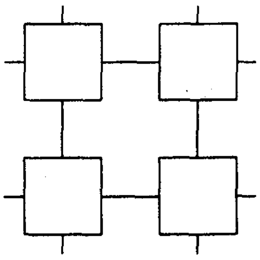
M. R. Civanlar et al 1-1-1

The Pixel Machine System Architecture

A Technical Report



AT&T

Pixel Machines
Crawfords Corner Road
Holmdel, New Jersey 07733

Introduction

Generating a realistic image from complex two and three dimensional data in real time demands a lot of computational power. Graphics and image processing algorithms, particularly rendering algorithms, often perform a set of operations to generate each pixel, with little or no interaction between pixels. These algorithms are candidates for mapping to a parallel architecture, with performance increasing nearly linearly with the number of processors.

In many display systems, a single custom processor handles the typical frame buffer operations. This approach is adequate for rendering simple two-dimensional images. However, when realistically shaded images must be displayed in real time, a single processor cannot provide the necessary computational power.

Pipelines or arrays of special purpose processors provide high performance at the expense of flexibility. Their performance improvements are limited to the narrow range of algorithms that they were designed to implement.

While the use of parallelism and pipelining gives a system the power needed to render high quality images in real time, the use of programmable processors provides the flexibility to attain high performance for a wide range of graphics and image processing algorithms. It is much easier to change a program from Gouraud shading to Phong shading, for example, than to redesign a customized processor.

The AT&T Pixel Machine combines the strengths of both coarse grain pipelining and multiple instruction/multiple datapath (MIMD) computing arrays. A pipeline of computing elements processes the serial tasks that precede pixel-level processing while a processor array provides high-bandwidth access to an integrated frame buffer and computes individual pixel values. The processors in both the pipeline and the array are programmable, with hardware floating point operations.

The programmability of the processors allows all algorithms to be implemented in software. A set of mapping functions translates frame buffer algorithms written for conventional serial computers to algorithms that execute in the pixel nodes and access the distributed frame buffer. The ability to use floating point computations in frame buffer operations such as antialiasing, ray tracing, and cascaded filtering, allows high quality image generation.

The Pixel Machine provides up to 820 megaflops of processing power and 48 megabytes of memory for data visualization applications, including three-dimensional object rendering and animation, image processing, and volume rendering.

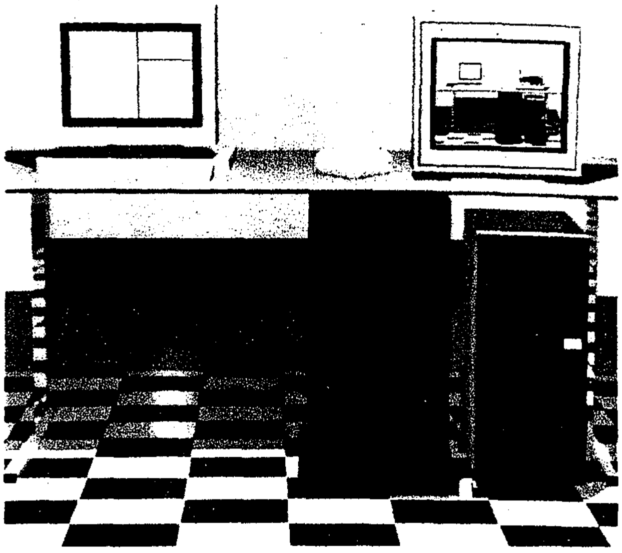


Figure 1. The Pixel Machine: A recursive self-portrait.

Pixel Machine Architecture

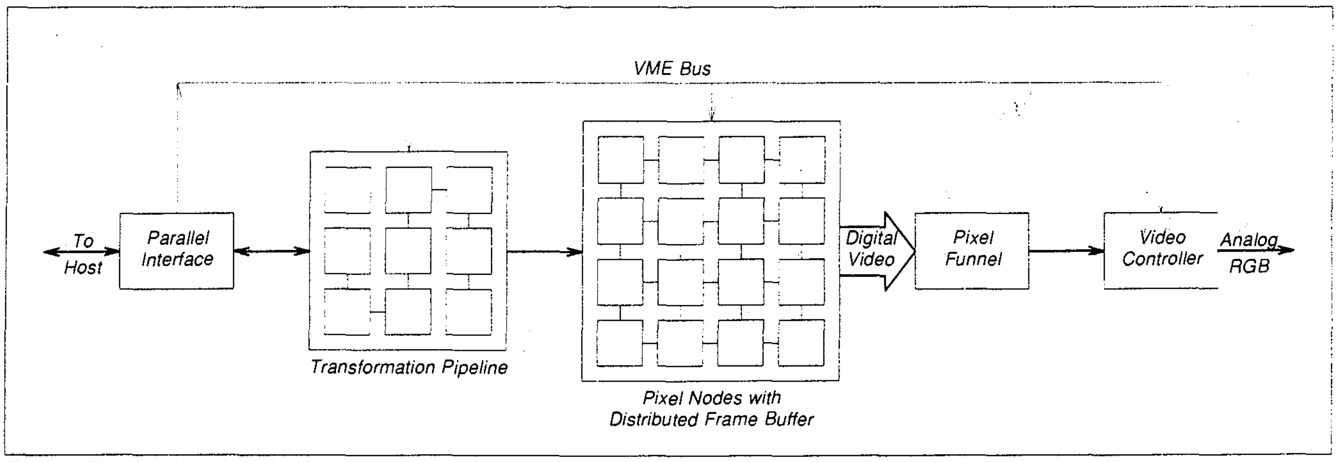


Figure 2. Pixel Machine block diagram.

The Pixel Machine combines the strengths of both coarse grain pipelining and MIMD computing arrays to provide the performance of a supercomputer for image synthesis and image analysis applications. Synthesis applications include the generation and display of two and three dimensional scenes as well as the visualization of scientific and engineering computations. Analysis applications include the processing and interpretation of image data from, say, a nuclear magnetic resonance machine or a satellite. The design philosophy is:

- Use floating-point computation and large image memories, which are useful for image processing.
- Design simple, modular processors that can be repeated a number of times to build a system.
- Implement all algorithms in software.

The modular approach enabled the Pixel Machine to be designed, built, and programmed in a short time, thereby reducing the duration of the design to product cycle. The decision to implement algorithms in software rather than special purpose VLSI chips gives wide functionality, faster implementation of new algorithms, and easier modification of existing ones.

The architecture has the following features:

- 32-bit programmable processors
- 9 or 18 pipe nodes, configurable as one or two pipelines
- 16, 20, 32, 40, or 64 pixel nodes
- 32-bit pixel and z-buffer data
- floating-point computation for pixel generation
- a frame buffer with pixel-interleaved parallel architecture
- 1280x1024 or 1024x1024 high-resolution 60 Hz non-interlaced display
- NTSC and PAL display modes
- a large image memory that allows single, double, or quadruple buffering
- uniform, upwardly compatible software that provides additional functionality as the number of pixel nodes increases

Both the pipe nodes and the pixel nodes include a WE® DSP32 Digital Signal Processor, a 32-bit, high speed, programmable device [K85]. Its features include:

- 20 MHz, 5 MIPS, 10 MFLOPS
- 4K bytes of on-chip memory
- 32-bit floating point arithmetic
- four 40-bit floating point accumulators
- twenty-one 16-bit integer and address registers
- an interface to off-chip expansion memory

- parallel and serial I/O ports with DMA

The DSP32 can be programmed in assembly language or in C [K78]. The software development environment includes a compiler, an assembler, a linking loader, and a simulator. All arithmetic operations on data are floating point operations. Only memory address generation and program control calculations use integer arithmetic. Software is developed on a host computer, typically a Sun Workstation®. The Pixel Machine is connected to the host computer via the VMEbus®.

Inside the Pixel Machine, there are 9 or 18 pipe nodes configured as one or two pipelines, a broadcast bus that transfers data from the end of the pipes to the pixel nodes, an array of 16, 20, 32, 40, or 64 pixel nodes that form a distributed frame buffer, and a *pixel funnel* that transfers digital video data from the frame buffer to the video processor, which controls the display monitor.

Each pipe and pixel node can be viewed as a small independent computer that executes its instructions and operates on data asynchronously with all the other nodes. Programs are loaded into the nodes by the host, using unique, software-defined node numbers to distinguish between them.

Pipe Nodes

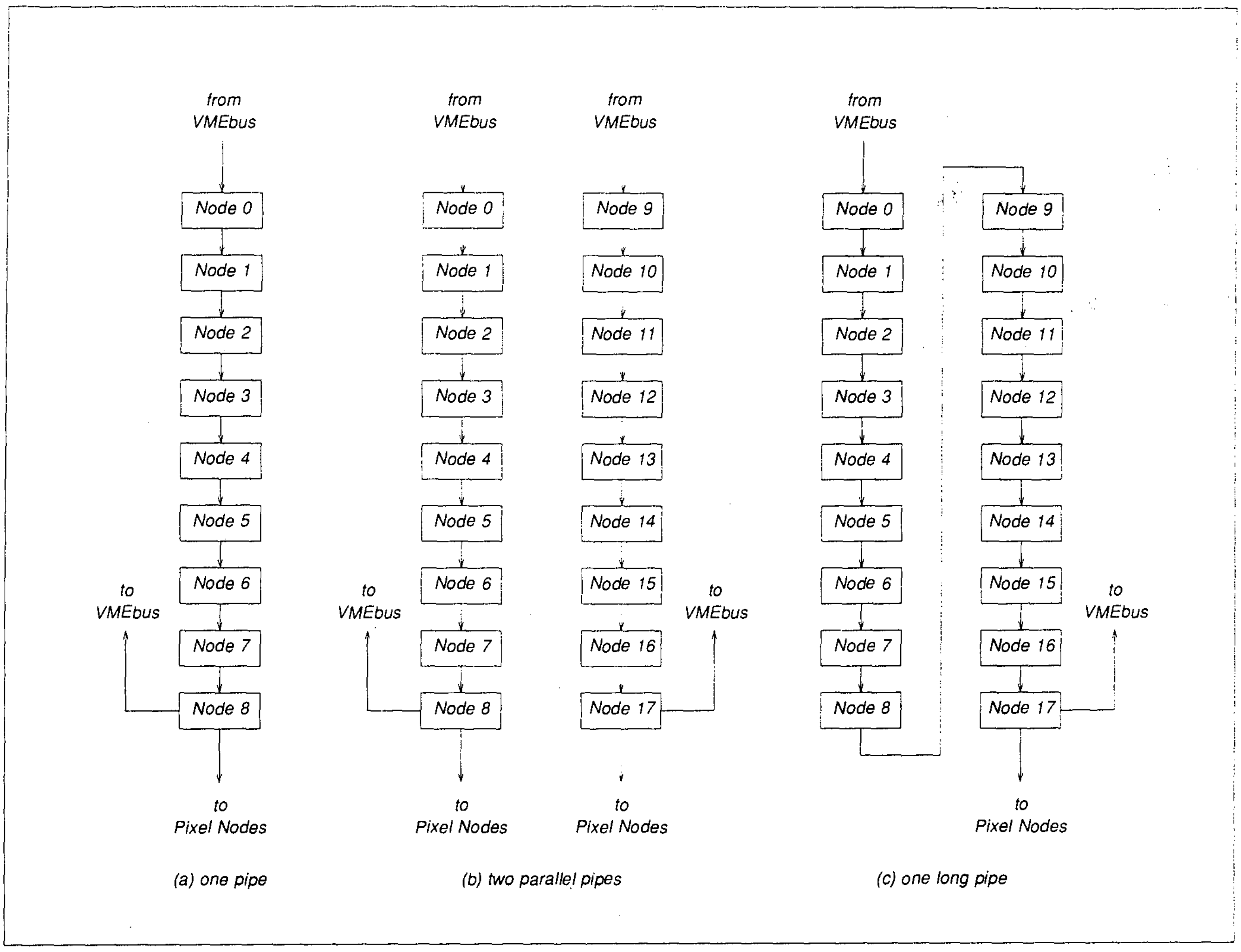


Figure 3. Pipeline configurations.

Figure 4 shows a block diagram for a pipe node. Each pipe node has a DSP32 processor that executes five million instructions or ten million floating point operations per second. The parallel DMA interface of each processor is connected to the VMEbus. The pipe nodes have 9K×32 bits of memory for instructions and data, a 512×32 bit input FIFO containing data written by the previous pipe node, a 512×32 bit output FIFO where all output is written, to be read by the next node in the pipeline.

The host computer provides input to the first pipe node via the VMEbus. The output from the last pipe node is broadcast to all of the pixel nodes. In addition, the last pipe node has a second output FIFO that is read by the host, again via the VMEbus.

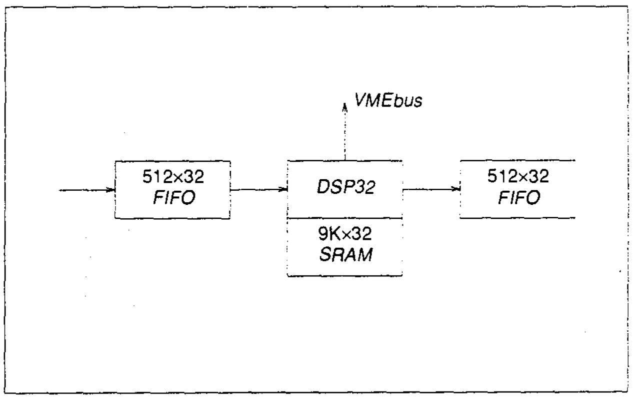


Figure 4. Pipe node block diagram.

A system can have 9 or 18 pipe nodes. The 18-node systems are software-configurable as either two nine-node parallel pipelines or one 18-node pipeline (see Figure 3). In a two pipeline system, the node in each pipeline has the ability to request, acquire, and release the broadcast bus. In the one pipeline system, the last node has continuous access to the broadcast bus.

The pipe nodes perform those parts of the algorithms which are serial in nature and can be pipelined. These include 3D transformations, clipping, projections, shading, and image filtering. The pipeline can also be used as a hardware subroutine by processes running in the host computer, which can send data to the first node and read results from the last one.

Pixel Nodes

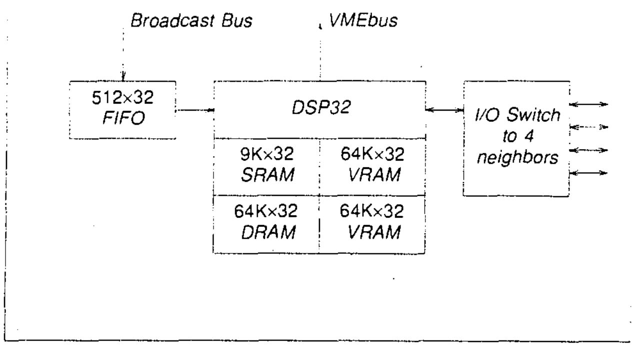


Figure 5. Pixel node block diagram.

The pixel nodes form an *n×m* array with a distributed frame buffer. They receive their data from the broadcast bus of the pipe nodes and store their output into the frame buffer or return it to the host computer. Mapping registers provide uniform access to the frame buffer across different configurations of pixel nodes, and a four-way multiplexed I/O switch and channel allows two-way communication with the four neighboring pixel nodes.

A DSP32 is the computing element in each pixel node, and just as in the pipe nodes, there is a 9K×32 bit static RAM in the node, in addition to the 1024×32 bits of on-chip storage. Figure 5 shows a block diagram.

Two banks of 64K×32 bit VRAMs form the pixel node's piece of the distributed frame buffer. The video RAMs store the red, green, blue, and alpha settings for the pixels. These memories can be displayed, or used as off-screen storage for images.

Pixel nodes also contain a 64K×32 bit dynamic RAM that can be used to hold floating-point z-buffer values, pixels in floating-point representation, display list segments, or code segments.

A Pixel Machine can be configured with 16, 20, 32, 40, or 64 pixel nodes. Table 1 summarizes these five configurations. The two video RAMs and the dynamic RAM (when used to store pixel data) are organized as three blocks of 256×256 32-bit pixels. Each bank can be logically divided further into smaller blocks, called *subscreens*.

Pixel	*Pixel Node Array*		*Display*	*Subscreens*		
Nodes	*physical*	*virtual*	*Resolution*	*Size*	*# pixels*	*per node*
16	4×4	8×8	1024×1024	128×128	*16384*	*4*
20	5×4	10×8	1280×1024	128×128	*16384*	*4*
32	8×4	8×8	1024×1024	128×128	*16384*	*2*
40	10×4	10×8	1280×1024	128×128	*16384*	*2*
64	8×8	8×8	1024×1024	128×128	*16384*	*1*
64	8×8	8×8	1280×1024	160×128	*20480*	*1*

Table 1. Pixel array configurations.

In order to allow configuration-independent software, the concept of *virtual* pixel nodes that reside inside physical nodes is introduced. Each virtual node accesses a single subscreen. All systems have either 64 or 80 virtual nodes, depending on the resolution of the display screen. In a 64-node system, each physical pixel node contains a single virtual node, while a 16- or 20-node system has four virtual nodes per physical node.

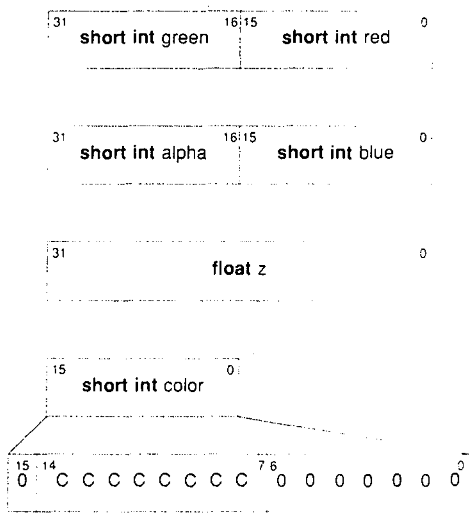


Figure 6. Pixel format.

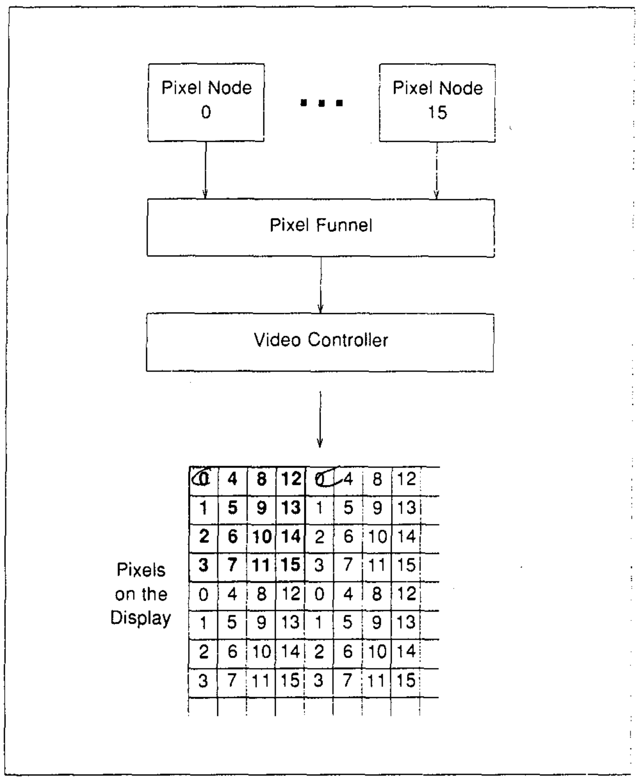
Pixel data is stored in the frame buffer as 16-bit signed integers. The four components of a pixel, (red, green, blue, α) form two 32-bit words, as shown in Figure 6. In the current implementation, only eight of the 16 bits in a pixel component are populated with memory. In future implementations, additional memories may extend the number of usable bits per color to 12 or 16 bits without requiring any modification to existing software.

Each pixel node has a serial input/output (SIO) channel that provides a communication path to its four nearest neighbors, allowing the Pixel Machine to function as a computing mesh. Node placement follows pixel interleaving conventions, as shown in Figure 7. Thus, in a 4×4 array of pixel nodes, node 5's neighbors are nodes 1, 4, 6, and 9. The edges of the mesh wrap around to form a torus, so node 0's neighbors are 1, 3, 4, and 12, for example.

The SIO compatibility at each node consists of one input and one output serial port that operate at peak rates of 12.5 MHz. Pixel data can be moved from node to node at a sustained rate of 5.25 Mbits per second, including the time spent buffering pixel data to and from the display memory. In practice, however, processor cycles will be shared between an application program and SIO, and the data transfer rate will be proportionately slower.

Video Display

The frame buffer is distributed throughout the array of pixel nodes. An example is shown in Figure 7. The *pixel funnel* rearranges pixels from the frame buffer into a properly ordered raster scan sequence. Both the video processor and the pixel funnel are software configurable for the five different pixel arrays.



0	**4**	**8**	**12**	0	4	8	12
1	**5**	**9**	**13**	1	5	9	13
2	**6**	**10**	**14**	2	6	10	14
3	**7**	**11**	**15**	3	7	11	15
0	4	8	12	0	4	8	12
1	5	9	13	1	5	9	13
2	6	10	14	2	6	10	14
3	7	11	15	3	7	11	15

Figure 7. Pixel mapping in the distributed frame buffer.

The frame buffer stores (red, green, blue, α) values for each pixel. Based on the value of α and the display mode setting, the pixel node may substitute the α value for the red, green, and blue values:

- $RGB_{out} = rgb_{in}$.

- $RGB_{out} = \begin{cases} rgb_{in} & \text{if } \alpha=0 \\ \overline{rgb_{in}} & \text{if } \alpha=255 \\ \alpha\alpha\alpha & \text{if } 0<\alpha<255 \end{cases}$

- $RGB_{out} = \begin{cases} rgb_{in} & \text{if } 0\leq\alpha<128 \\ \alpha\alpha\alpha & \text{if } 128<\alpha\leq255 \end{cases}$

The video processor uses six 256×10 lookup tables, or color maps, to translate 8-bit pixel color values to 10-bit video data. Three of the tables map red, green, and blue. The other three map α-type pixels to red, green, and blue values.

There are two sets of color maps. One set contains high-speed *video* tables that are used to convert video data. The other set are *shadow* tables which can be read and written via the VMEbus. The contents of the shadow tables are automatically copied to the video tables during a vertical retrace period, with copying enabled and disabled in software. The shadow tables prevent two problems common to many video systems from arising: snowy and sheared video because color maps are modified during active video periods, and distracting flashes on the screen because of partially modified color maps.

In high-resolution mode, the video system displays 1024 lines of either 1024 (in systems with 16, 32 or 64 pixel nodes) or 1280 (in systems with 20, 40, or 64 nodes) pixels, at 60 Hz non-interlaced. In NTSC mode, the video system uses the RS-170A timing format to display 485 lines of 720 pixels in all pixel node configurations. In PAL mode, 575 lines of 702 pixels are displayed.

The video timing generated by the system can be synchronized to and mixed with external video sources. The video signal can also be mixed with the Sun Workstation's color video signal and displayed on its color monitor.

System Configurations

As described above, the Pixel Machine can be configured with five different pixel array sizes and two different pipeline sizes. The ten models are described in Table 2. Models **964** and **964D** can be programmed to display either 1024x1024 or 1280x1024 pixels.

In high-resolution display mode, Models **916** and **920** have two *rgbα* frame buffers and one z-buffer, enough memory to render full-screen 24-bit images in double-buffered mode with a floating point depth buffer. Models **932** and **940** can be quadruple-buffered, and Model **964** has enough memory to store a 32-bit 2048x2048 image in the extended frame buffer while maintaining double buffers in the main video RAM.

In NTSC display mode, each pixel node has a single subscreen, regardless of configuration. The subscreen size in Model **916** is 180x122=21960 pixels, while in the **964**, it is one fourth as big. This means that a **916** can render full-screen NTSC images about as fast as a **964** can in high-resolution mode.

Model	*Nodes*		*Peak Performance*		*Memory*	*Buffers*		*Bytes*
Number	*pipe*	*pixel*	*MIPS*	*MFLOPS*	*(Mbytes)*	*rgbα*	*z*	*per pixel*
916	9	16	125	250	12	2	1	12
916D	18	16	170	340	12	2	1	12
920	9	20	145	290	15	2	1	12
920D	18	20	190	380	15	2	1	12
932	9	32	205	410	24	4	2	24
932D	18	32	250	500	24	4	2	24
940	9	40	245	490	30	4	2	24
940D	18	40	290	580	30	4	2	24
964	9	64	365	730	48	4/8	2/4	24/48
964D	18	64	410	820	48	4/8	2/4	24/48

Table 2. Pixel Machine configurations.

Physically, the Pixel Machine is a free-standing cabinet (see Figure 1) with a triple-height 21-slot VME chassis. There are four types of VME cards in the chassis:

- *pipeline* cards, each containing nine pipe nodes,
- *pixel array* cards, each containing four pixel nodes,
- *video processor* cards, and
- *parallel interface* cards.

The broadcast bus and the pixel funnel are part of the VME backplane.

The host computer accesses the Pixel Machine as a 64K byte block of memory on the VMEbus. The memory is mapped into user process space. The parallel I/O interface for each of the pipe and pixel nodes, the input and output FIFOs in the pipline, the shadow colormaps, and the video control registers are mapped directly into this memory block. The contents of all DSP memory maps, including the *rgbα* and z-buffer memories, are accessible to the host via DMA transfers, even while the processors are running.

Pixel Machine Software

The distinct architectural components of the Pixel Machine are the host computer, the pipe nodes, and the pixel nodes. The host computer allows an application program to access the power and functionality of the Pixel Machine, the pipe nodes are responsible for the serial parts of algorithms, and the pixel nodes execute parallel algorithms. The following paragraphs describe the software that supports each architectural component.

Host Software

A host-resident C-callable library is responsible for message creation and transmission, invocation of subprocesses that monitor external events, and machine initialization. The primary functions provided by the host are

- translating high-level function calls and macros into messages
- transmitting messages through a high-bandwidth channel to the Pixel Machine
- down-loading code to the pipe and pixel nodes and initializing them
- handling interactive functions like the mouse/cursor interface and feedback from the pipeline

All messages are sent to the first node in the pipeline. It processes any messages addressed to it and sends the rest to the next pipe node. Messages proceed serially down the pipe until the last node broadcasts them simultaneously to all of the pixel nodes.

Pipe Node Software

The pipe nodes are typically used to implement a set of algorithms that act serially on a set of data. For example, a rendering and modeling application might use the pipeline to generate objects, apply modeling and viewing transformation, cull and shade the objects, apply projection transformations, do *x*, *y*, and *z* clipping, and finally, map the image to a viewport on the screen.

A useful analogy is to think of the pipe nodes as UNIX® System filters [P87b]. Each node, like a filter, reads some input, transforms the input, and writes some output. The mechanism for controlling these actions is a command language interpreter that resides in every node.

The command language interpreter reads messages from the input FIFO. Each message consists of an opcode, the number of parameters, and the parameter list. When a message arrives at a pipe node, three actions can be triggered. The node can

- forward the message to the next node in the pipeline,
- modify the parameter list and send it down the pipe, or
- process the message, possibly generating new messages.

Each pipe node stores and executes routines that are invoked by message opcodes. In a typical polygonal rendering and modeling application, the pipe nodes perform geometric processing algorithms. For instance, three nodes could be assigned to clip the polygons in the *x*, *y*, and *z* planes, and one node to shade the polygon. In order to optimize the shading, however, the system could easily be re-configured to have three shading nodes and only one clipping node. Thus the pipeline can be optimized for any application through experimentation, and new functions can be added as needed. The customizing is done using DEVtools™, a collection of software that allows a user to develop applications for the Pixel Machine. DEVtools is described in more detail in a later section.

A Pixel Machine can contain a single 9-node pipe, or 18 pipe nodes that can be configured as either two parallel 9-node pipes or a single 18-node pipeline. In parallel mode, shown in Figure 3(b), the functions performed by a single pipeline are duplicated in both pipes. Distinct geometric primitives are explicitly assigned to each pipeline and processed simultaneously, ideally doubling the processing bandwidth of a single pipeline. The user can control which geometric primitives are routed to each pipeline. For example, one pipeline can be made to process bicubic patches while the other one does quadric surfaces. The processing of the two primitives occurs in parallel, with the last node of each pipeline arbitrating the use of the bus that broadcasts messages to the pixel nodes.

In serial mode, Figure 3(c), the geometric functions are distributed over 18 nodes instead of nine, spreading out the work load so that each node has less to do. This reduces the number of computational bottlenecks, and thus can reduce the overall processing time. Vectorization of the coarse-grain processes can alleviate bottlenecks. For example, a pipeline configuration might include ten nodes dedicated to shading, with the *i*-th node handling every tenth polygon.

Pixel Node Software

Pixel nodes provide functions and implement algorithms that can be done in parallel, like the raster-scan conversion of points, lines, and polygons, image compositing, and ray tracing. Because the frame buffer memory is distributed through the pixel node array [Figure 6], all routines that access the frame buffer are implemented here as well.

In the pixel node array, identical functions are usually replicated in each node. These include

- antialiased and depth-cued vector generation,
- flat, Gouraud-shaded, and texture-mapped polygon rendering,
- raster operations,
- buffer-to-buffer copy algorithms,
- antialiasing by supersampling, and
- image display.

Because the machine is completely programmable, the pixel nodes can contain a rich set of graphics functions that can be modified to include new algorithms or to improve existing ones.

User Level Software

There are three collections of software available with the Pixel Machine. All three are host-resident, but give different kinds of access to the Pixel Machine.

- **PIClib**™ is a complete 3D graphics package,
- **RAYlib**™ is a very fast ray-tracing package,
- **DEVtools**™ is a collection of tools for developing applications that run in the pipe and pixel nodes.

These three software packages are described in more detail in the sections that follow, including examples and timing information.

PIClib

PIClib is a library of interactive, high-resolution graphics routines for the Pixel Machine that are callable from application programs running on the host. PIClib uses the pipeline for object and normal vector generation and transformation, backfacing surface removal, shading, clipping, projection, and triangularization. The pixel nodes contain program modules that rasterize points, lines and polygons, do z-buffer calculations on polygons and spheres, draw cursors, perform raster operations, and display and read back images.

PIClib contains approximately 200 subroutines that provide the capability to

- draw points, lines, and polygons specified with either integer or floating point coordinates in two- and three-dimensional object spaces,
- draw arcs, circles, quadrics and super quadrics, cubic curves and bicubic patches,
- manipulate the *transformation* matrix stack: push, pop, and load it, pre- and post-multiply it by a given matrix, specify translation, rotation and scaling parameters, and read back the matrix, its inverse or a normal vector matrix,
- manipulate the *projection* matrix stack: push, pop, and load it, pre- and post-multiply it by a given matrix, specify perspective or orthographic projection parameters and viewports, and read back the matrix or its inverse,
- do lighting, shading and depth-cueing, hidden surface removal, and antialiasing
- manipulate color maps,
- control the frame buffer,
- manipulate the cursor, mouse, and other input devices,
- initialize the hardware and software, synchronize with vertical retrace, swap buffers and pipelines, and other control functions,
- load fonts and draw character strings,
- do operations that require feedback, such as picking or selecting objects.

In addition to points, lines, and polygons, the raster primitives include **atoms** and **voxel templates**.

Atoms are Phong-shaded spheres [F85] and have a directional light source vector and settable surface characteristics that affect ambient, diffuse, and specular lighting coefficients. Atoms are generated much faster than polygonal spheres and should be used as a building block for three dimensional modeling.

Templates are pre-rendered, z-buffered primitives that are stored in off-screen memory*. They can be spheres, cubes, or arbitrary *rgbαz* bitmaps, and are rasterized by copying the *rgb* bits into the frame buffer. They, too, are useful building blocks for applications such as molecular modeling and volume rendering, and, because they are pre-rendered, are much faster than atoms.

PIClib has a rich set of commands for controlling light sources and shading properties. Directional, point and spot light arrays, up to fifty of each type, can be specified, and selected lights turned off and on as desired. Surface properties that control reflection coefficients for ambient, diffuse and specular lighting as well as transparency can be specified, and the user can choose from among flat, Gouraud, and Phong shading* for polygons and surfaces.

*Voxel templates and Phong shaded polygons are not available in Release 1.0 of PIClib.

PIClib Examples

In addition to the commands mentioned in the previous section, PIClib includes routines for more advanced graphics algorithms like object generation, antialiasing, texture mapping, and image compositing. Examples of these capabilities and algorithms are presented below, including a simple PIClib program.

Object Generation

Object generation is the ability to produce a complex object from a high-level description. The polygonal rendering algorithm in PIClib supports the generation of two- and three-dimensional curves and surfaces: circles, arcs, cubic curves, quadrics, superquadrics, bicubic patches. These objects can be created by specifying points, lines, or polygons.

High-level object generation is performed by the first node in the pipeline, called *Euclid*. This node receives commands from the host that define the object. These commands are removed from the message stream and replaced with the corresponding *point*, *move*, *draw*, and *polygon* commands that will generate the object. For example, if the host sends a *sphere* command, Euclid will output a set of polygon commands describing a canonical sphere at the origin. The rest of the pipe nodes treat the polygons as if they had been sent directly from the host. However, the computational task of generating them and the memory required to store them has been off-loaded from the host to the Pixel Machine. This approach can significantly reduce the size of host-resident data bases.

A simple program to draw a set of rings using the torus routine in PIClib is shown in Example 1. Once the initialization steps are complete, the program sends commands to turn on object generation mode and set up the lighting model. The remaining commands set up the lighting model and viewing matrix and interactively draw the rings.

A more complicated PIClib program generated the image in Figure 9. The airplane consists of 2116 bicubic patches and 321 spheres, and was rendered at 1280×1024 resolution with two directional light sources in 3.5 seconds. Each patch was tessellated into eighteen Gouraud-shaded polygons. The rivets on the plane body were rendered as Phong-shaded spheres.

```
#include <stdio.h>
#include "piclib.h"

#define NRINGS          4
#define FROM            300.0, 0.0, 380.0
#define AT              0.0, 0.0, 60.0
#define TWIST           0.0
#define FIELD_OF_VIEW   40.0
#define ASPECT_RATIO    1.25
#define DEPTH           1.0, 2000.0
#define EXP             0.3
#define XYZDimensions   20.0, 20.0, 20.0

extern PICsurface_model surface[NRINGS];

main()
{
   PIClight_source light;
   register int i;
   register float rad;

   light.nx = 1.0;
   light.ny = -1.0;
   light.nz = 1.0;
   light.r = light.g = light.b = 1.0;

   /* initialization */
   PICinit();
   PICzbuffer(PIC_ON);
   PICeuclid_mode(PIC_EUCLID_POLYGON);
   PICshade_mode(PIC_SHADE_GOURAUD);
   PICput_viewport(192, 1088, 185, 899);

   /* set viewport to black */
   PICcolor_rgb(PIC_BLACK);
   PICclear_rgb();

   /* establish viewing and projection matrices */
   PICpersp_project(FIELD_OF_VIEW,ASPECT_RATIO,DEPTH);
   PIClookup_view(FROM, AT, TWIST);

   /* set up lighting parameters */
   PIClight_ambient(PIC_WHITE);
   PICput_light_source(PIC_LIGHT_DIRECT, 0, &light);
   PIClight_switch(PIC_LIGHT_DIRECT, 0, PIC_ON);

   /* iterate to draw the rings */
   for (i=0, rad=150; i<NRINGS; i++, rad-=25;) {
      PICput_surface_model(&surface[i]);
      PICsuperq_torus(XYZDimensions, rad, EXP, EXP);
      PICtranslate_z(40.0);
   }
   PICexit();
}
```

Example 1. Object generation: Rings.

Antialiasing

Antialiasing has been implemented in PIClib by supersampling. The number of samples per pixel is specified by the user, as well as coefficients of an arbitrarily-sized filter kernel. The samples may be distributed within a pixel or across neighboring pixels, and the filter can be box, pyramid, gaussian, or whatever the user desires. The entire

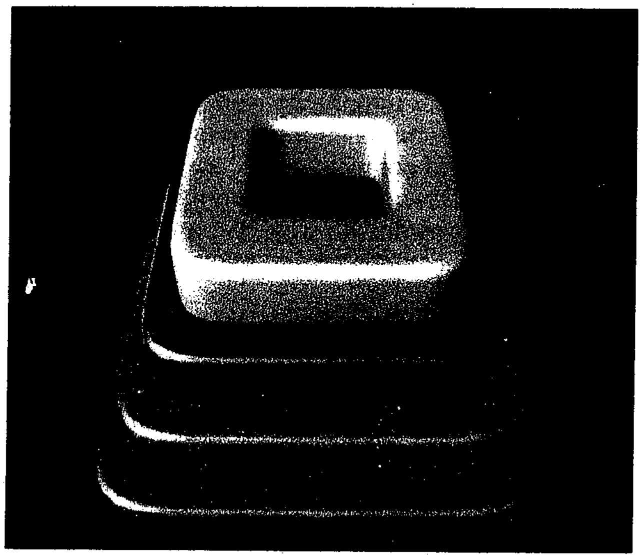


Figure 8. Object Generation: Rings generated by Example 1.

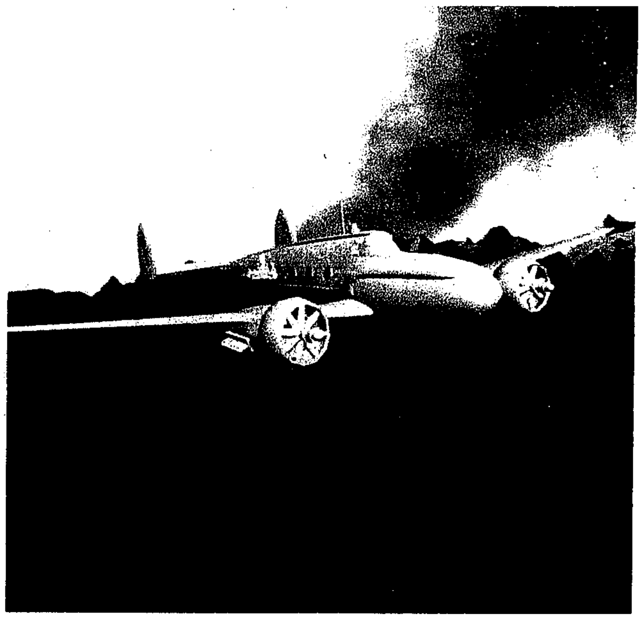
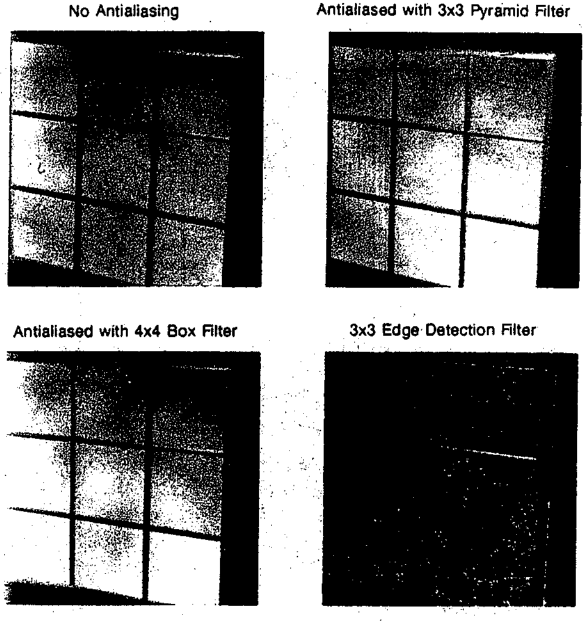


Figure 9. Object generation: Alias airplane with procedural clouds over Dog Mountain, Colorado.

image is rendered *n* times, where *n* is the number of samples per pixel. Each time the scene is rendered, the projection matrix is modified slightly, moving the center of a pixel to the current sampling point. The intensity values for each pixel are written into the frame buffer. At the end of each pass, the *rgb* contents of the frame buffer are multiplied by the current coefficient of the filter and accumulated in the external z-buffer in floating point format. After the final pass, the results of the convolution are converted from floating point to integer format and displayed in an *rgb* frame buffer.

One advantage to this approach is that it is general: special antialiasing rasterizers are not required for each primitive. It also eliminates the potential memory requirements of an a-buffer approach [C84]. Although it is brute-force, it quickly antialiases everything, including image regions that are not usually antialiased, such as specular highlights.



No Antialiasing

Antialiased with 3x3 Pyramid Filter

Antialiased with 4x4 Box Filter

3x3 Edge Detection Filter

Figure 10. Antialiasing: Pyramid, box, and edge-detection filters.

Figure 10 demonstrates antialiasing using three different filters. The original image, a brick wall with a window, is in the upper left corner of the figure. The scene is antialiased with a 3×3 pyramid filter (upper right corner), with a 4×4 box filter (lower left), and with a 3×3 edge-detection filter (lower right).

Texture Mapping

In real life, surfaces are seldom perfectly smooth, with constant color. More often, they are patterned or bumpy with color variations. Texture mapping, bump mapping, and environmental shading make a computer-generated scene look more realistic by adding variations to the surfaces.

PIClib provides a routine that allows the user to attach a texture map index, or *uv value*, to each vertex in a polygon. The texture maps, 256x256x32-bit images, are loaded into the video RAM of each pixel node via the broadcast bus. The texture maps can be antialiased using supersampling, as described above, and can be used to render any of the PIClib graphics primitives.

Of course, completely unrealistic but technically interesting effects can also be achieved with texture mapping algorithms. Figure 11 shows a Gouraud-shaded ellipsoid with a 256x256 image of a mandrill texture-mapped onto it. The ellipsoid, composed of 800 polygons, was rendered in 0.27 seconds at 600x600 pixel resolution.

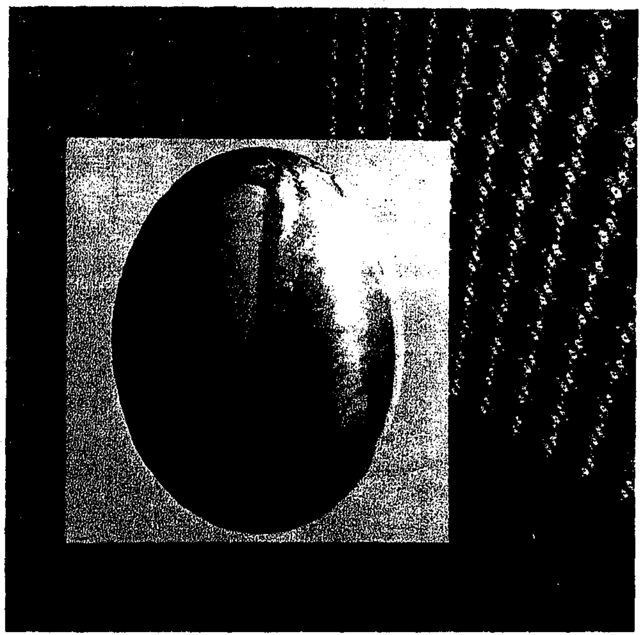


Figure 11. Texture mapping: Mandrill on an egg.

Mandrill on an Egg can be generated in the following way. First, the mandrill texture is stored in off-screen *rgbα* frame buffer memory Then, the host sends polygons, normals and *uv* texture indices at each vertex to the Pixel Machine. The polygons are transformed and shaded in the pipeline. The rasterizer in each pixel node receives messages containing polygon descriptions from the last node in the pipeline, including shading information and *uv* indices for each vertex. The surface intensity resulting from the shading calculations is combined with the intensity obtained from the texture map at each pixel and the *uv* values are bilinearly interpolated over the surface of the polygon.

Image Compositing

The large amount of video and dynamic memory in the frame buffer and the floating point capability of the processor facilitates the implementation of a powerful set of compositing tools. Figure 12, *A Dog Mountain Junkyard*, is a frame from an interactive program which allows a user to move a 3D cursor (located front and center in Figure 12) along the surface of the mountain, placing objects at various points along the route. The mountain range, consisting of approximately 350,000 polygons, is rendered only once. All *rgbα* and z-buffer information is saved in off-screen memory and copied to the current frame buffer at the beginning of each frame. The cursor is rendered on top of the mountain, while new objects are added to the stored *rgbα* and *z* data. Since the amount of computation per frame is minimal, this technique allows a user to manipulate the objects in this complex scene in real time.

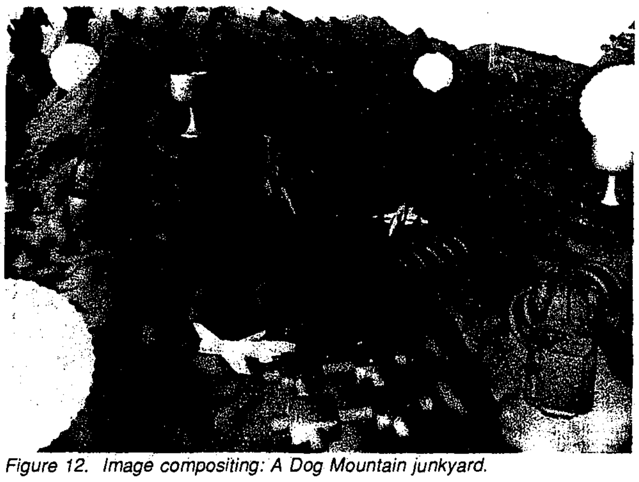


Figure 12. Image compositing: A Dog Mountain junkyard.

RAYlib

Ray tracing is a technique for rendering realistic, optically accurate images by tracing the paths of light rays to compute the illumination of objects. It accounts for reflection, transparency, refraction, and shadows in the generated image. Ray tracing is particularly well suited to the Pixel Machine, since it is a computationally intensive algorithm that is highly parallel in nature and requires large numbers of floating point operations.

The ray tracing algorithm involves three steps:

1. Generate the display list.
2. Form a tree of ray intersections with objects and generate new transmitted, reflective, and shadow rays.
3. Recursively apply the illumination model to nodes in the ray tree.

Rays are traced backwards from the viewpoint through each pixel in the screen to their origin at a light source. If there are *n*×*m* pixels on the display, then at least *n*×*m* rays will be traced, and each must be tested for intersection with each object in the image. Bounding volumes are used to reduce the complexity of the intersection test. If a ray intersects an object, the surface properties of the object determines whether the ray splits into several rays and what the paths of the rays will be.

Ray tracing is computationally very expensive. However, each ray can be traced independently of the others, making the algorithm well-suited for parallel implementation.

RAYlib is a ray-tracing library for the Pixel Machine. It contains approximately 100 routines, and includes the following features:

- area light sources that provide realistic soft shadows,
- multiple light sources of arbitrary color,
- adaptive stochastic antialiasing to remove jagged edges,
- texture mapping with independent surface qualities associated with each pixel, including color, reflectivity, and transparency,
- control over the degree of reflectivity and transparency of each object in the scene,
- the ability to independently turn on and off shadows, transparencies, antialiasing, and reflections for faster rendering.

Like PIClib, RAYlib is implemented as a C-callable library which provides an interface between an application program and the rendering functions in the Pixel Machine. The pipe nodes apply modeling transformations to objects and their bounding boxes, and other similar operations. Each pixel node keeps a copy of the display list, but ray-traces its portion of the image independently. Display lists that exceed the local memory of a pixel node are stored on the host and demand-paged to the pixel nodes. The z-buffer memory is used to store the display list and the texture maps are stored in off-screen *rgbα* memory. Texture maps larger than 256×256 are stored in the host and demand-paged to the pixel nodes. Up to 64 4K×4K textures are supported.

The image of the fractal *Sphereflake*, Figure 13, contains 7381 reflective spheres and three light sources. Up to 16 child rays were traced from each pixel to generate an image with 512×512 resolution. Table 3 lists Pixel Machine execution times for different numbers of pixel nodes. The last column in the table shows inverted and normalized timings illustrating the linear improvement in performance for this particular application. According to benchmarks by Eric Haines [H87], the Sphereflake can be rendered on high-performance workstations in four to five hours. A Pixel Machine with 64 pixel nodes took less than 16 seconds, three orders of magnitude faster.

Number of Pixel Nodes	*Actual Time*	*Normalized Time*
16	60.1	16.00
20	47.8	20.12
32	30.2	31.81
40	24.0	40.07
64	15.5	62.04

Table 3. Sphereflake execution times (seconds) for a 512×512 image.

Figure 1 is also a ray-traced image. It depicts a three dimensional model of the Pixel Machine, a workstation, a table, and two video monitors. The scene contains about 2000 polygons and is illuminated by two area light sources. The image is recursively mapped onto the Pixel Machine's monitor with a texture map.

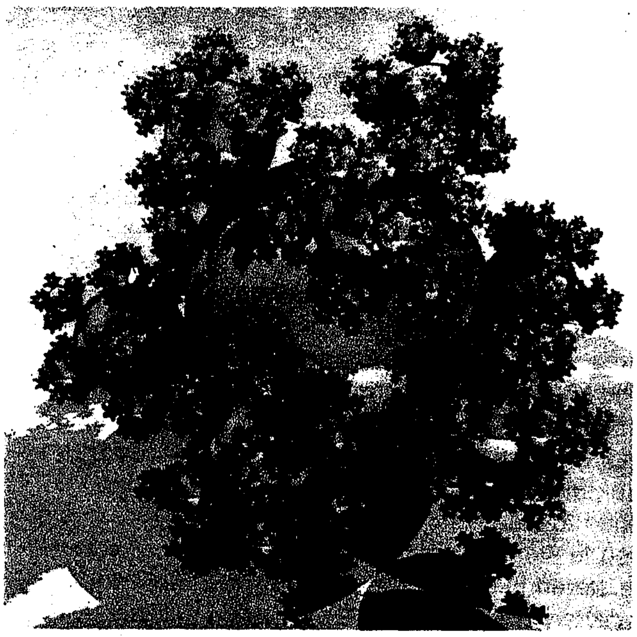


Figure 13. Ray tracing: Sphereflake.

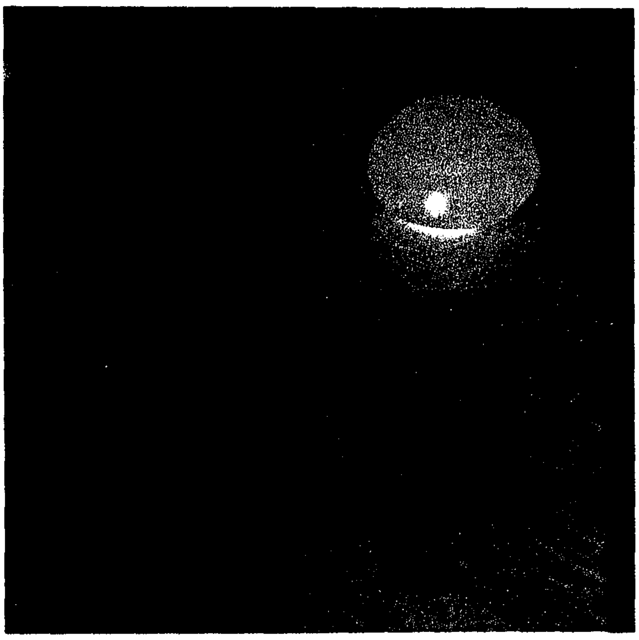


Figure 14. Ray tracing: the image generated by Example 2.

Example 2 shows a simple ray tracing program. The image it generates is pictured in Figure 14.

```
#include "raylib.h"

/* surface model definitions */
RAYsurface_model chrome = {
   0.2, 0.2, 0.2,  /* ambient color */
   0.2, 0.2, 0.2,  /* diffuse color */
   0.7, 0.8, 0.8,  /* specular color */
   100.0,          /* specular exponent */
   0.0, 0.8, 0.8,  /* transparency, specularity, reflectivity */
   1.0             /* refraction index */
};

RAYsurface_model matte = {
   0.0, 0.0, 0.0,
   0.0, 0.0, 0.0,
   0.0, 0.0, 0.0,
   0.0,
   0.0, 0.0, 0.0,
   1.0
};

struct { float x, y, z; } lightvertex[4] = {
   { 400.0, 400.0, 1000.0 },
   { 400.0, 600.0, 1000.0 },
   { 600.0, 600.0, 1000.0 },
   { 600.0, 400.0, 1000.0 }
};

main()
{  RAYlight_source light;

   /* initialization */
   if (RAYinit() != RAY_TRACE) exit(-1);

   RAYput_viewport(800, 1199, 50, 849);
   RAYlookat_view(1000.0,1000.0,1000.0,0.0,0.0,0.0,0.0);
   RAYpersp_project(10.0, 0.5, 0.0, 100000.0);
   RAYbackground_color(0.54, 0.75, 0.878);
   RAYlight_ambient(0.405, 0.562, 0.659);
   RAYambient_intensity(0.4);

   /* define light source */
   light.intensity = 500.0;
   light.r = light.g = light.b = 1.0;
   light.samples = 3;
   light.vertices = 4;
   light.vertex = (float *) lightvertex;
   RAYput_light_source(RAY_LIGHT_AREA, 0, &light);
   RAYlight_switch(RAY_LIGHT_AREA), 0, RAY_ON);

   /* specify matte surface model and
   /* draw textured polygon using default resident texture map */
   RAYput_surface_model(&matte);
   RAYpoly_point_uv(-1000.0, 0.0, -1000.0, 7.0, 7.0);
   RAYpoly_point_uv(1000.0, 0.0, -1000.0, 0.0, 7.0);
   RAYpoly_point_uv(1000.0, 0.0, 1000.0, 0.0, 0.0);
   RAYpoly_point_uv(-1000.0, 0.0, 1000.0, 7.0, 0.0);
   RAYpoly_close();

   /* specify chrome surface model and draw spherical atom */
   RAYput_surface_model(&matte);
   RAYatom(0.0, 70.0, 0.0, 50.0);

   /* define ray tracing parameters and start tracing! */
   RAYshade_mode(RAY_SHADOWS + RAY_ANTIALIAS);
   RAYsamples(5, 16, 0.25);
   RAYtrace();
   RAYexit();
}
```

Example 2. Ray tracing: A simple example.

DEVtools

DEVtools provides a set of tools and libraries necessary to develop application-specific code for the pipe and pixel nodes. They include

- a *host server* that allocates resources and provides services to the pipe and pixel nodes
- a C compiler, an assembler, and a linking loader for the DSP32
- libraries of I/O, graphics, and mathematics functions
- a symbolic debugger and dataflow simulator for the DSP32

The DEVtools host server handles all resource requests from the Pixel Machine. It monitors all or a specified subset of the nodes, waiting for messages. Whenever a message is received, a host function is invoked.

Some messages and functions are pre-defined. Examples are

- printing an ASCII message from a node to the standard output device, and
- setting the direction of the serial I/O links in the pixel nodes.

The user can add messages and functions to the server, customizing it to an application or an environment. By modifying the server, the Pixel Machine can be used for more than fast, fancy graphics. For example, intensive numerical computations can be programmed to run in the pipeline or the pixel node mesh with the results returned to the host in a user-defined message. This capability allows the Pixel Machine to serve as a general-purpose parallel computing engine.

DEVlib and a simple server program are included in DEVtools. DEVlib contains the pre-defined functions and the polling/dispatching routine that monitors communications from the nodes. Other functions help the user add messages and functions to the server. An example server program, DEVprint, is included and provides a skeleton which the user can customize to a particular application and environment.

The Distributed Frame Buffer

Programming the pixel nodes to access the interleaved frame buffer requires an understanding of two concepts:

- an algebraic *domain* transformation that maps from a screen space coordinate system to a *processor* space coordinate system, and
- techniques for rendering images in a *subscreen*, the small, contiguous frame buffer that is attached to each pixel node.

The domain transformation maps point from Cartesian (x,y) screen space to (i,j) processor space as follows:

$$i = \frac{1}{N_x}(x - O_x) \qquad j = \frac{1}{N_Y}(y - O_y)$$

where N_x and N_y are the number of processors per row and column, respectively, in the pixel node array, and are fixed for any given model of the Pixel Machine. O_x and O_y select a particular processor in the array, with O_x in the range $[0, N_x{-}1]$ and O_y varying between 0 and $N_y{-}1$.

The transformations from processor to screen space are:

$$x = i\,N_x + O_x \qquad y = j\,N_y + O_y$$

The effort required to parallelize an existing algorithm involves the restructuring of the algorithm so that it operates in the (i,j) processor space rather than screen space. Any algorithm that processes each pixel independently, such as fractal generation or ray tracing, requires very little modification, since no coherence is required from one pixel to the next. The number and complexity of modifications required increases with the degree of coherency between one pixel and the next or one scan line and the next. Writing a program so that it adheres to the domain transformation guarantees portability to single processor systems, where N_x=N_y=1 and O_x=O_y=0.

The pixel interleaving scheme presents an obstacle to applications that require a single pixel node to process and display a contiguous set of pixels. The serial I/O (SIO) capability of the pixel nodes provides continuous pixel manipulation. The set of pixels can be created in undisplayed memory and then routed, using SIO, to the pixel nodes that will display them. Table 4 shows timing information for distributing pixels for interleaved display using SIO. Three different pixel buffer sizes are used, as well as two different machine configurations.

Model		*Buffer Size*		
Number	*Nodes*	256×256	512×512	1024×1024
916	4×4	0.16	0.62	2.47
964	8×8	0.07	0.29	1.15

Table 4. Pixel interleaving times with SIO (seconds).

The pixel nodes are arranged in an *n*×*m* array, and the processor in the *i*th row, *j*th column handles every *n*th pixel on every *m*th scan line *(see Figure 7)*. Each processor addresses a portion of the frame buffer, which it sees as a contiguous subscreen. The coordinate system of the subscreen is called the *processor space*. DEVtools provides mapping functions from *(x,y)* screen space to *(i,j)* processor space.

`i = ILO(x)` returns the smallest i ≥ x.
`i = IHI(x)` returns the largest i ≤ x.

`j = JLO(y)` returns the smallest j ≥ x.
`j = JHI(y)` returns the largest j ≤ x.

Since there are more pixels in screen space than in processor space, the mapping is not one-to-one. To insure that the processor space pixel (i_1, j_1) is actually screen space pixel (x_1, y_1), the following condition must hold:

```
ILO(x1)==IHI(x1) && JLO(y1)==JHI(y1)
```

Here is a simple example. The code segment in Example 3 draws a set of vertical and horizontal lines in a screen space viewport defined by `xmin`, `xmax`, `ymin`, and `ymax`.

```
for (x=xmin; x<xmax; x+=delta)
for (y=ymin, y<ymax; y++)
   putpix(x, y, RED);

for (y=ymin, y<ymax; y+=delta)
for (x=xmin; x<xmax; x++)
   putpix(x, y, GREEN);
```

Example 3. Line drawing in screen space.

This code segment can be converted into code for the pixel nodes by adding a conditional statement to test for the pixel's presence in the processor space of this node, as shown below (Example 4).

```
for (x=xmin; x<xmax; x+=delta)
for (y=ymin, y<ymax; y++)
if ((i=ILO(x))==IHI(x)&&(j=JLO(y))==JHI(y))
   putpix(i, j, RED);

for (y=ymin, y<ymax; y+=delta)
for (x=xmin; x<xmax; x++)
if ((i=ILO(x))==IHI(x)&&(j=JLO(y))==JHI(y))
   putpix(i, j, GREEN);
```

Example 4. Line drawing in processor space.

The pixel node code shown above is straightforward but inefficient. It iterates across screen space, and does the processor space mapping and testing for each pixel. A better method is to iterate over processor space, as shown in Example 5.

```
imin = ILO(xmin);
imax = IHI(xmax);
jmin = JLO(ymin);
jmax = JHI(ymax);

for (i=imin; i<=imax; i+=delta)
for (j=jmin, j<=jmax; j++)
   putpix(i, j, RED);

for (j=jmin, j<=jmax; j+=delta)
for (i=xmin; i<=imax; i++)
   putpix(i, j, GREEN);
```

Example 5. Efficient line drawing in processor space.

In the next sections, two much more complicated examples of algorithms that might be implemented in the Pixel Machine nodes are presented.

Image Processing

Object generation, antialiasing, texture mapping, image compositing, and ray tracing examples demonstrate the Pixel Machine's ability to execute a wide range of *image synthesis* algorithms. The machine is equally well-suited for *image analysis* tasks, particularly since the pipe and pixel node processor, the DSP32, was designed for digital signal processing.

Adaptive histogram equalization [P87a] is a technique used to enhance the contrast in an image. At each pixel, a histogram of the intensities of neighboring pixels, called a *contextual region*, is examined. The histogram is equalized over the full intensity range and a new value is assigned to a pixel based on where its intensity falls relative to the neighboring values.

Figure 15 demonstrates adaptive histogram equalization. Figure 15(a) is the original image. Figures 15(b-d) illustrate the algorithm applied at each pixel with contextual regions of 63x63, 31x31, and 15x15 pixels, respectively.

The host and pipe nodes decode and download the image to the pixel nodes. Each pixel node has a copy of the image in its memory, and performs the adaptive histogram equalization algorithm asynchronously on the pixels in its own portion of the distributed frame buffer. Table 5 gives execution times for both 256x256 and 512x512 images using various contextual region sizes.

Region Size	256x256	512x512
15x15	0.35	1.10
31x31	1.22	4.20
63x63	4.20	17.00

Table 5. Adaptive histogram equalization execution times (in seconds).

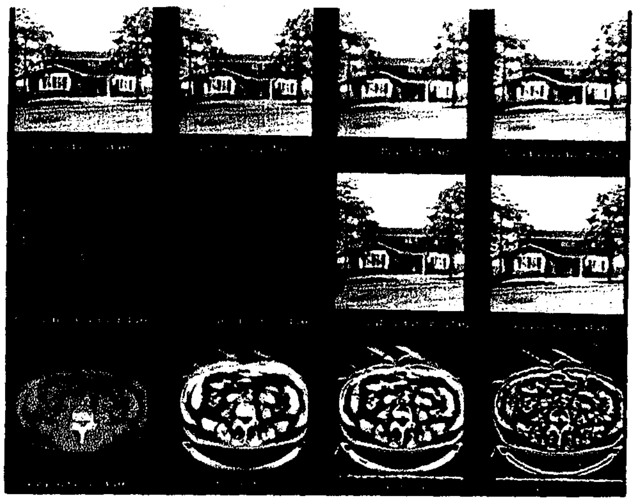


Figure 15. Image processing: Adaptive histogram equalization with (a) original image, (b) 63x63 region, (c) 31x31 region, and (d) 15x15 region.

Visualizing Complex Functions

Fractal geometry is a branch of mathematics used to describe self-similar structures of fractal dimension [M77]. The *Julia set* is a class of fractals in the complex plane.

A generating function is evaluated for discrete points in a rectangular region of the complex plane until the function diverges or a specified number of iterations is reached. The behavior of this function can be visualized by mapping the complex plane onto a raster display. The Julia set in Figure 16 was generated in 3.5 seconds at a resolution of 1280x1024. The color at each pixel is determined by the number of iterations of diverging points, and the function value for converging points, with up to 256 iterations per pixel.

This algorithm, shown in Example 6, has been implemented entirely in the pixel nodes. Each node computes the Julia set only at the complex points corresponding to pixels in its portion of the distributed frame buffer. The implementation is written in C and uses DEVlib routines that provide access to the frame buffer and apply the domain transformation from screen space to processor space.

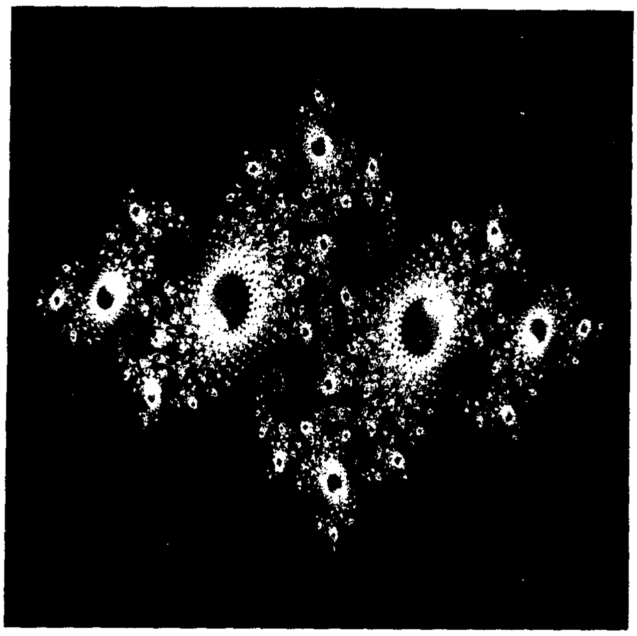


Figure 16. Fractal functions: A Julia set.

```
#include "pxm.h"

julia(screen,xmin,ymin,xmax,ymax,relo,rehi,imlo,imhi,P,Q)
{
   /* generate mapping functions */
   a1 = (rehi - relo)/(xmax - xmin);
   b1 = relo - a1 * xmin;
   FXTOFI(screen, a1, b1);

   a2 = (imhi - imlo)/(ymax - ymin);
   b2 = imlo - a2 * ymin;
   FYTOFJ(screen, a2, b2);

   /* convert screen coordinates to processor coordinates */
   imin = ILO(screen, xmin);
   jmin = JLO(screen, ymin);
   imax = IHI(screen, xmax);
   jmax = JHI(screen, ymax);

   for (j=jmin; j<=jmax; j++) {
      for (i=imin; i<imax; i++) {
         re = a1 * i + b1;
         im = a2 * j + b2;

         done=FALSE;
         for (n=0; n<MAX_ITER && !done; n++) {
            if ((z=re*re + im*im) <= MAX_Z) {
               temp_im = 2*re*im + Q;
               re = re*re - im*im + P;
               im = temp_im;
            }
            else
               done = TRUE;
         }
         if (done)
            putpix(screen, i, j, color_based_on_n);
         else
            putpix(screen, i, j, color_based_on_z);
      }
}
```

Example 6. Fractal functions: A Julia set renderer.

References

C84. Carpenter, L., "The A-buffer, An Antialiased Hidden Surface Method," *ACM Computer Graphics* **18**(3), pp. 103-108 (July, 1984).

F85. Fuchs, H.; Goldfeather, J.; Hultquist, J. P.; Spach, S.; Austin, J. D.; Brooks, F. P. Jr.; Eyles, J. G.; and Poulton, J., "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *ACM Computer Graphics* **19**(3), pp. 111-120 (July, 1985).

H87. Haines, E., "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Applications* **7**(11), pp. 3-5 (November, 1987).

K78. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language,* Prentice Hall, Englewood Cliffs, New Jersey (1978).

K85. Kershaw, R. N.; Bays, L. E.; Freyman, R. L.; Klinikowski, J. J.; Miller, C. R.; Mondal, K.; Moscovitz, H. S.; Stocker, W. A.; Tran, L. V.; Hays, W. P.; Boddie, J. R.; Fields, E. M.; Garen, C. J.; and Tow, J., "A Programmable Digital Signal Processor with 32b Floating Point Arithmetic," *Proceedings of the IEEE International Solid-State Circuits Conference*, pp. 92-93 (February, 1985).

M77. Mandelbrot, B., *The Fractal Nature of Geometry,* W. H. Freeman and Company, New York, NY (1977).

P87a. Pizer, S. M. and al., et., "Adaptive Histogram Equalization and Its Variations," *Computer Vision, Graphics, and Image Processing* **39**(3), pp. 85-93 (September, 1987).

P87b. Potmesil, M. and Hoffert, E., "FRAMES: Software Tools for Modeling, Rendering, and Animation of 3D Scenes," *ACM Computer Graphics* **21**(4), pp. 85-93 (July, 1987).

Credits

The airplane in Figure 9 is used courtesy of Alias Research, Inc.

The Sphereflake model in Figure 13 was designed by Eric Haines of 3D/Eye.