Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

# PixelView: A View-Independent Graphics Rendering Architecture

J. Stewart, E.P. Bennett, and L. McMillan

Department of Computer Science, The University of North Carolina at Chapel Hill, USA

**Abstract**

*We present a new computer graphics rendering architecture that allows "all possible views" to be extracted from a single traversal of a scene description. It supports a wide range of rendering primitives, including polygonal meshes, higher-order surface primitives (e.g. spheres, cylinders, and parametric patches), point-based models, and image-based representations. To demonstrate our concept, we have implemented a hardware prototype that includes a 4D, z-buffered frame-buffer supporting dynamic view selection at the time of raster scan-out. As a result, our implementation supports extremely low display-update latency. The PixelView architecture also supports rendering of the same scene for multiple eyes, which provides immediate benefits for stereo viewing methods like those used in today's virtual environments, particularly when there are multiple participants. In the future, view-independent graphics rendering hardware will also be essential to support the multitude of viewpoints required for real-time autostereoscopic and holographic display devices.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Viewing Algorithms; Display Algorithms; Bitmap and Frame Buffer Operations. I.3.1 [Computer Graphics]: Graphics Processors. I.3.6 [Computer Graphics]: Graphics Data Structures and Data Types.

## 1. Introduction

Viewpoint specification is fundamental to traditional computer graphics rendering. Both the transformation of a scene to eye space in the traditional graphics pipeline and the origination of viewing rays in a ray-casting system depend on the viewpoint. Moreover, many subsequent rendering steps are also impacted by the choice of viewpoint, including clipping, projection, illumination calculations, shading, and visibility determination. As a result, changing the viewpoint frequently gates the entire process of interactive rendering, as each rendered frame is initiated with the specification of a viewpoint, followed by the scene description, and culminating with the final displayed image.

There are many potential advantages to decoupling viewpoint specification from rendering. First, immediate efficiency improvements are available if rendering costs are amortized over multiple views. They result from reuse of shading calculations as well as exploiting the coherency of surface reflection with smooth variations in viewpoint. A second advantage results from beginning the rendering process before the viewing position is resolved, thereby reducing latency.

However, the ultimate advantage of separating rendering from viewpoint selection is that it becomes possible to render the same scene for multiple eyes. Possible applications include shared virtual environments (stereo viewing by many participants of a computer-generated scene). In the future, view-independent graphics rendering hardware will also be essential to support the multitude of viewpoints required for real-time autostereoscopic and holographic display devices.

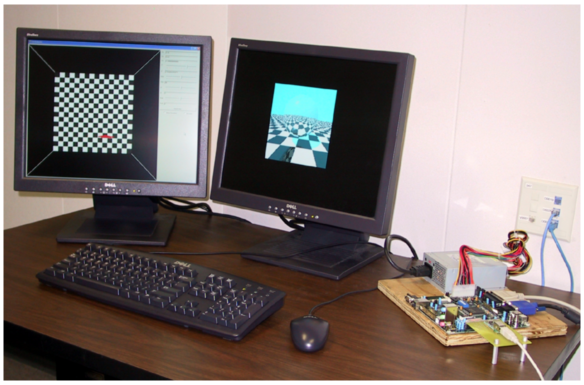
**Figure 1:** *Photograph of our hardware prototype of the PixelView architecture. The prototype is on the right, its VGA output is in the middle, and the PC controller is on the left.*

We have developed a new computer graphics rendering architecture, called PixelView, in which a single traversal of the scene description generates "all possible views" (or at least a wide range of views). PixelView is compatible with the scene descriptions used by traditional 3D rendering hardware (i.e. polygons). In addition, PixelView, like the Reyes rendering system and many ray-casting renderers, supports any higher-order surface primitive for which a world-space subdivision scheme exists. It also natively supports point-based models as well as image-based representations, all within a unified rendering architecture.

The primary contributions of our PixelView rendering system include:

- A scalable system architecture for supporting real-time, view-independent rendering of 3-D models
- A hardware prototype of a 4D, z-buffered frame buffer demonstrating the viability of dynamic view selection at scan-out
- Hardware rendering of primitives with view-dependent reflectance by a technique that we call "point casting"

In addition, we consider possible representations and compressions for the radiance of view-dependent points to reduce storage and bandwidth requirements.

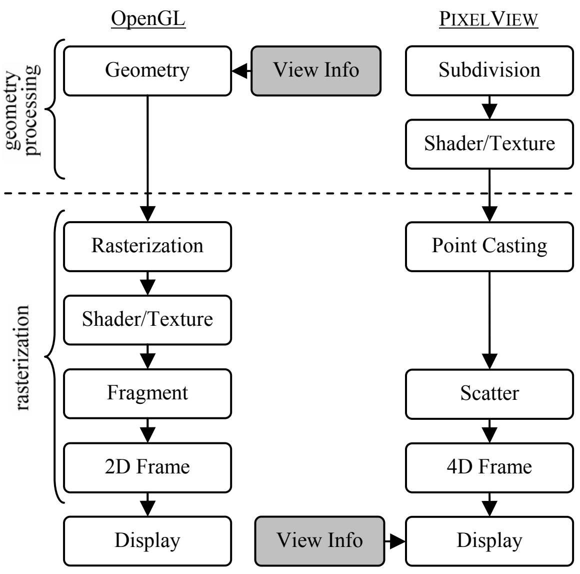

**Figure 2:** *High-level comparison between a typical OpenGL-style pipeline and PixelView. Only the display stage in PixelView requires knowledge of the viewpoint, allowing reuse of shading and rendering calculations.*

## 2. Overview

We begin with a high-level overview of PixelView via a comparison with the traditional graphics pipeline, with a focus on the role of view specification.

The left side of Figure 2 shows a traditional graphics pipeline. Polygon vertices (in object space) are fed into the geometry stage for transform and lighting. This stage outputs lit, screen-space triangles that are subsequently rasterized into fragments. These fragments are then shaded and textured. Various other raster operations, such as depth-comparison, compositing, and filtering, can then be performed before the final color values are written into the 2D frame buffer.

The right side of Figure 2 shows the PixelView pipeline. Our system supports any primitive that can be subdivided in world space. Primitives enter the subdivision stage, where they are first transformed from object space into world space. The primitives then undergo world-space subdivision and are shaded and textured. The shader/texture stage outputs fully shaded world-space points, which are then "point cast" into specific bins of the 4D frame buffer. Lastly, because a point will generally not fall exactly into a bin, its contribution is "scattered" into nearby bins. Specifics of the various pipeline stages are discussed in subsequent sections. For now, we will focus on the role that view information plays in each rendering approach.

When comparing the pipelines, note that view specification comes into play very early in the traditional graphics pipeline. Specifically, the first operation that typically happens to vertices in the geometry stage is to transform them into eye space via the modelview matrix. Many later steps in the pipeline are also affected by the viewpoint. Thus, changing the viewpoint requires re-traversal and re-rendering of the entire scene.

Alternatively, note that no view transform occurs in the PixelView geometry processing stages. In fact, no view transform occurs at any point along a primitive's path into the frame buffer. That is, no knowledge of the virtual camera's location and orientation is required to render primitives into the 4D frame buffer. We can "get away with this" because we render a sampled version of the outgoing radiance from each point of the scene. The viewpoint specification can thus move to the display stage, where it is used to reconstruct a particular view during scan-out. This allows the viewpoint to change without re-rendering the scene. Furthermore, this fundamental reorganization of rendering tasks allows us to re-visit the tradeoffs between rendering quality and frequency of scene updates.

## 3. Previous Work

The recent advent of flexible programmable graphics hardware has ignited renewed interest in alternative architectures for real-time rendering [OKTD02; PBMH02]. However, one of the least flexible remaining functions in existing architectures is the scan-out mechanism used to refresh the display. The only exposed controls for display refresh are for setting the resolution (and perhaps setting the origin of the memory block allocated for the frame buffer). There are compelling reasons for optimizing this functionality via a hardwired implementation. In particular, because the performance of graphics processing units is often dictated by available memory bandwidth, there is a significant benefit in optimizing the appreciable, constant, and real-time demands of display refresh. In other words, it makes sense to minimize the demands of refresh to free up memory bandwidth for other compelling rendering functions.

However, the lack of flexibility in display scan-out limits the ability to address certain problems related to dynamic view selection and latency compensation [OCMB95; BFMZ94]. Regan et al. recognized this limitation and constructed a novel 3D frame buffer with flexible scan-out circuitry for the specific purpose of studying display latency [RMRK99]. Although successful at reducing latency, their system lacked vertical parallax (i.e. it limited the viewing positions to points along a specific line), and it limited the image plane to the face of the display device. Moreover, it required off-line rendering to pre-compute the contents of the 3D frame buffer. The display stage of the PixelView architecture extends and generalizes Regan et al. into a 4D frame buffer. Its flexible scan-out stage supports both horizontal and vertical parallax. Furthermore, the PixelView architecture contains geometry and rasterization stages that allow primitives to be rendered into the 4D frame buffer.

Another active area of research has been the decoupling of slow rendering processes from the interactive demands of viewing and animation. Others have proposed special-purpose hardware [RP94] and software [MMB97; BWG03; Gla88] rendering systems to address this problem. Many of these systems also incorporate 3D [RMRK99] and 4D [War98; Bal99] frame buffers or ray caches, which are sampled and interpolated to produce view-specific renderings. Most of these systems operate as lazily evaluated caches, meaning that samples from previously rendered views are combined with samples from the current viewpoint. This approach generally requires no, or very little, variation in each point's reflectance as a function of viewpoint, with the notable exception of Bala [BWG03] who maintained a metric describing the range of views over which each radiance sample was valid.

Our approach renders the contribution of each primitive into all possible views. This affords a heretofore-unexploited type of coherency that is currently unavailable to traditional view-dependent rendering architectures, at the price of potentially rendering rays that might go unseen. There has even been some work on exploiting the coherence of rendering due to smooth variations in viewing position [Hal98]. This system effectively transformed and harnessed the power of 3D rendering to allow space-time or EPI rendering. We attempt to exploit the same sort of coherence in our shading approach. However, we do not focus on rendering epipolar planes one-at-a-time, but instead render the out-going radiance from each 3D point and use z-buffering to resolve visibility.

Our system relies on substantial preprocessing of display primitives much like Reyes [CCC87] and Talisman [TK96]. Specifically, we are able to render directly only those primitives that can be appropriately subdivided in world space. Moreover, as the average size of a rendering primitive shrinks, alternative primitives have been suggested. Examples of these include point-based models [RL00; PZvBG00; ZPKG02; PKKG03] and image-based models [SGHS98]. PixelView is capable of directly rendering and displaying point-based representations with view-dependent effects, as well as light fields [LH96] and lumigraphs [GGSC96]. Each display type can be easily combined with any of the others. Furthermore, the sampling and reconstruction strategies used in PixelView draw heavily on those developed for point-based primitives and light field rendering.

## 4. The PixelView Architecture

This section describes the various stages in the PixelView architecture. Note that this section is intended to provide a general, abstract description of the architecture, in contrast to the specific, concrete implementation presented in Section 5. Referring to Figure 2, we begin with the "lower half" of the pipeline (i.e. rasterization and display).

### 4.1. A 4D Frame Buffer

In PixelView, the standard 2D frame buffer is replaced with a 4D ray buffer. Frame buffers are commonly described as an array of pixels, but in the context of 3D rendering, they are more accurately characterized as an array of rays seen from a single viewpoint. This "frame buffer as ray buffer" concept is appropriate for both ray-casting and OpenGL-style renderers.

View independence is achieved by generalizing the 2D "ray buffer" into 4D. The resulting structure is, in essence, a light field/lumigraph, with rays specified by their intersection with two parallel planes [LH96; GGSC96]. Following the notation of Gortler et al. [GGSC96], we call these two planes the s-t plane and the u-v plane. Our frame buffer is thus a 4D collection of radiance values, a finite sampling of light rays parameterized by (s,t,u,v). Once the 4D frame buffer has been "populated", novel views can be generated without the need to re-render scene geometry.

### 4.2. Display/Scan-Out

New images can be created during scan-out by taking a 2D slice of the 4D frame buffer. This involves mapping scan-out pixel coordinates (i,j) into ray coordinates (s,t,u,v). Conceptually, the pixel coordinates are specified by rays, and the intersection of these rays with the s-t plane and u-v plane defines an (s,t,u,v) quadruple, as shown in Figure 3. The resulting mapping is given by the following four linear rational equations (a derivation is given in Appendix A).

$$s(i,j) = \frac{A_s i + B_s j + C_s}{A_w i + B_w j + C_w} \quad (1) \qquad t(i,j) = \frac{A_t i + B_t j + C_t}{A_w i + B_w j + C_w} \quad (2)$$

$$u(i,j) = \frac{A_u i + B_u j + C_u}{A_w i + B_w j + C_w} \quad (3) \qquad v(i,j) = \frac{A_v i + B_v j + C_v}{A_w i + B_w j + C_w} \quad (4)$$

The A, *B*, and *C* coefficients are defined with respect to the current position and orientation of the virtual camera (i.e. the current view). Each equation has a numerator of the same form, though the coefficients are different. The denominator is identical for all four equations.

These four equations represent the 2D planar slice of the 4D frame buffer, which maps an (i,j) pixel coordinate to an (s,t,u,v) ray index. As the scan-out iterates through i and j, these equations generate addresses into the 4D data structure. Note that the denominator varies with i and j, requiring per-pixel division. If, however, the image plane of the virtual camera is restricted such that it is always parallel to the s-t and u-v parameterization, the equations simplify to the following linear expressions.

$$s(i,j) = A'_s i + B'_s j + C'_s \quad (5) \qquad t(i,j) = A'_t i + B'_t j + C'_t \quad (6)$$

$$u(i,j) = A'_u i + B'_u j + C'_u \quad (7) \qquad v(i,j) = A'_v i + B'_v j + C'_v \quad (8)$$

Thus, we now have simple linear expressions in terms of pixel coordinates (i,j). Such expressions map well to hardware. The sacrifice for this elegance is that the orientation of our camera's image plane is now restricted. However, for applications such as autostereoscopic and CAVE-style virtual reality displays, it is practical to define the fixed viewing surface to be parallel to the s-t and u-v planes. That is, for these applications, this viewing configuration is inherent.

Our scan-out equations are similar to those used by Regan et al. in [RMRK99], but they are slightly more general. By limiting the viewing positions to points along a specific line, and by limiting the image plane to the face of the display device, Equations 5, 7, and 8 can be further simplified to those used in [RMRK99] (Equation 6 becomes unnecessary).

## 4.3. Point Casting and Scatter

Given a 4D frame buffer and the equations for scan-out, we must next tackle the issue of how to fill the frame buffer. The defining characteristic that separates the PixelView architecture from being "just a light-field viewer" is its ability to render geometric primitives into its 4D frame buffer. The geometry processing stage (i.e. subdivision and shader/texture) produces world-space points, each with an associated radiance map. These maps represent a sampled version of the outgoing radiance for each point. This radiance needs to be added to the 4D frame buffer for each s-t sample location. This process is dubbed "point casting" to indicate that a single point broadcasts its radiance out to a set of s-t sample points, instead of the more typical mapping to just a single camera's center of projection.

As shown is Figure 4, the process is performed by first iterating over the set of all s-t sample locations and finding the ray connecting the current sample location with the point primitive. This ray is then intersected with the u-v plane, and the 2D coordinate of that intersection determines the u-v sample location. This represents the (s,t,u,v) location in the 4D frame buffer where the radiance will be stored if it passes the 4D z-buffer test at that location. The

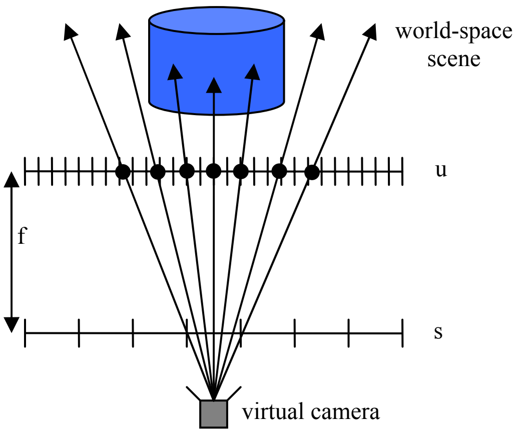

**Figure 3:** *2D depiction of rays displayed in scan-out. Each frame, PixelView evaluates a linear expression based on the camera matrix to determine which (s,t,u,v) rays are seen.*

intersection is given by the following two equations (a derivation is given in Appendix B).

$$u(s) = A''_u s + B''_u \quad (9) \qquad v(t) = A''_v t + B''_v \quad (10)$$

Thus, we must once again evaluate a linear expression, similar to the calculations required during scan-out. In this case, the coefficients are related to the (x,y,z) location of the scene point, the location of the u-v plane ($f$), and the terms in the matrices of Equation 16 (see Appendix B).

Note that, in general, the u-v intersection will not be an integer value. That is, the ray will usually not fall exactly in an (s,t,u,v) bin. Thus, the radiance contribution should be "scattered" into nearby bins.

## 4.4. Subdivision

Point primitives are ideal for the PixelView architecture because they specify a distinct ray to each s-t and u-v sample point, simplifying the point-casting process. However, it is possible to convert other primitives into this point-based representation through world-space subdivision. Note that all subdivision occurs without considering occlusion, which is handled through z-buffering at the time of point casting.

Polygonal models are subdivided into points until a sufficient density is achieved so that holes are not created when the points are mapped into the 4D frame buffer. The first pass of subdivision converts each polygon into a set of triangles to simplify subdivision. Subsequent passes perform a standard midpoint subdivision algorithm. The process continues until the length of each side of the triangle is less than half the size of a sample in the 4D frame buffer. This stopping criterion is not applied to the region as a whole, but instead to each individual subdivided triangle to allow for more precise sampling on large polygons. The method is similar to the Reyes approach [CCC87], but without explicit advance knowledge of the camera location, requiring uniform world-space subdivision.

Higher-order surfaces use subdivision algorithms more specific to their natural parameterizations. Thus, each of the supported higher-order surfaces is implemented with its own world-space subdivision procedure. For instance, a sphere is subdivided by algorithmically distributing points over the sphere's surface. By avoiding a traditional triangulation stage, the creation of geometry artifacts is avoided. Finding improved subdivision methods for primitives that are more complex remains an area for future research.

## 4.5. Shading

For point geometry fed directly into the hardware, PixelView needs color-shading information for each surface point's (x,y,z) coordinate. For polygonal and higher-order objects computed by PixelView, illumination must be applied.

Current rendering architectures depend on high-speed parallelism to accomplish the shading operations required at each point. However, the speed of a shading model is not as important when scan-out is decoupled from rendering/shading, as it is in PixelView. In order to create the illusion of smooth camera motion, traditional hardware must re-render an entire scene into a double buffer and then swap it at the next vertical refresh. This fill rate into the double buffer determines the frame rate of the system. Because PixelView can create user camera motion without resorting to reshading the entire scene, its frame rate is determined by the speed of the scan-out hardware, which is a constant independent of scene complexity. While the user is moving the view within a scene, a new scene is being shaded and is swapped in when complete. This independence allows implementation of more complex local and global shading algorithms with less concern for completing a frame by an arbitrary deadline, while still providing guaranteed view-update latency.

Hardware shading might be implemented using points output from subdivision and the normals generated during that process. For global illumination, this would imply a retained-mode graphics interface for primitive specification, or an immediate-mode interface for local illumination.

PixelView has the ability to interface with software renderers and offline storage of 4D frame buffers (e.g. 3D movies). This data could be loaded directly into the 4D frame buffer, bypassing PixelView's shading stage in a manner similar to how image-based rendering models are loaded. Another option is to transmit raw points and precomputed view-dependent global illumination, and then lets PixelView handle the placement of those points into the 4D frame buffer.

## 4.6. View-Dependency of Pixels

The ability to generate a 4D frame buffer without knowing the location of the user's camera in advance introduces the

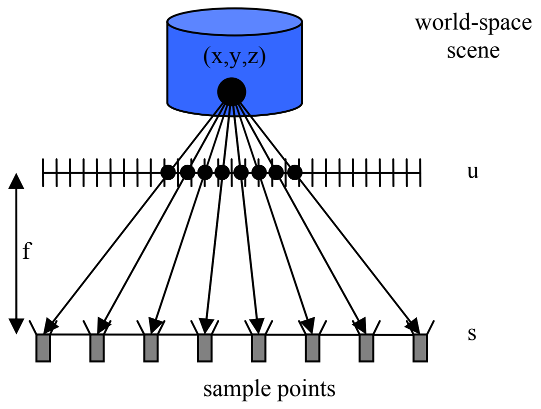

**Figure 4:** *2D depiction of a world-space point being rasterized into the 4D frame buffer via "point casting". The world-space point is tested against each s-t sample point to determine where u-v intersections occur. The point's radiance is then applied to those (s,t,u,v) samples.*

problem of accounting for scene elements with view-dependent specular reflections. Obviously, if all objects in the scene are diffuse, this is not an issue. If there is specular data, a method for shading, formatting, and point casting this extra information is needed.

The mapping of each outgoing view direction to the s-t sample points will be referred to as the *radiance map* of a point. For example, Phong radiance maps tend to be simpler than ray-traced reflective maps, which can resemble point-specific environment maps.

Dealing with point-specific radiance maps raises the issue of computing large numbers of samples for each point and transmitting them to the hardware. Luckily, a great deal of exploitable coherence exists in the raw maps [LH96]. There is much more radiance-map coherence for true object points than for arbitrary points in space. This coherence allows radiance maps to be transferred to and within the PixelView hardware as compressed data. For reflection and refraction, the data is not as structured, but in cases with a limited viewable range, the variation in illumination is far smaller than that found, for example, in a spherical environment map. This data could still be compressed using a more typical compression algorithm, because it exhibits the same type of spatial coherence seen when viewing the world through a normal camera.

The general structure of point-specific radiance-map coherence implies that wavelet-based compression would be a natural fit. However, there exists the opportunity to create compression algorithms and basis functions that are better suited to a specific shading model, such as for specular highlights.

The point-casting process is similar for both diffuse and specular points. PixelView acknowledges the difference between diffuse and specular points by requiring only a single color for a diffuse point and a full radiance map for all s-t samples for a specular point.

## 5. A PixelView Implementation

In order to demonstrate the feasibility and advantages of the PixelView architecture, we have chosen to implement a proof-of-concept prototype (see Figure 1), with the following objectives:

- To examine the feasibility of dynamic view selection at the time of pixel scan-out
- To measure the memory bandwidth utilization of dynamic view selection and investigate the impact of different memory organizations on the required bandwidth
- To investigate the addressing, coherency, and fill-rate implications of illuminating and shading each primitive from multiple viewing directions
- To explore the algorithm simplifications and tradeoffs implied by a practical hardware implementation

Some of these objectives could have been addressed entirely in simulation. However, modern system design tools make it quite easy to move rapidly from a functional simulation model to a FPGA prototype. Moreover, the advantages of physical prototypes are that they do not wallpaper over many of the engineering issues that can be easily overlooked in simulation, such as accurate memory models, signal distribution, routing delays, floor planning, and datapath complexity.

Another possible implementation avenue for the PixelView architecture would be to map it onto one of today's Graphics Processing Units (GPUs). It is clear that the trend in graphics hardware is towards increasing flexibility and generality. In effect, GPUs can be viewed as programmable parallel-processing units. We expect this to become increasingly true in the future. However, there are many aspects of GPUs that, at present, are rendering architecture specific or unexposed. Examples of this include frame buffer scan-out logic and memory organization, both of which are critical to demonstrating the feasibility of PixelView.

We have chosen to prototype only a limited subset of the PixelView architecture. Specifically, our system implements only one "light slab" [LH96] of an all-views architecture. A complete all-views implementation would include at least 4 (and more likely 6) slabs to enclose a region of empty space. More slabs would be necessary in scenes with intervening occluders. Of course, there are tradeoffs between the range of actual view-independence and the frequency of re-rendering, and we plan to investigate those tradeoffs more in the future. Even so, a single light-slab implementation is still technically interesting, because it maps directly to a through-the-window viewing paradigm. Thus, it would be able to support interactions similar to those of a single-wall CAVE VR architecture, but with support for correct stereo viewing for multiple participants.

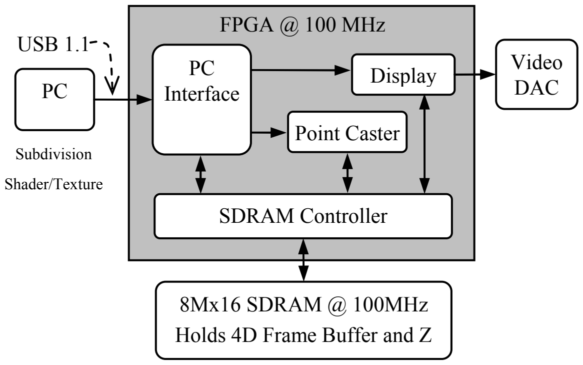

**Figure 5:** *Block diagram of the PixelView prototype. Point casting and display occur independently, allowing scan-out rate to be decoupled from rendering rate. A host PC implements the subdivision and shader/texture stages.*

Lastly, we have also chosen to use a general-purpose host processor to emulate the geometry processing stages of the PixelView pipeline. These stages generally require floating point computation and involve more decision-making and variations in control flow than the later stages of the PixelView pipeline. We intend to investigate a hardware version of this front-end geometry processing in the future.

### 5.1. Hardware Prototype

The hardware begins with the point-casting stage. Recall that the shader/texture stage outputs world-space (x,y,z) points along with their associated radiance maps that capture view-dependent reflectance. The hardware "rasterizes" these points into the 4D frame buffer. Visibility is determined via a corresponding 4D z-buffer. Scan-out is an independent function that uses view updates from the host PC to determine the latest view parameters at the start of each frame. This section gives details about how this hardware is implemented.

Figure 5 features a block diagram of the hardware system. The primary components are a single Xilinx XC2S300E Field Programmable Gate Array (FPGA), a single 8Mx16 SDRAM, a 10-bit video DAC, and a USB 1.1 port. The Xilinx and the SDRAM are both clocked at 100 MHz. The 4D frame buffer and 4D z-buffer reside in the SDRAM, each occupying 4Mx16. The frame buffer contains 16-bit RGB565 color values, and the z-buffer contains 16-bit two's complement depth values. Since we are limited to 4Mx16, the 4D buffers are organized as 8x8 s-t sample points, each with a corresponding set of 256x256 u-v samples.

We will first focus on point casting. The point caster receives (x,y,z,color) information from the PC. It uses this information to iterate over the s-t sample points and calculate the corresponding u-v bins. Each s-t sample point generates a conditional write to update the 4D frame buffer at the closest u-v sample. However, the write occurs only if the z value of the current point is "nearer" than the value currently at the corresponding location in the 4D z-buffer. Given enough (x,y,z) points, the 4D frame buffer will be “fully populated” and appear as a solid image.

Recall from Section 4.3 that determining u and v is a relatively straightforward evaluation of a linear expression. Specifically, it is an incremental calculation that utilizes accumulators. No multiplication or division is necessary.

Turning now to the scan-out/display hardware, it runs at a constant 640x480 at 60Hz. After vertical sync (i.e. at the start of each frame), the hardware grabs the latest view parameters from the PC. It then begins to iterate through (i,j) pixel coordinates just like a normal 2D scan-out. The difference is that the pixel coordinates for conventional scan-out map directly to frame buffer addresses, whereas our scan-out must calculate the equivalent (s,t,u,v) quadruple. The mechanisms for doing these calculations are similar to those for point casting, as they both involve the evaluation of a linear expression.

## 5.2. Scan-Out Memory Bandwidth

Bandwidth is always a primary concern in any graphics hardware system. It is reasonable to wonder how our 4D memory organization affects DRAM performance. Our hardware system has two main users of memory bandwidth: the point-casting stage and the display stage. The display stage generates sequential (i,j) pixel values as in normal scan-out, but these values are then mapped into 4D. Referring back to Figure 3, note that the s-t plane is sparsely sampled, whereas the u-v plane is finely sampled. Thus, scan-out tends to stay in a single s-t bin for several pixels. We can use this to our advantage by “swizzling” (s,t,u,v) such that the bank/row address of the DRAM stays in the same page for several iterations (Figure 6). Using this scheme, we achieve page-hit rates well over 90% during scan-out, and scan-out consumes only 4 percent of available bandwidth.

## 5.3. Point-Casting Performance

The prototype system is able to transmit and rasterize over 80,000 points per second. This equates to over five million rays per second written into the 4D frame buffer. Note that this rate is limited by the USB 1.1 interface between the host PC and the prototype, not by the point-casting hardware. At this rate, point casting consumes about 17 percent of available bandwidth. Factoring in scan-out and SDRAM auto-refresh, we have consumed less than 25 percent of available bandwidth. Thus, we have sufficient bandwidth to add additional point-casting units to process incoming points in parallel and increase point-casting performance. However, we would still ultimately be limited by the speed bottleneck of the USB interface, and so we chose not to do this.

## 6. Results

An important result of our research is the completion of the working prototype that demonstrates the benefits, capabilities, and plausibility of the PixelView architecture. This section provides examples of all supported primitive types captured from the PixelView prototype.

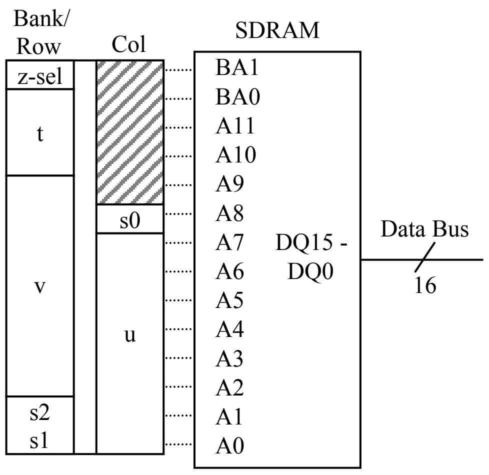

**Figure 6:** *Swizzling of (s,t,u,v) samples into SDRAM addresses to achieve increased scan-out coherence. Half the banks are used for z-buffer data and half for color data, determined by the high order bank pin.*

The example images are captured from a VGA scan converter connected to the VGA output of the PixelView prototype. All examples run at a constant frame rate of 60 frames per second within a 256x256 window inlaid in a 640x480 display. The images shown here have been enlarged to show detail. A better idea of the final output can be gained from watching the supporting video. The subdivision for these models has been computed on the host PC and transmitted to the board in a point stream format consistent with what a hardware-based subdivider would output.

The quality of reconstruction in these results is largely a function of memory size. The SDRAM can store only 8x8 s-t samples in the 4D frame buffer at a resolution of 256x256 and still allow room for 16-bit z-buffering. In addition, the reconstruction method used here is nearest neighbor. Better reconstruction methods, such as quadrilinear interpolation, could be implemented at the cost of additional memory bandwidth.

Our polygonal models are subdivided by the host PC and transmitted to the prototype as specular point samples. The model in Figure 8 shows a 3263-faced polygonal model of a cow with vertex normals. This is subdivided into 600899 Phong-shaded point samples, which are sufficient to render the scene for any view. The excess points are removed by z-buffering and all occlusions are properly resolved.

The point based terrain map data shown in Figure 9 represents a dataset of 2048x2048 (x,y,z) texture mapped points forming a model of Puget Sound in Washington. This data is easy for PixelView to work with because it does not need to subdivide or shade the points before point casting.

Image-based rendering datasets can be loaded directly into the 4D frame buffer and then viewed from a virtual camera position. The light field in Figure 10 was taken with a real-world light-field capture rig and digitized. The data set is comprised of 8x8 camera images, each with a resolution of 256x256 pixels, which map directly into the 4D frame buffer.

The final example, Figure 11, depicts 14 reflective and refractive spheres against a texture-mapped background; all described as higher-order surfaces. Ray tracing was used to compute the view-dependant radiance maps of each point in this data set. Each point and its radiance map is point cast and then scanned out at 60 frames per second.

## 7. Discussion and Future Work

While our prototype system addresses many of the issues posed by the PixelView architecture it leaves many unaddressed. Two issues of particular interest are how the PixelView architecture scales to a significantly larger 4D frame buffer, and how its rendering performance scales as multiple simultaneous views are supported. How to animate objects is another area of interest.

### 7.1. Scalability

We envision that a minimal practical PixelView system would incorporate at least 4G ray samples. Addressing such a large memory block poses immediate issues. Consider the cost of clearing the buffer's z and color values alone. We believe that the solution to this problem is to distribute the 4D frame buffer over multiple functional units, each with a slice of the 4D frame buffer that shares small overlap regions with its adjacent neighbors (Figure 7). We envision that the granularity of these modules would match the prevailing memory density, providing one geometry processing and rasterization/display unit per memory chip in order to maximize the system bandwidth.

However, this need for high memory bandwidth is further aggravated by the need to support multiple simultaneous views. If a PixelView-like system were employed to drive an autostereoscopic or holographic display, then it is conceivable that every ray in the frame buffer would need to be read out at the update rate. While the update rate might be minimal for a static scene, we would still like to support animations and dynamic scenes at interactive rates. We believe the solution here is redundancy. Each ray within the 4D frame buffer might need to be replicated as many as 16 times to support in excess of 1024 views. This further increases the memory requirements by a factor of 16 to 64G rays. While this seems unimaginable now, in ten years time it might be both achievable and affordable.

### 7.2. Animation

In a 4D frame buffer, the notion of a partial screen clear seems especially helpful, particularly for supporting animation. Ideally, one would like to remove all contributions from a given point set in a past frame and re-render only those portions of the frame buffer that have changed. An old double-buffering trick used in traditional systems might prove to be particularly useful for the PixelView system. Specifically, one can render static elements into a background buffer and then reload them to initialize each rendered frame. This trick is seldom used because it fixes the viewpoint in favor of fast animation, but it is particularly inviting for a view-independent architecture, like PixelView.

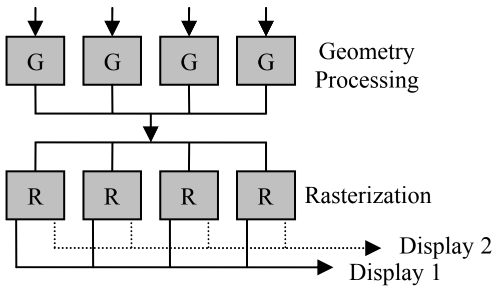

**Figure 7:** *Block diagram of scalability via multiple PixelView devices. Geometry processing and rasterization are performed in parallel, with each rasterizer controlling part of the light field, and subsequently part of each scan-out.*

### 7.3. Other Issues

A more general-purpose implementation of PixelView than our prototype would include the complete linear rational addressing implied by Equations 1 through 4. This seems achievable, because it is similar to the per-pixel divide required for perspective correct interpolation in today's hardware. We only require that similar circuitry be used in the frame buffer scan-out stage, where it could be heavily pipelined.

The image quality of PixelView could be improved with better reconstruction. A naïve implementation of quadrilinear interpolation causes an additional 16x increase in memory bandwidth. However, caching methods could be employed to reduce bandwidth requirements.

The point-casting bandwidth of PixelView could also be reduced considerably by including compression of the radiance maps. Off-the-shelf methods, such as wavelet compression, would probably do a good job, but we are particularly interested in the possibility of compression methods with basis functions optimized for radiance modeling. Any new compression bases would probably need to be used in combination with compression methods for general images, because the limiting case of a mirror reflector leads to a general image. Adding compression would increase computation in the primitive processing front end, but this increase is probably offset by the reduction in communication bandwidth.

## 8. Conclusions

We have presented PixelView, a new hardware rendering architecture that generates all possible views of a scene after a single traversal of the scene description. PixelView supports a wide range of rendering primitives, including polygonal meshes and higher-order surfaces. It also supports recently developed point and image-based rendering primitives. Moreover, we have developed a working hardware prototype of a 4D, z-buffered frame buffer that supports dynamic view selection at raster scan-out to demonstrate the viability of such a system. Graphics systems like PixelView are capable of supporting extremely low display update latencies. They will also enable efficient rendering of shared virtual environments supporting multiple simultaneous participants. In the future, view-independent graphics rendering architectures, like PixelView, will also be essential to provide the multitude of viewpoints required for real-time autostereoscopic and holographic display devices.

## References

[Bal99] Bala, K., “Radiance interpolants for interactive scene editing and ray tracing” Doctorial Thesis, Dept. of Computer Science and Electrical Engineering, Massachusetts Institute of Technology (MIT), 1999.

[BWG03] Bala, K., Walter, B., and Greenberg, D., “Combining edges and points for interactive high-quality rendering”, Proc. SIGGRAPH 2003.

[BFMZ94] Bishop, G., Fuchs, H., McMillan, L., and Zagier, E.S., “Frameless rendering: Double buffering considered harmful”, Proc. SIGGRAPH 1994, pp. 175-176.

[CCC87] Cook, R. Carpenter, L, and Catmull, E., “The Reyes image rendering architecture,” Proc. SIGGRAPH 1987, pp. 95 102.

[Gla88] Glassner, A.S., “Spacetime ray tracing for animation”, IEEE Computer Graphics and Applications, 60-70, 1988.

[GGSC96] Gortler, S.J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F., “The lumigraph”, Proc. SIGGRAPH 1996, pp. 43-54.

[Hal98] Halle, M., “Multiple viewpoint rendering”, Proc. SIGGRAPH 1998, pp. 243-254.

[LH96] Levoy, M. and Hanrahan, P., “Light field rendering”, Proc. SIGGRAPH 1996, pp. 31-42.

[MMB97] Mark, W. R., McMillan, L., and Bishop G., “Post-rendering 3d warping”, Proc. 1997 Symposium on Interactive 3D Graphics, pp. 7-16, 1997.

[OCMB95] Olano, M., Cohen, J., Mine, M., and Bishop, G., “Combating rendering latency”, 1995 Symposium on Interactive 3D Graphics, 1995.

[OKTD02] Owens, J.D., Khailany, B., Towles, B., and Dally, W.J., "Comparing Reyes and OpenGL on a stream architecture," Proc. SIGGRAPH/ EUROGRAPHICS conference on graphics hardware, pp. 47-56, 2002.

[PKKG03] Pauly, M., Keiser, R., Kobbelt, L.P., and Gross, M., “Shape Modeling with Point-Sampled Geometry”, Proc. SIGGRAPH 2003.

[PZvBG00] Pfister, H., Zwicker, M., van Baar, J., and Gross, M., "Surfels: Surface elements as rendering primitives", Proc. SIGGRAPH 2000, pp. 335-342.

[PBMH02] Purcell, T.J., Buck, I., Mark, W.R., and Hanrahan, P., “Ray tracing on programmable graphics hardware”, Proc. SIGGRAPH 2002, pp. 703-712.

[RMRK99] Regan M., Miller G., Rubin S., and Kogelnik C., "A Real Time Low-Latency Hardware Light-Field Renderer", Proc. SIGGRAPH 1999, pp. 287-290.

[RP94] Regan M. and Pose R., "Priority rendering with a virtual reality address recalculation pipeline", Proc. SIGGRAPH 1994, pp. 155-162.

[RL00] Rusinkiewicz, S. and Levoy, M., “QSplat: A multiresolution point rendering system for large meshes”, Proc. SIGGRAPH 2000.

[SGHS98] Shade, J., Gortler, S., He, L., and Szeliski, R., "Layered depth images," Proc. SIGGRAPH 1998, pp.231-242, 1998.

[TK96] Torborg, J. and Kajiya, J.T., “Talisman: Commondity realtime 3D graphics for the PC”, Proc. SIGGRAPH 1996, pp. 353-363.

[War98] Ward, G.J., “The Holodeck: a parallel ray-caching rendering system”, 2nd Eurographics Workshop on Parallel Graphics and Visualization, 1998.

[ZPKG02] Zwicker, M., Pauly, M., Knoll, O., and Gross, M., “Pointshop 3D: an interactive system for point-based surface editing, Proc. SIGGRAPH 2002, pp. 322-329.

## Appendix A. Derivation of Scan-Out Equations

Equation 11 below gives a model of the “image plane” of the virtual camera used to specify the current view. Specifically, given a point ***E*** (the eye point), a vector ***P*** that spans the image plane in the i-direction, a vector ***Q*** that spans the image plane in the j-direction, and a vector ***O*** from the eye point to the (0,0) pixel of the image plane, Equation 11 maps pixel coordinates (i,j) to (x,y,z) points in world space.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix} + \lambda \cdot \begin{bmatrix} P_x & Q_x & O_x \\ P_y & Q_y & O_y \\ P_z & Q_z & O_z \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \qquad (11)$$

The following two equations represent a similar mapping for the s-t and u-v planes. Specifically, they define the world-space points that fall on the planes. In Equation 12, vector ***L*** spans the s-t plane in the s-direction, vector ***M*** spans the s-t plane in the t-direction, and vector ***S*** goes from the world origin to (s,t) = (0,0). Similar definitions exist for Equation 13.

$$\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}L_x & M_x & S_x\\L_y & M_y & S_y\\L_z & M_z & S_z\end{bmatrix}\cdot\begin{bmatrix}s\\t\\1\end{bmatrix}\quad(12)\qquad\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}F_x & G_x & U_x\\F_y & G_y & U_y\\F_z & G_z & U_z\end{bmatrix}\cdot\begin{bmatrix}u\\v\\1\end{bmatrix}\quad(13)$$

If we place the s-t plane at $z = 0$, define ***L*** parallel to the x-axis, and define ***M*** parallel to the y-axis, Equation 12 simplifies to Equation 14. Similarly, if we place the u-v plane at $z = f$, define ***F*** parallel to the x-axis, and define ***G*** parallel to the y-axis, Equation 13 simplifies to Equation 15.

$$\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}L_x & 0 & S_x\\0 & M_y & S_y\\0 & 0 & 0\end{bmatrix}\cdot\begin{bmatrix}s\\t\\1\end{bmatrix}\quad(14)\qquad\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}F_x & 0 & U_x\\0 & G_y & U_y\\0 & 0 & f\end{bmatrix}\cdot\begin{bmatrix}u\\v\\1\end{bmatrix}\quad(15)$$

Setting the right side of Equation 11 equal to the right side of Equation 14, we can solve for s and t in terms of i and j. Likewise, we can solve for u and v in terms of i and j using Equations 11 and 15. This yields Equations 1 through 4 in Section 4.2.

Moreover, if the image plane of the virtual camera is restricted such that it is always parallel to the s-t and u-v planes, the $P_z$ and $Q_z$ terms in Equation 11 become zero, and consequently, the denominator becomes a constant. This results in Equations 5 through 8 in Section 4.2.

## Appendix B. Derivation of Point-Casting Equations

Assuming the simplified s-t and u-v configuration from Appendix A, the following equation represents the ray starting at a particular s-t sample point and passing through a particular point on the u-v plane:

$$\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}L_x & 0 & S_x\\0 & M_y & S_y\\0 & 0 & 0\end{bmatrix}\cdot\begin{bmatrix}s\\t\\1\end{bmatrix}+\tau\left(\begin{bmatrix}F_x & 0 & U_x\\0 & G_y & U_y\\0 & 0 & f\end{bmatrix}\cdot\begin{bmatrix}u\\v\\1\end{bmatrix}-\begin{bmatrix}L_x & 0 & S_x\\0 & M_y & S_y\\0 & 0 & 0\end{bmatrix}\cdot\begin{bmatrix}s\\t\\1\end{bmatrix}\right)\quad(16)$$

Viewing this expression as a system of three equations, note that the last equation is simply $z = \tau{\cdot}f$. Consequently, you can easily solve for $\tau$ and substitute back to solve for u and v in terms of s and t. This yields Equations 9 and 10 in Section 4.3.

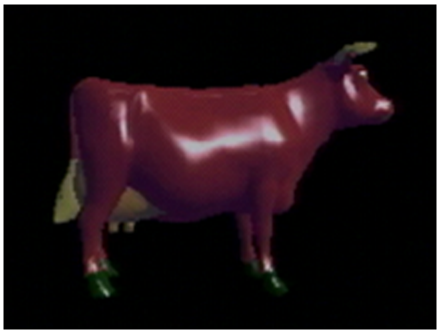
**Figure 8:** *Screen capture of a subdivided polygonal model.*

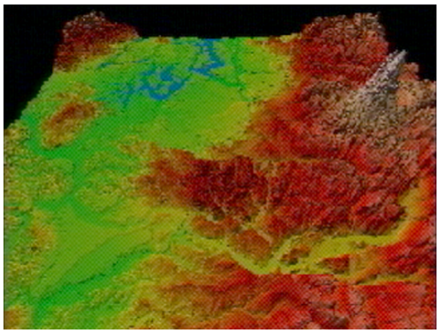
**Figure 9:** *4-million point-sample model of Puget Sound.*

**Figure 10:** *A "real-life" 8x8 by 256x256 light field.*

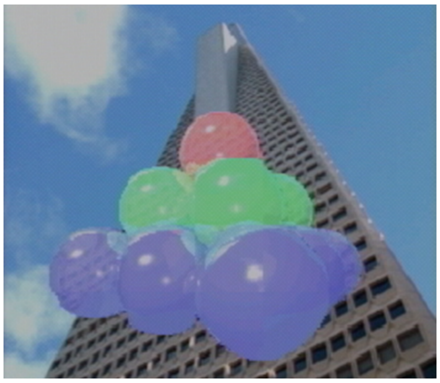
**Figure 11:** *Ray-traced scene with view-dependent points.*