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 $Source : /tmp_mnt/net/hydra/pp0/doc/software/opengl/tex/RCS/pxgl.tex, v$. 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_LIGHTi`, 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