Chapter 5

PixelFlow Shading

Marc Olano

OpenGL Extensions and Restrictions for PixelFlow

Jon Leech
University of North Carolina

April 20, 1998

Abstract

This document describes the extensions to OpenGL supported by the PixelFlow API, restrictions forced by the architecture, and as-yet unimplemented features.

READERS OUTSIDE UNC-CH AND HEWLETT-PACKARD PLEASE NOTE: This is an internal working document. The final implementation may differ substantially.

Contents

List of Tables

1 Introduction

This document describes the *PxGL* graphics API for the UNC/Hewlett-Packard *PixelFlow* [3] architecture. PxGL is based on the OpenGL [1] API with extensions, restrictions, and unimplemented features[1]. Only material which differs between PxGL and a conformant OpenGL implementation is covered; readers are expected to be conversant with OpenGL proper.

PixelFlow has enormous flexibility because almost all stages of the graphics pipeline - transformation, rasterization, and shading - are implemented with user-programmable hardware. In order to exploit this capability in the framework of a traditional graphics API, we have extended OpenGL to specify

- When to **load** and **invoke** application-defined code (rather than built-in functionality, such as rendering lit, Gouraud-shaded triangles).
- Which **stage** of the pipeline to invoke it at.
- What **parameters** to pass when the code is executed.

To optimize performance of OpenGL code on PixelFlow, some architectural details of the machine are exposed to the API. Using these features may relax some OpenGL guarantees or invariants in return for greatly improved performance. They include

- **Primitive and state distribution**, which balances rendering load across the parallel geometry processors while affecting the order in which primitives are composited into the frame buffer.
- **Display list optimization**, which increases performance of upper stages of the pipeline while relaxing knowledge of global state.

While PixelFlow has far more flexibility in most respects than more traditional graphics accelerators, it also has certain constraints not present in those machines. Most notably, the nature of the image-composition architecture forces a *frame oriented* paradigm on the API, and implies that there is no valid frame buffer containing pixel colors until after rasterization and shading of all primitives in that frame is complete. PixelFlow also uses a *deferred shading* model, in which pixel color is not computed until after visibility determination. The consequences of these and other minor architectural and design decisions are that

- Additional, non-standard OpenGL calls are required to delimit the start and end of frame generation.
- Much of the global rendering state (textures, lights, view matrices, and other state which is not associated to individual primitives) must be defined prior to start of frame and may not change within the frame.
- Many API calls are only allowed at specific points in the process of generating a frame.

[1] PixelFlow will support a fully conformant OpenGL API, but in general that mode will not be used because of its expected substantial performance cost.

- Most types of blending and stenciling are not supported, and composition order of primitives is not guaranteed.
- Access to the frame buffer may only take place after end of frame.

Finally, many features of the rich OpenGL API are not implemented in PxGL at this time, though they may be added later.

1.1 Roadmap

The remainder of this document will address the following areas:

- Frame generation (§2).
- Controlling primitive and state distribution (§3).
- Extending the OpenGL namespace (§4).
- Loading application-defined code (§5).
- Programmable rasterization (§6).
- Programmable shading (§7).
- Programmable lighting (§8).
- Used-defined functions (§??).
- Other programmable pipeline stages (§9).
- Transparency and shadows effects (§10).
- Display list optimization (§11).
- Multiple application threads (§12).
- OpenGL variances (§13).
- Unsupported OpenGL features (§14).

1.2 Change Log

This is revision *Revision* : 1.9 of *Source* : */tmp*$_m$*nt/net/hydra/pp0/doc/software/opengl/tex/RCS/pxgl.tex*, *v*. Changes from the next most recent revision are delimited by change bars (or approximations thereof in the HTML version).

Changes in revision 1.9 (July 22, 1997):

- Changed all uses of **glInquireParameterEXT()** to **glGetMaterialParameterNameEXT()** or **glGetRastParameterNameEXT()** as appropriate.

- Note that **glGetLightParameterNameEXT()** and other stage-specific inquiry functions will need to be documented and created.
- Added to section on primitive and state distribution, including **pxDistributionMode()** and **glGenDataEXT()**.
- Added section on user-defined functions.

Changes in revision 1.8 (August 1, 1996):

- Changed references from Division to Hewlett-Packard to reflect PFX sale to HP.
- Added new inquiry calls for rasterizer and shader parameters (though details remain to be documented).
- Rearranged glossary entries in section 7 to group parameter terminology together, at Rich Holloway's suggestion.
- Added section on transparency and blending effects, including **glTransparencyEXT()**.

Changes in revision 1.7 (March 22, 1996):

- **glShaderEXT()** now allows different shaders on front and back faces of primitives.
- Added discussion to **glSurfaceEXT()** definition of the restriction of a single value for uniform and nonvarying parameters, regardless of whether the front or back face of a primitive is being rasterized.
- Added discussion to **glMaterialVaryingEXT()** definition of the reason for the apparently-redundant *shaderid* argument.
- Added **glLightModelEXT()** to lighting chapter, specifying that user-defined shader parameters are handled in the same way as OpenGL material parameters.

Changes in revision 1.6 (February 12, 1996):

- First version released to outside readers; added disclaimers.
- Removed definitions of hardware-specific terms like composition/geometry network parameters, and changed definitions of varying/nonvarying/uniform parameters to eliminate dependence on those terms.
- Added *face* argument to **glSurfaceEXT()**.

Changes in revision 1.5 (December 17, 1995):

- Added calls for light groups and loadable light functions.
- Removed **glGenShaderEXT()** and folded its functionality into **glNewShaderEXT()**.

- Added sections (though little text yet) for atmospheric and image warping shader stages.
- Changed **glSurfaceParamEXT()** to **glRastParamEXT()** to avoid too-close similarity to **glSurfaceEXT()**.
- Updated to reflect separate-namespace model for parameters and separation of instance and current values. In particular, **glBindParameterEXT()** has been replaced by **glSurfaceEXT()**, although the name of the latter may change.
- Rewrote interpolator introduction.

Changes in revision 1.4 (November 14, 1995):

- Moved document from LaTeX 2.09 to LaTeX 2_ε.
- Added changebars using changebar.sty.

Changes in revision 1.3 (November 11, 1995):

- Added flat interpolator for per-primitive constant parameters.
- Added **glBindParameterEXT()** and **glGetParameterEXT()**.
- **glShaderEXT()** now takes a face argument. Added `GL_FRONT_SHADER_EXT` and `GL_BACK_SHADER_EXT` as targets to **glGet()**.
- Worked on definitions of composition network and geometry network parameters; more work is needed.

2 Frame Generation

The underlying hardware model in OpenGL is that primitives are specified by the application and immediately drawn - vertices are transformed and lit, rasterization and texturing are done, and final pixel colors are copied into the frame buffer, or blended with existing frame buffer contents. Global parameters affecting transformation, rasterization, and shading of primitives, such as the projection matrix, light bindings, blending modes, and so on, may be changed at any time.

This model is not compatible with PixelFlow's image composition and deferred shading paradigms. In order to achieve good performance on the machine, the API must be *frame-oriented*; that is, it must specify several *stages* in the process of generating a frame, and different types of OpenGL operations may occur only during specific stages. The stages and the types of calls that may take place during them are:

- **Frame setup** - establish viewing, lighting, and shading parameters that will apply throughout the frame.
- **Geometry definition** - traverse the database, rasterizing primitives.
- **End of frame** - perform image composition, shade pixels, and read/write directly to the frame buffer.

2.1 Frame Setup

The setup stage begins by calling **glBeginFrameEXT()**. In this stage, parameters which globally affect the scene are defined. This includes defining the projection matrix, loading light functions, creating lights and light groups, changing light source parameters, loading shader functions, creating shaders, changing nonvarying shader parameters, loading rasterizer functions, binding textures, and any other operations that must be known before primitives can be rasterized and shaded (a complete list of OpenGL calls and the stages they may be called for is in section 13). Parameters of the scene such as the viewport size, antialiasing kernel, and background color are also set here; these must be known to define the *rendering recipe*.

PxGL currently allows only a single projection matrix to be used during a frame. Many lighting environments may be used, but they must be defined as *light groups*. Many textures may be used, but they must be defined during frame setup using the *texture object* calls[2].

2.2 Geometry Definition

The geometry stage begins by calling **glStartGeometryEXT()**. In this stage, primitives are defined and rasterized by different *rasterizer boards*. Valid calls include operations on the modelling and texture matrices, setting material values and other attributes, changing the current texture, and other changes to global state which affect only transformation and rasterization. Display lists may be compiled and executed, or primitives may be issued in immediate mode.

2.3 End of Frame

The final stage begins when **glEndFrameEXT()** is called. Once it returns, the frame buffer is defined. At this time it may be accessed using functions like **glReadPixels()** or **glCopyTexture()**[3]. We expect to support other frame buffer operations such as **glAccum()** at a later date.

2.4 Example

This code fragment draws a frame containing a single red triangle. Lights are assumed to have been defined previously.

```
glBeginFrameEXT();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 3.0);
```

[2]The reason for these restrictions is that while performing deferred shading, the viewing, lighting, and texturing environment is assumed to be the same for all samples. If this were not the case, such information would have to be encoded along with each sample, which would enormously increase the amount of pixel memory needed for a sample. By creating named objects representing these environments, we regain this capability, although not at OpenGL's per-primitive granularity.

[3]Hopefully, for e.g. shadow maps.

```
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.0, 0.0, -2.0);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glStartGeometryEXT();

    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        glVertex3f(-1.0, -1.0, 0.0);
        glVertex3f( 0.0,  1.0, 0.0);
        glVertex3f( 1.0, -1.0, 0.0);
    glEnd();

glEndFrameEXT();
```

Example - Frame generation

3 Controlling Primitive and State Distribution

The PixelFlow architecture achieves scalability by using many parallel *rasterizers*, each of which is responsible for transforming and rasterizing a portion of the database, and *shaders*, each of which is responsible for lighting and shading a portion of the pixels in the image. However, primitives are defined in sequential order by the application. So to achieve good rasterization performance, all the primitives defined in the course of a frame must be *distributed* among the rasterizers.

PxGL has a built-in distribution algorithm, and in most cases, an application does not need to be aware of or make changes in this algorithm. However, in some cases application performance can be increased by modifying how primitives are distributed.

This section describes how primitives are distributed, the implications of the distribution algorithm on graphical state maintenance and performance, and how applications may control distribution.

3.1 Primitive Distribution Algorithm

In the remainder of this section, we assume that a PixelFlow system with N rasterizer boards is being used, and that M geometric primitives are to be distributed, where $M \gg N$.

To be done: call to specify processor groups + comments on ordering implications of distributing primitives, state maintenance, per-vertex state not neccessarily affecting global state.

The calls controlling distribution are[4]

`GLenum pxDistributionMode(GLenum` *`type`*`, GLenum` *`mode`*`, GLint` *`param`*`)`

[4]The pbase headers don't use GL types for the prototypes, and return void - this inconsistency needs to be resolved.

Changes how GL commands are distributed to rasterizers and shaders. *type* specifies the type of commands to be affected, and may take on the following values:

> `PX_PRIMITIVE_EXT` - affects sequences of commands delimited by a **glBegin()** . . . **glEnd()** block, which are normally rasterizer primitives such as triangles.
>
> `PX_STATE_EXT` - affects all other commands not within a block[5].
>
> `PX_TEXTURE_EXT` - affects textures[6].

mode specifies how that type of command is distributed, and may take on the following values:

> `PX_DEFAULT_EXT` - commands are sent in according a default mapping scheme.
>
> `PX_BROADCAST_EXT` - commands are sent to all rasterizers that may use them.
>
> `PX_ROUND_ROBIN_EXT` - commands are sent to a single rasterizer or shader, but successive commands are sent to different rasterizers or shaders in a simple sequence specified by *param*, for load balancing purposes.
>
> `PX_ROUND_ROBIN_WEIGHTED_EXT` - commands are sent to a single rasterizer or shader, but successive commands are sent to different rasterizers or shaders in a sequence determined by the cost of the commands, for load balancing purposes[7].
>
> `PX_SPECIFIED_GPS_EXT` - commands are sent to a set of rasterizers and shaders specified by *param*[8].

param controls details of the distribution. For `PX_ROUND_ROBIN_EXT` mode, it is the *blocking factor* - *param* commands are sent to each rasterizer or shader before shifting to the next. For `PX_SPECIFIED_GPS_EXT` mode, it is the rasterizer to send commands to. *param* is currently ignored for the other modes.

`GL_INVALID_ENUM` is generated if *type* or *mode* are not one of the allowed values.

`GL_INVALID_VALUE` is generated if *param* is less than 1 (for `GL_ROUND_ROBIN_EXT` mode) or an invalid rasterizer or shader ID (for `GL_SPECIFIED_GPS_EXT` mode).

`GLenum pxGetDistributionMode(GLenum` *`type`*`, GLenum *`*`mode`*`, GLint *`*`param`*`)`

Returns the distribution *mode* and *param* used for the specified *type* of command.

This call may not be placed in a display list.

`GL_INVALID_ENUM` is generated if *type* is not one of the valid command types passed to **pxDistributionMode()**.

[5] Not implemented; may never be implemented

[6] Which commands are "textures", exactly?

[7] How might this be parameterized?

[8] Eventually, *param* will specify a *processor group ID*, referring to an arbitrary set of processors established with other pxgl calls. At present, it is just a rasterizer number, with rasterizers numbered starting at 0.

4 Extending the OpenGL Namespace

The C language binding of OpenGL [2] includes several namespaces: *functions*, *types*, and *enumerants*. PxGL extends the function and enumerant namespaces and adds several new namespaces: *shader parameters*, *shader functions*, *light parameters*, *light functions*, *rasterizer parameters*, *rasterizer functions*, and *interpolators*. Examples of these namespaces are given.

In accordance with the ARB[9] guidelines for extensions to OpenGL, all additions to the existing namespaces are postfixed by **EXT** for functions and **_EXT** for enumerants.

4.1 Functions

The function namespace refers to C calls made by an application, such as **glBegin()** and **glEnable()**. About 20 new calls are introduced in PxGL, such as **glStartGeometryEXT()** and **glShaderEXT()**. New calls are discussed in detail elsewhere in this document.

4.2 Enumerants

The enumerant namespace refers to compile-time integral constants used to denote options, values, flags, and other parameters to API functions. PxGL adds enumerants for the new calls it introduces, such as `GL_ALL_PRIMITIVES_EXT` (an allowed parameter to the function **glMaterialInterpEXT()**). PxGL also allows some existing functions to *accept* additional enumerant values in the context of extensions, such as passing an enumerant denoting a user-defined sphere rasterizer to **glBegin()** (which normally accepts only enumerants corresponding to the primitives defined in OpenGL). Finally, some existing functions will *generate* or *return* new enumerant values, such as `GL_UNSUPPORTED_OPERATION_EXT` (which may be generated by calling functions in unsupported modes, and later returned by **glGetError()**).

4.3 New Namespaces

Application-defined code may be inserted at many stages of the graphics pipeline, primarily for rasterization, surface shading, and lighting. To call this code and pass appropriate values to it, several new namespaces are introduced corresponding to the various types of code and parameters.

Because such code (with the exception of built-in functionality like triangle rasterizers or the OpenGL shading model) is not known at compile time, a way to dynamically define the namespaces is needed. This is accomplished by functions which map from ASCII string **names** of code and parameters to numeric **identifiers**[10] which are passed to PxGL calls[11].

The new namespaces and the sections in which their uses are discussed are

- Rasterizer functions and parameters, and parameter interpolators (§6).

[9] OpenGL Architecture Review Board.

[10] Should generated IDs be `GLenum` or `GLuint`? Adding enumerants at runtime is of questionable legality; using integers causes incompatibilities with existing calls like **glMaterial()**.

[11] It would be possible to pass names everywhere and avoid this mapping, at enormous performance cost.

- Shader functions, instances, and parameters (§7).
- Light functions, instances, and parameters (§8).
- Atmospheric functions and parameters (§9.1).
- Image manipulation functions and parameters (§9.2).

4.3.1 Names of OpenGL Objects

OpenGL parameters such as light and material properties are given string names (§15). There are unique parameter IDs corresponding to the different parameters, such as ambient light color and ambient surface color. This differs from OpenGL, where the same *pname*, such as `GL_AMBIENT`, may be used to refer to both light and material properties. For backwards compatibility, the OpenGL IDs are accepted as aliases of the actual parameter IDs.

Stuff to be done...

- Querying instance/global, interpolator, and default value for shader parameters
- Built-in shader function, shader parameters (also for rasterizers, lights, etc.)
- Specifying transformation of parameters (also for rasterizers, lights, etc.)
- Talk some more about the parameter namespaces and how they relate to OpenGL *pnames*.

5 Loading Application-Defined Code

Adding application-defined code written in the PixelFlow *shading language* [5] to the PxGL graphics pipeline is done at runtime[12].

The application identifies such code using string *names* that symbolically refer to different modules; the API hides details of how the names are mapped into object files which are loaded into the hardware[13]. For example, a light function using the Torrance-Sparrow model might be named `torrance`; a sphere rasterizer function might be named `sphere`; and a marble shader function might be named `marble`.

Application-defined code may be loaded using this call:

`GLenum glLoadExtensionCodeEXT(GLenum` *stage* [14]`, const GLubyte *`*name*`)`

[12] The mechanism used involves compiling code in the *shading language* into shared object files that are loaded on demand.

[13] Although we can expect that the name will either be a Unix filename component, or a key to look up a filename.

[14] Do we want to load code for different stages with a single interface? We distinguish between stages with **glGetMaterialParameterNameEXT()** and **glGetRastParameterNameEXT()** for example.

Loads application-defined code for the specified pipeline *stage* identified by *name*. Returns an enumerated `id` which is passed to other calls controlling when the code is to be used.

May be called with a built-in function or called again for application-defined code that's already been loaded. No action is taken, but the same valid `id` is returned.

stage may take on the following values:

`GL_LIGHT_FUNCTION_EXT` - load a light function. `id` is passed to **glNewLightEXT()**.

`GL_RASTERIZER_FUNCTION_EXT` - load a rasterizer function. `id` is passed to **glBegin()**.

`GL_SHADER_FUNCTION_EXT` - load a shading function. `id` is passed to **glNewShaderEXT()**.

`GL_ATMOSPHERIC_FUNCTION_EXT` - load an atmospheric function. `id` is passed to[15].

`GL_WARPING_FUNCTION_EXT` - load an image warping function. `id` is passed to[16].

`GL_INVALID_ENUM` is generated if *stage* is not one of the allowed values, and 0 is returned.

`GL_INVALID_VALUE` is generated if *name* does not exist, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

Code loaded with **glLoadExtensionCodeEXT()** usually has associated *parameters*; rasterizers may also have associated *interpolators*. Loading code may have the side effect of extending those namespaces. At present, there is a single namespace for parameters even though they are accessed by different calls depending on the stage in which those parameters are used. Thus, we require user-defined namespace scoping to distinguish both the stage and the specific object within that stage which the parameter applies to; for example, `rast_sphere_radius` and `shader_polkadot_radius`[17].

To map parameter names into identifiers, use the calls **glGetMaterialParameterNameEXT()** or **glGetRastParameterNameEXT()**.

6 Programmable Rasterization

The programmable rasterization model used in PxGL extends the **glBegin() / glEnd()** mechanism used to define built-in primitive types such as triangles and lines. These new terms are introduced:

[15] Yes, to what?
[16] And again, to what?
[17] We should recommend namespace conventions.

Interpolator - A method for combining parameter values specified at one or more discrete locations on a primitive being rasterized to generate values for that parameter at all other locations on the primitive where it is not specified. The most common interpolators are named `constant` (corresponding to flat shading on a primitive), `flat` (corresponding to **glShadeModel(**`GL_FLAT`**)**, e.g. flat shading on individual polygons within a primitive), and and `linear` (corresponding to **glShadeModel(**`GL_SMOOTH`**)**, e.g. Gouraud shading on polygons within a primitive). Other interpolator types may be defined for user-specified rasterizer functions.

Since interpolation considered as a mathematical process is tightly bound to the geometrical definition of a surface, most interpolators are only defined for specific types of primitives. Interpolators have string *names* and corresponding enumerated *parameter IDs*, referred to as `interpname` and `interpid` in code examples

Rasterizer Function - A function which takes as input a set of *rasterizer parameters* and generates screen-space samples at which the function is visible. A rasterizer function represents a type of geometric primitive; its parameters determine a specific instance of that geometry. In abstract terms, the function creates geometry, transforms it according to the current model-view and projection matrices, and samples it. At visible samples, *shader parameters* defined for the current shader are computed using a specified *parameter interpolator* and copied into the *sample buffer*.

Rasterizer Parameter - A parameter to a rasterizer function. Some examples include vertices of polygons, sphere radii, or control points of parametric patches.

Sequence Point - Specifies the binding time for a group of rasterizer and shader parameters. A *rasterizer function* may require one or more sequence points to define a specific instance of its geometry. In many cases, including all the OpenGL primitive types, the *rasterizer parameters* bound at the sequence point will simply be vertices of a surface. Other examples include center and radii of spheres, twist vectors of Hermite patches, or coefficients of general quadric surfaces[18].

6.1 Loading and Using Rasterizer Functions

To use an application-defined rasterizer function, the following steps must be taken:

- Load the rasterizer function and obtain its ID with **glLoadExtensionCodeEXT()**
- Obtain parameter IDs of rasterizer parameters using **glGetRastParameterNameEXT()**.
- Call **glBegin()** with the rasterizer ID to start delimiting sequence points of a rasterizer function.
- Specify rasterizer parameters using **glRastParamEXT()** and bind them using **glSequencePointEXT()**.

[18] Rasterizer writers will have to document which parameters are per-block and which are per-sequence-point.

- Call **glEnd()** to finish delimiting sequence points of the function and call the rasterizer function.

6.1.1 Example

In the following example, a rasterizer function named `spheres` is loaded. The function has two parameters, the `center` and `radius` of the sphere; each sequence point defines a separate sphere. Two unit-radius spheres which touch at the origin and are centered at (1,0,0) and (-1,0,0) are drawn.

```
// Load the rasterizer and obtain its ID
GLenum spherefuncid =
    glLoadExtensionCodeEXT(GL_RASTERIZER_FUNCTION_EXT, "spheres");

// Obtain IDs for named parameters
GLenum centerid = glGetRastParameterNameParameterEXT("rast_sphere_center");
GLenum radiusid = glGetRastParameterNameParameterEXT("rast_sphere_radius");

glBeginFrameEXT();
    glStartGeometryEXT();

    GLfloat vertminus[3] = { -1, 0, 0 };
    GLfloat vertplus[3] = { 1, 0, 0 };

    // Draw the two spheres
    glRastParamfEXT(radiusid, 1.0);
    glBegin(spherefuncid);
        glRastParamfvEXT(centerid, &vertminus);
        glSequencePointEXT();

        glRastParamfvEXT(centerid, &vertplus);
        glSequencePointEXT();
    glEnd();
```

Example - Using rasterizer functions

6.2 Rasterizer API Definitions

There is currently an naming inconsistency where some calls use **RastParam** and others use **RastParameter**. This should be resolved, probably in favor of the latter.

`void glGetRastParamEXT(GLenum` *`paramid`*`, TYPE *`*`params`*`)`

Returns the value of the specified parameter in *params*.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid rasterizer parameter.

`GLenum glGetRastParameterNameEXT(GLchar *name_string)`

Returns the parameter ID corresponding to the string *name*.

`GL_INVALID_NAME_STRING_EXT` is generated if *string* is not a parameter of any rasterizer, and 0 is returned.

`GLchar * glGetRastParameterStringEXT(GLenum pname)`

Returns the string name corresponding to the specified parameter ID.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `NULL` is returned.

`void glSequencePointEXT()`

Binds parameters of the rasterizer and shader functions in use.

`GL_INVALID_OPERATION` is generated when **glSequencePointEXT()** is called other than between **glBegin()** and **glEnd()**.

`void glRastParamEXT(GLenum paramid, TYPE params)`

glRastParam assigns values to rasterizer parameters. *paramid* specifies which parameter will be modified. *params* specifies what value or values will be assigned to the parameter.

`GL_INVALID_VALUE` is generated if *paramid* is not a defined rasterizer parameter ID.

6.3 glVertex() and Sequence Points

Vertices defining built-in primitive types are rasterizer parameters. The following two code sequences have identical effects:

```
glVertex3f(x,y,z);
```

Defining a vertex using glVertex()

```
GLenum vertid = glGetRastParameterNameEXT("gl_vertex");
GLfloat point[4] = { x, y, z, 1.0 };
...
glRastParamfvEXT(vertid, &point);
glSequencePointEXT();
```

Defining a vertex using rasterizer extensions

6.4 Vertex Array Extensions for Rasterizers and Shaders

These will be needed, but can't be finalized until the GL 1.1 specification is out.

6.5 Interpolators

Every rasterizer function has one or more interpolators associated with its geometry, which take shader parameters specified at control points and generate parameter values at all samples. All rasterizers may use the *constant* interpolator, which copies a single value into all samples. Rasterizers defined by OpenGL all support the *flat* interpolator, which copies a separate constant value into each successive primitive (triangle, line segment, quadrilateral, etc.) in a group, and the *linear* interpolator, which fits a linear function (possibly perspective-corrected) to the first two or three vertices of a primitive.

There is also an *implicit* interpolator, which ignores parameter values specified at sequence points. Its exact function varies depending on the rasterizer and parameter type. For built-in rasterizers, the implicit interpolator can only be applied to texture coordinates, implementing the functionality of **glTexGen()**.

Other types of rasterizers may use these interpolators, if they make sense, or define new interpolators corresponding to their geometry[19]. For example, a triangle with 3 additional sequence points at the midpoints of its edges might define a *quadratic* interpolator, to allow smoother shading between triangles. A parametric patch might define an interpolator which applies the same weights to shader parameters as to control points. A sphere or general quadric surface rasterizer might interpret the *implicit* interpolator to generate texture coordinates and normals based on the intrinsic geometry of the surface.

6.6 Interpolator API Definitions

`void glGetMaterialInterpEXT(GLenum` *`paramid`*`, GLenum` *`primtype`*`, GLenum *`interpid`*`)`

> Returns the interpolator used for rasterizing the specified shader parameter for the specified primitive type.
>
> `GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter or if *primtype* is not a valid primitive type.

`void glMaterialInterpEXT(GLenum` *`paramid`*`, GLenum` *`primtype`*`, GLenum` *`interpid`*`)`

> Sets the *interpolator* to be used for rasterizing the specified shader parameter for the specified primitive type. A primitive type is required because most interpolators are defined only for specific types of geometry.
>
> *interpid* is usually an interpolator ID for a specific primitive. Five interpolators are built-into PxGL:
>
> `GL_IMPLICIT_INTERPOLATOR_EXT` is implemented for texture coordinates in built-in rasterizers, according to the **glTexGen()** parameters[20]. When rasterizing user defined primitives, it is intended to allow generating normals and texture coordinates based on the intrinsic geometry of the object.
>
> `GL_CONSTANT_INTERPOLATOR_EXT` copies the parameter value current when

[19] We don't have a way to get IDs for interpolators loaded as part of rasterizers, yet - something like a **glGetInterpolatorNameEXT()** call is needed.

[20] Do we want to implement it for surface normals, too?

glBegin() is called into all samples rasterized for that primitive or group of primitives. It is guaranteed to be implemented for all primitive types and all parameter types.

`GL_FLAT_INTERPOLATOR_EXT` copies the parameter value current when the last vertex or sequence point defining a primitive is called into all samples rasterized for that primitive. Unlike the constant interpolator, a group of primitives defined in a **glBegin() / glEnd()** block may have a different value specified for each primitive. This corresponds to **glShadeModelEXT(**`GL_FLAT`**)**.

`GL_LINEAR_INTERPOLATOR_EXT` is implemented for all built-in primitive types and parameters, and corresponds to **glShadeModel(**`GL_SMOOTH`**)**[21].

`GL_DEFAULT_INTERPOLATOR_EXT` is a way to specify the most "natural" type of interpolator for a primitive; linear for a polygon, implicit for a sphere, bicubic for a patch, and so on.

primtype is either a valid primitive type or the special value `GL_ALL_PRIMITIVES_EXT`. In the latter case, only `GL_CONSTANT_INTERPOLATOR_EXT`, `GL_FLAT_INTERPOLATOR_EXT`, or `GL_DEFAULT_INTERPOLATOR_EXT` may be specified.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter, if *primtype* is neither a valid primitive type nor `GL_ALL_PRIMITIVES_EXT`, or if *interpid* is not a valid interpolator.

`GL_INVALID_OPERATION` is generated if *interpid* is not defined for the specified *paramid* and *primtype*.

To be added: glGenDataEXT() and **glDeleteDataEXT()**.

7 Programmable Shading

The programmable shading model used in PxGL is based on the RenderMan [4] shading language, but use of some terms differ and these new terms are introduced:

Shader Function - A function, either built-in to PxGL or loaded at runtime, which takes as input a set of *shader parameters* and generates as output a color. A shader function is conceptually applied to each sample of a primitive which was rasterized with a corresponding *shader* applied[22]. Shader functions have string *names* and corresponding enumerated IDs, referred to as `shaderfunc` and `shaderfuncid` in code examples.

Shader - An instance of a shader function which binds a subset of the function's parameters to be *nonvarying* for all samples to which the shader is applied. This is done primarily to increase rasterization and shading speed and to reduce traffic on the PixelFlow image composition network. Shaders have enumerated IDs, referred to as `shaderid` in code examples.

[21] Note that in PxGL, interpolation is applied to shading parameters *before* lighting, rather than to color *after* lighting, as in OpenGL. This allows true Phong shading, avoiding the artifacts caused by OpenGL's Gouraud interpolation of Phong-lit vertices.

[22] Deferred shading means that in practice, only samples which affect visibility are actually shaded.

Shader Parameter - An input argument to a shader function. These fall into three types depending on how they arrive at the shading hardware: *uniform*, *nonvarying*, and *varying* parameters. Shader parameters have string *names* and corresponding enumerated IDs, referred to as `paramname` and `paramid`[23] in code examples.

Nonvarying Parameter - A shader parameter whose value is the same for all samples rasterized using that shader. A non-*uniform* parameter of a *shader function* may be chosen to be either nonvarying or *varying* on a per-*shader* basis using **glMaterialVaryingEXT()**.

Uniform Parameter - A shader parameter whose value is the same for all samples rasterized using that shader. Uniform parameters cannot be made *varying*[24].

Varying Parameter - A shader parameter whose value may be different in each sample rasterized using that shader.

7.1 Creating Shaders

To create a shader, the following steps must be taken:

- Load a shader function and obtain its ID with **glLoadExtensionCodeEXT()**.
- Create the new shader and obtain a shader ID using **glNewShaderEXT()**.
- Obtain parameter IDs of shader parameters using **glGetMaterialParameterNameEXT()**.
- Specify which shader parameters are varying using **glMaterialVaryingEXT()** (all parameters not otherwise specified are assumed to be uniform).
- Instantiate the shader with **glEndShaderEXT()**.

After creating the shader, nonvarying parameter values may be set using **glSurfaceEXT()**. These parameter values can be changed at any time before start of geometry.

7.1.1 Example

This code fragment loads a hypothetical shader function named `phong_shader`. The shader function has two parameters, named `gl_shader_color` (intrinsic color) and

[23] OpenGL uses *pname* to refer to material parameters such as emission color, which are shader parameters of the builtin OpenGL shading model. This discrepancy should be resolved; Rich suggests an explanation of parameter *names* vs. parameter *IDs*.

[24] The distinction between uniform parameters and nonvarying parameters is subtle from the user's point of view, and these definitions need work: both are sent to the shader GPs over the geometry network, but uniform parameters are held on the GP during shading code execution, while nonvarying parameters are copied into pixel memory. The distinction is primarily an efficiency measure to reduce composition network bandwidth requirements.

`gl_shader_normal` (surface normal)[25]. Two shaders are created. The first, `phongshader`, allows both color and normal to vary. The second, `redshader`, has a nonvarying intrinsic color of red.

```
// Load the named shader and obtain its ID
GLenum phongfuncid =
    glLoadExtensionCodeEXT(GL_SHADER_FUNCTION_EXT, "phong_shader");

// Obtain IDs for named parameters
GLenum colorid = glGetMaterialParameterNameEXT("gl_shader_color");
Glenum normalid = glGetMaterialParameterNameEXT("gl_shader_normal");

// Create a shader with ID 'phongshader', allowing both parameters to vary
GLenum phongshader = glNewShaderEXT(phongfuncid);
    glMaterialVaryingEXT(phongshader, colorid);
    glMaterialVaryingEXT(phongshader, normalid);
glEndShaderEXT();

// Create 'redshader', allowing only normals to vary and
//  binding the nonvarying color to red.
GLfloat red[3] = { 1, 0, 0 };
GLenum redshader = glNewShaderEXT(phongfuncid);
    glMaterialVaryingEXT(redshader, normalid);
glEndShaderEXT();
glSurfacefvEXT(redshader, colorid, &red);
```

Example - Creating shaders

7.2 Using Shaders

To use a shader once it has been created, the following steps must be taken:

- Select the shader using **glShaderEXT()**.
- Specify the interpolation method to be used for *varying* shader parameters using **glMaterialInterpEXT()**.
- Define a primitive, setting values of varying shader parameters using **glMaterial()**.

7.2.1 Example

This continues the previous example, defining three triangles. The first uses `redshader` to draw a red phong-lit triangle with linearly interpolated normals. The second uses `phongshader` to draw a vertex-colored triangle using linear interpolation of the vertex colors. The third uses `phongshader` to draw a green triangle using constant interpolation.

[25] Note that these parameters are also parameters of the built-in OpenGL shader; they are used by the loadable shader so the example can make shortcut calls like **glNormal()** and **glColor()** to specify shader parameters, rather than **glMaterial()**.

```
// Select the red-colored shader
glShaderEXT(GL_FRONT_AND_BACK, redshader);

// Choose a linear interpolator for normals and draw a red
//  phong-shaded triangle.
glMaterialInterpEXT(normalid, GL_TRIANGLES, GL_LINEAR_INTERPOLATOR_EXT);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();

// Select the phong shader, use linear interpolation for color,
//  and draw a vertex-colored phong-shaded triangle
glShaderEXT(GL_FRONT_AND_BACK, phongshader);

glMaterialInterpEXT(colorid, GL_TRIANGLES, GL_LINEAR_INTERPOLATOR_EXT);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glColor3fv(color[i]);
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();

// Change to constant interpolation for color, and draw a green
//  phong-shaded triangle.
glMaterialInterpEXT(colorid, GL_TRIANGLES, GL_CONSTANT_INTERPOLATOR_EXT);

GLfloat green[3] = { 0, 1, 0 };
glColor3fv(green);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();
```

Example - Using shaders

There is a subtle difference between the first and third triangles: the first uses a shader where color is *nonvarying*, so that all primitives rendered using that shader will be red. The third triangle uses a shader where color is *varying*, but the constant interpolator causes the

color to be fixed on that particular triangle[26].

7.3 Shading API Definitions

`void glDeleteShaderEXT(GLuint shaderid)`

Removes the definition of the specified shader; *shaderid* is unused after this call.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glEndShaderEXT()`

Instantiates a shader created by **glNewShaderEXT()**. All shader parameters which are not explicitly specified in previous calls to **glMaterialVaryingEXT()** are made *nonvarying*; values of these parameters are set with **glSurfaceEXT()**.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, or when not preceded by a corresponding **glNewShaderEXT()**.

`void glGet(GLenum pname, TYPE *params)`

glGet() is extended to accept parameters `GL_FRONT_SHADER_EXT` and `GL_BACK_SHADER_EXT`, which return the current front and back face shaders as specified via **glShaderEXT()**.

`void glGetMaterial(GLenum face, GLenum paramid, TYPE *params)`

glGetMaterial() is extended so that *paramid* can refer to shader parameters defined by dynamically loaded shaders.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter.

`GLenum glGetMaterialParameterNameEXT(GLchar *name_string)`

Returns the parameter ID corresponding to the string *name_string*.

`GL_INVALID_NAME_STRING_EXT` is generated if *name_string* is not a parameter of any shader, and 0 is returned.

`void glGetMaterialParametersEXT(GLuint shaderid, GLenum *pnames)`

Returns a list of parameter IDs used by the specified shader. *pnames* must have room for at least the number of IDs specified by **glGetNumMaterialParametersEXT()**.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

[26] The purpose of the constant interpolator is to reduce work done during rasterization; it's appropriate when performing (for example) flat shading. The same visual effect could also be achieved by using the linear interpolator and specifying the same color at each vertex, but rasterization speed would be lower.

`GLchar * glGetMaterialParameterStringEXT(GLenum` *`pname`*`)`

Returns the string name corresponding to the specified parameter ID.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `NULL` is returned.

`GLuint glGetNumMaterialParametersEXT(GLuint` *`shaderid`*`)`

Returns the number of material parameters accepted by the specified shader. Used in conjunction with **glGetMaterialParametersEXT()**.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`void glGetSurfaceEXT(GLuint` *`shaderid`*`, GLenum` *`face`*`, GLenum` *`paramid`*`, TYPE *`params`*`)`

Retrieves the value of a *nonvarying* parameter of the specified shader. Bound values are set by **glSurfaceEXT()**.

`GL_INVALID_ENUM` is generated if *face* is not `GL_FRONT` or `GL_BACK`, or if *paramid* is not a bound parameter of *shaderid*.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GLboolean glIsMaterialParameterEXT(GLuint` *`shaderid`*`, GLenum` *`pname`*`)`

Returns `TRUE` if *pname* is a parameter of the specified shader, `FALSE` otherwise.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID, and `FALSE` is returned.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `FALSE` is returned.

`GLboolean glIsMaterialUniformEXT(GLuint` *`shaderid`*`, GLenum` *`pname`*`)`

Returns `TRUE` if *pname* is a *uniform* parameter of the specified shader, `FALSE` otherwise.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID, and `FALSE` is returned.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `FALSE` is returned.

`GLboolean glIsShaderEXT(GLuint` *`shaderid`*`)`

Returns `TRUE` if *shaderid* is used for an existing shader, `FALSE` otherwise.

`void glMaterial(GLenum` *`face`*`, GLenum` *`paramid`*`, TYPE` *`params`*`)`

glMaterial() is extended so that *paramid* can refer to shader parameters defined by dynamically loaded shader functions.

`GL_INVALID_ENUM` is generated if *paramid* is not a shader parameter either of the built-in OpenGL shading function or of a shader function previously loaded.

`void glMaterialVaryingEXT(GLuint` *`shaderid`*`, GLenum` *`paramid`*`)`

Specifies that a parameter is *varying* for this shader. All parameters of a shader are *uniform* or *nonvarying* unless specified as varying by the time **glEndShaderEXT()** is called[27].

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter or a *uniform* parameter.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called other than between **glNewShaderEXT()** and **glEndShaderEXT()**.

`GLuint glNewShaderEXT(GLenum` *`shaderfuncid`*`)`

Creates and returns a shader ID for a new instance of the specified shader function.

`GL_INVALID_ENUM` is generated if *shaderfuncid* does not refer to a valid shader function, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

`void glShaderEXT(GLenum` *`face`*`, GLuint` *`shaderid`*`)`

Sets the shader to be used for shading the specified face of primitives defined following the call. *face* may be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

`GL_INVALID_ENUM` is generated if *face* is not one of the allowed values.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`void glSurfaceEXT(GLunit` *`shaderid`*`, GLenum` *`paramid`*`, TYPE` *`params`*`)`

Sets the value of **nonvarying** parameters of a shader instance. The values of **varying** parameters are set with **glMaterial()**.

Nonvarying parameters cannot be specified separately for front and back faces; there is a single value used regardless of whether the front or back face of a primitive is rasterized. This can be addressed by using different shaders on front and back faces.

A nonvarying parameter has an initial value defined by the shader using that parameter. The value is set when the shader is loaded.

`GL_INVALID_ENUM` is generated if *paramid* does not refer to a nonvarying parameter of the specified shader.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

[27] While *shaderid* appears redundant, keeping the parameter allows the possibility of changing a parameter between varying and nonvarying on the fly, in a possible future implementation.

7.4 To Be Done

- Parameter Transformation (normals, texture matrix).
- Parameter Generation (**glTexCoord()**, sphere normals).
- Implicit Parameters (texture scale factors, texture ID, normals).
- `GL_FRONT_AND_BACK` vs. uniform parameters and optimized lists.

8 Programmable Lighting

The programmable lighting model used in PxGL introduces these new terms:

Light Function - A function which takes as input a set of *light source parameters* and a set of *shader parameters* at a sample, and generates an illumination at that sample which is used by a *shader function* to compute color of the sample.

Light Group - A subset of all existing light instances, used to illuminate specified primitives during shading. Only one light group may be active at any time.

8.1 Creating Lights

To create a light, the following steps must be taken:

- Load a light function and obtain its ID with **glLoadExtensionCodeEXT()**
- Create the new light and obtain a light ID using **glNewLightEXT()**.
- Obtain parameter IDs of light source parameters using **glGetLightParameterName**[28]**EXT()**.
- Call **glLight()** to specify light source parameters.

8.1.1 Example

I don't have a good example of a user-defined light function. This example just creates a new instance of the built-in OpenGL light function, which is named `gl_light_function`. The light is made a red, diffuse, infinite light in direction -Z.

```
glBeginFrameEXT();

    // Get the light function ID for the built-in light model
    //  by "loading" it.
    GLenum lightfuncid =
        glLoadExtensionCodeEXT(GL_LIGHT_FUNCTION_EXT, "gl_light_function");

    // Create a new instance of the OpenGL light function
```

[28] This call needs to be added.

```
GLenum lightid = glNewLightEXT(lightfuncid);

// Get IDs of light source parameters. We do not really
//  need to do this for the built-in light function; GL_POSITION
//  and GL_DIFFUSE could be used instead.
GLenum positionid = glGetLightParameterNameEXT("gl_light_position");
GLenum diffuseid = glGetLightParameterNameEXT("gl_light_direction");

GLfloat position[4] = { 0.0, 0.0, -1.0, 0.0 };
GLfloat diffusecolor[4] = { 1.0, 0.0, 0.0, 1.0 };

glLightfv(lightid, positionid, &position);
glLightfv(lightid, diffuseid, &diffusecolor);
```

Example - Creating a light

8.2 Using Lights

There is no limit on the number of lights which may be created (above and beyond the built-in OpenGL lights). Lights are placed in *light groups*, which are arbitrary subsets of the defined lights with enumerated IDs; the *current light group* may be changed at any time and that set of lights is applied when shading primitives. Initially a single light group, `GL_DEFAULT_LIGHT_GROUP_EXT`, exists and is the current light group.

To change the lighting environment, the following steps must be taken:

- Optionally create a new light group.
- Place desired lights in the light group.
- Specify the current light group.
- Render primitives with the specified light group illuminating them.

8.2.1 Example

This continues the previous example, placing the new light in a new light group, selecting that as the current light group, and drawing a triangle.

```
// Create a new light group
GLuint groupid = glNewLightGroupEXT();

// Add the new light to this group
glEnableLightGroupEXT(groupid, lightid);

glStartGeometryEXT();

glLightGroupEXT(groupid);

// Primitives drawn now are lit by the new light
```

Example - Using a light

8.3 Light API Definitions

`void glDeleteLightEXT(GLenum` *`lightid`*`)`

Removes the definition of the specified light; *lightid* is unused after this call.

`GL_INVALID_VALUE` is generated if *lightid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDeleteLightGroupEXT(GLuint` *`groupid`*`)`

Removes the definition of the specified light group; *groupid* is unused after this call.

`GL_INVALID_VALUE` is generated if *groupid* is not a defined light group.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDisable(GLenum` *`cap`*`)`

`void glEnable(GLenum` *`cap`*`)`

glDisable() and **glEnable()** are extended to operate on light groups. When *cap* is `GL_LIGHT`*i*, the specified built-in light is removed from or added to the current light group[29].

`void glDisableLightGroupEXT(GLuint` *`groupid`*`, GLenum` *`lightid`*`)`

`void glEnableLightGroupEXT(GLuint` *`groupid`*`, GLenum` *`lightid`*`)`

Removes or adds the specified light to the specified light group.

`GL_INVALID_VALUE` is generated if *groupid* is not a valid light group ID or *lightid* is not a valid light ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glGet(GLenum` *`pname`*`, TYPE *`*`params`*`)`

glGet() is extended to accept parameter `GL_LIGHT_GROUP_EXT`, which returns the current light group as specified via **glLightGroupEXT()**.

`void glGetLight(GLenum` *`lightid`*`, GLenum` *`paramid`*`, TYPE *`*`param`*`)`

glGetLight() is extended so that *paramid* can refer to light source parameters defined by dynamically loaded light functions.

`GL_INVALID_ENUM` is generated if *lightid* is not a valid light or if *paramid* is not a light source parameter of the light

[29] `GL_LIGHTING` could be implemented as a flag on the entire light group; at present it has no effect.

`void glGetLightFunctionEXT(GLenum` *`lightid`*`, GLenum *`*`lightfuncid`*`)`

Returns in *lightfuncid* the light function used by the specified *light*.

`GL_INVALID_ENUM` is generated if *lightid* is not a valid light.

`GLboolean glIsLightEXT(GLenum` *`lightid`*`)`

Returns `TRUE` if *lightid* is used for an existing light, `FALSE` otherwise.

`GLboolean glIsLightGroupEXT(GLuint` *`groupid`*`)`

Returns `TRUE` if *groupid* is used for an existing light group, `FALSE` otherwise.

`void glLight(GLenum` *`lightid`*`, GLenum` *`paramid`*`, TYPE` *`param`*`)`

glLight() is extended so that *paramid* can refer to light source parameters defined by dynamically loaded light functions.

`GL_INVALID_ENUM` is generated if *paramid* is not a light source parameter either of the built-in OpenGL light function or of a light function previously loaded.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glLightGroupEXT(GLuint` *`groupid`*`)`

Sets the light group to be used for lighting primitives specified following the call.

`GL_INVALID_VALUE` is generated if *groupid* is not a defined light group ID.

`void glLightModelEXT(GLenum` *`pname`*`, TYPE` *`param`*`)`

glLightModel() is extended so that when two-sided lighting is enabled via `GL_LIGHT_MODEL_TWO_SIDE`, it includes all **varying** parameters of the shader being used for a primitive. This allows texture coordinates, texture IDs, and user-defined shader parameters to differ on front and back faces of a primitive.

`GLenum glNewLightEXT(GLenum` *`lightfuncid`*`)`

Creates and returns a light ID for a new instance of the specified light function.

`GL_INVALID_ENUM` is generated if *lightfuncid* does not refer to a valid light function, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

`GLuint glNewLightGroupEXT()`

Creates a new light group and returns the group ID. Initially no lights are in the group; lights may be added with **glEnableLightGroupEXT()**.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

9 Programming Other Pipeline Stages - *to be written*

9.1 Atmospheric

Talk about **glFog()** *here.*

9.2 Warping

To be defined.

10 Transparency and Other Blending Effects

Because PixelFlow is an image composition architecture, in which there is not a single frame buffer during rasterization, the effects possible via blending in OpenGL must be done via alternate methods.

Further discussion about blending across frame boundaries and such will go here later.

10.1 Transparency

Transparent primitives may be handled in one of two ways. The first is screen-door transparency. This supports a limited number of levels of transparency, depending on the number of samples/pixel being rasterized, but is the most general method. The second method is a multipass algorithm which extracts all transparent primitives and renders them properly in sorted order using multiple rendering passes to resolve visibility (*Apgar paper citation goes here*). Unlike alpha blending in OpenGL, neither approach relies on the database being traversed in any particular order.

To use transparent primitives, several steps must be taken:

- Enable transparency on a per-frame basis using **glTransparencyEXT()**.
- Enable transparency on a per-primitive basis using **glEnable()**.
- Specify transparent primitives by defining colors with non-unitary alpha components.

The new calls are:

`void glTransparencyEXT(GLenum` *mode*`)`

Specifies the method by which transparent primitives are rendered. Must be called during the frame setup stage (section 2.1).

mode may take on the following values:

`GL_TRANSPARENCY_NONE_EXT` - transparency is not handled. All primitives are treated as opaque regardless of alpha values.

`GL_TRANSPARENCY_SCREEN_DOOR_EXT` - transparency is done by turning on a fraction of the samples in each pixel corresponding to the alpha value of

that fragment. This is usually the fastest and lowest quality mode.

`GL_TRANSPARENCY_MULTIPASS_EXT` - transparency is done by multipass rendering of potentially transparent primitives. This is usually the slowest and highest quality mode.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDisable(GLenum` *cap*`)`

`void glEnable(GLenum` *cap*`)`

glDisable() and **glEnable()** are extended to support potentially transparent primitives. When *cap* is `GL_TRANSPARENCY_EXT` and is enabled, primitives may be handled using the transparency mode determined by **glTransparencyEXT()**. When disabled, primitives are treated as opaque regardless of their alpha values.

For maximum performance, `GL_TRANSPARENCY_EXT` should be enabled only when potentially transparent primitives are being rasterized.

10.1.1 Determining Transparency

Determining whether or not primitives are transparent at rasterization time is difficult in a deferred-shading architecture, since user-defined shaders need not have an input parameter analogous to the alpha value used by OpenGL. At present, transparency is only handled for primitives using the built-in OpenGL shader[30].

11 Display List Optimization - *to be written*

- How to specify optimization; types of optimizations.
- Inheriting state from environment for constant-interpolated params, binding at **glBegin()**.
- Interaction with **glShadeModelEXT()**.

12 Multiple Application Threads - *to be written*

Discuss multiple AP contexts, ordering issues, frame synchronization points, global namespaces for lights, shaders, and rasterizers, local (perhaps) namespaces for display lists.

13 OpenGL Variances - *to be written*

Tables of (enumerant,relevant calls) and (call,valid frame stages) will go here.

[30] Is this true? We've gone around on possible approaches to shaders generating transparent samples before, but there has been no resolution yet. What does the current implementation do?

- Depth buffer always enabled.
- Depth function always `GL_LESS`.
- Transparency specially handled (see section 10.1).
- And lots more...

14 Unsupported OpenGL Features - *to be written*

Lee's lengthy document should be referenced here.

15 Function, Enumerant, and Name Tables

Parameters of the built-in light, shader, and rasterizer functions have all been assigned string names which map to enumerated IDs. Existing OpenGL enumerants (such as `GL_AMBIENT` or `GL_LIGHT0`) are recognized as aliases for the actual IDs. String names of built-in parameters, and the corresponding OpenGL enumerants, are listed below.

15.1 Light Function and Parameter Names

There is a single built-in light function corresponding to the OpenGL lighting model, named `gl_light_function`. Table 1 lists parameters of this function, which correspond to OpenGL light source parameters.

String Name	OpenGL ID
gl_light_ambient	GL_AMBIENT
gl_light_diffuse	GL_DIFFUSE
gl_light_specular	GL_SPECULAR
gl_light_position	GL_POSITION
gl_light_spot_direction	GL_SPOT_DIRECTION
gl_light_spot_exponent	GL_SPOT_EXPONENT
gl_light_spot_cutoff	GL_SPOT_CUTOFF
gl_light_constant_attenuation	GL_CONSTANT_ATTENUATION
gl_light_linear_attenuation	GL_LINEAR_ATTENUATION
gl_light_quadratic_attenuation	GL_QUADRATIC_ATTENUATION

Table 1: Built-in light source parameter names

15.2 Rasterizer Function and Parameter Names

Table 2 lists the built-in rasterizer function names and the corresponding OpenGL IDs.

String Name	OpenGL ID
gl_rasterizer_points	GL_POINTS
gl_rasterizer_lines	GL_LINES
gl_rasterizer_line_strip	GL_LINE_STRIP
gl_rasterizer_line_loop	GL_LINE_LOOP
gl_rasterizer_triangles	GL_TRIANGLES
gl_rasterizer_triangle_strip	GL_TRIANGLE_STRIP
gl_rasterizer_triangle_fan	GL_TRIANGLE_FAN
gl_rasterizer_quads	GL_QUADS
gl_rasterizer_quad_strip	GL_QUAD_STRIP
gl_rasterizer_polygon	GL_POLYGON

Table 2: Built-in rasterizer functions

There is a single parameter of built-in rasterizers, named `gl_vertex`. Vertices are normally specified using **glVertex()** rather than **glRastParamEXT()** (§6.3).

15.3 Shader Function and Parameter Names

There is a single built-in shader function corresponding to the OpenGL shading model, called `gl_shader_function`. Table 3 lists parameters of this function and the corresponding OpenGL material parameter names.

15.4 Atmospheric Function and Parameter Names

There is a single built-in atmospheric function corresponding to the OpenGL fog model, called `gl_fog_function`. Table 4 lists parameters of this function and the corresponding OpenGL fog parameter names.

15.5 Interpolator Names

Table 5 lists the built-in interpolator functions which may be used with the built-in rasterizer functions. The **constant** and **implicit** interpolators may also be used with any application-defined rasterizer function.

String Name	OpenGL ID
gl_shader_ambient	GL_AMBIENT
gl_shader_diffuse	GL_DIFFUSE
gl_shader_color	Use **glColor()**
gl_shader_specular	GL_SPECULAR
gl_shader_emission	GL_EMISSION
gl_shader_shininess	GL_SHININESS
gl_shader_textureid	Use texture object calls
gl_shader_normal	Use **glNormal()**
gl_shader_u, gl_shader_v	Use **glTexCoord()**
gl_shader_du, gl_shader_dv	Implicitly generated

Table 3: Built-in material parameters

String Name	OpenGL ID
gl_fog_mode	GL_FOG_MODE
gl_fog_density	GL_FOG_DENSITY
gl_fog_start	GL_FOG_START
gl_fog_end	GL_FOG_END
gl_fog_color	GL_FOG_COLOR

Table 4: Built-in atmospheric parameters

String Name	OpenGL ID
gl_interpolator_implicit	GL_IMPLICIT_INTERPOLATOR_EXT
gl_interpolator_constant	GL_CONSTANT_INTERPOLATOR_EXT
gl_interpolator_flat	GL_FLAT_INTERPOLATOR_EXT
gl_interpolator_linear	GL_LINEAR_INTERPOLATOR_EXT
gl_interpolator_default	GL_DEFAULT_INTERPOLATOR_EXT

Table 5: Built-in interpolator names

15.6 Defined Constants

Table 6 lists manifest constants in PxGL which are not in OpenGL, along with the corresponding commands these constants are used in.

Constant	Associated Commands
GL_ALL_PRIMITIVES_EXT	**glMaterialInterpEXT()**
GL_BACK_SHADER_EXT, GL_FRONT_SHADER_EXT, GL_LIGHT_GROUP_EXT	**glGet()**
GL_DEFAULT_LIGHT_GROUP_EXT	**glLightGroupEXT()**
GL_CONSTANT_INTERPOLATOR_EXT, GL_DEFAULT_INTERPOLATOR_EXT, GL_FLAT_INTERPOLATOR_EXT, GL_IMPLICIT_INTERPOLATOR_EXT, GL_LINEAR_INTERPOLATOR_EXT	**glMaterialInterpEXT()**
GL_ATMOSPHERIC_FUNCTION_EXT, GL_LIGHT_FUNCTION_EXT, GL_RASTERIZER_FUNCTION_EXT, GL_SHADER_FUNCTION_EXT, GL_WARPING_FUNCTION_EXT	**glLoadExtensionCodeEXT()**
GL_TRANSPARENCY_EXT	**glEnable()**
GL_TRANSPARENCY_NONE_EXT, GL_TRANSPARENCY_SCREEN_DOOR_EXT, GL_TRANSPARENCY_MULTIPASS_EXT	**glTransparencyEXT()**
GL_UNSUPPORTED_OPERATION_EXT	many

Table 6: Defined constants

16 Glossary

Interpolator - A method for combining parameter values specified at one or more discrete locations on a primitive being rasterized to generate values for that parameter at all other locations on the primitive where it is not specified.

Light Function - A function which takes as input a set of *light source parameters* and a set of *shader parameters* at a sample, and generates an illumination at that sample which is used by a *shader function* to compute color of the sample.

Light Group - A subset of all existing light instances, used to illuminate specified primitives during shading. Only one light group may be active at any time.

Nonvarying Parameter - A shader parameter whose value is the same for all samples rasterized using that shader.

Rasterizer Function - A function which takes as input a set of *rasterizer parameters* and generates screen-space samples at which the function is visible.

Rasterizer Parameter - A parameter to a rasterizer function.

Sequence Point - Specifies the binding time for a group of rasterizer and shader parameters.

Shader Function - A function, either built-in to PxGL or loaded at runtime, which takes as input a set of *shader parameters* and generates as output a color.

Shader Parameter - An input argument to a shader function.

Shader - An instance of a shader function which binds a subset of the function's parameters to be *nonvarying* for all samples to which the shader is applied.

Uniform Parameter - A shader parameter whose value is the same for all samples rasterized using that shader.

Varying Parameter - A shader parameter whose value may be different in each sample rasterized using that shader.

Rasterizer Boards - Hybrid MIMD/SIMD parallel processors which transform subsets of the primitives making up an image, rasterizing *shader parameters* into local *sample buffers* These buffers are later combined using the image composition network as directed by the rendering recipe.

Rendering Recipe - A list of instructions describing how to combine rasterized *screen regions* containing shading parameters using the image composition network, shade the resulting visible samples, and combine shaded samples into the frame buffer. The rendering recipe is normally defined by state such as viewport size and number of supersamples used for antialiasing.

Sample Buffer - buffers on rasterizer boards which contain samples of locally-visible surfaces and shading parameters for those samples.

17 Credits

The PixelFlow API has developed by discussion among the following people[31]:

Dan Aliaga, Jon Cohen, Lawrence Kestleoot, Anselmo Lastra, Jon Leech, Jonathan McAllister, Steve Molnar, Marc Olano, Greg Pruett, Yulan Wang, and Rob Wheeler (UNC), and Rich Holloway, Roman Kuchkuda, and Lee Westover (HP)

[31] I think this covers everyone who had significant input, but please correct me - JPL.

References

[1] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.1).* Silicon Graphics, Inc., 1995. Unpublished; available at UNC in file:/home/pxfl/doc/software/SGI/glspec.ps

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual.* Addison-Wesley Publishing Company, Inc., 1992.

[3] Steve Molnar, John Eyles, and John Poulton. *PixelFlow: High-Speed Rendering Using Image Composition.* Computer Graphics vol. 26 no. 2, July 1992.

[4] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley Publishing Company, Inc., 1990.

[5] Marc Olano. *PixelFlow Shading Language.* Unpublished; talk to Marc for a copy.

Index

PixelFlow Shading Language
Marc Olano
Revised: 15 September 1997

1 Overview

The PixelFlow shading language is a special purpose C-like language for describing the shading of surfaces on the PixelFlow graphics system. On PixelFlow, some shading function written in the shading language is associated with each primitive. The shading function is executed for each visible pixel (or sample for antialiasing) to determine its color. The language is based heavily on the RenderMan shading language[1].

2 Data

2.1 Built in types

Only a few simple data types are supported. The simplest type is **void**. As with C, it is only used as a return type for functions that have no return value. There is a floating point type, **float**, used for most scalar values. There is a fixed point type, **fixed**, provided for efficiency. And there are literal strings, useful for print formatting[2]. Note that, unlike RenderMan, the string type is not used as an identifier for texture maps, instead a scalar ID is used.

The **fixed** type has two parameters: the size in bits and an exponent. So it is really a class of types, given as **fixed<size, exponent>**. For exponents between zero and the bit size, the exponent can also be thought of as the number of fractional bits. Note however, that an exponent larger than the size or less than zero is perfectly legal. A two byte integer would be **fixed<16, 0>**, while a two byte pure fraction would be **fixed<16, 16>**. It is possible to translate back and forth between the real value and stored value using these equations:

$$\text{real\_value} = \text{stored\_value}^{-\text{exponent}}$$
$$\text{stored\_value} = \text{real\_value}^{\text{exponent}}$$

However, it is much less confusing to always think of the real value. For example, with a **fixed<8,8>**, never think of the value as 128, instead think 0.5. An unspecified fixed point type can also be used, declared simply as **fixed**, and its size and exponent will be chosen automatically[3].

It is also possible to have arrays of these basic types, declared in a C-like syntax (i.e. **float color[3]**). The declaration **float color[3]**, declares color to be a 1D array of three **float**s, **color[0]**, **color[1]**, and **color[2]**. You can also look at **color** as a variable of type **float[3]**, and an equivalent definition would be **float[3] color**. Note the behavior of mixing these two types of definitions: **float[2][3] color_list**, **float[3] color_list[2]** and **float color_list[2][3]** are all equivalent. As with C, it is not necessary to give all of the indices for an array at once. While **color_list[1][1]** is a **float**, **color_list[0]** and **color_list[1]** are each **float[3]** 1D arrays. Where RenderMan uses separate types for points, vectors, normals, and colors, pfman uses arrays.

2.2 Type attributes

As with RenderMan, types may be declared to be either **uniform** or **varying**. A **varying** variable is one that might vary from pixel to pixel, similar **plural** in MasPar's mpl. A **uniform** variable is one that will not vary from pixel to pixel, similar to **singular** in MasPar's mpl. It deserves mentioning again that declaring a variable to be **varying** does not imply that it will vary, only that it might. If not specified, shader parameters default to **uniform** and local variables default to **varying**.

Variables of the **fixed** type may be declared **signed** or **unsigned**. The size of a fixed point type does not include the extra sign bit added by **signed**. So a **signed fixed<15,0>** takes 16 bits. If not specified, all fixed point variables default to **signed**.

---

[1] Upsill, Steve, The RenderMan Companion, Addison-Wesley, 1990.

[2] As of September 13, 1997, strings for calls to printf are not supported.

[3] As of September 13, 1997, automatic fixed point variables are not supported. The sizes produced by automatic fixed types will have to be pessimistic in their size estimation. Error analysis and explicit fixed point sizes is sure to make better use of memory.

There are a number of additional attributes for shader parameters. One transformation type can be given for any parameter. These are **transform_as_vector**, **transform_as_normal**, **transform_as_point**, **transform_as_plane**, or **transform_as_texture**[4]. A parameter can also be declared to be **unit** if it should be unit length[5]. For example, you might declare a parameter

unit transform_as_vector float v[3];

These attributes only affect what happens to the parameter before it is passed to the shader. They do not affect how the parameter is used inside the shader. For example, a **unit** parameter will not remain unit length. These attributes also cannot be used to distinguish versions of an overloaded function.

2.3 User defined types

Aliases can be defined for types with a C-like **typedef** statement. **typedef** is only legal outside function definitions. The **typedef** statement only provides aliases for types, no distinction is made between equivalent types with different names. The statement

typedef float Point[3], Normal[3];

declares **Point** and **Normal** to both be types which can be used completely interchangably with **float[3]**.

3 Expressions

3.1 Operators

The set of operators and operator precedence is fairly similar to that of C (it was based on a grammar for ANSI C). The full list of operators and their precedence is given in Figure 1.

Operation	Associativity	Purpose
`( )`	—	expression grouping
`++ -- []`	—	postfix increment and decrement, array index
`++ -- - !`	—	prefix increment and decrement, arithmetic and logical negation
`( )`	—	type cast
`^`	left	xor / cross product / wedge product[6]
`* / %`	left	multiplication, division, mod
`+ -`	left	addition, subtraction
`&`	left	bitwise and[7]
`|`	left	bitwise or[8]
`<< >>`	left	shift[9]
`< <= >= >`	left	comparison
`== !=`	left	comparison
`&&`	left	logical and
`||`	left	logical or
`?:`	right	conditional expression
`= += -= *= /= ^=`	right	assignment[10]
`,`	—	expression list

Figure 1. Operator precedence

3.2 Operations on arrays[11]

Operations on arrays are defined as the corresponding vecor, matrix, or tensor operation. The unary operations act on all elements of the array. Addition, subtraction, and assignment require arrays of equal

---

[4] As of March 4, 1995, vectors and points are transformed the same and normals and planes are transformed the same.

[5] As of September 13, 1997, **unit** has no affect (parameters declared **unit** are not normalized).

[6] As of March 4, 1995, none of xor, cross product, or wedge product are implemented.

[7] & only works between identical fixed point types.

[8] | only works between identical fixed point types.

[9] As of September 13, 1997, left and right shift are only implemented for varying integer shift values

[10] As of September 13, 1997, ^= is not implemented

[11] As of September 13, 1997, Array cross product, and inverse do not work.

dimension and do the operation between corresponding elements (i.e. **a + b** gives the standard matrix addition of **a** and **b**). The comparison operations also require arrays of equal dimension, though only == and **!=** are defined.

Multiplication between vectors gives a dot product, between vector and matrix, matrix and vector, or matrix and matrix gives the appopriate matrix multiplication. More generally, multiplication between any two arrays gives the tensor contraction of the last index of the first array against the first index of the second array. In other words, for **float a[3][3][3]**, **float b[3][3][3]** and **float c[3][3][3][3]**,

```
c = a * b;
```

is equivalent to

```
float i, j, k, l;
for(i=0; i<3; i++)
    for(j=0; j<3; j++)
        for(k=0; k<3; k++)
            for(l=0; l<3; l++) {
                c[i][j][k][l] = 0;
                for(m=0; m<3; m++)
                    c[i][j][k][l] += a[i][j][m] * b[m][k][l];
            }
```

Division can also be used as a matrix inverse. **1 / a** is the inverse of a square matrix **a** and **b / a** multiplies **b** by the inverse of square matrix **a**.

Finally, the ^ operator gives the cross product between two vectors or the tensor wedge product between two arrays.

3.3 Inline arrays[12]

C-style array initializers are allowed in any expression as an anonymous array. So a 3x3 identity matrix might be coded as **{{1,0,0},{0,1,0},{0,0,1}}**, while the computed elements of a point on a paraboloid might be filled in with **{x, y, x*x+y*y}**.

3.4 Einstein summation notation[13]

Inside any statement block, the uniform integer variables **$1**, **$2**, … are automatically defined. For example for **float a[3], b[3]**, the expression **a[$1] * b[$1]** is equivalent to **a[0]*b[0] + a[1]*b[1] + a[2]*b[2]** (which in this case, is equivalent to **a * b**).

4 Statements

4.1 Compound statements

As with C, anywhere a statement is legal, a compound statement is legal as well. A compound statement is just a list of statements delimited by **{** and **}**.

4.2 Expression statements

Any expression followed by a **;** is a legal statement.

4.3 Standard control statements

Most of the control statements are borrowed directly from C.[14]

if (condition_expression) statement_for_true
if (condition_expression) statement_for_true **else** statement_for_false
while (condition_expression) loop_statement
do loop_statement **until** (condition_expression)**;**
for (initial_expression**;** condition_expr**;** increment_expression) loop_statement
break;
continue;

---

[12] As of September 13, 1997, inline arrays can only have constants for their array elements.
[13] As of September 13, 1997, Einstein summation notation is not implemented
[14] Due to limitations of PixelFlow, the condition_expression's must be **uniform** for all of the looping control statements. The condition for an **if** can be either **uniform** or **varying**.

return;
return return_value_expression**;**

In addition, there are several control statements taken from the RenderMan shading language to aid in shading. They are **illuminance**, **illuminate**, and **solar**.

The **illuminance** statement,

illuminance () statement
illuminance (position_expression) statement
illuminance (position_expression**,** axis_expression**,** angle_expression) statement

acts like a loop over the available light sources. It can also be thought of as an integral over the incoming light. For each light that can hit a pixel at the given position, or can hit a surface at the given position with the given orientation and visibility angle, the light source function is run, returning a light color and intensity that can be used in the statement. The light direction can be accessed using the **px_rc_l** parameter to the shader. The light color can be accessed using the **px_rc_cl** parameter to the shader.

The **illuminate** and **solar** statements,

illuminate (position_expression) statement
illuminate (position_expression**,** axis_angle**,** angle_expression) statement
solar (axis_angle**,** angle_expression) statement
solar ( ) statement

provide the information the **illuminace** statement uses to tell if a light source function should be run or not. They can also be thought of as conditional statements that only execute the associated statement if the current pixel position falls within the light's area. The four statements above correspond to a point light, a spot light, a directional light, and an ambient light[15].

4.4 Declaration statements

Variable declarations can occur anywhere a statement can. They consist of a type and a list of new variable names to declare. Each variable name can have additional array dimensions and an expression for the initial value.

float a[3], b=2*x, c;

declares **a** as an uninitialized 1D **float** array with 3 elements, **b** as a **float** with an initial value twice whatever is in the **x** variable at the declaration time, and **c** as an uninitialized **float**.

Each compound statement defines a new scope, so variables can be redefined within a compound statement without conflicting with function or variable names in other scopes. It is illegal, however, to have a variable in any scope with the same name as any user defined type. This is true even if the **typedef** occurs after the variable declaration.

5 Functions

5.1 Overloading

Function overloading similar to C++ is supported. So functions of the same name that can be destinguished by their input parameters are considered distinct. This provides the ability to have seperate versions of functions for **uniform** and **varying** parameters, **float** and **fixed**, or different fixed point types. Note that functions cannot be overloaded based on their return parameters and operator overloading is not supported.

5.2 Definition

A function definition gives the return type, name, parameters, and body that define the function. Function definitions cannot be nested. By default, function parameters and return types are **uniform**. A simple function definition:

```
float factorial(float n) {
  if (n > 1)
    return n * factorial(n);
```

[15] I don't really like the way this works in RenderMan. Is there a use to placing some of the light code within an **illuminate** statement and some outside? Is it too specialized for a couple of particular light types? Whether I understand it or not, it's there.

```
    else
        return 1;
}
```

The formal parameters to a function have their own scope level between the global scope and the function body, so their names can hide the global function names. As with variables, it is illegal to have a function or parameter with the same name as a user defined type, regardless of where in the source the **typedef** occurs.

5.3 Shading functions[16]

There are several special return types to indicate that a function has some special rendering purpose and may need to be called by the PixelFlow rendering library. These are **primitive**, **interpolator**, **surface**, **light**, and **image**. A **primitive** function computes which pixels are in some rendering primitive like a triangle or sphere; an **interpolator** function computes the value for some shading parameter across a number of pixels; a **surface** function computes the shading on a surface (the archetypal shading function); a **light** function computes the color and intensity of a light; and an **image** computes the final color and location of the image pixels (handling image warping, fog effects, etc.). For all of these functions, each parameter can have a default value in case the graphics library is not given a value for that parameter. These are given just by putting an **= value** (just like variable initialization) in the parameter list. These default values must be compile-time constants. It is perfectly legal to call a surface shading function from inside another surface shading function[17]. In this case, only one illuminance statement can occur in either the original surface shader or any called by it.

5.4 Prototypes

Any function that is to be used before it is defined, or that is defined in a different source file, must have a prototype. A function prototype is just like a function definition, but with a **;** instead of the function body

```
float factorial(float n);
```

5.5 Internal details and External linkage

The **pfman** shading language compiler turns shading language source code into C++ source code that must be further compiled with a C++ compiler. The function definitions created by the compiler and function calls made by it correspond directly to C++ function definitions and function calls. It is possible (and supported) to call C++ functions from shading language functions and to call shading language functions from C++. This facility is limited to functions using types that the shading language supports.

Pfman adds some additional arguments added by the compiler. The new first argument is a pointer to the PixelFlow IGCStream where the instruction stream for the pixel processors should go. The new second argument is a pointer to a PixelFlow GLStage class, which contains information about the rendering context. The new third argument is a pointer to the PixelFlow pixel memory map class. For functions with a varying return value, the new third argument is the address for the return value. All the other arguments follow. There are C++ classes for varying float and fixed parameters giving their address, and in the case of fixed parameters, their size and binary point position. Details of these types and the prototypes for the different kinds of shading functions are beyond the scope of this document.

Standard C or C++ functions can be used by pfman by prefixing their prototype with **extern "C"** or **extern "C++"**. All of the uniform math library routines are declared this way. These tell pfman not to add the extra function parameters. Similarly, pfman functions that contain only uniform operations can be declared **extern "C"** or **extern "C++"** for use by code outside of pfman.

---

[16] As of September 13, 1997, only surface and light are supported.

[17] As of September 13, 1997, it is not possible to call either surface shaders from inside surface shaders.

Implementing PixelFlow Shading

Marc Olano

In the previous sections of this chapter, we covered the interface seen by both application and shader writers. In this section, we cover the basic knowledge of the PixelFlow hardware required to understand the implementation issues. For more details on the PixelFlow architecture, see [Molnar91][Molnar92][Eyles97]. We also cover some intermediate levels of abstraction between PixelFlow and an abstract graphics pipeline and explain how our procedural stages fit into the real PixelFlow pipeline.

Our abstract pipeline consists of procedures for each stage in the rendering process. Since these can be programmed completely independently, it is possible (and expected) that a particular hardware implementation may not have procedural interfaces for all stages. While PixelFlow is theoretically capable of programmability at every stage of the abstract pipeline, our implementation only provided high-level language support for surface shading, lighting, and primitives. The underlying PixelFlow software includes provisions for programmable testbed-style atmospheric and image warping functions, but we did not supply any special-purpose language support for these.

1. High-level view

PixelFlow consists of a host workstation, a number of rendering nodes, a number of shading nodes, and a frame buffer node. The hardware and lower level software handle the scheduling and task assignment between the nodes, so we can consider the flow of data in the system as the pipeline shown in Figure 1. This view is based on the passage of a single displayed pixel through the system. Neighboring pixels may have been operated on by different physical nodes at each stage of this simplified pipeline. This will be covered in more detail later in this chapter. For the purposes of mapping the abstract pipeline onto PixelFlow, the simplified view of the physical PixelFlow pipeline is sufficient.

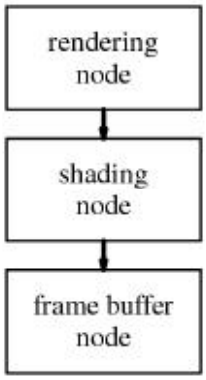

Figure 1. Simplified view of the PixelFlow system

1.1. Applying the abstract pipeline

The mapping of an abstract pipeline onto PixelFlow is shown in Figure 2. This abstract pipeline is divided into stages based on a set of logical rendering tasks. Contrast this with the abstract model presented later in Chapter 8, in which a single shader spans several computational units.

The modeling, transformation, primitive, and interpolation stages are handled by the rendering node. The shading, lighting, and atmospheric stages are handled by the shading node. Finally, the image warping stage is handled by the frame buffer node.

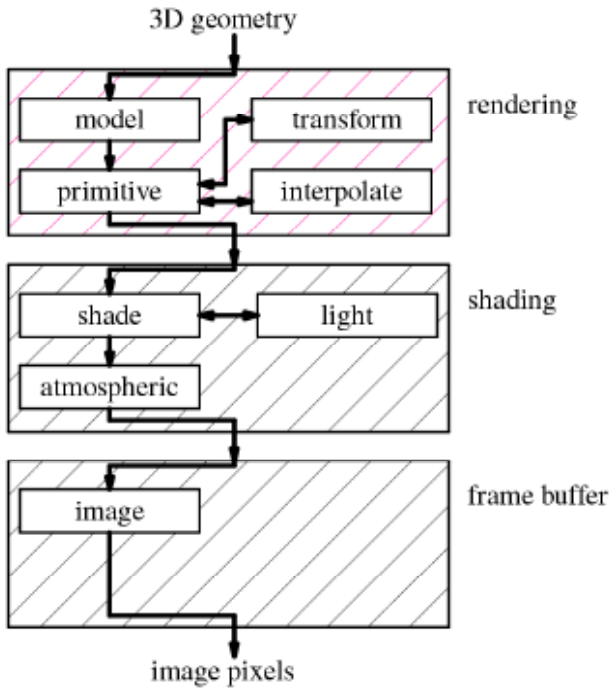

Figure 2. Procedure pipeline.

When mapping the abstract pipeline onto PixelFlow, we maintain the interfaces to the pipeline stages. Thus, the procedures written for PixelFlow should look exactly the same as the procedures written for a different machine with a different organization. The code for each stage is written just as if it were part of some arbitrary rendering system implementing the abstract pipeline.

It is important to notice that the abstract pipeline only provides a conceptual view for programming the stages. It allows the procedure programmer to pretend that the machine is just a simple pipeline instead of a large multicomputer. The real stages do not need to be executed strictly in the order given (and, in fact, are not). The user writing code for one of the stages does not need to know the differences between the execution order given in the abstract pipeline and the true execution order. The mapping of the abstract pipeline onto PixelFlow exhibits several different forms of this.

The first example is the overall organization of the processes on PixelFlow. PixelFlow completes all of the modeling, transformation, primitives, and interpolation in the rendering nodes before sending the shading parameters for the visible pixels on to a shading node. PixelFlow then completes all of the shading, lighting, and atmospheric

effects before sending the completed pixels on to the frame buffer node for warping. On a different graphics architecture, it might make more sense to complete all of the stages for every pixel in a primitive before moving on to the next primitive. Either choice appears the same to users who write the procedures. The abstract pipeline does not include information about the stage scheduling to allow just such implementation flexibility.

The procedures running on the PixelFlow rendering nodes provide another example. The abstract pipeline presents transformation, primitive, and interpolation as if they were a sequential chain of processes. On PixelFlow, the primitive stage drives transformation and interpolation. A procedural primitive function is invoked for each primitive to be rendered. This function calls both transformation and interpolation functions on demand as needed. The results stored for each pixel include its depth, an identifier for which procedural shader to use and the shading parameters for that procedural shader. Once again, the user writes procedures as if they were independent sequential stages and is not aware of the true ordering within the PixelFlow implementation.

The final example is with the shading and lighting stages. The abstract pipeline presents shading and lighting as if the shading stage called the lighting stage for each light. On PixelFlow, the linkage between these stages is not as direct. These two stages run with an interleaved execution scheduled by the PixelFlow software system. This interleaving is explained in more detail in [Olano98]. And again, the interleaved scheduling is hidden from anyone who writes a shading or lighting procedure.

1.2. Parameter manager

Supporting this pipeline is a software framework that handles the details of the rendering process and the communication between the programmable procedures. That communication is assisted by a global parameter manager, implemented on PixelFlow by Rich Holloway. The parameter manager allows each node in the system to find values or pixel memory addresses of the parameters. It also keeps track of other attributes of each parameter - its type and size, default values, whether it needs to be transformed (and how), etc. Whenever a procedure is compiled, an extra *load function* is generated. This load function is run when the procedure is loaded by the application. The load function registers all of the parameters used or produced by the procedure. The parameter manager collects this information and makes sure each parameter is available when it is needed. This global parameter space is similar to the shared memory "blackboard" idea used by MENV [Reeves90].

2. Low-level view

The PixelFlow system data-flow was show in Figure 1. A view of the hardware at that level was sufficient to understand how the abstract pipeline maps onto PixelFlow. We must delve deeper to understand some of the issues that impacted our implementation. Where Figure 1 showed only a single stage for rendering and shading, PixelFlow may have many nodes (see Figure 3). There are also two networks connecting the nodes in the PixelFlow system, the geometry network and composition network. The rendering nodes and shading nodes are identical, so the balance between rendering performance and shading performance can be decided on an application by application basis. The frame buffer node is also the same, though it includes an additional *daughter card* to produce

video output.

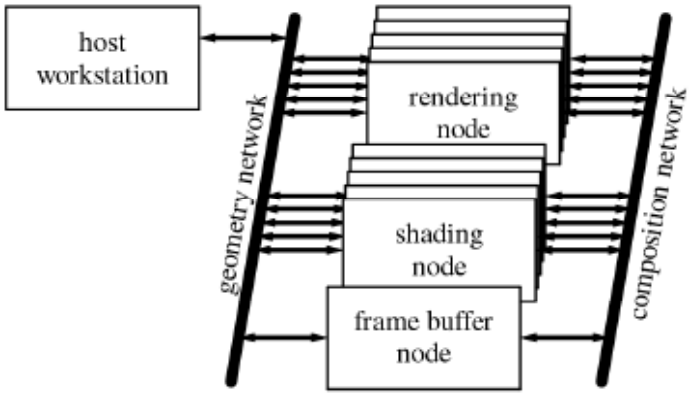

Figure 3. PixelFlow machine organization.

Each rendering node is responsible for rasterizing an effectively randomly chosen subset of the primitives in the scene. The rendering nodes work on one 128x64 pixel *region* at a time (or 128x64 image samples when antialiasing). Many of our examples and tests are based on either an NTSC video screen size of 640x512 pixels with four samples per pixel, or a high-resolution screen size of 1280x1024 pixels. There are 40 regions in an NTSC image with no antialiasing. With antialiasing using four samples per pixel, the NTSC image has 160 regions. Without antialiasing, the high-resolution image also has 160 regions. Therefore, our target is to be able to handle 160-128x64 regions at NTSC video rates of 30 frames per second.

Since each rendering node has only a subset of the primitives, a region rendered by one node will have holes and missing polygons. The different versions of the region are merged using *image composition*. PixelFlow includes a special high-bandwidth network called the *composition network* with hardware support for these comparisons. As all of the rendering nodes simultaneously transmit their data for a region, the network hardware on each node compares, pixel-by-pixel, the data it is transmitting with the data coming in from the upstream nodes. It keeps only the closest of each pair of pixels to send downstream. By the time all of the pixels reach their destination, one of the shading nodes, the composition is complete.

Once a shading node has received the data, it does the surface shading for the entire region. The technique of shading after the pixel visibility has been determined is called *deferred shading* [Deering88][Ellsworth91]. Deferred shading only spends time shading the pixels that are actually visible, and allows us to do shading computations for many more pixels in parallel. With non-deferred shading, each primitive is shaded separately. With deferred shading, all primitives in a region that use the same procedural shader can be shaded at the same time.

In a PixelFlow system with n shading nodes, each shades every n^{th} region. Once each region has been shaded, it is sent over the composition network (without compositing) to the frame buffer node, where the regions are collected and displayed.

3. PixelFlow node

The nodes on PixelFlow all look quite similar (See Figure 4). Each node of the PixelFlow system has two RISC processors (HP-PA 8000's), a 128x64 custom SIMD array of pixel processors, and a texture memory store. Only the rendering nodes make use of the second RISC processor, where the primitives assigned to the node are divided between the processors. The existence of the second RISC processor does not impact our implementation, so we can take the simplified view that there is only one processor on the node and let the lower level software handle the scheduling between the physical processors. The RISC processors share 128 MB of memory, while each pixel processor has access to 256 bytes of local memory. The texture memory exists in several replicated banks for access speed, but the apparent size is 64 MB.

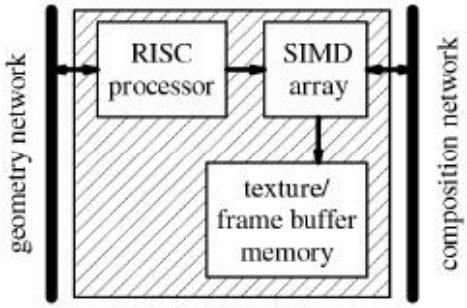

Figure 4. Simple block diagram of a PixelFlow node

Each node is connected to two communication networks. The geometry network, carries information about the scene geometry and other data bound for the RISC processors. This network is four bytes wide and operates at 200 MHz. It can simultaneously send data in both directions, giving a total bandwidth of 800 MB/s in each direction. The composition network handles transfers of pixel data from node to node. It also operates in simultaneously in both directions at 200 MHz. However, the composition network is 32 bytes wide, giving a bandwidth of 6.4 GB/s in each direction. Four bytes of every transfer is reserved for the pixel depth, reducing the effective bandwidth to 5.6 GB/s.

3.1. Compiler target

Every procedural stage on PixelFlow has a testbed-style interface, which allows new stage procedures to be created using the internal libraries of the PixelFlow system. Writing code new procedures using this interface requires a deep understanding of the implementation and operation of PixelFlow, more than will be provided in this dissertation. We provide a high-level, special-purpose language so the users who write new procedures will not need to have that level of understanding of PixelFlow. It also makes rapid prototyping and porting procedures to other systems possible.

The compiler for our special-purpose language produces C++ code that exactly conforms to the testbed interface. This code consists of two functions, a load function (mentioned in section 1.2), and the actual code for the procedure. The code for the procedure is run on the RISC processor and includes embedded *EMC functions*. Each EMC function puts one SIMD instruction into an instruction stream buffer. The EMC prefix that appears on all of these functions stands for *enhanced memory controller*, from the Pixel-Planes SIMD array's origin as a processor-enhanced memory; we use it here just to identify the functions that generate the SIMD instruction stream.

When the C++ code for a procedure is run, the result is a buffer full of instructions for the SIMD array. This instruction stream buffer can be sent to the SIMD array several times without requiring the original C++ code to be re-executed.

There are two forms of EMC function used in PixelFlow. The form used on the shading nodes checks the available space in the instruction stream buffer with each instruction and can re-allocate the buffer on the fly. The form used in the rendering nodes requires a buffer of sufficient size to be allocated at the beginning of the procedure. The reason for this difference, and the issues that result, are discussed in Section [Olano99].