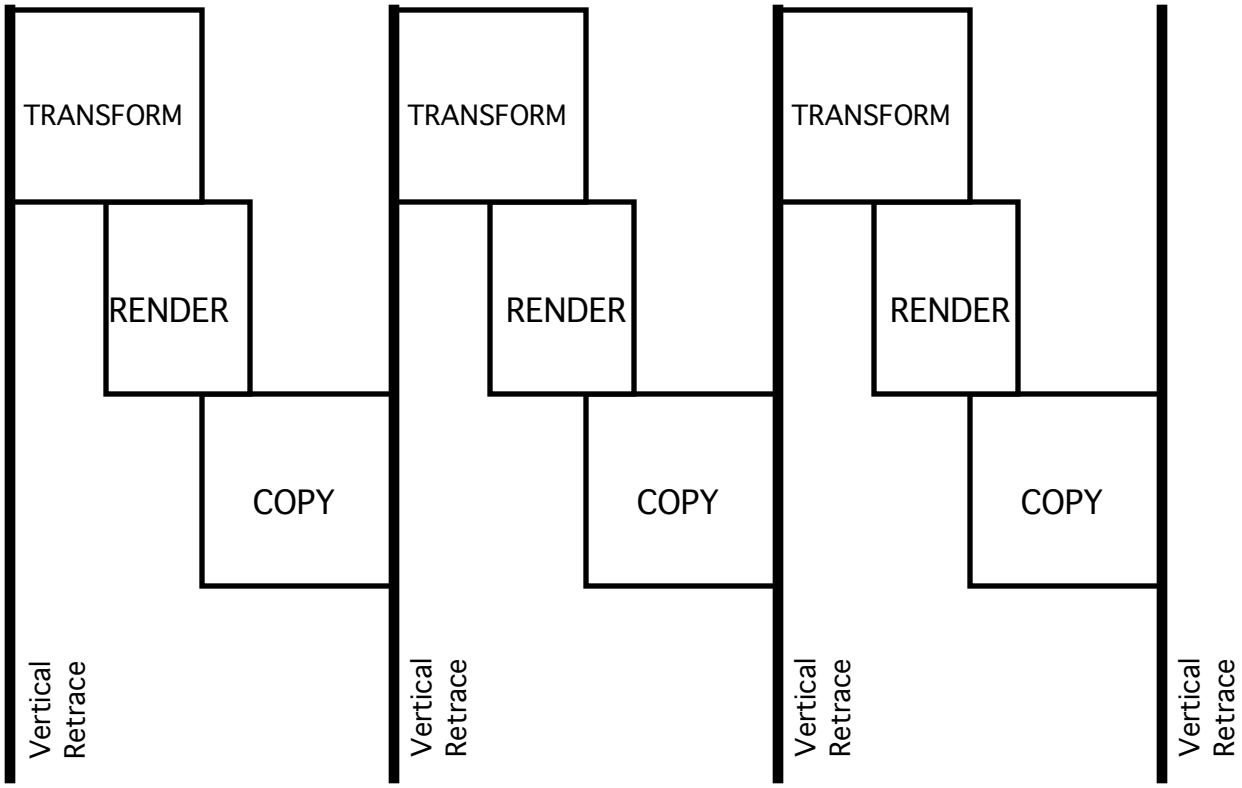


Figure 5: Slats pipeline without COPY optimization

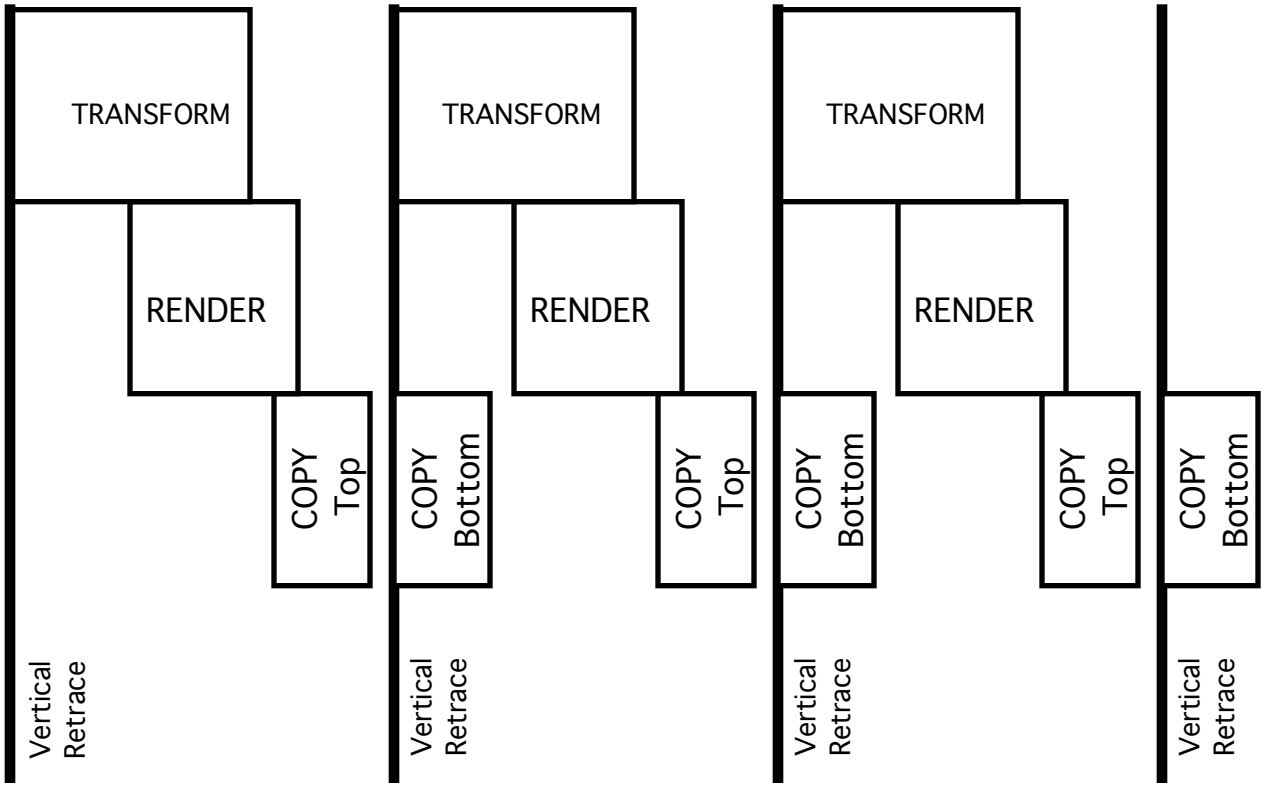


Figure 6: Slats pipeline with COPY optimization