agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the GL (by **gl**, `GL_`, and `GL`, respectively in `C`) to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

2.1.1 Numeric Computation

The GL must perform a number of numeric computations during the course of its operation.

Implementations of the Common profile will normally perform computations in floating-point, and must meet the range and precision requirements defined under **"Floating-Point Computation"** below.

Implementations of the Common-Lite profile will normally perform computations in fixed-point, and must meet the more relaxed range and precision requirements defined under **"Fixed-Point Computation"** below. However, Common-Lite implementations are free to use floating-point computation if they wish.

Floating-Point Computation

We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least 2^{32}; the maximum representable magnitude for colors or texture coordinates must be at least 2^{10}. The maximum representable magnitude for all other floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results. The identities specified above do not hold if the value of x is not a floating-point number.

Fixed-Point Computation

2.9.1 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options supported for client-side vertex arrays.

The client state associated with each vertex array type includes a buffer object binding point. The commands that specify the locations and organizations of vertex arrays copy the buffer object name that is bound to ARRAY_BUFFER to the binding point corresponding to the vertex array of the type being specified. For example, the **NormalPointer** command copies the value of ARRAY_BUFFER_BINDING (the queriable name of the buffer binding corresponding to the target ARRAY_BUFFER) to the client state variable NORMAL_ARRAY_BUFFER_BINDING.

Rendering commands **DrawArrays** and **DrawElements** operate as previously defined, except that data for enabled vertex arrays are sourced from buffers if the array's buffer binding is non-zero. When an array is sourced from a buffer object, the pointer value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units[2].

It is acceptable for vertex arrays to be sourced from any combination of client memory and various buffer objects during a single rendering operation.

2.9.2 Array Indices in Buffer Objects

Blocks of array indices may be stored in buffer objects with the same format options that are supported for client-side index arrays. Initially zero is bound to ELEMENT_ARRAY_BUFFER, indicating that **DrawElements** is to source its indices from arrays passed as the *indices* parameters.

A buffer object is bound to ELEMENT_ARRAY_BUFFER by calling **BindBuffer** with *target* set to ELEMENT_ARRAY_BUFFER, and *buffer* set to the name of the buffer object. If no corresponding buffer object exists, one is initialized as defined in section 2.9.

The commands **BufferData** and **BufferSubData** may be used with *target* set to ELEMENT_ARRAY_BUFFER. In such event, these commands operate in the same fashion as described in section 2.9, but on the buffer currently bound to the ELEMENT_ARRAY_BUFFER target.

[2]To resume using client-side vertex arrays after a buffer object has been bound, call **BindBuffer**(ARRAY_BUFFER,0) and then specify the client vertex array pointer using the appropriate command from section 2.8.

the plane equation coefficients in eye coordinates. All points with eye coordinates $(\, x_e \quad y_e \quad z_e \quad w_e \,)^T$ that satisfy

$$(\, p'_1 \quad p'_2 \quad p'_3 \quad p'_4 \,) \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_PLANE`i where i is an integer between 0 and n; specifying a value of i enables or disables the plane equation with index i. The constants obey `CLIP_PLANE`i = `CLIP_PLANE0` + i.

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$ and $\mathbf{P}_2$, then t is given by

$$\mathbf{P} = t\mathbf{P}_1 + (1-t)\mathbf{P}_2.$$

The value of t is used in color and texture coordinate clipping (section 2.12.7).

If the primitive is a triangle, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Clipping may cause triangle edges to be clipped, but because connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a triangle, creating a more general *polygon*.

If it happens that a triangle intersects an edge of the clip volume's boundary, then the clipped triangle must include a point on this boundary edge.

A line segment or triangle whose vertices have w_c values of differing signs may generate multiple connected components after clipping. GL implementations are

Let the colors assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.11) for a clipped point $\mathbf{P}$ is used to obtain the color associated with $\mathbf{P}$ as[4]

$$\mathbf{c} = t\mathbf{c}_1 + (1-t)\mathbf{c}_2.$$

(Multiplying a color by a scalar means multiplying each of R, G, B, and A by the scalar.) Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

Texture coordinates must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

2.12.8 Final Color Processing

Each color component (which lies in $[0, 1]$) is converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer.

Because a number of the form $k/(2^m - 1)$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of color components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and the color was specified with unsigned byte or integer values. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the command defining it: if m is less than the number of bits b with which the component was specified, then the converted value must equal the most significant m bits of the specified value; otherwise, the most significant b bits of the converted value must equal the specified value.

[4]Since this computation is performed in clip space before division by w_c, clipped colors and texture coordinates are perspective-correct.

version 1.1.10, draft of 2007/02/06 Noted in section 2.10.3 that normal vectors are treated as row vectors transformed by matrix postmultiplication, which may be unfamiliar to some graphics programmers. Removed X Window System trademark information from the copyright pages.

version 1.1.10, draft of 2007/03/31 Document scaling of integer to fixed-point parameters. Polygon smooth mode is not supported. Front and back material colors exist in terms of the API, but are constrained to always have the same values. General polygons are not supported. Remove references to texture borders. Many other minor fixes and clarifications from WG review - see Khronos member Bugzilla bugs 1247, 1257, 1258, 1259.

version 1.1.10, draft of 2007/04/04 Clarify that floating-point identities do not hold for infinite or NaN values in section 2.1.1. Remove advice about preferred vertex buffer object formats in section 2.9.1. Mandate point sprite clipping (do not allow scissoring) in section 2.11, pending Working Group resolution of the open issue. Clarify that color and texture coordinate clipping defined in section 2.12.7 is already perspective correct.