# 9

# PROOF: An Architecture for Rendering in Object Space

Bengt-Olaf Schneider, and Ute Claussen

This paper gives a short introduction into the field of computer image generation in hardware. It discusses the two main approaches, namely partitioning in image space and in object space. Based on the object space partitioning approach we have defined the PROOF architecture. PROOF is a system that aims at high performance and high quality rendering of raster images. High performance means that up to 30 pictures are generated in one second. The pictures are shaded and anti-aliased, giving the images a high degree of realism. The architecture comprises three stages which are responsible for hidden surface removal, shading, and filtering respectively.

The first of these stages is a pipeline of object processors. Each of these processors stores and scan converts one object. Furthermore, it interpolates the depth and the normal vector across the object. Each object processor is able to handle objects of a certain primitive type. The specialization of an object processor to a certain primitive type is encapsulated in a single block called primitive processor.

The output of the object processor pipeline is the input to a stage for shading. The illumination model employed takes into account both diffuse and specular reflections. The paper reviews Gouraud and Phong shading with regard to their suitability for a hardware implementation.

The final stage of the PROOF system is formed by a stage for filtering the colours of those objects that contribute to a pixel. This is done by constructing a subpixel mask and filtering across an area of $2 \times 2$ pixels.

At the end, the paper briefly reports on the current state of the project.

**Computing Reviews Classification:**

C.1.2 *[Processor Architectures]* : Multiple Data Stream Architectures — *Pipeline processors*
C.3 *[Special-purpose and Application-based Systems]* : — *Real-time systems*
I.3.1 *[Computer Graphics]* : Hardware Architectures — *Raster display devices*
I.3.2 *[Computer Graphics]* : Graphics Systems — *Distributed/network graphics*

**Keywords**: rendering, scan conversion, hidden surface removal, interpolation, shading, illumination models, filtering, anti-aliasing, object processor, object-oriented

## 1. Introduction

There are various applications that demand very fast graphics. They include flight simulators, animation, and process control. The constraint that the pictures must be updated in only a few frame times —in the worst case in only one frame time ($\approx 20\,\text{ms} \cdots 30\,\text{ms}$)— is a great challenge to the graphics hardware. In this time slot the following functions have to be carried out by the graphics system:

- Geometric and perspective transformations, which define how the objects of the scene will appear on the screen. In order to reduce the number of objects to be processed, clipping of the objects takes place.
- Visibility of the objects has to be determined by hidden surface removal calculation.
- The mapping of the continuous scene definition to the raster screen by scan conversion.
- Apply shading and anti-aliasing in order to give a natural appearance to the generated image.

These tasks can be divided into two groups: the geometrical processing (transformations and clipping) and the pixel oriented processing. There are different opinions whether visibility computations and shading should take place in the geometric part or in the pixel oriented part. In most systems that are aiming at fast image generation these parts are allocated to the pixel processing. In this case hidden surface removal is done employing a (distributed) z-buffer, and shading is accomplished with incremental methods like Gouraud or Phong shading.

The paper starts with a short review of approaches to high speed image generation using VLSI. Section 3 describes the overall structure of our approach, the PROOF system. The next sections give details about the single stages in PROOF that are responsible for visibility computations, shading, and anti-aliasing. Each of these sections briefly discusses the algorithms to be employed. The paper concludes with a short report of the current state of our project.

## 2. Approaches to Partitioning the Pixel Processing Step

As a consequence of the large amount of data to be handled the pixel processing is computationally most intensive in the process of image generation. Many work has been done to speed up this part of the image generation. There are at least two approaches to this problem. Both try to distribute the necessary computations on a multiprocessor network [AF86], [GF85].

The first one that has been investigated very thoroughly in the past, tackles the problem by partitioning the screen into smaller areas, in the extreme case into the single pixels. A processor is assigned to each such area. The processors receive the transformed and clipped objects from a geometry processor. Every processor looks whether an object covers the associated pixel and conditionally stores the depth and the colour of that object. The

depth is used to determine the object lying closest to the viewer. This is done by comparing the stored depth value with that of the new object. After all objects have been sent, the portions of the screen that are associated with the processors contain an image of the scene with removed hidden surfaces. This solution is a distributed implementation of the popular z-buffer algorithm [Eyl87], [Dem85].

The second approach distributes the scene's objects amongst several processors. For every pixel, each processor scan converts its objects and computes their depth value. In a second step, the depth values of all the objects at a certain pixel are compared and the closest one is put forward to the screen. This approach constitutes a parallel realization of the z-buffer algorithm, too [CD81], [FR82], [Wei81].

A comparison of these solutions shows that they are complementary. For a given number of processing elements, the image space partitioning is restricted in the number of pixels that can be handled whereas object space partitioning limits the numbers of objects in the scene. On the other hand, the performance (e.g. update rate of the image) of a system employing image space partitioning decreases with an increasing number of objects. Performance of a system partitioned in object space degrades when the screen resolution is raised.

We are exploring the object oriented approach because we think that, in the future, scene complexities will increase so much that a pixel oriented partitioning will not give the desired performance.

PROOF (**P**ipeline for **R**endering in an **O**bject **O**riented **F**ramework) is a system that speeds up the rendering task by handling the objects of the scene independently and in parallel. Because our system forms a distributed z-buffer it will show aliasing deficiencies if no care is taken to avoid them. Therefore, special circuitry is provided to avoid aliasing. In order to generate pictures with a high degree of realism, we have defined the PROOF-architecture such that a Phong-like shading can be incorporated into the system.

## 3. Description of PROOF

### 3.1. System Overview

PROOF consists of three main building blocks (Figure 1). The task of the *geometry processor* consists of all calculations that are required by the following stages to render the image. This comprises the transformations, the clipping, and some shading calculations. The design of the geometry processor is out of the scope of our project. There already exist designs that are capable of computing these data [Cla82], [F*88].

The first stage after the geometry processor is formed by a *pipeline of object processors.* The tasks of the object processor pipeline (henceforward called OPP) is scan conversion and hidden surface removal of the objects in the scene. Furthermore, it provides data that are needed by the following stages for shading and filtering. The OPP is an

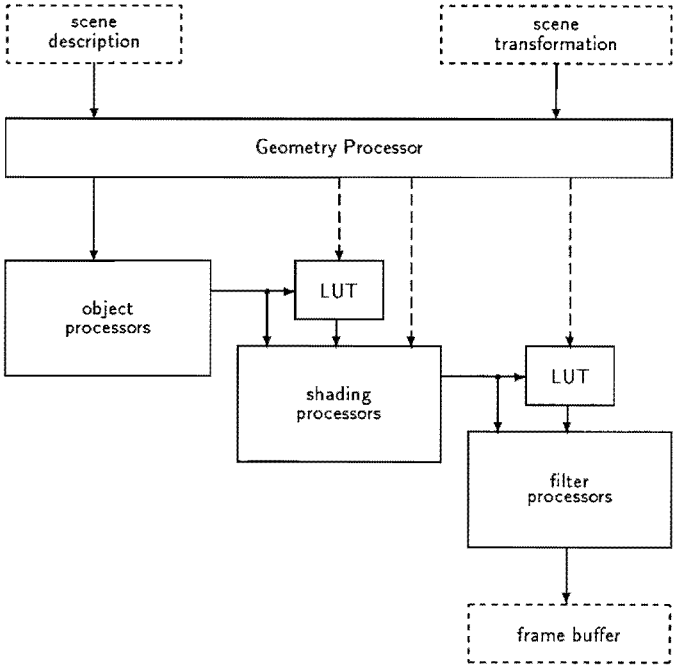


Figure 1: The system architecture of PROOF

implementation of the object oriented solution to computer image generation. Every element of the OPP, called object processor (henceforward referenced as OP), can store one object. The geometry engine is pushing "pixels" into the OPP. The output of the OPP is, for every pixel, a list of objects that are potentially visible at this pixel. The construction of this list is explained in the following section.

The *shading stage* calculates the colour of all objects in this list. The shading algorithm will incorporate ambient light as well as diffuse and specular reflections. For every pixel, the shading stage produces a list of shaded objects.

The final pixel colour is computed by the *filter stage.* The filter takes the list of objects and computes a subpixel mask that indicates which object is actually visible at which subpixel. The colours of the contributing objects are combined by weighted filtering resulting in the final pixel colour.

Between these three stages there are look-up tables (LUT) that provide information necessary for the following stage. The LUTs reduce the amount of data that has to be propagated through the system. If necessary, the LUTs are updated by the geometry processor before the start of a new picture.

The coloured pixels that are produced by the filter stage are written into a frame buffer. Our ultimate goal is real-time generation of pictures, which implies the generation of pixels at video rate. In this case a frame buffer would be obsolete. However, there are several reasons to incorporate a frame buffer. The first versions of our system will probably not work at the intended speed, which will make a frame buffer necessary. The second reason is that there are many algorithms, e.g. in image processing, that work on the basis of pixel images. The frame buffer will allow an interface to such applications. The frame buffer will also serve as a common interface of our system to other graphics systems. We are planning to use low cost graphics hardware for the generation of text, cursor symbols, etc. These graphics systems would write their output directly into the frame buffer.

### 3.2. The Object List

As already mentioned, a list of objects that are potential contributors to a pixel is propagated through the pipeline. The reason for this procedure is to compute more accurately the pixel colour in order to get rid of the aliasing problems. Because these problems stem from the fact that only one object is associated with each pixel, we collect all objects that may influence the pixel's colour. Such objects are characterized by the following properties:

- The object covers the pixel at least partially, that is the pixel is a part of the object or it is intersected by the edge of the object.
- There is no opaque object closer to the viewer that covers the pixel completely.

Objects that have these properties are included in the list such that the list is sorted by depth with the closest object first.

The construction of the list is carried out by the OPP. The compression of the list to one colour takes place in the filter stage.

### 3.3. Communication in the System

PROOF will be composed out of many integrated circuits. High speed communication in such a large system constitutes a big problem because of several effects like wire delay, clock skews, noise, waveform distortion, etc. The most severe of these are wire delay and clock skews. Fortunately, our architecture is mostly organized in a pipelined manner. We took advantage of this fact by stressing the locality of the interconnections in our communication scheme. Although there is no global clock in the system, all processing elements are synchronous circuits. The clock is distributed along the pipeline. This guarantees that neighbours in the pipeline always share the same clock with only little clock skew. This simplifies the design of the interfaces since there are no synchronization problems.

## 4. The Object Processor Pipeline

### 4.1. Task of the OPP

The OPP is responsible for scan conversion and hidden surface removal. This is accomplished by a pipeline of object processors [Sch88]. Each of these object processors store the description of one object and scan converts this object. The depth sorted list of objects for every pixel is constructed in the OPP. The list entries describe the properties of the object at the current pixel. This comprises the object identifier, the depth value, interpolated normal vectors (i.e. normal to the surface of the object), and geometric data describing the position of the object in relation to the pixel centre.

### 4.2. Properties of the OPP

The object-oriented approach exhibits some advantages and some drawbacks.

- In contrast to image space partitioning, the OPP allows to render rather complex scenes with only modest degradation in performance.
- The OPP is capable of generating pictures in real-time (30 frames per second) at medium screen resolutions (512×512).
- In its principle architecture the number of objects in the scene is limited by the number of OPs.

The architecture offers considerable potential for overcoming this last limitation. We are currently investigating several ways that lead to a graceful degradation in performance if the number of objects exceeds the number of OPs. One could be to establish a feedback of the output of the OPP to its input. Another alternative is to utilize a single OP several times per frame by objects that do not overlap in the y-coordinate.

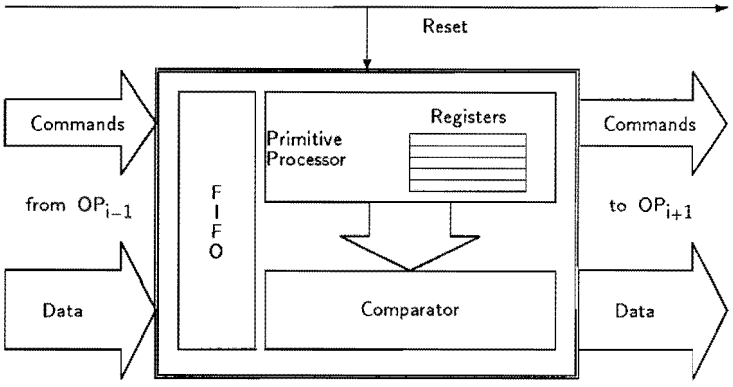


Figure 2: Block diagram of an object processor

### 4.3. Description of the Object Processor

The elements of the OPP are the object processors. They store and scan convert objects of a certain type of primitive. At the present time, the OPs can only handle triangles and vectors. However, the architecture of the OP can easily be changed to process primitives of another type. This has been achieved by encapsulating the parts of the OP that are dependent on the primitive type. This part is called the *primitive processor.* The rest of the OP is not influenced by the actual primitive type. This enables us to design a family of object processors that only differ in a specially designed primitive processor.

The internal structure of the OP consists of two main blocks (primitive processor and comparator) and some auxiliary functions (FIFO and communication) (Figure 2). The primitive processor contains registers that store the parameters which describe the stored object. These parameters describe the geometry of the object (e.g. vertices, control points, etc.), the degree of coverage of the pixel (completely, partially, not at all), and the normal vectors to the surface. The contents of these registers is used to scan convert the object and to linearly interpolate depth value and normal vector across the object.

The output of the primitive processor, i.e. object identifier, depth value, and normal vector are sent forward to the *comparator.* Here, a comparison of the depth value of the stored object and that of the incoming object takes place. The object being closer to the viewer is selected for the propagation to the next OP. The degree of overlapping of the current pixel and the object sent rules how the retained object is processed further. If the object sent covers the pixel partially the object held back will be sent in the next time-slot, because this object may still contribute to the current pixel. If the object sent covers the pixel totally the object kept is discarded. In case that it is the incoming object that is discarded implies that the objects in the rest of the list of a particular pixel can be discarded as well. This procedure guarantees that all lists in the OPP are constructed according to the rules defined above.

Obviously, insertion/removal of objects to/from the list changes the length of the list. This causes jams (in case of insertion of an object) and holes (in case of removal of objects) in the data flow. In order to absorb these variations in the rate of data flow we have included a FIFO in our architecture. It serves as a buffer for incoming data if the stored object is inserted; in case of the deletion of a list tail it supplies data quickly, thus implementing a kind of cache memory. The length of the FIFO depends lightly on the statistical properties of the scene to be rendered and the kind of distribution of the objects along the pipeline. Simulations indicate that a FIFO that can store the description of only one object is sufficient (increasing the length of the FIFO will result in performance gains less than 2 %).

### 4.4. An Object Processor for Triangles

It has been said before that the concept of the object processor is suited for a large variety of primitives types. At present, we are designing an OP with a primitive processor for triangles and vectors (in fact, vectors are described like triangles, but only one edge is displayed). The triangles are described by the bounding lines. These lines are given in the normal form

$$l : x\cos\alpha + y\sin\alpha - p = 0$$

The distance d of a pixel at $(x_0, y_0)$ and l is given by:

$$d(x_0, y_0) = x_0\cos\alpha + y_0\sin\alpha - p \tag{1}$$

The distance d can be easily computed incrementally from pixel to pixel and from scanline to scanline [CD81].

### 4.5. Coverage Calculations

To enable the comparator to manage the lists correctly it is necessary that the primitive processor determines how much the current pixel is covered. This decision is ternary: An object may cover a pixel *not at all, partially,* or *completely.* The pixel is covered partially if an edge of the object is intersecting the pixel. Although this seems to be trivial, it is not simple to decide whether an object covers the pixel partially or not. This is due to several items:

- Most important is the way the object is described. The kind of description may be well suited for scan conversion and interpolation (as it is with the outlined description of the triangles) but it may complicate the classification of the coverage.
- Usually, all computations are related to pixel centres. Thus, coverage normally means, the pixel centre is covered. Unfortunately, we are not concerned with the pixel centre but with the boundary of the pixel.
- To allow a reasonable anti-aliasing the objects have to be defined with subpixel precision. This also implies that coordinates, e.g. object vertices, might be located on subpixel centers and not on pixel centers. This gives rise to some problems because the OPs are calculating on pixel coordinates.
- Rounding errors may result in wrong coverage decisions. This kind of errors can be eliminated on the expense of some extra bits in the description of the objects at the cost of more hardware and an increase of time for computations.

The concrete example of the primitive processor for triangles might illustrate this. Using the description of the triangle as outlined above it is very simple to decide whether the pixel centre is inside the triangle or not. This is established by testing if, for all edges i, the distances $d_i$ have the same sign. It is far more difficult to decide whether any part of the pixel is covered by the object. We have tried several solutions, none of them being entirely satisfying. One way is to let the triangles grow by half the diameter of a pixel. This new

triangle has the property that the pixel centre of any pixel that is intersected by the original triangle is located inside the grown triangle. Thus, the test for "pixel area inside a triangle" has been transformed to the simpler test for "pixel centre inside a triangle". The drawback of this procedure is that in case of very acute angles many pixels are classified as partially covered although they are uncovered.

This leads to the issue of the behaviour of the coverage algorithm: A coverage algorithm is said to be *good-natured* if all pixels that are covered partially by the object are classified as such. A good-natured algorithm might classify pixels as partially covered although they are covered completely or not at all. Such algorithms are called good-natured because they do not miss any object that should be included in the list of objects. Objects that have been included into the list unnecessarily will be invisible in the picture because of the filtering step at the end.

In contrary, an algorithm is *malicious* if it does not detect all pixels that are covered partially. This behaviour is unwanted because this would result in a loss of information — either about that object, if the pixel has been classified as uncovered, or about the objects lying behind that object, if the object has been (incorrectly) identified as completely covering the pixel.

What we are searching are, of course, good-natured algorithms. Currently, our investigations are aiming for two goals. The first is the development of algorithms that are good-natured but exhibit only little erroneous behaviour. This is done by refining the algorithm outlined above (grown triangle). In parallel to this, we are estimating the loss of performance that is caused by such algorithms. We are, therefore, studying various pictures according to their common statistical properties. On the basis of these studies we will decide how much extra effort is reasonable for the improvement of the algorithm.

## 5. The Shading Stage

Shading methods consist of two parts: the computation of an *illumination model,* which determines the radiant intensity at a given point, and the *shading algorithm,* which gives us an instruction where to use which illumination model with which parameter values. The task of the shading stage is to give a certain pixel an adequate colour, that is: to compute the illumination model.

As the shading algorithm is strongly connected to the computations that are performed by the object processor pipeline, we will now present the options to integrate a shading stage into the PROOF system architecture. Afterwards, we will discuss some approaches to the computation of an illumination model, assuming that we want to integrate a Phong algorithm.

### 5.1. Shading Algorithms and their Impact on the Architecture

For obvious reasons, we have to restrict ourselves to incremental, interpolating shading algorithms: The database consists of objects that approximate the true geometry of the objects, thus, to obtain a smooth appearance, interpolation schemes have to be used. Our ultimate goal of PROOF is to build a real time image generation system, due to which we have to consider algorithms with low computational complexity. Algorithms of that type have been introduced in literature by Gouraud [Gou71] and Phong [Pho75]. A modification of Phong's algorithm was presented by Bishop and Weimer [BW86]. We want to examine these three approaches and see, how they would fit into the architecture.

**Gouraud algorithm.** The interpolation schemes as performed by the object processors can be used to *interpolate colour values.* The colour values at the vertices of a polygon have to be computed before the interpolation takes place. As a consequence, we have to integrate the shading stage between geometry engine and object processor pipeline. In fact, this could also mean that the shading calculations are done by the geometry engine.

**Phong algorithm.** Exploiting the interpolating capabilities of the object processors, *interpolation of normals* can be handled, too. These normal vectors are input for the illumination model. This results in locating the shading stage between object processor pipeline and filter stage.

**Fast Phong algorithm.** The approach of fast Phong shading is an approximation of Phong's algorithm and an illumination model by developing the formula into a Taylor series. The resulting scheme —for simple illumination models— is a *quadratic function* in x and y. As the illumination model is already included, no separate shading stage has to be implemented. All computations can take place within the object processors.

Several pros and cons can be stated for the approaches examined before:

- It is common sense that for most cases, Phong shading is better than Gouraud shading considering the visual effects.
- The evaluation of a quadratic function in the object processors is not desirable for the reasons first that it increases computation times and hence decreases the performance and second that it doubles the load time for the parameters.
- In the case of Gouraud shading, the computation of the illumination model has to be performed once for each vertex of the scene. In contrast, for Phong shading, this computation has to be done approximately the number of pixels times the average number of objects in a list. This results in much more parallel shading substages.

Our conclusion is that Phong shading is desirable but that Gouraud shading fits better into the architecture proposed. In the near future, development of the illumination model will continue to reduce computational costs. Therefore, we will now discuss the impact of illumination models.

### 5.2. Illumination Models and their Impact on the Architecture

The complete illumination model we want to consider, includes ambient light, diffuse and specular reflection.

$$I = k_d \cdot I_a + \sum_{l=1}^{L} [k_d \cdot I_l \cdot (\vec{N} \cdot \vec{L}) + k_s \cdot I_l \cdot (\vec{N} \cdot \vec{H})^m]$$

$k_d$ and $k_s$ denote the diffuse and the specular reflectance, respectively, $m$ is the glossiness of the object and $I_a$ and $I_l$ are the ambient and point light source intensities. The normal to the surface $\vec{N}$ and the direction of the light source $\vec{L}$, together with the highlight vector $\vec{H}$ form the rest of the parameters.

We want to look at the computation of I, assuming that we implement a Phong algorithm and integrate the shading stage between OPP and filter stage. This assumption leads to the fact that we cannot perform computations incrementally. For every pixel and every object in the pixel list, the equation for I has to be evaluated independently. This independence can be an advantage, too: parallel computations are easily feasible.

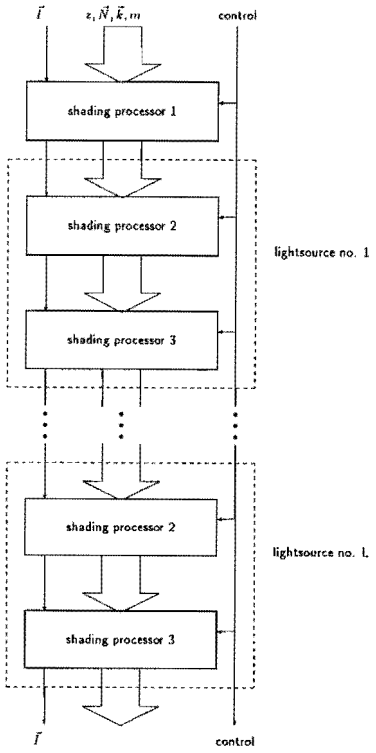


Figure 3: One pipeline out of the shading stage.

**5.2.1. Shading Stage Design.** As $\vec{L}$ and $\vec{H}$ change for every light source, the single terms of the sum can be computed independently. It is reasonable to have a pipeline of light source stages, so that the overall sum can be broken up into sums of two operands (see Figure 3). The initial value, which is fed into the pipeline, is the ambient term. A single part of that pipeline is called light source processor. Investigations into possible parallel and/or pipeline arrangements of these processors have been made before [Cla88a]. They indicate that —depending on the computing times— up to 100 of such processors would be needed. This demonstrates clearly the need for a time reduction in a single light source processor by restricting e.g. the illumination model.

There will be three types of light source processors, one for the computation of the ambient term, one for the computation of the diffuse term of a specific light source, and one for the computation of the specular term.

As the timing constraints are such that the pixel rate, given by the OPP is valid for the output of this stage, too, only one pipeline will not be sufficient. Because the computations can be performed independently, a simple round robin arbitration of the incoming data onto the shading pipelines is a possible arrangement. Only the control signals have to be propagated to all light source processors. Another round robin unit will sort the signals for sending them to the filter stage in the correct order.

**5.2.2. Design of a Shading Processor.** The task of a single light source processor is to compute a colour value for a particular pixel, a particular object, and a specific light source, that means to take the parameters $k_d$ , $k_s$ , $x$ , $y$ , $z$ , and $\vec{N}$ coming from the OPP or a look-up table, respectively, and to compute a part of the illumination model.

For the sake of the generality of our processor design, we have chosen to develop only one processor that can compute the formula :

$$\vec{I}_{t+1} = \vec{k}\cdot\vec{I}\cdot(\vec{N}\cdot\vec{B})^m + \vec{I}_t$$

This processor can have three forms, depending on its position in the pipeline (see Figure 3). A first design is presented in Figure 4. Optional parts, for which it is not yet decided if they are necessary, are shown with dashed lines.

At first sight, it seems to be attractive to implement a light source processor as a look-up table. Unfortunately, this look-up table would have to have a 216 bit entry and a word length of 24 bit, which is practically not implementable. But look-up tables still remain attractive for the implementation of parts of the formula. For example, $x^y$ can be realized using this technique.

Further investigations showed that the normalization of the vectors $\vec{N}$ , $\vec{L}$ and $\vec{H}$ are cost-intensive. Restricting ourselves to directional instead of positional light sources leads to constant values for $\vec{L}$ and $\vec{H}$. If we can decide for this restriction, the normalization of $\vec{N}$ remains in the critical path of the computation. Experiments lead us to the hypothesis that *the normalization* of the normal *can hopefully be omitted,* and hence, the computational

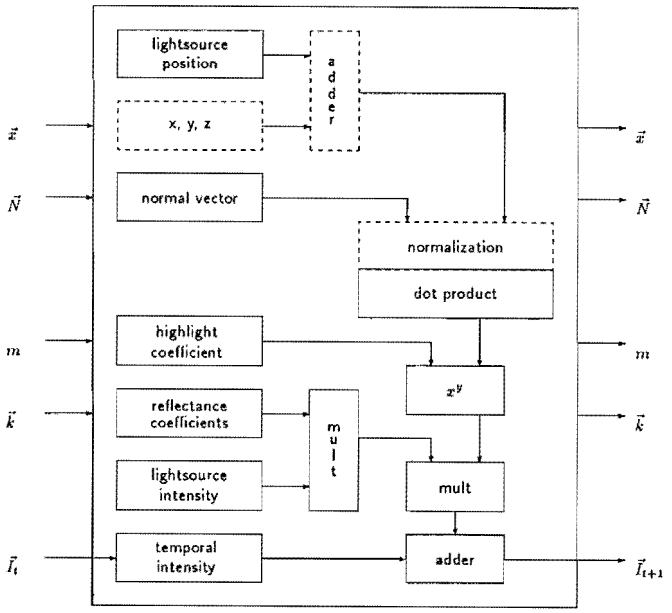


Figure 4: Basic design of a shading processor.

intensive calculations would be reduced. Some pictures illustrating this hypothesis are shown in colour plates included in the appendix.

The next step evaluating this hypothesis is to determine constraints that have to be taken into account during the triangulation. These steps consist of visual and mathematical investigations of the formulas. The results will enable us to decide, if the reduction of the illumination model is feasible.

## 6. The Filter Stage

In order to avoid aliasing errors, filtering of the colours of the contributing objects is performed in the final stage of the PROOF system. Our filter algorithms are based on a subpixel grid with a subpixel resolution of $8 \times 8$ subpixels. The filter area will be $2 \times 2$ pixels. The filter algorithm is able to process both opaque and transparent objects.

### 6.1. Task of a Filter Processor

The filter stage is built from processing elements each of which is responsible for one subscanline. The processing elements determine the subpixel mask, i.e. the array that states which object is visible at a certain subpixel. The construction of the subpixel mask starts with the first object in the list and determines which subpixels are covered. These subpixels are tagged. Then, the next object is examined. In the end, the subpixel mask indicates, for every subpixel, the visible object. The determination of the subpixels

covered by an object is similar to the coverage algorithm used in the OPs: For every subpixel, a decision is made whether the subpixel (centre) lies inside the object.

In order to speed up the calculation of the subpixel mask, the filter processors take advantage of results from the OPP [Rom88]. The OPP delivers at its output the distances d of the centre of the current pixel and the edges describing an object. Starting from these values, the filter processors calculate the distances from the edges to the subpixel centres. This is done by using that

$$d_{sp}(x_i, y_i) = (8x + x_i)\cos\alpha + (8y + y_i)\sin\alpha - p$$

where $d_{sp}$ is the distance from the edge to a subpixel with the offset $(x_i, y_i)$ from the pixel centre. The formula assumes a grid of $8 \times 8$ subpixels and that the pixel centre is at $(x,y)$. Using (1) this formula can be simplified to

$$d_{sp}(x_i, y_i) = d(x, y) + x_i \cdot \cos\alpha + y_i \cdot \sin\alpha$$

This formula shows that $d_{sp}$ can be obtained from d by adding the appropriate multiples of $\sin\alpha$ and $\cos\alpha$. The parallel calculation of these multiples can be advantageously mapped on a tree structure. The tree is constructed from nodes with two inputs and two outputs. The inputs are called the main input and the auxiliary input. The tree structure is formed by the outputs and the main inputs, i.e. the left output of a node is connected to the main input of its left child node and the right output to the main input of its right child node. The left output of a node takes the value of the main input. The value of the right output is the sum of the two inputs. The value at the second, auxiliary input of the node depends on the level of the node in the tree. If we assume that the leaves of the tree form the level 0 of the tree the auxiliary inputs are supplied with the value $2^L \cdot \cos\alpha$ or $2^L \cdot \sin\alpha$ respectively (here L is the level of the node in the tree). The multiples to be calculated are the values at the leaves of the tree. Using this tree structure, two trees and some negators are required to compute the distances of the edges to the subpixels.

After the subpixel mask has been determined the colours of the contributing objects are combined. This is established using a filter function. We are currently investigating different filter functions. With regard to a hardware implementation we will probably choose a filter function that can be easily expressed in terms of powers of two of the contributing subpixel colours.

### 6.2. Filter Stage Design

Designing the filter stage demands exactly the same reflections as the design of the shading stage. Those considerations have been published elsewhere [Cla88b]. They resulted in an amount of at least ten up to hundred filter processors that will be needed, depending of the computing time needed by a single processor. These processors can easily perform their work in parallel, controlled by a simple round robin arbitration. Hence, the goal of the filter stage design has to be a reduction of time for the single processor, too.

## 7. Current Work

It became already clear in the previous chapters and sections that the stages of PROOF are defined at different levels of detail. Of all stages, the development of the OPP is the most advanced. We are currently finishing an architectural simulation of the OP. A gate level design of the OP will start soon.

In contrast to the OPP, we have not yet finished the algorithmic research for the shading stage. Different algorithmic alternatives are currently investigated for their suitability for both a hardware implementation and their adaptability to the PROOF architecture. The study of algorithms will be followed by the design and simulation of the architecture of a shading processor.

At the present, behavioural and architectural simulations are carried out for the filter stage.

## 8. Conclusion

We presented the system architecture of PROOF. PROOF is a system that aims at high speed image generation. It performs all tasks of the pixel processing part of the computer image generation. This includes scan conversion, hidden surface removal, shading, and anti-aliasing. PROOF stands for **P**ipeline in an **O**bject **O**riented **F**ramework which indicates that the architecture divides the rendering process in object space. This results in the object processor pipeline that consists of many processing elements each of them being responsible for one object. Pixels are pushed through the pipeline. A list of objects that are potential contributors to the pixel colour are associated with each pixel.

A stage for shading the objects in these lists follows after the object processor pipeline. It is planned to implement a Phong-like shading algorithm together with the modeling of diffuse and specular reflections in order to produce pictures with a high degree of realism.

The list of shaded objects is fed into a filtering stage. The colours of all objects in the list are combined in order to produce the final pixel colour. This combination is done on the basis of a subpixel grid. The single colours are weighted in accordance to the number of subpixels they cover.

## References

[AF86] Gregory D. Abram, and Henry Fuchs, "VLSI Architectures for Computer Graphics", In G. Enderle, editor, *Advances in Computer Graphics I*, pp 6-21, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1986.

[BW86] G. Bishop and D.M. Weimer, "Fast Phong Shading", *ACM Computer Graphics, vol. 20, no. 4, pp 103-106, August 1986.*

[C*87] R. L. Cook et al., "The Reyes Image Rendering Architecture", *ACM Computer Graphics*, vol. 21, no. 4, pp 95-102, July 1987.

[CD81] D. Cohen, and S. Demetrescu,"A VLSI approach to Computer Image Generation." Technical Report, Information Sciences Institute, University of Southern California, 1981. Presentation at SIGGRAPH 1980.

[Cla82] James H. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics", *Computer Graphics*, vol. 16, no. 3, pp 127-133, July 1982.

[Cla88a] Ute Claussen, "Parallel Scan Conversion", In M. Cosnard et al., editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, IFIP, Elsevier Science Publishers (North-Holland), Amsterdam, 1988.

[Cla88b] Ute Claussen, "Parallel Subpixel Scanconversion", In A. A. M. Kuijk, and W. Strasser, editors, *Advances in Computer Graphics Hardware II, EurographicSeminars*, pp 155-166, Springer, Berlin, Heidelberg, New York, Tokyo, 1988.

[Cro87] F. Crow, "The Origins of the Teapot", *IEEE Computer Graphics and Applications*, pp 8-17, January 1987.

[Dem85] Stefan Demetrescu, "High Speed Image Rasterization Using Scan Line Access Memories", In Henry Fuchs, editor, *Proceedings of the Chapel Hill Conference on VLSI*, pp 221-243, Computer Science Press Inc., 1803 Research Blvd., Rockville, Maryland 20850, 1985.

[Duf79] T. Duff, "Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays", *ACM Computer Graphics*, vol. 13, no. 2, pp 270-275, 1979.

[Eyl87] J. Eyles, "Pixel Planes Project", In A. A. M. Kuijk, and W. Strasser, editors, *Advances in Computer Graphics Hardware II, EurographicSeminars*, pp 183-207, Springer, Berlin, Heidelberg, New York, Tokyo, 1988.

[F*88] H.R. Finch et al., "A Multiple Application Graphics Integrated Circuit", In A. A. M. Kuijk, and W. Strasser, editors, *Advances in Computer Graphics Hardware II, EurographicSeminars*, pp 81-92, Springer, Berlin, Heidelberg, New York, Tokyo, 1988.

[FR82] Donald Fussell, and Bharat Deep Rathi,"A VLSI-Oriented Architecture for Real-Time Raster Display of Shaded Polygons", In *Proceedings of Graphics Interface '82*, pp 373-380, The National Research Council of Canada, 1982.

[GF85] Andrew Glassner, and Henry Fuchs, "Hardware Enhancements for Raster Graphics", In Rae A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, pp 631-658, NATO ASI, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[Gou71] H. Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Transactions on Computers*, vol. C-20, no. 6, pp 623-628, June 1971.

[Pho75] B. T. Phong, "Illumination for Computer Generated Images", *Communications of the ACM*, vol. 18, no. 6, pp 311-317, June 1975.

[Rom88] Claudia Romanova, "Effektives Anti-Aliasing für die Bilderzeugung auf Rastersichtgeräten", In *Proceedings of Fachgespräch Visualisierungstechniken und Algorithmen*, Gesellschaft für Informatik, Osterreichische Computer Gesellschaft, 1988. To appear in Informatik-Fachberichte, Springer-Verlag.

[Sch88] Bengt-Olaf Schneider, "A Processor for an Object-Oriented Rendering System" In *Computer Graphics Forum*, vol. 7, no. 4, pp 301-309, December 1988.

[Wei81] Richard Weinberg, "Parallel Processing Image Synthesis and Anti-Aliasing", *Computer Graphics*, vol. 15, no. 3, pp 55-62, August 1981.

# Appendix

This appendix presents some figures, on which we base our assumption, that in some cases, normalization of the normal vectors can be omitted. As an example, we have chosen the teapot from [Cro87], because it is often used to demonstrate shading methods. Figure 5 shows a coarsely triangulated pot, rendered with a Phong shading algorithm, using the normalized normal vectors. In contrast, normalization has been omitted in the next figure. The second teapot seems to be "textured" in the regions where it is highlighted. Here, the triangulation is "visible". This is due to the fact, that using the unnormalized normal vector results exactly in the same effects as with a Gouraud shading [Duf79]. What Duff suppressed is the fact that this is only true for the diffuse components, but *not* for the specular component.

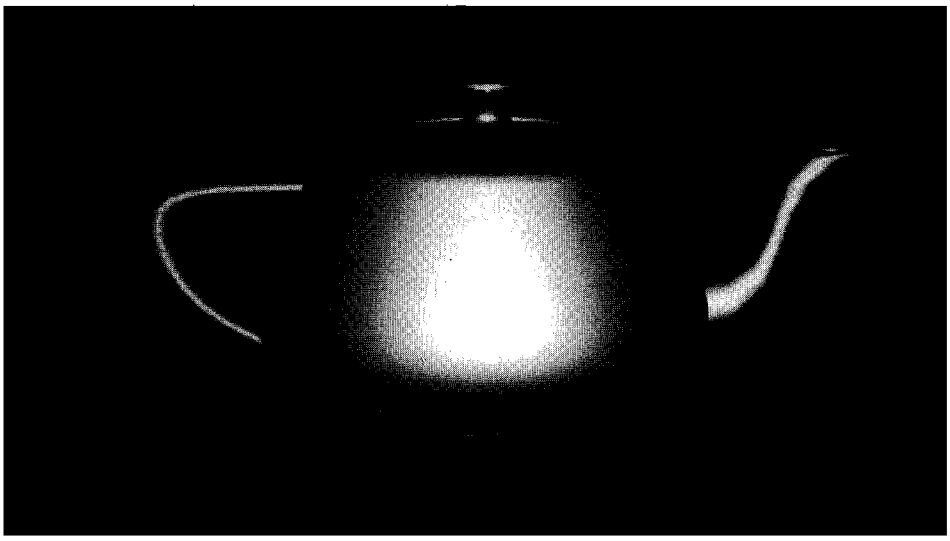

**Figure 5:** "Coarse" teapot rendered with Phong shading using the normalized normal vectors.

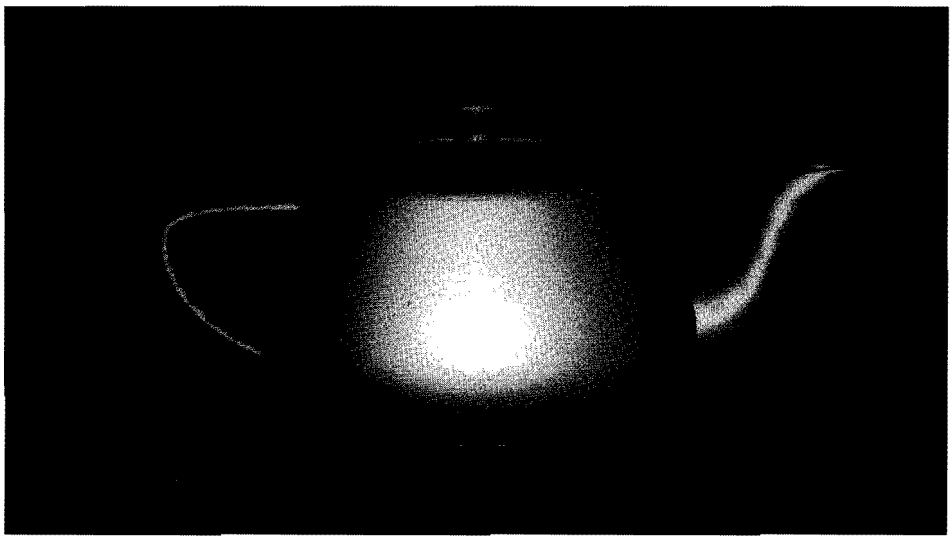

Figure 6: "Coarse" teapot rendered with Phong shading using the unnormalized normal vectors.

The next two figures are rendered from a teapot database with a higher granularity. As can be seen, both figures where normalized normal vectors are used, do not differ very much from each other. But, on the other hand, the last figure, showing the finer teapot rendered using the unnormalized normal vector, is only "textured" at some critical points, where the slopes of the normal vectors of adjacent triangles differ very much. This is due to the fact that a little difference in the angle between normal vectors will be raised to a power and thus amplified. Hence, we can omit normalization, if the diffuse component predominates, and if differences between adjacent normal vectors are small.

Hopefully, this could lead us to an adaptive triangulation of the scene. Or, in consequence, these ideas result in a triangulation of such fine parts as the micropolygons, that only have to be flat shaded [C*87].

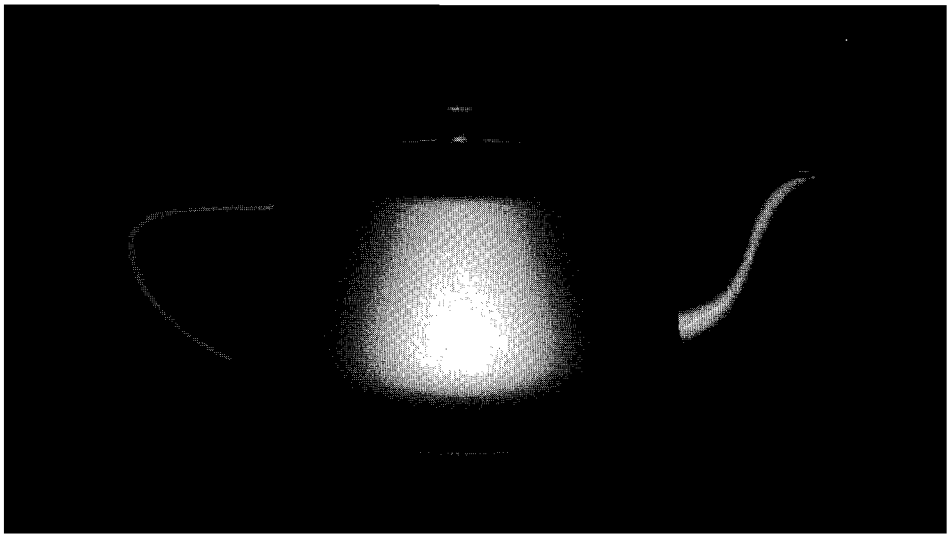

Figure 7: "Fine" teapot rendered with Phong shading using the normalized normal vectors.

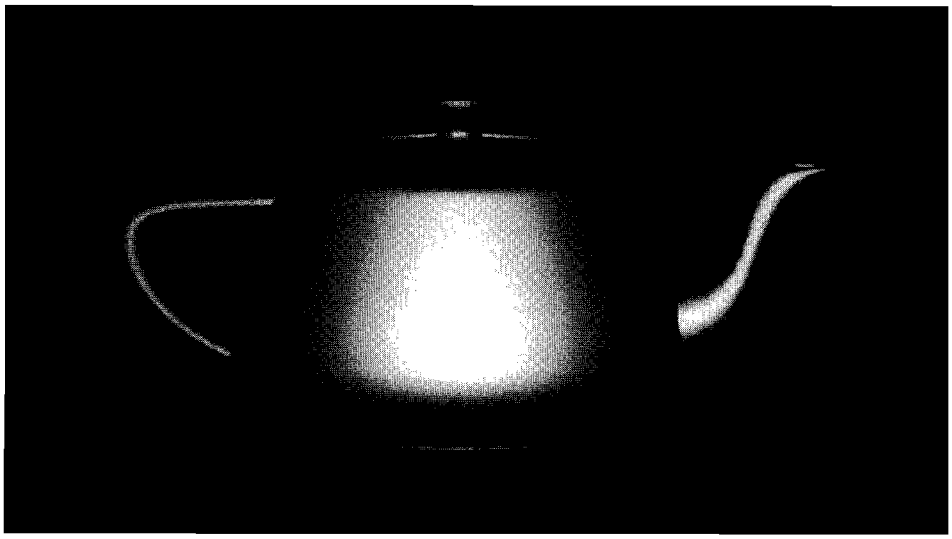

Figure 8: "Fine" teapot rendered with Phong shading using the unnormalized normal vectors.