RASTER GRAPHICS ARCHITECTURES

The field of computer graphics involves rendering or visualizing of concrete or abstract data, for example, the 3-D display of a mechanical part, the visualization of the airflow over a wing, or the visualization of weather patterns. Since the very early days of computer graphics it has been apparent that the computational complexity and the bandwidth requirements

for generating images exceeded by a wide margin the capabilities of general-purpose processors. Therefore, dedicated hardware was developed to accelerate the rendering of computer-generated images. Only recently have general-purpose microprocessors reached a performance level that permits to build interactive 3-D graphics systems without dedicated graphics hardware. This article starts with a survey of the historical evolution of graphics hardware and a description of the overall architecture of a graphics system. Next, we examine in more detail the structure of raster graphics architectures for polygon rendering. We discuss the basic algorithms and some exemplary, concrete architectures. The article concludes with a list of references for further reading that enable the reader to study the field of graphics hardware in more detail.

HISTORY

Research and development of computer graphics hardware started in 1950 with the Whirlwind Computer at the Massachusetts Institute of Technology (MIT). It employed a modified oscilloscope to visualize and analyze the stability of aircrafts. A few years later the SAGE air-defense system used a vector screen to display radar information. SAGE employed a light pen to allow users to identify objects on the screen.

Ivan Sutherland's seminal doctoral dissertation (1) marks the birth of modern computer graphics. He introduced many concepts that are still in use, for example, hierarchical data structures to define geometric information to be rendered onto the screen, instantiation of prototype objects, and many interaction and user-interface techniques. Sutherland's Sketchpad drawing system demonstrated the utility of these techniques.

Throughout the 1960s and early 1970s vector displays were the only devices available for interactive graphics displays. During that time the evolution was influenced by the fact that most computing environments were based on terminals attached to a central mainframe over low-bandwidth connections. Consequently, substantial intelligence and compute power had to be put into the terminals to process a stream of high-level commands and data coming from the host. Due to the substantial cost of graphics subsystems, they were mostly restricted to defense and industrial applications [e.g., flight simulators, CAD (computer aided design), or simulations analysis].

In the mid 1970s affordable semiconductor memories became available, leading to the introduction of the first raster graphics systems. Initially, raster graphics systems were very expensive, had a low resolution, and were fairly slow. The development of specialized raster graphics algorithms and dedicated hardware in the 1980s enabled interactive raster graphics (2,3).

The final breakthrough of raster graphics occurred with the introduction of workstations and personal computers. These machines introduced the new computing model of decentralized, autonomous computers. Now, the graphics subsystem was integrated more closely with the CPU (central processing unit) and became an integral part of the system architecture instead of being just another peripheral device.

Since about 1990, the evolution of raster graphics systems has moved toward higher performance using parallel processors and higher functions, in particular texture mapping and image processing.

BASIC RASTER GRAPHICS ARCHITECTURE

Figure 1 shows the basic block diagram of a graphics computer (4). The host computer runs the operating system and application software. The application responds to commands and values provided by the input devices and generates the graphics data to be displayed on the output device. The graphics subsystem converts the graphics data received from the host computer into a data stream that can be displayed by the output device.

Graphics subsystems are specialized to operate on a set of graphics primitives. Different graphics subsystems support different sets of primitives. Such primitive objects are, for instance, line segments, triangles, general polygons, text, or curved surfaces. Figure 2 shows a more detailed view of the graphics subsystem for a raster graphics subsystem.

Geometry Subsystem

The geometric operations manipulate the geometric primitives in the scene, for example, triangles or line segments, to prepare them for rendering. The results of the geometric operations are primitives that are transformed into the coordinate system of the output device and carry associated color or intensity information. The geometric operations include several or all of the following steps. More details on these operations can be found in every standard text on computer graphics, for example, Ref. 5.

Modeling Transformations. The modeling transformations convert the description of a geometric primitive from its intrinsic modeling coordinate system to a world coordinate system that is used to describe the entire scene. Modeling transformations are commonly specified as affine transformations using homogeneous coordinates and are expressed as 4×4 matrices. Usually, the coordinates and the parameters of the matrices are specified as floating-point numbers.

Lighting. The effects of light sources in the scene are computed taking into account the position of the light source, the position of the viewer (eye point), the orientation of the primitive (normal vector) and the primitive's surface properties (e.g., reflectivity or shininess). The lighting model according to Phong is frequently used to determine the intensity/color of a point on a surface.

$$I = I_a \cdot k_a + I_l \cdot [k_d \cdot (\boldsymbol{N} \cdot \boldsymbol{L}) + k_s \cdot (\boldsymbol{R} \cdot \boldsymbol{E})^n] \qquad (1)$$

Equation (1) describes the Phong model using the geometry shown in Fig. 3.

I is the intensity received by the viewer. I_a and I_l are the intensity of the ambient light and the light source. The coef-

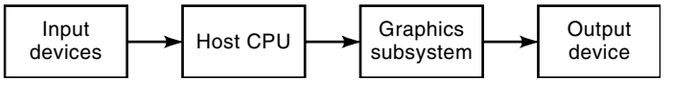

Figure 1. Basic structure of a graphics computer. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

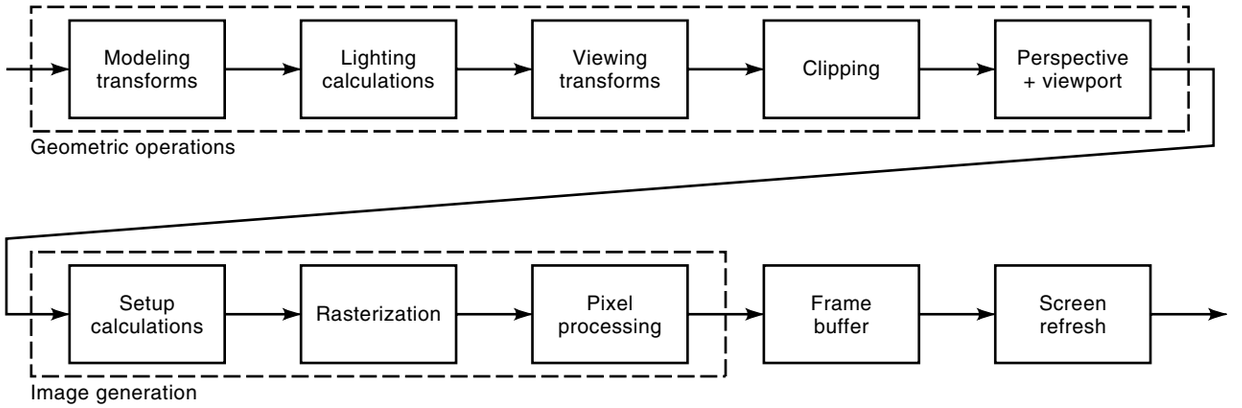

Figure 2. Components of a raster graphics system. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

ficients k_a, k_d, and k_s specify the surface reflectivity for ambient, diffuse and specular light. The coefficient n is the glossiness, controlling the extent of specular highlights.

Viewing Transformation. The viewing transformation positions and orients the scene according to the viewing parameters (or camera parameters), for example, viewing direction or field of view. Like the modeling transformations, the viewing transformation is expressed using a 4×4 matrix.

Clipping. Parts of the scene that fall outside of the viewing frustum do not need to be processed by any subsequent steps. The viewing frustum is described by planes enclosing the volume visible for the given viewing parameters. Clipping geometric primitives against the viewing frustum requires splitting them along the clipping planes. Such computations are performed using floating-point calculations.

Another technique similar to clipping is known as culling. Entire objects composed of many primitives are tested against the viewing frustum. If an entire object is outside of the viewing frustum the object is not considered further for rendering. Culling is usually performed at the application level, but can be performed in hardware if the object structure is preserved, that is, if the relationship between an object and its primitives is not destroyed.

Perspective Transformation and Viewport Mapping. Optionally, the viewing transformation and clipping are followed by a perspective projection to account for perspective foreshortening. The perspective projection converts homogeneous coordinates into real 3-D coordinates by dividing out the w component of the coordinate. Viewing transformations and perspective projection are performed using floating-point calculations. Finally, the viewport mapping scales and translates the coordinates from world coordinates to device coordinates. Device coordinates are device-specific and are typically specified with integer or fixed-point values, for example, pixel positions. Hence viewport mapping involves casting the coordinate values from floating-point to fixed-point representation.

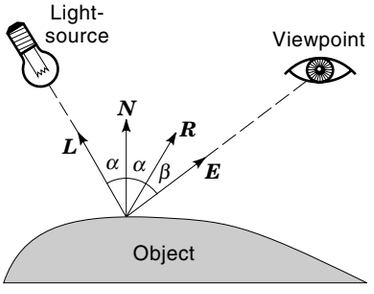

Figure 3. Geometry of the Phong lighting model. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

Image Generation Subsystem

Raster displays build the screen image from many small dots on the screen, the picture elements, or pixels. As in a mosaic, each of the pixels can be controlled individually to assume a color and intensity. An overview over the principles of raster graphics can be found, for instance in Ref. 5.

Today, raster displays are dominant in workstations and personal computers. Their principal advantage over vector displays is that they can display images of arbitrary complexity without flicker and they can display shaded images instead of only wireframes. They obviously provide a much larger utility than vector displays. The main component of a raster graphics system is the frame buffer. The frame buffer is a special memory that provides storage for every pixel on the screen. In the simplest case the frame buffer stores the color of every screen pixel. Advanced raster graphics systems store additional information for every pixel, for example, a depth value for visible surface determination or transparency information (see the following section).

The setup calculation takes as input the coordinates of the triangle vertices and the colors at the vertices and computes the coefficients of the bilinear expression for the color components and the depth value. The rasterization step generates the pixels covered by the triangle. Pixel processing operations are arranged as a sequence of operations applied to the pixels generated during rasterization. The screen refresh is performed by periodically scanning out the contents of the frame buffer onto the screen. The frame buffer decouples the image generation from the screen refresh so that these two functions

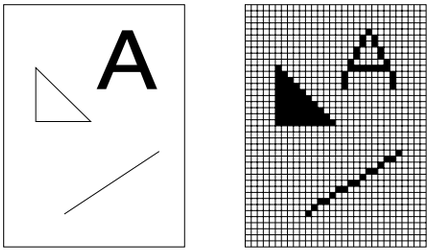

Figure 4. Rasterization of geometric primitives. (a) vector-based, ideal shape of the primitives; (b) rasterized primitives. (Reprinted from (4) by courtesy of Marcel Dekker Inc.).

can be performed asynchronously, which is why the time for generating the entire image does not affect the screen refresh rate.

Rasterization. To fill the frame buffer the drawing primitives are subdivided into pixels, a process known as scan-conversion or rasterization. Figure 4 illustrates this process. The pixels covered by the primitives are determined and the corresponding locations in the frame buffer are overwritten with the color of the primitives. Several algorithms have been developed to efficiently rasterize simple geometric shapes like lines or triangles. These algorithms exploit coherence to compute the pixels covered by a primitive in an incremental fashion. A line segment connecting two pixels ($x1$, $y1$) and ($x2$, $y2$) is described by a linear equation $y = mx + b$, where m is the slope of the line segment and b denotes where the line intercepts the y axis.

The simple approach to drawing that line segment is to evaluate line equation for every x between $x1$ and $x2$, as shown in the following sample C code.

Algorithm 1. Pseudo-code for simplistic line drawer

```
for (x=x1 ; x<=x2 ; x++) {
  y = m*x + b ;
    /* evaluate the line equation at (x,y) */
  set_pixel (x,y) ;
    /* store pixel into the frame buffer */
  }
```

This procedure, even though functionally correct, is computationally expensive because it relies on floating-point computations and requires several multiplications.

A more efficient implementation of the line drawing algorithm is shown below. It is known as digital differential analyzer (DDA). The sample C code which follows illustrates how the multiplication operation is avoided using an iterative scheme employing successive additions. (This algorithm will generate connected pixels if the line has a slope between -1 and $+1$.)

Algorithm 2. DDA line drawing algorithm

```
y=y1 ; /* initialize y with the start value */
for (x=x1 ; x<=x2 ; x++) {
  set_pixel (x,y) ;
    /* store the pixel into frame buffer */
  y = y+m ;
    /* incrementally compute next y value */
  }
```

This algorithm is not optimal as it still uses floating-point numbers but it illustrates how incremental computations can simplify and speed up the rasterization process. There exist a number of algorithms that use only integer or fixed-point numbers, for example, the Bresenham algorithm (6). Incremental algorithms are more easily implemented in hardware, see Fig. 5. Note how the structure inside the shaded blocks is used to incrementally compute the x and y coordinates of the line. This block is a linear interpolator to compute values of an expression $Ax + B$ for successive $x = 0, 1, 2, \ldots$

Another important primitive in graphics is the triangle. Triangles are the simplest 2-D primitives and are used as the basic rendering primitive in many raster systems. To display a triangle its interior must be filled with a color. The simplest case is flat shading where all pixels covered by the triangle are assigned the same color. The Gouraud shading algorithm interpolates colors specified at the vertices across the triangle. The C code which follows illustrates how incremental computations can be applied to rasterize a triangle while interpolating its color using Gouraud shading.

Algorithm 3. Algorithm for rasterizing a Gouraud-shaded triangle (see also Fig. 6)

```
/* Assume scan-aligned bottom triangle */
x_left = x1 ;
x_right = x2 ;
c_left = c1 ;
for (y=yt ; y <= yb ; y++) {
  color = c_left ;
  for (x=x_left ; x <= x_right ; x++) {
    fill_pixel (x,y, color) ;
    color += mc_x ;
  }
  x_left += m_left ;
  x_right += m_right ;
  c_left += mc_left ;
}
```

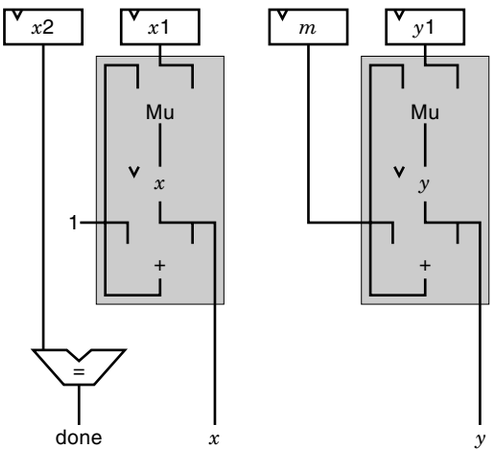

Figure 5. Block diagram of DDA line drawer. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

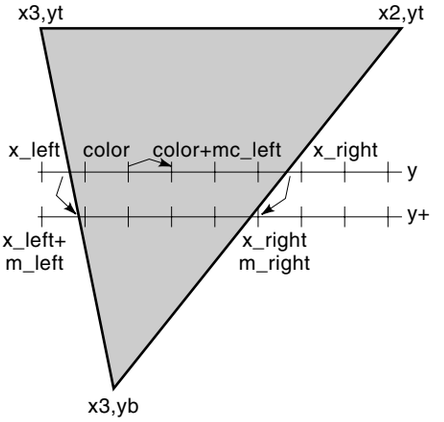

Figure 6. Principle of triangle rasterization (see text). (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

The algorithm assumes a triangle with a horizontal (scanline-aligned) top edge. A DDA line drawing algorithm computes the pixels on the leading and trailing edge within each scanline. Then the pixels between starting and trailing edge are filled. Figure 6 illustrates the procedure. The algorithm relies on the fact that the color (and depth) changes linearly across the triangle, that is, all colors in the triangle lie on a plane that is described by a linear expression $Ax + By + C$ in the screen coordinate system. The values `mc_left` and `mc_x` are therefore the gradients of the color plane along the left edge of the triangle and the x direction respectively. The evaluation of a linear expression $Ax + By + C$ occurs frequently in computer graphics. The computation of this expression can be implemented using two linear interpolators, and it is therefore also known as a bilinear interpolation. Figure 7 shows the block diagram of a bilinear interpolator.

Pixel Processing. The most common pixel operations are z-buffering, texture mapping, and alpha blending. z-buffering is an algorithm for hidden surface removal. The z-buffer is a memory that stores a depth value at every pixel. These depth values indicate the distance from the viewer to the object visible at each pixel. During rasterization a depth value is generated for each pixel. The stored and the new depth values are compared. Only if the new depth value is smaller than the stored depth value, that is, the new object is closer to the viewer at that pixel, is the new depth value stored in the z-buffer and the new color is stored in the frame buffer. Otherwise the new pixel is discarded.

The main cost of z-buffering is the amount of extra memory required to implement the z-buffer. Typically, depth values are represented with 2 to 4 bytes per pixel. Depending on the display size, the z-buffer can therefore require several MB of memory. The advantage of z-buffering is that it allows objects to be rendered in any order because visibility is determined at every pixel. Other hidden-surface removal algorithms require objects to be sorted prior to rendering; this can incur significant overhead.

Texture mapping applies image information, the texture, to the surface of rendered objects. It is used to enhance the visual realism of rendered images without increasing the number primitives in the scene. Figure 8 shows the steps for texture mapping an image onto a surface.

Similar to color and depth values, texture coordinates are computed for every pixel covered by the primitive. These texture coordinates are used to access pixels in the texture map. The color value retrieved from the texture map is applied to the surface. In practice, this basic texture mapping algorithm is refined to avoid perspective distortion of the texture and to reduce aliasing effects due to undersampling the texture map. More details can be found for instance in Refs. 5 and 7.

Alpha blending mixes the new pixel color with the color stored in the frame buffer using a weighting factor called alpha. Equation (2) describes one of several alpha blending techniques:

$$c = \alpha \cdot c_{\text{new}} + (1 - \alpha) \cdot c; \quad 0 \le \alpha \le 1 \tag{2}$$

The value alpha determines how much of the new pixel color c_{new} affects the color c already stored in the frame buffer. Alpha blending is primarily used for two purposes, transparency and anti-aliasing. To simulate transparency of a primitive, the alpha value represents how transparent the object is (0 for fully transparent, 1 for fully opaque). The transparency value can be computed similarly to color values by interpolation across the triangle.

For anti-aliasing, alpha values represent coverage information (0: object doesn't cover the pixel; 1: object covers the pixel fully). The rasterizer computes this coverage information for all pixels along the edges of a triangle, while interior pixels always cover the entire pixel. Using the coverage information, aliasing effects like *staircases* along the edges are reduced. Figure 9 illustrates this.

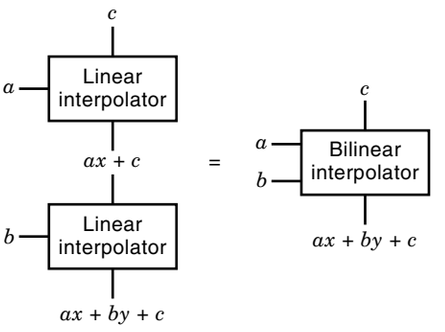

Figure 7. Bilinear interpolator. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

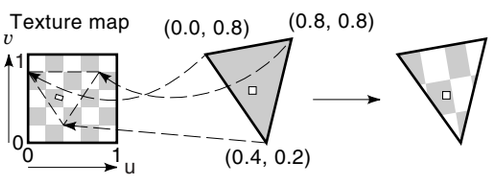

Figure 8. Texture mapping. The texture coordinates (u, v) defined at the vertices are interpolated across the triangle. The pixel color is determined by looking up the texture color for the interpolated texture coordinates. Note how the highlighted pixel is distorted when mapped into the texture map. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

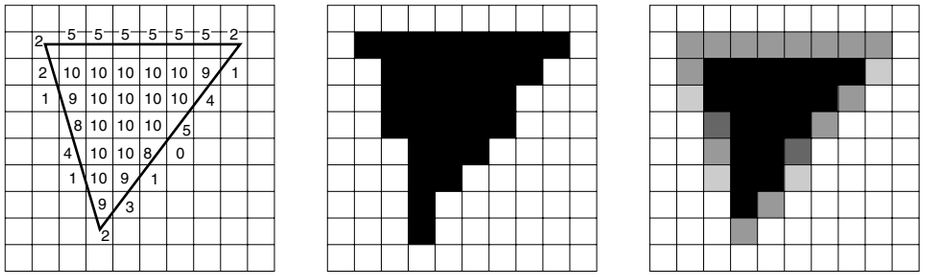

Figure 9. Anti-aliasing. (a) Pixel grid with triangle. The numbers denote coverage information. (b) Aliased raster image. (c) Anti-aliased raster image. (Reprinted from (4) courtesy of Marcel Dekker Inc.)

Screen Refresh and Frame Buffer Memory. Certain output devices, in particular cathode ray tubes (CRTs), require continuous rewriting of the display surface to create the impression of a steady, flicker-free image at a rate of at least 30 times a second. Ergonomic standards require much higher refresh rates of 80 Hz and more. Figure 10 shows the block diagram of a video controller that performs the screen refresh in a raster graphics system.

The frame buffer is scanned out periodically by the video controller that addresses the pixels in the frame buffer in scanline order. The pixel values retrieved from the frame buffer are converted to analog signals using digital-to-analog converters (DAC). The output of the DACs is fed to the monitor synchronized with the trace of the electron beam scanning across the screen.

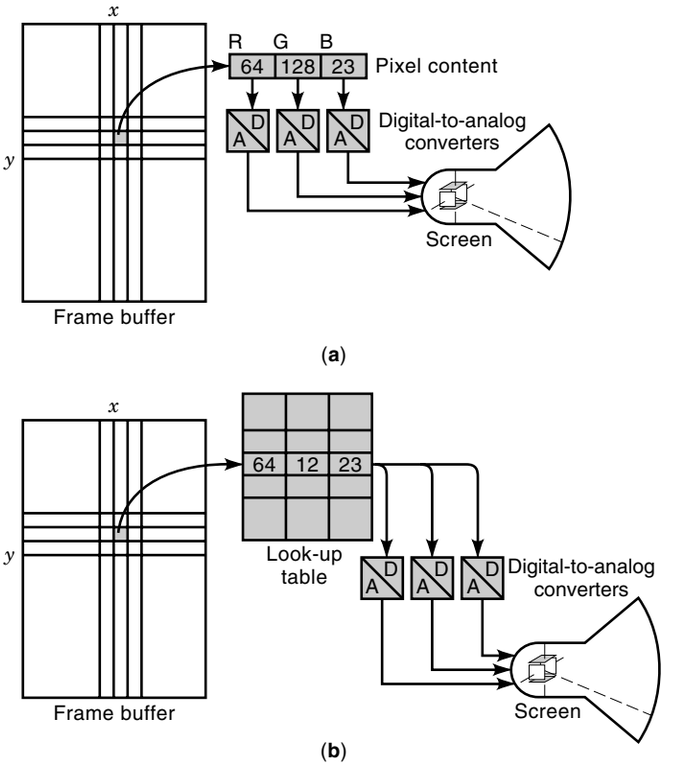

Figure 10. Operating principle of (a) a true-color graphics system, and (b) a color mapped graphics system. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

The frame buffer output can be interpreted as a color (or gray value) to be displayed on the screen. Then the output of the frame buffer is connected directly to the DAC as shown in Fig. 10(a). Such displays are called true-color displays because the color stored in the frame buffer is the color that appears on the screen. The advantage of true-color displays is that every pixel on the screen can be assigned a different color. This is very important for generating realistic looking images with many shades of colors. However, true-color displays demand a significant amount of frame-buffer memory because colors are stored with 16 or 24 bits per pixel.

Another way to interpret the frame buffer contents is to use the pixel values as an index into a look-up table (or palette) of color values [Fig. 10(b)]. The look-up table contains color values that are selected by the pixel values. Those color values are then fed into the DACs to be displayed on the screen. Such displays are known as index-color displays or as color-mapped displays. Color-mapped displays require less frame-buffer memory than true-color displays but offer only a limited number of colors that can be displayed simultaneously. The actual number of simultaneous colors depends on the length of the color map. In typical color-mapped systems the frame buffer stores 8 bpp which allows selection from 256 values in the color map. The number of available colors depends on the width of the color map, typically 24 bits (approximately 16.7 million colors).

Look-up tables are also used to correct for the nonlinear response of the CRT. Equation (3) shows the relationship between the computed pixel color C and the displayed intensity I on the CRT screen.

$$I = k \cdot C^{\gamma} \tag{3}$$

The constants k and γ are dependent on the monitor. Gamma correction [named after the exponent in Eq. (3)] linearizes the monitor characteristic by predistorting the color value presented to the monitor. A video look-up table (VLT) is loaded with values that produce a linear monitor response.

In raster graphics systems the screen refresh requires reading out the entire frame-buffer contents at the refresh rate. This puts a very high demand on the memory bandwidth for the frame buffer. Equation (4) describes time available for accessing a single pixel in the frame buffer during screen refresh.

$$t_p = \frac{\left(\frac{1}{f_r} - t_v\right) \Big/ V - t_h}{H} \tag{4}$$

The following symbols are used in Eq. (4):

f_r: Refresh rate, typically between 60 Hz and 120 Hz.

t_v: Vertical retrace time for bringing the electron beam from the last scanline to the first scanline, typically between 600 and 1250 μs.

t_h: Horizontal retrace time for bringing the electron beam from the end of a scanline to the start of the next scanline, typically between 4 and 10 μs.

V: Vertical screen resolution, that is, the number of scanlines.

H: Horizontal screen resolution, that is, the number of pixels per scanline.

The bandwidth for accessing the frame buffer depends on the pixel access time t_p and how many bytes (bpp) must be accessed for each pixel. Equation (5) is used to determine the bandwidth B_r for transferring the pixel data from the frame buffer to the display.

$$B_r = \frac{bpp}{t_p} \tag{5}$$

Note that B_r denotes the peak bandwidth at which pixel information must be supplied to the display. The average bandwidth is computed as $H \cdot V \cdot \text{bpp} \cdot f_r$.

In addition to the video logic, the rasterizer must access the frame buffer to write the image into the frame buffer. This means that additional bandwidth into the frame buffer is required to give the rasterizer access to the frame buffer. Frame-buffer designs must ensure that there is enough bandwidth available that the rasterizer is not slowed down because access to the frame buffer is denied. A good description of the problems of frame-buffer design can be found in Ref. 8. Several strategies are available to accomplish this goal. To satisfy the bandwidth requirement frame buffers are highly optimized memory subsystems that are built from multiple banks of memory that are accessed in parallel. The values read from these banks of frame-buffer memory are stored in a shift register from where the pixel values are read serially to the display. The shift register acts as a fast intermediate memory between the frame buffer and the display. The benefit is that the internal memory array is accessed less often, thus its availability for access from the rasterizer is increased.

This concept has been integrated into single-chip memory architecture known as video-random-access memories (VRAM). Figure 11 shows a block diagram of a VRAM chip.

VRAMs are dual-ported dynamic random-access memories (DRAM). One port behaves like a standard DRAM interface and is used for filling the frame buffer. The other interface is a specialized graphics port that allows the one-step loading of an entire row of the internal memory array into a shift register from where the pixel values can be read at high speed. Only one port can have access to the internal memory array of the VRAM chip. The serial port has priority over the parallel port because the screen refresh must follow a fixed timing and must not be delayed.

Another way to increase the availability of the frame buffer is double-buffering. Here, the system is equipped with two full-frame buffers. At any given time, one is connected to the video logic for screen refresh. This buffer is often referred to as the front-buffer because its contents are initially displayed on the screen. The second buffer (back-buffer) is filled by the rasterizer. When the image is completed the buffers are swapped, that is the second buffer is displayed while a new image is rendered into the first buffer. Double-buffering completely decouples image generation and screen refresh. An additional advantage of double-buffering is that the buffers are only swapped when the next image is finished. Therefore, the screen always contains a complete image, that is, the process of building the image from its primitives is completely hidden from the viewer. Double-buffered systems are often used for animated images. The biggest disadvantage of double-buffered systems is the extra cost for the second frame buffer.

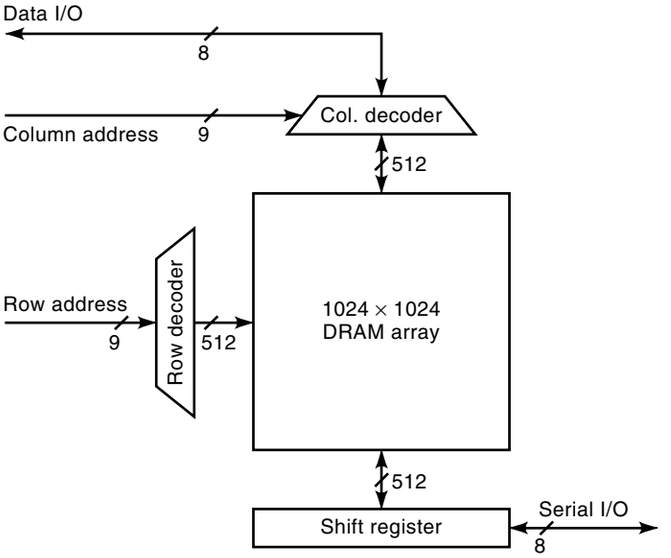

Figure 11. Basic architecture of Video-RAM (VRAM). The example shows a 1 Mb VRAM, organized as 128 K × 8. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

In virtual reality (VR) systems it is important to keep the latency between user input, for example, head motion, and display update to a minimum in order to avoid possible motion sickness. Since double-buffering adds up to one full frame-time to that latency, VR systems attempt to avoid double-buffered systems.

PARALLEL RASTER GRAPHICS ARCHITECTURES

Raster graphics forms the basis of most modern graphics systems. A description of the basic functional units forming a raster graphics system has already been given. We will discuss different classification systems for raster graphics architecture. Finally, we will describe small numbers of actual raster graphics architecture and highlight different optimizations in the design of graphics systems.

Definition and Classification

The large variety of actual graphics systems has proven to resist characterization in a single, comprehensive taxonomy. Several partial attempts at such taxonomies have been de-

scribed in the literature (9,10,11,12). We will describe classifications that emphasize different aspects of rendering architectures in the following section.

Sorting. In his dissertation Molnar (10) suggests rendering systems classification based on a simple model of the rendering pipeline as shown in Fig. 12.

To accelerate the rendering process both the geometry stage and the rasterization stage are usually implemented using multiple processors working in parallel. Polygons are sorted and assigned to these processors. According to where in the rendering pipeline this sorting is performed rendering systems are classified as *sort-first, sort-middle* or *sort-last.*

Sort-first architectures statically assign a pair containing one geometry engine (executing the geometry operations, see Fig. 2) and one rasterizer to a screen region [Fig. 13(a)]. We will refer to such a pair as a *rendering engine.* The application assigns polygons to each rendering engine randomly. Model and view transformations determine quickly which screen region a polygon covers. The polygon is then transferred to the rendering engine for that screen region and the geometry processing is completed. If a polygon covers several screen regions, the polygon is transferred to all affected rendering engines. Sort-first architectures tend to suffer from poor load balancing if polygons are not evenly distributed across the screen, thus resulting in higher load for some rendering engines.

In sort-middle architectures only the rasterizers are statically assigned to screen regions. Unlike sort-first architectures, geometry engines and rasterizers are not rigidly coupled into pairs [Fig. 13(b)]. Polygons are assigned to geometry engines in some arbitrary fashion that balances the load among the geometry engines, for example, random or round robin. After model and view transformation, polygons are transferred to the rasterizer(s) responsible for the screen region(s) covered by the polygons. Sort-middle architectures achieve good load balancing among the geometry engines but may also suffer from uneven utilization of the rasterizers if polygons are not distributed uniformly across the screen. Another potential problem with sort-middle architectures lies in the many-to-many communication between geometry engines and rasterizers. For large numbers of geometry engines and rasterizers building such interconnects becomes very expensive.

Sort-last architectures revert again to a static pairing of geometry engines and rasterizers [Fig. 13(c)]. However, each rendering engine covers the entire screen area. Each rendering engine computes a full-screen image containing all polygons it has been assigned. After all rendering engines have finished rendering, the partial images are merged in a compositing step to produce the final image. Sort-last architectures exhibit good load balancing, as the spatial distribution of polygons across the screen does not factor into the utilization of the rendering engines. However, the cost of sort-last architectures lies in the final compositing step that has to determine for every pixel the final color by selecting the frontmost object from all the partial images computed by the rendering engines. If the compositing step has to be performed at frame-rate, very stringent timing conditions have to be met in order to ensure synchronous operation with the screen refresh.

Parallelization. Parallel rendering architectures can also be distinguished by how the rendering pipeline is partitioned among the processing elements (9,13).

Given the model of the rendering pipeline, the most natural partitioning is know as functional decomposition or pipelining. Every processor performs one or several steps of the rendering pipeline. Most graphics systems employ this partitioning strategy at a high level, that is, by having one processor perform the geometric calculations while another performs the rasterization. This parallelization strategy is straightforward as it relies on the natural sequence of steps in the rendering process. Early graphics engines employed this parallelization strategy for geometry processing, for example, the Silicon Graphics GTX series (2,14). However, the disadvantages of pipelining are a low degree of parallelism and limited flexibility to achieve load balancing. The latter problem is a consequence of the rigidity of a pipeline as it is difficult for one processor to assume part of the workload of another processor. Most of today's graphics engines use parallel processors to accelerate the geometry operations, for example, the Silicon Graphics VGX processor.

Parallel architectures can be classified by what data are treated in parallel. Object parallel architectures distribute objects among the processors, that is, several objects are processed in parallel. This partitioning is frequently used for geometry processing. For instance, the Silicon Graphics VGX uses four parallel processors to transform and light the vertices in a triangle in one step. A few architectures have applied object-space partitioning to the rasterization stage.

Image-parallel architectures assign portions of the screen to the processors. In the extreme, each processor handles only one pixel, for example, the Pixel-planes 4 system (see the next section). More typically, a single processor handles rectangular regions comprising several pixels. Image-space partitioning is mostly used for parallel rasterizers.

Example Architectures

Pixel-Planes 4. Pixel-planes 4 (3) is a rasterization engine that uses a smart frame buffer (SFB) to compute all pixels inside a triangle in a fixed number of steps. The SFB consists of special memory modules that provide a simple processing element (PE) for every pixel. All PEs work in single-instruc-

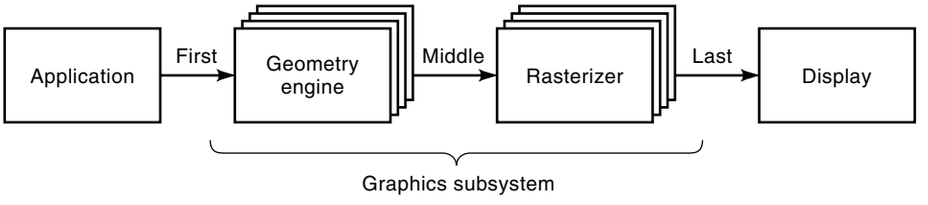

Figure 12. Rendering pipeline model used to classify parallel rendering architectures. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

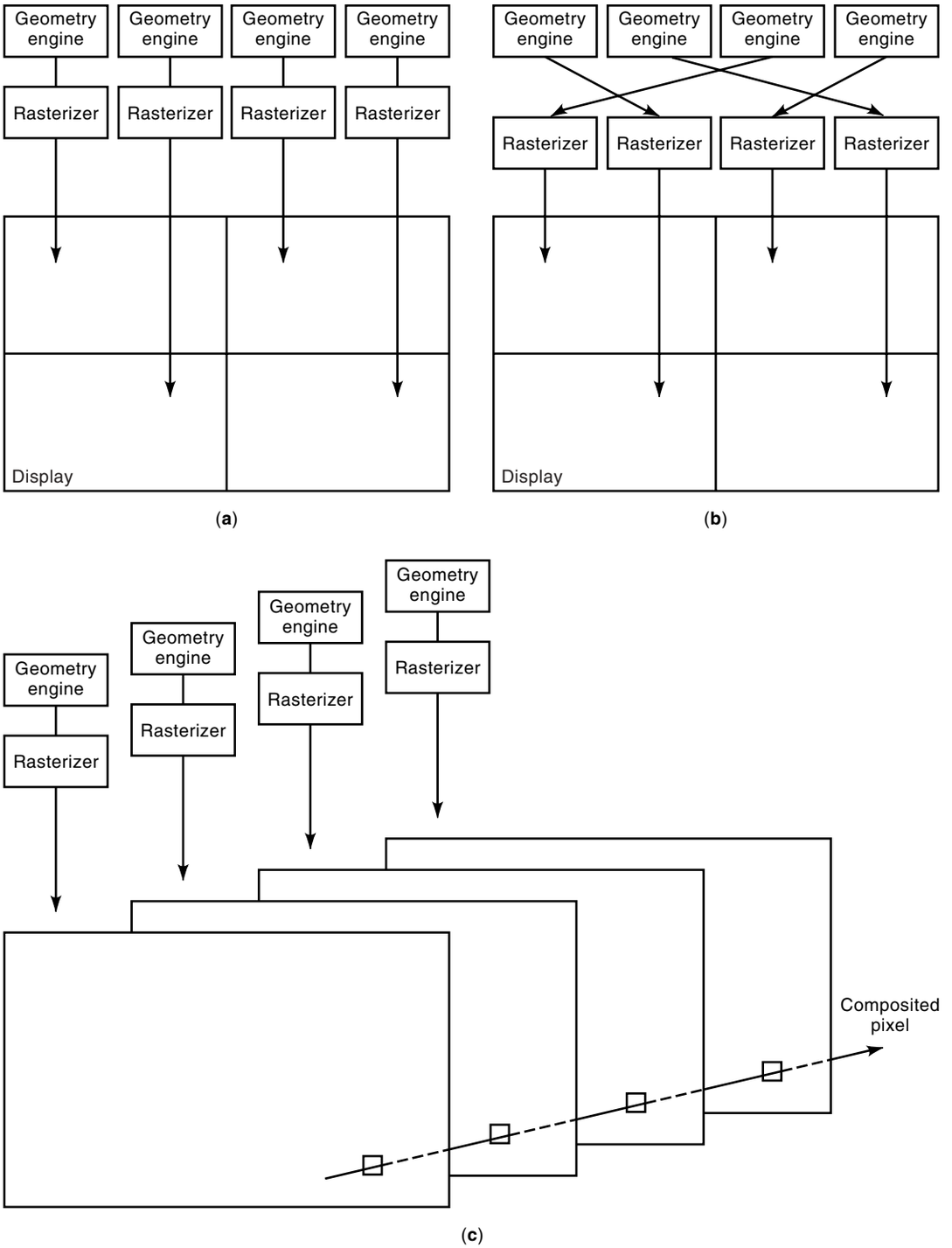

Figure 13. Raster graphics architectures for (a) sort-first, (b) sort-middle, and (c) sort-last.

tion multiple-data (SIMD) fashion, that is, they perform the same operations at every pixel.

To exploit the SFB, Pixel-planes 4 uses a different rasterization algorithm than the one described earlier. A triangle is not described by its vertices but by the lines passing through its edges (Fig. 14).

Equation (6) shows the linear edge equation that describes each line ℓ by a linear expression with the line parameters A, B, and C.

$$\ell : Ax + By + C = 0 \tag{6}$$

Evaluating this equation for points to the left of the line produces positive values, while points on the right side of the line produce negative values. A point is inside a triangle if all three edge equations return a positive value for this point.

Pixel-planes 4 computes z-values and Red–Green–Blue (RGB-values) by evaluating another linear expression. Equation (7) describes the plane supporting the triangle and is used to compute the z-values for pixels inside the triangle.

$$z = A_z x + B_z y + C_z \tag{7}$$

Pixel-planes 4 uses simple one-bit processing elements (bit-serial computation) to evaluate the edge equations for all three edges, the z-values, and the RGB-values. Figure 15 shows how multiplier trees are used to compute the partial sums Ax and $By + C$. The pixel processors add those partial sums to compute the final sum.

Pixel-planes 4 delivers a rasterization performance of 40,000 z-buffered and Gouraud-shaded triangles per second. As a consequence of its operating principles, this performance is independent of the actual size of the triangles as all pixels are computed in parallel.

The Pixel-planes 4 implementation uses 2048 SFB modules, each containing an 8×8 pixel array, to provide a 512×512 pixel frame buffer. The modules are addressed by external multiplier trees that precompute the base sums for each module.

Besides its size (and cost), one of the principal problems with Pixel-planes 4 is its low efficiency. Typical scenes contain many small triangles (less than 100 pixels). For such triangles, only very few pixel processors actually contribute to the final rendering of a triangle. Most processors are running idle. The successor to Pixel-planes 4 addresses these problems.

Pixel-Planes 5. Pixel-planes 5 (15) builds on the concepts developed in the Pixel-planes 4 architecture but is much more than just an extension of its predecessor. Pixel-planes 5 is a complete graphics subsystem, containing both geometry and raster processing capabilities. It was conceived as a flexible testbed environment to prototype parallel rendering algorithms. Figure 16 gives an overview of the Pixel-planes 5 architecture. The backbone of the system is a high-speed ring network that connects up to 32 graphics processors (GP) and up to 16 rasterization processors (RP). The GPs are built using general purpose microprocessors (Intel 860). They perform geometry processing operations and the setup calculations for the RPs. The central piece of the RPs is a 128×128 pixel-processor array that adds to the SFB modules of Pixel-planes 4 the capability to evaluate biquadratic expressions of the form shown in Equation (8).

$$z = Ax + By + C + Dx^2 + Exy + Fy^2 \tag{8}$$

This enables rasterization of quadratic patches instead of triangles, thus allowing rendering of complex scenes with less geometric primitives.

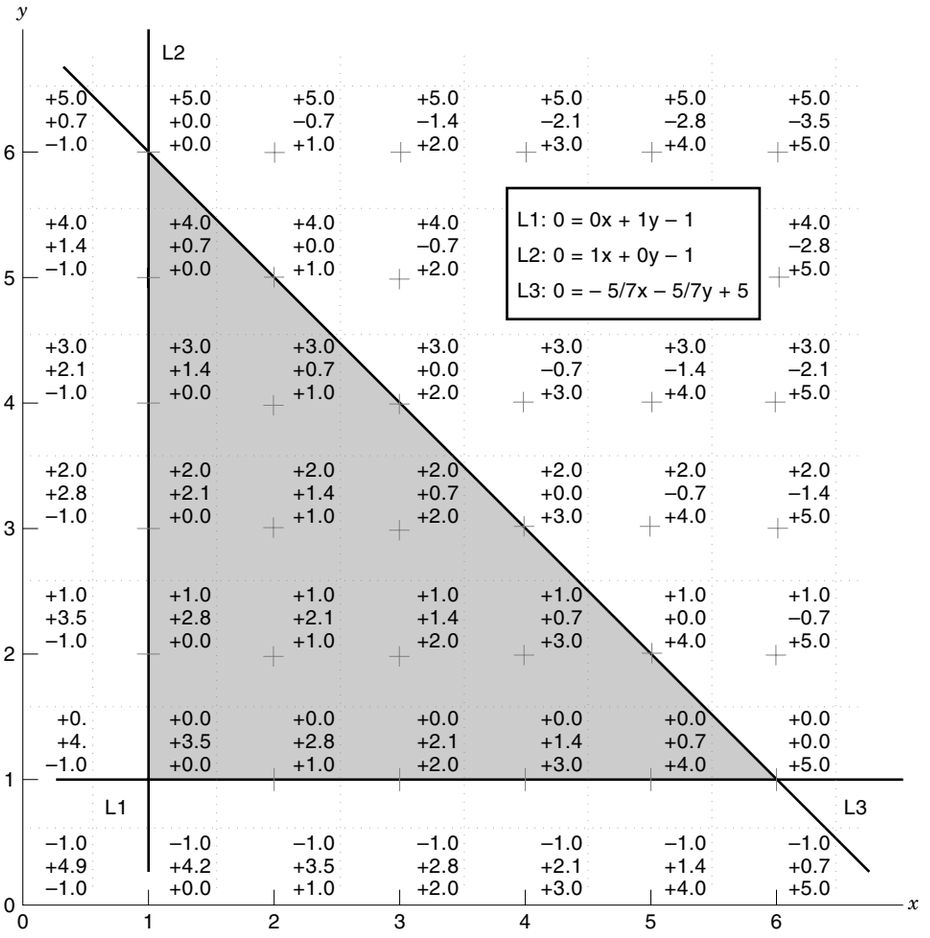

Figure 14. Triangle rasterization using edge equations. Each pixel contains the distance of the pixel center to each of the lines bounding the triangle. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

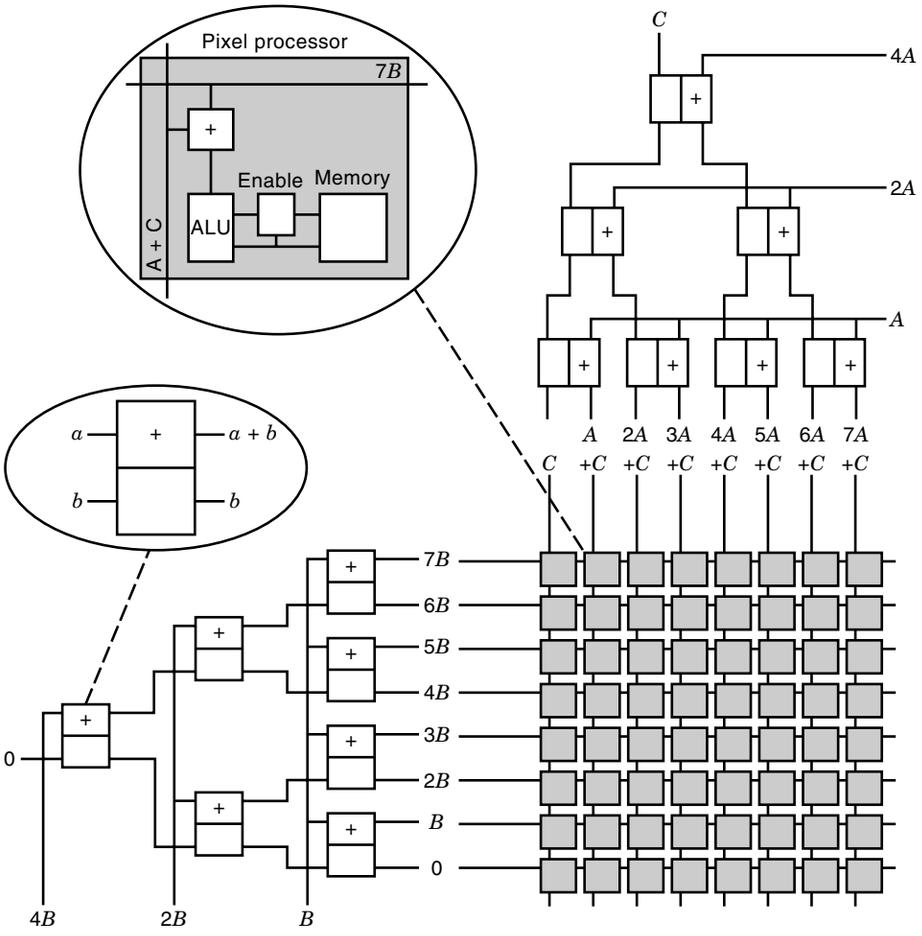

Figure 15. Basic architecture of Pixel-planes 4. (Reprinted from (9) with permission by Springer Verlag.)

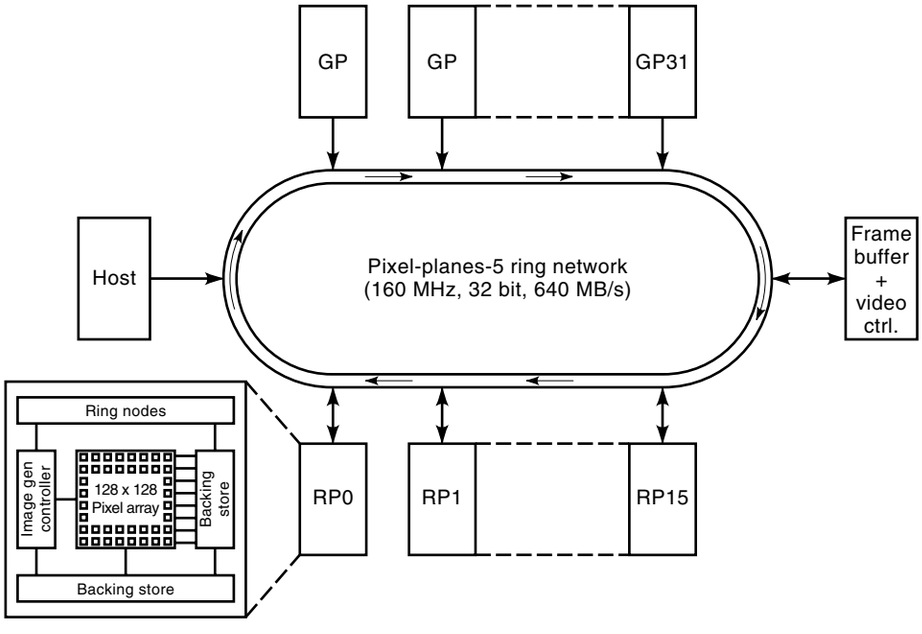

Figure 16. Basic architecture of Pixel-planes 5. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

Pixel-planes 5 implements a sort-middle architecture, that is, GPs operate on a part of the scene database and sort the transformed and clipped triangles into bins for different screen regions. For a display with 1280 × 1024 pixels, there are 80 such screen regions each covering 128 × 128 pixels. There are much fewer RPs than screen regions. An idle RP is assigned the next unprocessed screen region for rasterization, thus load-balancing between very busy screen regions and screen regions with few objects.

The unconventional architecture of Pixel-planes 5 has enabled programmers to implement interesting variations of the standard graphics pipeline. Deferred shading delays the lighting computations until after the hidden-surface removal. Now, lighting and shading are only performed once at every pixel, namely for the visible surface. In addition to the constant cost of the shading computations, the advantage of deferred shading is that it makes possible more complicated lighting models, for example, Phong shading or procedural shading. The obvious drawback of this technique is that all parameters required for lighting, for example, material and normal vectors, have to be stored at every pixel until after *z*-buffering.

Object-Parallel Pipelines. In object-parallel pipelines objects are assigned to one processor. Each object processor rasterizes its objects and injects the rasterized pixels into the stream of pixels traveling through the pipeline (Fig. 17).

Pixels proceed through the pipeline in scan order. The object processors accomplish hidden surface removal by determining for every pixel received at the input whether it is closer to the viewer than the triangle stored in the object processor. At the end of the pipeline the stream of pixels carries for every pixel the color and depth value of the visible object.

The principal advantage of object-processor pipelines is that the rendering performance is independent of the number of objects as long as there are enough processors to store all objects. At the same time, the maximum display resolution is limited by the speed with which the object processors can process pixels. Typically such systems are reported to be limited to display resolutions of 512 × 512 pixels.

Several research projects have studied pipelines of object-processors (16,17,18,19). We will give a short description of the PROOF system (18) as it combines many features found in the other systems. Figure 18 shows a block diagram of PROOF (pipeline for rendering in an object-oriented framework).

PROOF's graphics subsystem consists of three main stages. The first and principal stage is the object-processor pipeline (OPP) that performs rasterization, shading, and *z*-buffering to determine the visible objects for every pixel. The basic mode of operation of the OPP is Gouraud shading where the object processors compute object colors.

The optional shading stage adds Phong shading to the functionality. For Phong shading the OPP computes a normal vector instead of colors for every pixel. In order to reduce the bandwidth requirements for the OPP, the material properties are looked up in the look-up table 1 (LUT1) before data enter the shading stage.

To improve image quality, PROOF supports anti-aliasing by implementing the A-buffer algorithm. The A-buffer is an extension of the *z*-buffer algorithm that constructs for every pixel a depth-sorted list of potentially visible objects (20). For instance, if the edge of an object intersects a pixel, that object and the object behind it are entered into the list. The filter stage determines the final pixel color by computing an area-weighted average of the colors of the objects in the list.

The pixel color at the output of the filter stage is passed through a color look-up table (CLUT) to enable gamma correction.

Pixel-Flow. Pixel-flow (10,21,22) is similar to object-parallel pipelines in that it determines the final image by combining the partial images formed by subsets of the entire scene. It is a typical sort-last architecture. Figure 19 shows the overall architecture of a Pixel-flow system.

Besides the host interface and the frame buffer, the Pixel-flow system contains a number of renderer/shader boards. Depending on their configuration these boards are working either as rasterizers for polygons or as shading engines. Each renderer/shader board is a single-board graphics processor that is capable of transforming and rasterizing approximately

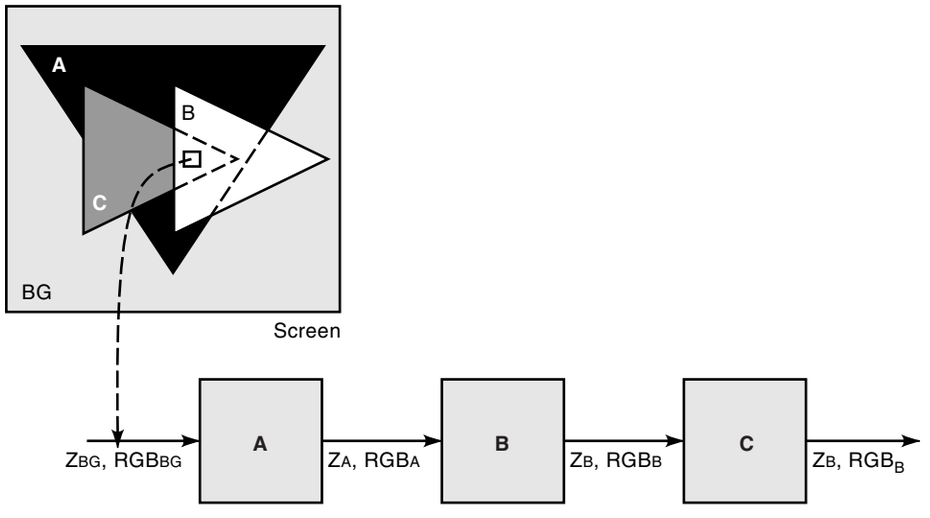

Figure 17. Basic architecture of object-processor pipelines. At the top, the position of the objects on the screen is shown. The assignment of objects to processors is shown at the bottom. At every stage in the pipeline it is indicated which object is visible at the highlighted pixel. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

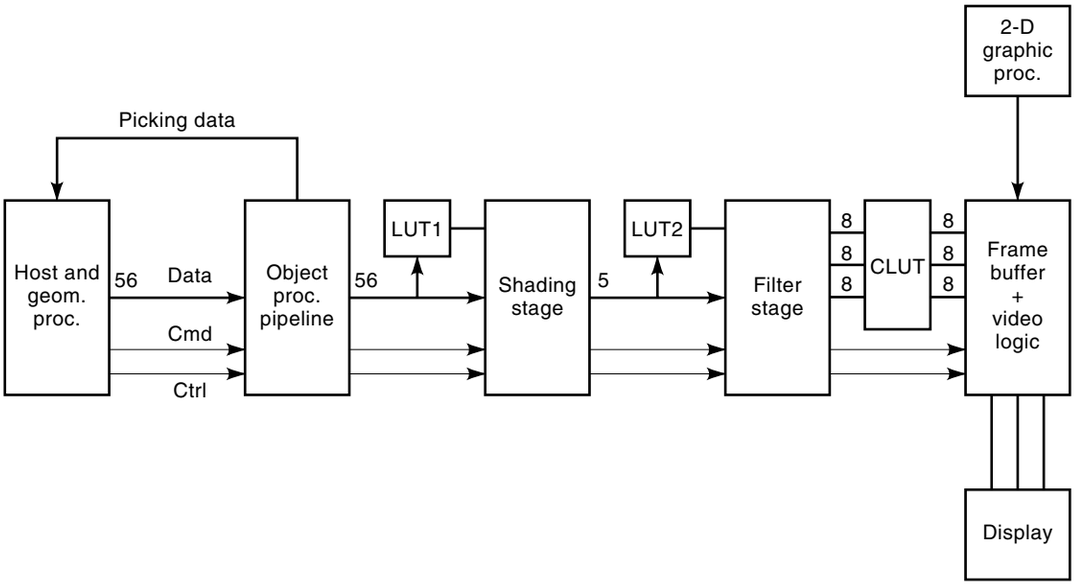

Figure 18. Basic architecture of the PROOF system. Connections drawn with heavy lines denote data buses, thin lines denote control and communication signals. (Reprinted from (4) by courtesy of Marcel Dekker Inc.)

70,000 triangles per second using a 160 × 128 pixel SIMD rasterizer similar to the one used in Pixel-planes 5. The renderer/shader boards are connected through a shared interconnect structure that provides two communication networks, the message-passing network and the image-composition network. The message-passing network allows every board in the system to communicate with any other board in the system. The message-passing network is used for system initialization and for sending changes to the geometry information or the view parameters. The image-composition network is a bidirectional high-speed data channel for moving pixels between adjacent boards.

The operating principle of Pixel-flow is to distribute over the message-passing network subsets of the scene database to each of the rendering boards. Each renderer generates a partial image based on the objects that it was assigned. The completed partial image is shipped over the image-composition network to the next renderer. A compositor stage in each renderer merges the image generated by the renderer with the image received over the image-composition channel. The merge operation determines for corresponding pixels the visible pixel by performing a comparison of depth values. The output of the last renderer is the final image of the entire scene. This image is then sent to the shading nodes. Similar to Pixel-planes 5 and PROOF, Pixel-flow employs deferred shading to determine the final pixel color.

Pixel-flow implements anti-aliasing by super-sampling the image, that is, by sampling every pixel at multiple, slightly different locations. The samples are combined into the final pixel color by the shading modules using a weighted-average filter.

Pixel-flow exhibits a number of advantageous characteristics. The system is very scalable to different performance levels by choosing the number of renderer/shader boards. Since each renderer/shader board is programmable, different rendering and shading algorithms can be realized by the system.

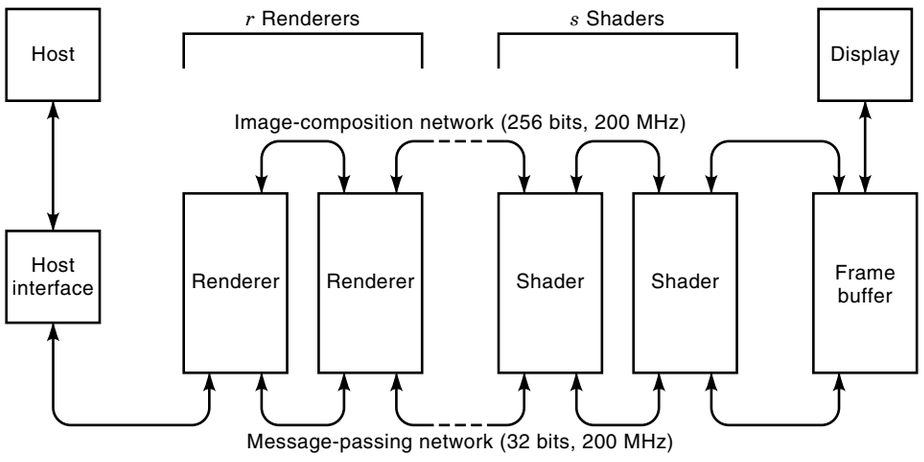

Figure 19. Basic architecture of Pixel-flow; revised from Molnar (10).

Silicon Graphics RealityEngine. The SGI RealityEngine (23) extends the concepts introduced with the VGX architecture. Figure 20 shows a block diagram of the RealityEngine graphics subsystem.

The geometry subsystem contains a command processor and, depending on the configuration, 6, 8 or 12 geometry engines (GE). The command processor interprets the incoming command stream, updates internal state and, if necessary, subdivides long triangle-strips. Each GE contains an Intel 860 processor, 2 MB of local memory and circuitry for data buffering and data format conversion. The GEs decompose incoming polygons into triangles and perform transformation and lighting calculations. The transformed, lit, and clipped triangles are then sent over the triangle bus to the raster subsystem.

The raster subsystem can be equipped with one, two, or four raster boards. Each raster board contains five fragment generators. The fragment generators perform rasterization, texture mapping, blending, and fog computations. The generated pixels, the so-called fragments, are then sent to the image engines (IE). Each raster board holds 80 IEs, 16 for each fragment generator. Each IE performs depth buffering and stores colors and z-values into the local frame buffer memory. The local frame buffer memory is built from 4 MB of DRAM that can be organized as 256, 512, or 1024 bpp, depending on the display resolution and the number of raster boards.

The RealityEngine architecture improves over its predecessor, the VGX architecture, by computing for each pixel an 8×8 subpixel coverage mask that indicates which subpixels are actually covered by a triangle. This subpixel information is used in the IE compute to perform anti-aliasing by blending the fragment color with the pixel color according to the coverage mask.

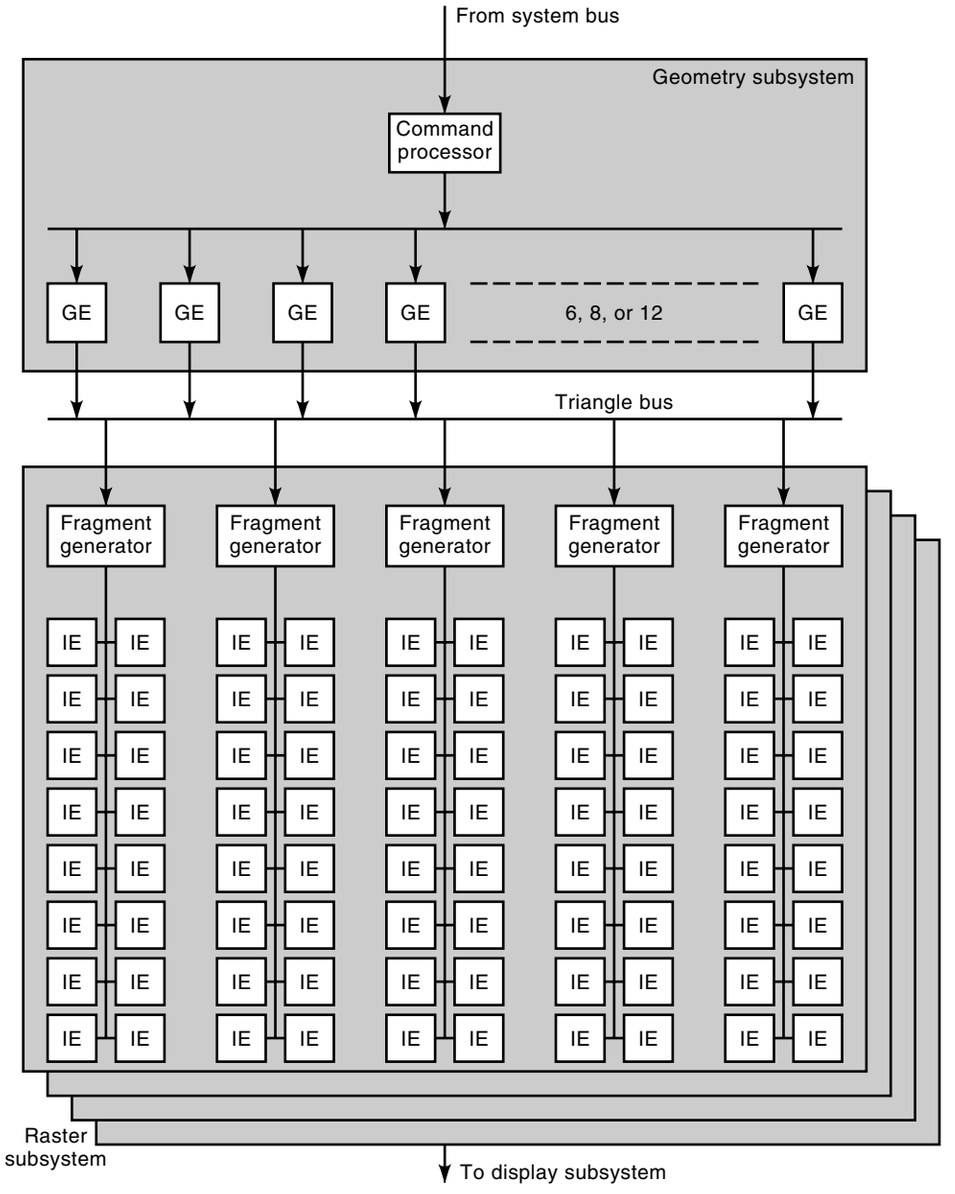

Figure 20. Basic architecture of the Silicon Graphics RealityEngine; adapted from Akeley (23).

At a high level the RealityEngine has abandoned the use of pipelining in favor of parallel geometry engines and fragment generators. The fragment generators themselves, however, are designed as deep pipelines. The RealityEngine is a typical sort-middle architecture. The sorting step occurs across the triangle bus that forms a potential bottleneck. The bandwidth of the triangle bus has been designed to accommodate the maximum workload generated by a fully configured system with 12 geometry engines.

SPECIAL PURPOSE RASTER GRAPHICS ARCHITECTURES

Constructive Solid Geometry

Even though most current raster graphics architectures operate according to the principles laid out in the previous sections, several research efforts have investigated raster graphics architectures for primitive types other than triangles. Constructive solid geometry (CSG) is a modeling technique frequently used in CAD. CSG describes objects as a sequence of Boolean set operations, for example, union, intersection, and difference, on simple point sets, for example, half-spaces (24). The ray-casting engine (RCE) (25) renders CSG objects by directly evaluating the CSG expression for a set of rays from the eye through the screen pixels. The RCE uses two types of processors to reflect the structure of a CSG object. The primitive classifier (PC) processors store the description of a half-space and compute the intersection of a viewing ray with that half-space. The classification combine (CC) processors perform set operations on ray segments received from its children, either PCs or CCs.

Volume Rendering

Several medical, geophysical, and meteorological applications generate volume data. Volume data are a set of samples defined over a 3-D grid. Each sample represents data for a small surrounding volume, usually called a *voxel.* Voxel data frequently are interpreted as volume densities. For rendering, viewing rays are sent into the volume, where they are attenuated according to the densities in the traversed voxels.

Volume-rendering architectures store the volume data set in a 3-D frame buffer with a resolution of 256^3 to 1024^3 voxels. Special purpose hardware generates the viewing rays, calculates for each ray the voxels along the ray, and combines the densities of those voxels into a single value that is displayed on the screen. Because of the huge amounts of data (16 MB to 1 GB) one of the critical design issues for volume-rendering architectures is to construct efficient memory architectures that allow parallel access to several voxels in order to speed up the rendering process. A more detail discussion of volume rendering can be found in Ref. 26.

BIBLIOGRAPHY

1. I. E. Sutherland, Sketchpad: a man–machine graphical communication system, in *Proc. Spring Joint Comput. Conf.,* Baltimore, MD: Spartan Books, 1963.
2. J. H. Clark, The geometry engine: a VLSI geometry system for graphics, *Comput. Graphics (Proc. Siggraph),* **16** (3): 349–355, 1982.
3. J. Poulton et al., Pixel-planes: building a VLSI-based graphic system, *Proc. Chapel Hill Conf. VLSI,* 1985, pp. 35–61.
4. B.-O. Schneider, Computer graphics hardware, in A. Kent, H. Lancour and W. Z. Nasri (eds.), *Encyclopedia of Library and Information Science,* New York: Marcel Dekker, in press.
5. J. D. Foley et al., *Computer Graphics, Principles and Practice,* 2nd ed., Reading: Addison-Wesley, 1990.
6. J. E. Bresenham, Algorithm for computer control of a digital plotter, *IBM Syst. J.,* **4** (1): 25–30, 1965.
7. P. S. Heckbert, Survey of texture mapping, *IEEE Comput. Graphics Appl.,* **6** (11): 56–67, 1986.
8. M. C. Whitton, Memory design for raster graphics displays, *IEEE Comput. Graphics Appl.,* **4** (3): 48–65, 1984.
9. B.-O. Schneider, Towards a taxonomy for display processors, in R. L. Grimsdale, W. Straßer (eds.), *Advances in Computer Graphics Hardware IV,* New York: Springer-Verlag, 1991, pp. 3–36.
10. S. Molnar, *Image composition architectures for real-time graphics,* Ph.D. dissertation, Dept. of Comput. Sci., Univ. North Carolina, Chapel Hill, 1991.
11. N. Gharachorloo, et al., A characterization of ten rasterization techniques, *Comput. Graphics (Proc. Siggraph),* **23** (3): 355–368, 1989.
12. T. Whitted, Architectures for 3D graphics display hardware, in Proceedings of the International Summer Institute, *State of the Art in Computer Graphics—Aspects of Visualization,* New York: Springer-Verlag, July 1992.
13. G. Abram and H. Fuchs, VLSI architectures for computer graphics, in G. Enderle (ed.), *Advances in Comput. Graphics I,* New York: Springer-Verlag, 1986.
14. K. Akeley and T. Jermoluk, High-performance polygon rendering, *Comput. Graphics (Proc. Siggraph),* **22** (4): 239–246, 1988.
15. H. Fuchs et al., Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories, *Comput. Graphics (Proc. Siggraph),* **23** (3): 79–88, 1989.
16. D. Cohen, A VLSI approach to the CIG problem, *Siggraph Conf.,* 1980.
17. R. Weinberg, Parallel processing image synthesis and antialiasing, *Comput. Graphics (Proc. Siggraph),* **15** (3): 55–62, 1981.
18. B.-O. Schneider, A processor for an object-oriented rendering system, *Comput. Graphics Forum,* **7**: 1988, pp. 301–310.
19. M. Deering et al., The triangle processor and normal vector shader: a VLSI system for high performance graphics, *Comput. Graphics (Proc. Siggraph),* **22** (4): 21–30, 1988.
20. L. Carpenter, The A-buffer, an antialiased hidden surface method, *Comput. Graphics (Proc. Siggraph),* **18** (3): 103–108, 1984.
21. S. Molnar et al., PixelFlow: high-speed rendering using image composition, *Comput. Graphics (Proc. Siggraph),* **26** (2): 231–240, 1992.
22. John Eyles et al., PixelFlow: the realization, *Proc. 1997 Siggraph / Eurographics Workshop Graphics Hardware,* Los Angeles, ACM, 1997, pp. 57–68.
23. K. Akeley, RealityEngine graphics, *Proc. Siggraph 93,* Anaheim, CA, 1993, *Computer Graphics Proc., Annual Conf. Ser.,* 1993, ACM Siggraph, New York, 1993, pp. 109–116.
24. A. A. G. Requicha, Mathematical models of rigid solids: theory, methods, and systems, *ACM Comput. Surveys,* **12** (4): 437–464, 1980.
25. G. Kedem and J. L. Ellis, The Raycasting Engine, *Proc.1984 Int. Conf. Comput. Des. (ICCCD),* 1984, pp. 533–538.
26. J. Hesser et al., Three architectures for volume rendering, *Computer Graphics Forum (Proc. Eurographics),* **14** (3): pp. 111–122, 1995.

Reading List

J. Foley et al., *Computer Graphics: Principles and Practice,* Reading, MA: Addison-Wesley.

This book is the standard textbook on computer graphics. It describes many of the issues discussed in this article.

Conrac Corporation, *Raster Graphics Handbook,* New York: Van Nostrand Reinhold, 1985.

Although somewhat dated, this book summarizes in an excellent fashion many practical aspect of the design of a raster graphics system.

IEEE Computer Graphics & Applications

ACM Computer Graphics

Computer Graphics Forum

These journals publish articles on all aspects of computer graphics, including hardware, algorithms, and graphics systems. In particular, the proceedings of the annual Siggraph conference are published as an issue of ACM Computer Graphics. Siggraph is the leading conference on computer graphics. The proceedings of the annual European computer graphics conference, Eurographics, appear as an issue of Computer Graphics Forum.

Eurographics Workshop on Graphics Hardware

This annual workshop is a forum for the latest developments in graphics hardware and presents work about all aspects of graphics architectures. Compared to Siggraph and Eurographics, this workshop tends to provide a platform for early results and reports of work in progress.

H. K. Reghbati and A. Y. C. Lee, *Tutorial: computer graphics hardware, Image generation and display,* Los Alamitos, CA: IEEE Comput. Soc. Press, 1988.

This tutorial is a collection of many of the early papers on the subject of graphics hardware. It contains many of the seminal papers describing pioneering work in graphics architectures.

BENGT-OLAF SCHNEIDER
IBM Thomas J. Watson Research Center

RATE-ADAPTIVE PACEMAKERS. See PACEMAKERS.

RATE DISTORTION THEORY. See DATA COMPRESSION, LOSSY.

RATE WINDOW. See DEEP LEVEL TRANSIENT SPECTROSCOPY.