PhD Dissertation Proposal
Marc Olano
30 August 1994

Thesis Statement

The decomposition of the graphics pipeline into a coherent set of programmable functions provides greater flexibility and valuable new tools to the graphics programmer. Furthermore, this enhanced flexibility can be implemented efficiently to yield systems that maintain interactive frame rates.

Abstract

It has been recognized that given the wealth of shading possibilities, no single parameter-based shading model could ever be sufficient [Hanrahan90]. In response to this, procedural shading and shading languages have arisen to give the graphics programmer full access to the range of shading algorithms. However, the existence of this kind of flexibility at other points in the graphics pipeline is in its infancy. I intend to show that it is worthwhile to allow programmability throughout the graphics process. I will show this by first implementing programmable hooks for the PixelFlow graphics library [Molnar92], and then finding one or two people to use the added capabilities or solving a couple of suitable problems proposed by the committee.

Contributions

- A new decomposition of the rendering process into a framework and a logical and orthogonal set of functions.
- The idea that such a system can and should be designed for use by the graphics programmer/user.
- The idea that such a system can be implemented efficiently enough to be interactive on near-term graphics hardware.
- The design and demonstration of such a system.
- Separation into rasterizers and interpolators to decouple scan conversion from the interpolation of shading parameters.
- Data structure for efficient caching of composited linear transformations while still allowing general programmable transformations.

Motivation

Examples

All current graphics machines are designed as essentially closed systems. It is difficult, if possible at all, to do tasks not originally envisioned and programmed in by the system designers. Yet there are tasks that could run efficiently within the graphics pipeline architecture if there were only a way to modify the behavior of the right piece of the system. The purpose of this dissertation is to design a rendering system providing this kind of power, and to show that it can be done efficiently enough to use in an interactive system. What follows are some examples of what could be done with this capability in a graphics system. For each I will briefly discuss how it might be implemented with a typical closed architecture graphics pipeline and with a programmable pipeline.

- Many current head mounted displays (HMDs) introduce significant optical distortion in order to create a wide field of view. Images created assuming a rectangular array of pixels will be warped in the HMD. To present undistorted images, it is necessary to produce an image with the inverse warping.
- Frameless rendering is another task that does not fit well within closed graphics pipelines. If updating pixels is a limiting factor for graphics speed, a randomized subset of the pixels can be updated. This can allow a faster frame rate (zero in the limit) at the cost of some image quality during motion.
- Most real-time animation of hands, faces, or figures use intersecting rigid parts. Yet the intersections do not look very natural. But figures seen in production animation for commercials and film usually don't have these seams. One way they avoid the intersections is to use a single connected model which they manipulate using deformations.
- Another method to solve the same problem (not as widely used) is to use an implicit surface shell around an articulated skeleton.
- In the flight simulator Trey Greer wrote on Pixel-Planes 5 for IVEX, he had a interesting landing light primitive. The light is rendered as a single pixel. As the intensity of the light increases, so does the intensity of the pixel. Once the single pixel is at full intensity, the size increases to keep the aggregate intensity correct.
- John Alspaugh used his access to the Pixel-Planes 5 graphics library source to add a new primitive to the system. He added featured polygons, convex polygons with convex holes. Featured polygons are based heavily on the existing polygon rendering code and are a full-fledged addition to the system.

• David Banks used the Pixel-Planes 5 GP callback facility (designed to provide some degree of immediate mode graphics on a retained mode graphics system) to run his own version of the polygon rendering code. His Phong shaded polygons could find and mark their intersections and silhouettes.

HMD warp

Closed pipeline

To produce an image that will end up undistorted in the HMD, it is necessary to produce an image with the inverse warping. The graphics pipeline system will produce an image with a rectangular pixel grid. If an image large enough to cover the field of view of the HMD is rendered, it can be warped by a post process into the image that will be undistorted in the HMD. The HMD optics will probably have the highest pixel density in the center of the image. The original pixel grid will need to be of a fine enough resolution to adequately cover this region. The warp will take extra processing power, so to remain interactive it will need to either be done on multiple application processors or in some way take advantage of the graphics hardware. If image texturing is supported (and loading texture memory is cheap enough), the warp could be done with a textured polygonal grid. Of course if some graphics power is used for the warp, less will be available for the graphics.

Programmable pipeline

A programmable pipeline can use the same attack with the image warp as a final stage in the pipeline. As a stage in the pipeline, excess image copies may be avoided. Being in the pipeline, the warping can happen before anti-aliasing. The rendered image is stored as a texture map, then the texture coordinate of each pixel in the final image is computed from the warp function (more likely it will have been precomputed since it never changes), so each pixel determines its color from the texture map.

If the HMD optics also introduce variations in intensity across the image, each pixel can have an intensity modifier as well as a texture location. By warping each color seperately, it is even possible to do some compensation for chromatic aberration in the optics where a slight prism effect spreads the colors.

Alternate programmable pipeline solution

Another way to approach the same problem is to warp the pixel locations during scan conversion. This can be easily accomplished with a new scan conversion function that uses these warped locations instead of the pixel locations from the rectangular grid. Since the pixel to pixel coherency will be lost (or at least much harder to take advantage of), the scan conversion will be slower than standard scan conversion for almost any graphics system. However, for a wide field of view rectangular image, some pixels land

entirely outside the warped image and many of the peripheral pixels have only minor effects image in the HMD. The additional resampling also complicates anti-aliasing. Using warped pixel locations has neither of these problems. Which solution is superior will depend on the graphics hardware and the type and degree of image warping.

Frameless rendering

Closed pipeline

I can think of no possible solution for a closed pipeline. The effects can be simulated, as they were on Pixel-Planes 5. The Pixel-Planes simulation of frameless rendering generated multiple frames and discarded all but a random subset of the pixels. With the full frame computed anyway, there is no frame performance gain to showing only some of the pixels. I have overheard people familiar with only closed pipelines say that frameless rendering is only a useful method for ray tracing and volume rendering.

Programmable pipeline

Frameless rendering can be realized in a programmable pipeline with two cooperating procedures. A new rendering procedure scan converts random pixels instead of the normal rectangular frame, and a warping procedure descrambles them into their correct positions in the image. Since the rendering procedure throws away the advantages of pixel to pixel coherence, it will run more slowly. Descrambling the pixels may take some processing as well. But factored over a quarter million pixel NTSC screen or a couple million pixel HDTV screen, it can still achieve its purpose of higher frame rates at the cost of degraded images.

Deformation of a rigid object

Closed pipeline

To handle deformations in a closed pipeline, each vertex or control point of the model must be transformed by the application code. The deformed object can then be sent through the normal rendering system. This does not take advantage of any of the rendering system to do the deformation.

Programmable pipeline

Deformations are just a form of non-linear transformation. As a programmable transformation, they are trivially incorporated into a programmable pipeline.

Implicit skin around a rigid skeleton

Closed pipeline

First, the skeleton is transformed by the application code. Then a polygonal mesh is found for the implicit skin. Methods for this are marching cubes, shrink wrapping, or point repulsion, all of which involve quite a bit of computation. Finally the resulting polygonal mesh is sent through the rendering system.

Programmable pipeline

What is needed is an implicit surface primitive. But even this may be hard to create interactively. For all of the algorithms I can think of, the entire skeleton needs to be one primitive which the primitive procedure transforms. One option for rendering is to have each pixel run a ray casting to find if and where it intersects the model. Another option is to use a Newton iteration based root finder to take advantage of the pixel coherence.

Landing lights

Closed pipeline

If the rendering system has a point primitive, the light can be rendered as a point as long as the intensity remains low enough. However, as soon as it gets bright enough to start expanding (or if no point primitive is available), more work is required. One method would be to have the application first transform the light into screen space, then knowing where the light needs to end up on the screen, use the inverse transform to find out what object coordinates to give the rendering process.

Programmable pipeline

In a programmable pipeline, it is simple to create a new primitive whose position is based on an object location but whose size is purely screen space. The code for the primitive takes in the location of the light, transforms it to screen space, then draws a disk around that screen space point.

Featured polygons

Closed pipeline

About the only alternative on a closed pipeline that only supports convex polygons is to break the featured polygon up into several convex pieces. Unfortunately, doing that optimally is hard (Hard enough to make the UNC Walkthrough project decide to modify the graphics library rather than do it as part of their database conversion). A naive procedure that splits into trapezoidal segments at each vertex should be relatively easy, though it may produce an excessive number of polygons. These polygons can then be passed to the rendering system.

Programmable pipeline

It is surprisingly easy to extend the standard polygon scan conversion code to handle convex and complex polygons. The scan converter just keeps track of the number of edge crossings as it crosses the span to know if it is inside or outside the polygon. Even Pixel-Planes style scan conversion can cut out a convex piece about as easily as it can draw another convex polygon. So a new scan conversion function in a programmable pipeline can easily handle featured polygons.

Surface intersection and silhouette

Closed pipeline

This is very difficult in a closed pipeline. Polygons with silhouettes will be the ones with some normals with a positive Z in screen space and some with a negative Z. To find these, the application must transform every polygon's normals into screen space. The silhouette curve will be where the Z component of the normal is 0. To find intersections, the application must check every polygon against every other (though it need not transform into screen space for that step). After all of this, the polygons and markings must be sent to the graphics system (which will do all of those transformations over again).

Programmable pipeline

A new primitive can be used to find both intersections and silhouettes. To find silhouettes, it is possible to look at the normal on a pixel by pixel basis as it is scan converted. The normal needs to be computed anyway, so marking it is just a matter of watching when it gets near zero. A fancy procedure could also look at the rate of change of the normal to make even width markings. To find silhouettes, the primitive compares not just whether its Z is in front or behind what is already drawn, but also how close. If the pixel is just in front it is marked as being on an intersection; if it is just behind, the pixel that is already there is marked as an intersection (with appropriate checks to make sure it also came from the right kind of polygon). Once again, a fancy procedure could also look at the rate of change of Z to make even width markings.

Procedure points

Maps

Procedural maps can provide one or two parameter functions. They are the programmable version of the common image maps used for textures, shadow maps, reflection maps, and bump maps. At some point the shading process uses the map to look up a color, number, or vector.

Transformation

Procedural transformation allows deformations and other nonlinear transforms. There is a progression of shading complexity from hard-coded simple shading models through bump and reflection mapping to full programmable shading. There is a similar progression for transformations from the standard affine transformations through free form deformations to full programmable transformations.

There are some problems presented by the warping of space caused by non-linear transformations, for example flat polygons remain flat only under projective transformations. However, they are still a powerful tool that should not be ignored.

Modeling

Procedural models use object parameters to generate a lower level description of the model. Most polygon based systems implement spline patches in this way, but do not allow users to write their own modeling procedures. A few possibilities are particle systems, fractals, L-systems, graftals, hypertextures, spline patches, and generative models.

Primitives

Procedural primitives differ from procedural models in that they are directly rendered instead of being converted into simpler primitives. There is consequently a good deal of overlap between the objects which can be created as procedural primitives and those that can be supported as procedural models.

The Pixel-Planes 5 graphics system provides good anecdotal support for the need for procedural primitives. No user-level support is provided for writing primitives, yet a number of users have gone through the trouble to write their own custom primitives.

Given the number of primitive types currently available, the chances are good that any given rendering system will have missed several of them. A few possibilities are polygons, spline patches, spheres, quadrics, superquadrics, metaballs and other implicit models, generative models, smart points, and volume elements. Some of the more interesting candidates can be found in [Lane80], [Max81], [Blinn82], and [Kajiya83].

Volume and Atmospheric Effects

Procedural volume and atmospheric shaders handle the behavior of light as it passes through a medium. A few examples are fog, haze, atmospheric color shift, and density thresholding.

Shading

Procedural shading describes the shading of a surface by the code used to turn the surface attributes and shading parameters into a color. Over the past several years there has been a trend among high-quality rendering programs to include shading languages and procedural shading capabilities.

Lighting

Procedural lighting functions determine the intensity and color of light that hits a surface point from a light source. They can be used for a variety of shadow and slide projector techniques.

Image Warping

Procedural warping can be used to support a host of video warping special effects as well as non-rectangular pixel grids as commonly seen in head mounted displays.

Image Filtering

Image filtering is similar to image warping in that it may require non-local access to pixel samples. The main filtering effect of interest in graphics is anti-aliasing.

1.10. Application to a Typical Rendering Pipeline

Figure 1, a typical rendering pipeline, was based on the OpenGL software architecture [OpenGL92]. The lighting and texturing stages are covered by what I have called procedural shading and lighting; the primitive assembly stage by procedural modeling; the modeling, perspective, and view transform stages by procedural transformation; the clip and scan convert stages are covered by procedural primitives; the fog stage is covered by procedural volume and atmospheric affects; And the anti-aliasing stage is covered by procedural image filtering.

Research Plan

Thesis Statement Revisited

A traditional graphics pipeline with programmable hooks throughout the process can be implemented efficiently and provides the graphics programmer an easier and faster way to do tasks that would otherwise be difficult or impossible.

Demonstration of Thesis

I plan to demonstrate that the graphics pipeline with programmable hooks mentioned above are practical, can be implemented efficiently, and are powerful.

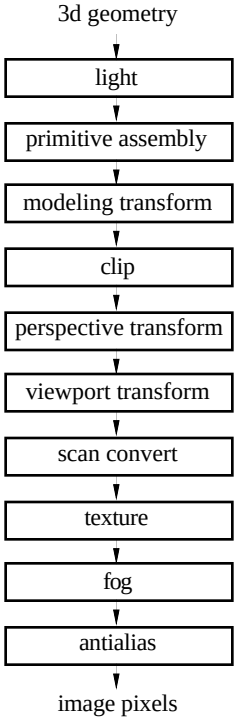


Figure 1. A typical rendering pipeline

Practicality and Efficiency

To show practicality and efficiency, I will add programmable hooks to the PixelFlow graphics system. There are some limitations caused by this decision: I will not be able to support displacement mapping; most transparency will have to be screen door transparency; any multipass algorithm (transparency, reflection maps, shadow maps) will probably have a weird interface caused by the PixelFlow architecture; and most of the procedures will have to be programmed in a special purpose language.

The special purpose language is modeled after the RenderMan shading language. This is a matter of convenience and the use of special purpose languages for these procedures are not really part of my thesis. RenderMan uses a special purpose language for portability and to allow unusual optimizations. I also have unusual

optimizations in mind for my special purpose language, but I am not interested in portability. There is no compiler for C or any other high level language for the PixelFlow renderers. If I want to provide any interface above assembly language level, I must write my own compiler. So I have somewhat selfishly decided to go for a simpler compiler instead of spending my time implementing a full C compiler.

Points where I will provide procedural hooks in the PixelFlow implementation are transformation, primitive creation, shading, lighting, atmospheric effects, and image warping and filtering.

Power

To show the increased capabilities and ease of use, I would like to find one or two people in the department to use the procedural capability on PixelFlow. I do not have anyone in mind yet, but I believe that the chances are good that I will be able to find people. Several people have made modifications directly to the Pixel-Planes 5 software who would have been perfect candidates. I do not imagine that the need for these tools will disappear.

Test users will have to be fairly committed to learning and using the system. While I am optimistic, I have no assurance that I will be able to find any. If I fail to find willing participants, I think it should be sufficient for me to solve a couple of reasonably difficult problems posed by the committee.

Background

Varying degrees of programmability have been previously provided at points in the graphics process. The examples range from high level shading languages down to graphics systems that are designed to be reprogrammed only by the original author.

Maps

The procedural shading capability described in [Rhoades92] are really just an instance of procedural maps in a fixed Phong shader.

Transformations

Two different kinds of nonlinear deformations have been proposed [Barr84], [Sederberg86]. Barr defines deformations based on linear transformations where the transformation parameters are a function of position. Sederberg defines free form deformations as spline warping of the space around an object. The full range of deformations is much larger than the subset covered by these two types, and they do not even provide very intuitive control. In practice, simplified parameterization of their controls are preferred [Watt92], [Reeves90].

Some definitions and implementations of procedural transformation have been made. Fleischer and Witkin [Fleischer88] defined true user-accessible programmable transformations as point, normal, and inverse transform functions. The RenderMan specification defines transformation shaders, though they are not implemented in PIXAR's PhotoRealistic RenderMan. Displacement mapping can be used as a form of procedural transformation. And PIXAR's MENV system allows procedural definitions of deformation parameters [Reeves90].

Modeling

Examples of some procedural models can be found in [Lane80], [Kajiya83], [Blinn85]. I have not yet seen a standardized procedural model description language (unless you count graphics libraries like OpenGL), but several systems supporting procedural models have been created [Hedelman84], [Amburn86], [Upstill90], [Green88], [Perlin89].

Primitives

Procedural primitives are easily supported in ray tracers and examples can be found in [Rubin80], [Hall83], [Wyvill85], [Kuchkuda88], and [Kolb92]. Examples are much harder to find for non-ray traced renderers. A handful of "testbed" systems have provided some degree of primitive programmability: [Whitted82], [Crow82], [Hall83], [Fleischer87], [Nadas87]. The key shortcoming of these testbed systems seems to be that they are designed either for testing out new primitive algorithms or for use only by the original author.

Volume and Atmospheric Effects

Cook [Cook84] defines atmospheric shade trees to handle fog and similar color effects between the surfaces and the viewer. RenderMan [Hanrahan90] extends this to two types of shaders, volume and atmospheric, to handle the passage of light through and outside of objects in the scene.

Shading

In the earliest systems, programmability was supported by rewriting the shading code for the renderer [Max81]. Sometimes this was specifically allowed by the design of a testbed system [Whitted81], [Hall83]. More recently easier access to procedural shading capability has been provided to the graphics programmer [Cook84], [Perlin85], [Abram90], [Hanrahan90]. The RenderMan shading language [Hanrahan90] is even presented as a standard so shaders can be portable to any conforming implementation (though the rush to write conforming renderers has yet to happen).

Lighting

RenderMan [Hanrahan90] and Cook's shade trees [Cook84] make an important conceptual distinction between lighting and shading. The same light procedure may be used by all of the shading procedure. A prime example of the power of lighting procedures is the window pane light in PIXAR's Tin Toy [Upstill90]

Image Warping and Filtering

Image filtering is incorporated in the RenderMan image shaders. Both filtering and warping are provided by Adobe Photoshop's plug in capability [Knoll90] (though this is not part of a rendering system).

Systems

Most of the programmable systems were mentioned in the primitives section since fully programmable systems are pretty much the only context in which procedural primitives have been explored. [Whitted82] allowed programmable primitives and shaders. [Crow82] used a separate process for each primitive. [Hall83] and [Trumbore93] were systems designed for Cornell's global illumination research. [Hedelman84] was a data flow based system allowing procedural models, transformations, and shaders to be connected together. [Fleischer87] was a conceptually elegant LISP system. It has a nice generalization for transformations, but can only deal with surfaces with both implicit and parametric formulations and is clumsy for shading. [Nadas87] was a data flow based system allowing C functions to be hooked together in a directed acyclic graph. [Glassner93] was a similar system based on C++. [Reeves90] and [Hanrahan90] describe most of PIXAR's internal rendering system which supports procedural modeling and shading. [Kolb92] is a ray tracer based on [Kuchkuda88], both of which are designed to be easily modified.

Implementation

I have already begun implementation of the demonstration system. It will be part of the standard PixelFlow graphics library.

PixelFlow

PixelFlow consists of a host, a number of renderer boards, a number of shader boards, and one or a small number of frame buffer boards (Figure 2a). The hardware and lower level software handle the details of scheduling primitives for the renderer boards, compositing pixel samples, assigning them to shader boards, and moving the shaded pixel information to the appropriate frame buffer. Consequently, it is possible to take the simplified view of PixelFlow as a simple pipeline (Figure 2b).

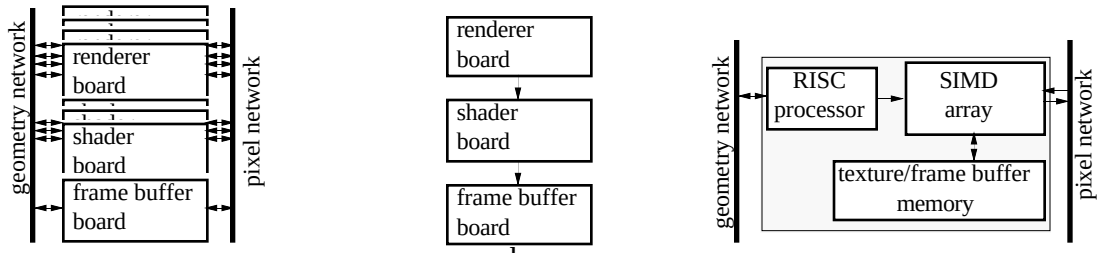


Figure 2. Three views of PixelFlow: a) hardware system block diagram. b) simplified view of the system. c) simplified view of a single board.

The boards on PixelFlow all look quite similar. Each board of the PixelFlow system has two RISC processors, a SIMD array of pixel processors, and a texture memory store (Figure 2c). Part of the implementation is a compiler for a simple C or RenderMan like language that produces C++ for one of the RISC processors with embedded commands to the SIMD processors. This will remove many of the hardware specific quirks, making the demonstration system more generally applicable.

Pixel information can easily be passed from board to board in the SIMD processor memory and frame information can easily be passed from RISC processor to RISC processor.

Pipeline

PixelFlow will have an API based on OpenGL, though our pipeline differs from the OpenGL pipeline presented earlier. It differs to better fit the PixelFlow architecture and to make the procedural hooks make more sense. Figure 3 shows the processor pipeline and one of our proposed procedure organizations. To summarize the points in the pipeline:

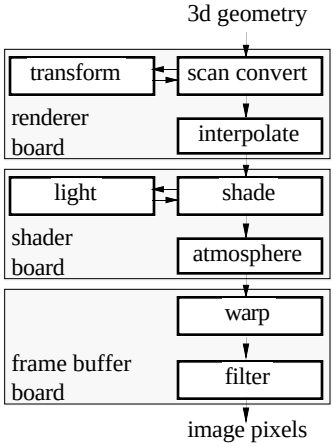


Figure 3. procedure pipeline

- Procedural modeling is not covered since it can be done effectively with the API.
- The scan conversion procedure turns 3D geometry data from the API into a set of enabled samples to participate in the later processing.
- Each transformation has a set of functions to transform points, planes, normals, and tangents which are called as needed by the scan converter.
- The interpolator procedure interpolates arbitrary parameters across the primitive for the later stages. (Separation of the interpolation from scan conversion makes handling of arbitrary parameters easier and allows reuse of a number of standard interpolators for multiple primitive types.)

• Shading, lighting, and atmosphere act in fairly standard ways.
• The warp procedure can access data from other samples in the image to determine the new sample contents.
• The filter procedure combines samples for a pixel into the final pixel value.

Procedures will communicate through a shared parameter space. This is similar to the shared memory "blackboard" idea used by MENV [Reeves90]. Procedures will have their own default parameters. These may be overridden by the parameters given in the OpenGL code which may be overridden by replacement values from earlier procedures.

Framework

Supporting this pipeline is a software framework that handles the details of the rendering process and the communication between the programmable procedures. For example, it is part of this framework that allows the procedure writer to pretend PixelFlow is a simple pipeline instead of a large multicomputer. Of particular interest is the collection and redistribution of procedure parameters.

Most of the details of parameter routing and distribution are handled by the host in the extensions to the OpenGL API. Each procedure has a set of parameters with default values. The host collects a table of these parameters and keeps track of the current value for each. The parameters are collected through query functions provided along with the procedures. Query functions are used to allow procedures to be added and removed on the fly. Each parameter is accessed through an ID number so all parameter operations reduce to table lookups.

The transformation functions provide a good example of the use of these parameter tables. When a transformation function is called, it is linked into the list of active transformations. The values of all of its parameters are taken from the parameter tables and stored as well. This information on transformation state is sent to the rendering boards over the geometry network. When a scan conversion procedure tries to transform a point, each transformation on the list is called in turn with its parameters and the partially transformed point.

Scan conversion and parameter interpolation operate slightly differently. When the glBegin call is issued in the application code, the host begins to collect parameters for a primitive. It can determine, based on the currently active procedures, what parameters will be needed, either for scan conversion or in the pixels for later procedures. It builds lists of these parameters, one entry in each for each vertex or control point. These, along with any single valued parameters for scan conversion or interpolation, are packaged up and sent to a rendering board. On the rendering board, the scan converter is called once

to compute the participating pixels, then an interpolator is called for each pixel parameter needed by the later stages.

Shading procedures may have parameters that vary across the pixels. As an added complication, their execution is not triggered directly by the application code, so all of their parameters must be saved and delivered to them when they are run. To deal with this, the parameters are labeled when the procedure is written as either *varying* or *uniform* (a terminology taken from [Hanrahan90]). Varying parameters are assumed to reside in the pixels, but uniform parameters must be sent over the geometry network to the shader boards. To keep the assignment of uniform parameters and shaders straight, a *shading function* and its uniform parameters are bound together in a manner similar to the OpenGL display list and given a *shader ID*. Each pixel on the screen includes the shader ID in pixel memory, so on the shader board its uniform parameters can be found.

Only one procedure per frame is allowed for each of atmosphere, warp, and filter. When one of these procedures is specified, the parameters that do not vary across the pixels are sent directly to the appropriate boards along with the procedure ID. Information on parameters that vary across the pixels, but do so in a uniform way over the screen are also sent to the appropriate board to be dealt with later. Parameters that vary on a primitive by primitive basis are rasterized into the pixel memory on the renderer boards. The same approach is used for lighting functions, but pixel parameters for lights will be discouraged on PixelFlow if they are supported at all since they will be quite expensive.

PixelFlow composition bandwidth is limited, so it is best to keep the amount of information passed at a pixel level as small as possible. Several things are done to help with this goal. Each shader has its own pixel memory map. The size of the composition is determined by the requirements of the largest shader. To keep the memory packed, the memory map is rearranged after each procedure to prepare for the next.

Efficiency

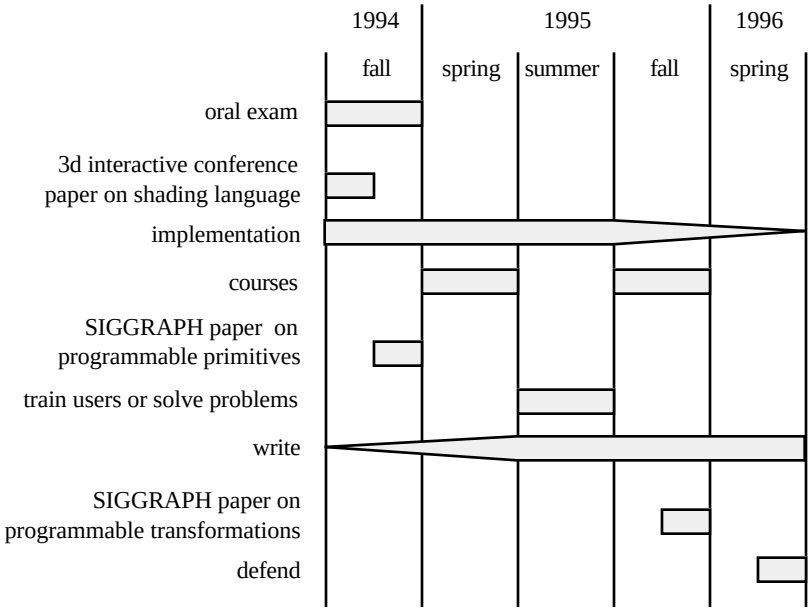
There are several techniques that can be used to make the system efficient and usable. They will be briefly mentioned here:

- Time critical procedures can be replaced by versions that have been hand coded in C++ or assembly language. This is planned from the start so the output of the compiler will be reasonable for hand generation.
- Careful choice of how to pass parameters can reduce the amount of computation that needs to be done. Parameters which change over a primitive must be sent across the pixel network. Parameters that are the same over many primitives are more efficiently sent over the geometry network. Parameters which do not change across a primitive,

but do change often between primitives are more efficiently sent over the pixel network. If only pixel parameters have changed, the SIMD code is still valid and procedure does not need to be rerun on the RISC processors.

- Certain computations can be factored out from several procedures and run simultaneously. On a large scale this could include execution of light procedures for all shaders simultaneously. On a smaller scale, this could include common execution of noise functions or other common expressions.
- A special case is made for affine and projective transformations since they are the most common. It is possible to cache a composite transformation when several matrix transformations appear in sequence without loosing the ability to have arbitrary procedural transformations. In the case where there are no procedural transformations, it would cost only a single comparison over the standard matrix formulation.

Schedule



Annotated Bibliography

[Abram90] Gregory D. Abram and Turner Whitted, “Building Block Shaders”, *Computer Graphics (SIGGRAPH '90 Proceedings)* , volume 24(4), August 1990, 283–288

An implementation of Cook's shade trees.

[Amburn86] Phil Amburn and Eric Grant and Turner Whitted, “Managing Geometric Complexity with Enhanced Procedural Models”, *Computer Graphics (SIGGRAPH '86 Proceedings)* , volume 20(4), August 1986, 189–195

Describes two ideas for procedural modeling. First is generalized subdivision, where a representation is subdivided until a transition test triggers a change of representation. The new representation may undergo further subdivision or be rendered as a primitive. Second is communication between models. Two interacting models send messages to adjust to each other.

[Banks92] David Banks, “Interactive manipulation and display of two-dimensional surfaces in four-dimensional space”, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)* , volume 25(2), March 1992, 197–207

Uses a custom primitive on Pixel-Planes to find intersection and silhouette curves.

[Barr84] Alan H. Barr, “Global and Local Deformations of Solid Primitives”, *Computer Graphics (SIGGRAPH '84 Proceedings)* , volume 18, July 1984, 21–30

Basic deformations like twist, taper, bend, etc. Standard transformations parameterized by position.

[Bentley88] Jon Bentley, “Little Languages”, *More Programming Pearls* , Addison-Wesley 1988, 83-100

A definition of little languages and an argument in favor of their use.

[Blinn82] James F. Blinn, “A Generalization of Algebraic Surface Drawing”, *ACM Transactions on Graphics* , volume 1(3), July 1982, 235–256

Ray tracing of blobby models. Blobs are implicit functions based on a Gaussian density function. Uses bounding spheres to limit the number of active blobs and solves for intersection using a hybrid Newton/regula falsi method.

[Blinn85] James F. Blinn, “The Ancient Chinese Art of Chi-Ting”, *SIGGRAPH '85 Image Rendering Tricks seminar notes* , 1985

A collection of rendering tricks from JPL space movies and the Mechanical Universe. The first section has text, the rest is in outline form. Section 1.4 is titled Specialized Primitives and describes his 2 polygon cylinders (also published as “Jim Blinn's Corner: Optimal tubes” in CG&A, September 1989)

[Blinn89] James F. Blinn, “Jim Blinn's Corner: Optimal tubes”, *IEEE Computer Graphics and Applications* , volume 9(5), September 1989, 8–13

Derives two or three polygon cylinders used in JPL space movies and the Mechanical Universe. Previously published in “The Ancient Chinese Art of Chi-Ting”, SIGGRAPH '85 Image Rendering Tricks seminar notes

[Cook84] Robert L. Cook, “Shade trees”, *Computer Graphics (SIGGRAPH '84 Proceedings)* , volume 18(3), July 1984, 223–231

Parses arbitrary simple expressions into a parse tree form which is interpreted for shading. Reasonable speed is obtained though a powerful set of precompiled library functions.

[Coquillart90] Sabine Coquillart, “Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling”, *Computer Graphics (SIGGRAPH '90 Proceedings)* , volume 24, 1990, 187–196

FFD with non-rectangular lattices.

[Coquillart91] Sabine Coquillart and Pierre Jancéne, “Animated free-form deformation: An interactive animation technique”, *Computer Graphics (SIGGRAPH '91 Proceedings)* , volume 25, 1991, 23–26

Animation of objects by either animating an EFFD containing the object or moving the object through the space of an existing EFFD.

[Crow82] F. C. Crow, “A More Flexible Image Generation Environment”, *Computer Graphics (SIGGRAPH '82 Proceedings)* , volume 16(3), July 1982, 9–18

Control process does some sorting and forks off separate processes for each primitive. These are later composited together. Lots of technical problems no longer relevant.

[Dunlavey79] M. R. Dunlavey, “The Procedural Approach to Interactive Design Graphics”, *Computer Graphics* , volume 13(2), March 1979, 110–147

Presents a language based CAD system. The user communicates through the DL language, a simple stack based language with registers. Objects are created by controlling a turtle with a milling tool attached.

[Fleischer87] Kurt Fleischer, “Implementation of a modeling testbed”, *SIGGRAPH '87 Object-Oriented Geometric Modeling and Rendering seminar notes* , volume 14, July 1987

Much of the text of Fleischer and Witkin's Graphics Interface '88 paper with a little more detail

[Fleischer88] K. Fleischer and A. Witkin, “A modeling testbed”, *Proceedings of Graphics Interface '88* , Canadian Inf. Process. Society 1988, 137–137

LISP based system. Transformations generalized to functions for position, normal, and an inverse position transform. Objects generalized to functions for parametric surface position and normal and implicit surface position. Shaders use a function of position, normal, and parametric position to get the appearance parameters for a reasonably standard shader definition. Also a GUI similar to building block shaders.

[Fuchs89] Henry Fuchs and John Poulton and John Eyles and Trey Greer and Jack Goldfeather and David Ellsworth and Steve Molnar and Greg Turk and Brice Tebbs and Laura Israel, “Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories”, *Computer Graphics (SIGGRAPH '89 Proceedings)* , volume 23, 1989, 79–88

Outlines Pixel-Planes 5 hardware and software.

[Glassner91] Andrew S. Glassner, "*Spectrum*: A Proposed Image Synthesis Architecture", *SIGGRAPH '93 Developing Large-scale Graphics Software Toolkits seminar notes* , 1991

Initial proposal for the *spectrum* system described more completely in [Glassner93]. Includes code from initial MESA implementation.

[Glassner93] Andrew S. Glassner, "*Spectrum*: An Architecture for Image Synthesis Research, Education, and Practice", *SIGGRAPH '93 Developing Large-scale Graphics Software Toolkits seminar notes* , 1993

This is probably my closest sibling. It defines a generic architecture for entirely procedural graphics which is being implemented in C++ to be freely distributed. It's designed for radiosity and ray tracing, but the modules scheduling is also programmable so it could probably wash your shirts for you. Normal module types: cameras, samplers, reconstructors, seeders, shaders, shapes.

[Grant86] Eric Grant and Phil Amburn and Turner Whitted, "Exploiting classes in modeling and display software", *IEEE Computer Graphics and Applications* , volume 6(11), November 1986, 13–20

Discusses class hierarchy for their C++ system. Not much of use to me.

[Grant87] Eric Grant, "Class Design for a Modeling Testbed", *SIGGRAPH '87 Object-Oriented Geometric Modeling and Rendering seminar notes* , volume 14, July 1987

Discussion of object-oriented approach in the Grant, Amburn, and Whitted system. Display classes divisions exist for z-buffer, a-buffer, etc. It appears that primitives must either reduce to polygons or use custom code for each.

[Green88] Mark Green and Hanqiu Sun, "MML: A language and system for procedural modeling and motion", *Proceedings of Graphics Interface '88* , June 1988, 16–25

Based on C, MML is just a preprocessor. Can generate rule or grammar based models (particle systems, L-systems, graftals, fractals). Objects have state, code for procedural generation, code for motion verbs, and code for rendering. Their examples render using lower-level primitives.

[Hall83] R. A. Hall and D. P. Greenberg, "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics And Applications* , volume 3, November 1983, 10–20

Ray tracing system concerned primarily with the ability to use new illumination models. Includes spectral distributions instead of just RGB. Brief mention of the typical ray tracing use of functions to intersect with an arbitrary ray. Claims it could use other rendering techniques, but that has to be just for illumination calculations, the rendering code would need to be replaced.

[Hanrahan90] Pat Hanrahan and Jim Lawson, "A Language for Shading and Lighting Calculations", *Computer Graphics (SIGGRAPH '90 Proceedings)* , volume 24, August 1990, 289–298

Introduces and explains RenderMan. Includes some details on compilation issues

[Hedelman84] H. Hedelman, "A Data Flow Approach to Procedural Modeling", *IEEE Computer Graphics and Applications* , volume 3, January 1984, 16–26

Can hook together procedural models, transformations, and shaders into a tree. Discusses parallel execution of nodes of the tree. Also discusses high level smart

culling by procedural models and multiple representations provided by one model procedure.

[Kajiya83] J. T. Kajiya, “New techniques for ray tracing procedurally defined objects”, *ACM Transactions on Graphics* , volume 2(3), July 1983, 161–181

Gives three related ray tracing techniques. Fractals are contained within hierarchical bounding volumes. The sub-volumes (and final primitives) are only computed when a ray intersects the parent volume. For prisms, the ray is projected into the plane of the prism base and intersection proceeds in 2D. The 2D base is contained by a strip tree. For surfaces of revolution, the ray is transformed to a parabola in a squared space and intersection proceeds in 2D as for prisms.

[Kiesewetter80] H. Kiesewetter, “ALGRA, an algebraic-graphic programming language for modeling”, *Eurographics '80* , North-Holland September 1980, 249–254

ALGRA is a follow on to DIGRA 73 – algebraic description of models with some control structures.

[Knoll90] Thomas Knoll, *Filter Module Interface for Adobe Photoshop* , Adobe Photoshop Developers Kit 1990

Technical specification for writing plug-in filters for Photoshop.

[Knoll90] Thomas Knoll, *Writing Plug-in Modules for Adobe Photoshop* , Adobe Photoshop Developers Kit 1990

Overview of the workings of Photoshop plug-ins.

[Kolb92] Craig E. Kolb, *Rayshade User's Guide and Reference Manual* , January 1992

Description of Rayshade ray tracer. Rayshade is designed to be extensible, but new primitives or shading functions require understanding and changing the code. Very little is said in the documentation on how to go about doing either.

[Kuchkuda88] Roman Kuchkuda, “An introduction to ray tracing”, *Theoretical Foundations of Computer Graphics and CAD* , volume F40, Springer-Verlag 1988, 1039–1060

Includes C code for a simple ray tracer. The code is designed to make it simple to add new primitives. Each primitive requires instancing, intersection, and normal calculation functions and a small amount of extra lex/yacc code and support code.

[Lane80] J. Lane and L. Carpenter and T. Whitted and J. Blinn, “Scan line methods for displaying parametrically defined surfaces”, *Communications of the ACM* , volume 23(1), 1980, 23–34

Three algorithms for scan converting spline surfaces. Blinn: track edges and silhouettes with Newton's method. Whitted: approximate edges and silhouettes with cubic curves. Subdivide when not monotonic or not accurate enough. Lane-Carpenter: subdivide to flatness or size limit.

[Lewis89] John-Peter Lewis, “Algorithms for Solid Noise Synthesis”, *Computer Graphics (SIGGRAPH '89 Proceedings)* , volume 23, July 1989, 263–270

An excellent overview of noise functions.

[Max81] N. L. Max, “Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset”, *Computer Graphics (SIGGRAPH '81 Proceedings)* , volume 15(3), August 1981, 317–324

Gives details for creation of “Waves and Islands in the Sunset” animation. Some color table hacks to allow frame reuse. Uses procedural models for the waves and islands.

[Max89] Nelson Max, “Smooth appearance for polygonal surfaces”, *The Visual Computer* , volume 5(3), 3 1989, 160–173

Uses polygon mesh and vertex normals to define a quadratic Beziér triangle mesh. Uses the Beziér mesh to render c1 smooth silhouettes, normals, shadows, and texture coordinates

[Max90] Nelson L. Max, “Cone-Spheres”, *Computer Graphics (SIGGRAPH '90 Proceedings)* , volume 24, 1990, 59–62

Presents a generalized cylinder primitive: two spheres and a section of cone tangent to both connecting them.

[Middleton78] T. Middleton, “A Language for Regular Operations in Graphics”, *Computer Graphics* , volume 11, March 1978, 39–57

Argues that a base language without support for graphics makes writing graphics code unnatural and inconvenient. Presents a 2d system built on ALGOL 68 with a handful of special data types and operations.

[Molnar92] Steven Molnar and John Eyles and John Poulton, “PixelFlow: High-speed rendering using image composition”, *Computer Graphics (SIGGRAPH '92 Proceedings)* , volume 26, 1992, 231–240

Describes the PixelFlow hardware.

[Nadas87] Tom Nadas and Alain Fournier, “GRAPE: An Environment to Build Display Processes”, *Computer Graphics (SIGGRAPH '87 Proceedings)* , volume 21, July 1987, 75–84

A data flow based C testbed system. C functions are hooked together in a directed acyclic graph. Includes excellent overview of other systems and a good set of references.

[Nakamae90] Eihachiro Nakamae and Kazufumi Kaneda and Takashi Okamoto and Tomoyuki Nishita, “A Lighting Model Aiming at Drive Simulators”, *Computer Graphics (SIGGRAPH '90 Proceedings)* , volume 24, 1990, 395–404

Presents a reflection model for wet roads. Also, of more interest to me, presents a 2d model for diffraction of bright light sources.

[Ostby93] Eben F. Ostby, “Implementation of MENV”, *SIGGRAPH '93 Developing Large-scale Graphics Software Toolkits seminar notes* , 1993

Background and implementation details on MENV. Includes some information not found in [Reeves90] on partial updates of models from avar changes.

[Perlin85] Ken Perlin, “An Image Synthesizer”, *Computer Graphics (SIGGRAPH '85 Proceedings)* , volume 19, July 1985, 287–296

He presents a *pixel stream editor*. Essentially expands shade tree work to a full language. This is the earliest example I have of a full language for shading. Includes Perlin noise function.

[Perlin89] Ken Perlin and Eric M. Hoffert, “Hypertexture”, *Computer Graphics (SIGGRAPH '89 Proceedings)* , volume 23, July 1989, 253–262

Procedurally defined solid objects (volume rendered).

[Pixar89] Pixar, *The RenderMan Interface* , September 1989

Technical description of RenderMan, RIB, and the shading language

[Reeves90] William T. Reeves and Eben F. Ostby and Samuel J. Leffler, “The MENV Modeling and Animation Environment”, *Journal of Visualization and Computer Animation* , volume 1(1), August 1990, 33–40

Describes the “MENV” system in use at PIXAR. MENV provides a set of cooperating tools that communicate through shared memory, semaphores, and message passing. All modeling is done using a special purpose modeling language ML. ML consists of C-like statements, calls to geometric primitives, and calls to geometric operations. Includes distinct concepts of variable scoping hierarchy, object hierarchy, and transformation hierarchy. Animation is accomplished through the use of articulated variables (“avars”).

[Rhoades92] John Rhoades and Greg Turk and Andrew Bell and Andrei State and Ulrich Neumann and Amitabh Varshney, “Real-time procedural textures”, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)* , volume 25(2), March 1992, 95–100

Assembler-like interpreted texture language for Pixel-Planes 5

[Rubin80] S. M. Rubin and T. Whitted, “A 3-Dimensional Representation for Fast Rendering of Complex Scenes”, *Computer Graphics* , volume 14(3), July 1980, 110–116

Hierarchical bounding boxes for ray tracing. Ray intersection only done with bounding boxes, surfaces rendered by recursive subdivision. Mentions application to recursive subdivision of procedurally defined surfaces.

[Sabin79] M. A. Sabin and R. A. Guedj and H. Tucker, “Software Interfaces for Graphics”, *Methodology in Computer Graphics* , North-Holland 1979, 49–78

Part of the proceedings of the IFIP Workshop on Methodology in Computer Graphics, Seillac, France, 1976. Overview and comparison of picture description languages, stream protocols, and procedure libraries. Covers languages DRAW, ARTIST, GPDL, G439, PDL2, MONSTER, and DIGRA. The article is followed by a critique by L. Kjelldahl, and notes taken during Sabin and Kjelldahl's presentations and the following panel.

[Trumbore93] Ben Trumbore and Wayne Lyttle and Donald P. Greenberg, “A Testbed for Image Synthesis”, *SIGGRAPH '93 Developing Large-scale Graphics Software Toolkits seminar notes* , 1993

A collection of library routines called by user code to facilitate global illumination research.

[Upstill90] Steve Upstill, *The RenderMan Companion* , Addison-Wesley 1990

User's guide description of the RenderMan scene description interface and shading language.

[Watt92] Alan Watt and Mark Watt, *Advanced Animation and Rendering Techniques: Theory and Practice* , Addison-Wesley Publishing Company 1992

An *excellent* general graphics book including coverage of just about everything. For my purposes, chapters on procedural texture mapping and modeling, shading languages and RenderMan, deformations, procedural animation, shadowing, shading, and parametric surfaces. Wow.

[Whitted81] T. Whitted and D. M. Weimer, “A software test-bed for the development of 3-D raster graphics systems”, *Computer Graphics (SIGGRAPH '81 Proceedings)* , volume 15(3), August 1981, 271–277

They describe a generalized 3d scan line rendering program with support for C coded shaders. This is the earliest example of programmable shading I have. They can interpolate arbitrary parameters.

[Whitted82] T. Whitted and D. M. Weimer, “A Software Testbed for the Development of 3D Raster Graphics Systems”, *ACM Trans. on Graphics (USA)* , volume 1(1), January 1982, 43–57

More detailed version of their SIGGRAPH 81 paper. Of particular personal interest, it has more detail on structure for the primitive half of the testbed. Primitives have a bounding box to determine the span when they are activated and may deposit new primitives for processing in later spans (ideal for subdivision algorithms).

[Wyvill85] Geoff Wyvill and Tosiyasu L. Kunii, “A Functional Model for Constructive Solid Geometry”, *The Visual Computer* , volume 1(1), July 1985, 3–14

CSG system rendered with ray tracing. Primitives are defined with functional definitions. CSG operations are done on an octree representation, and ray tracing using the octree and arbitrary functional definitions. The CSG cells can be full, empty, point to a single object, or “nasty” (at the octree resolution limit with more than one object in the cell).