OpenGL Conformance

Chris Frazier, Paul Ho, and Lesley Kalmin

1.0 Goal.

SGI has derived a set of conformance tests in order to guarantee predictability among various implementations of OpenGL. The conformance tests were designed to insure that an implementation is qualitatively consistent with the intent of the OpenGL specification, even if it is not quantitatively identical. The OpenGL specification dictates how an image is rendered not by pixel exactness but rather through a description of good procedures. The same idea applies to the way the conformance tests examines an implementation. They do not do pixel exact comparisons but rather scrutinize a rendered image for those specific details that are defined in the OpenGL specification. Wherever there is an allowance for latitude in the OpenGL specification on the way an implementation can behave, the conformance tests will also reflect this allowance.

1.1 What the conformance tests don't do.

They are not an exhaustive test suite:

Individual licensees are responsible for ensuring that their implementations function properly. Implementors are of course free to use the conformance tests to help locate bugs, as long as they realize that this is not the fundamental purpose of the tests.

They do not necessarily run quickly:

Little effort was made to optimize these tests. More emphasis was placed on readability and portability.

They do not guarantee exact conformity between different OpenGL implementations:

The OpenGL specification is intentionally permissive about how certain problems should be solved and the conformance tests are designed to respect this permissiveness. In order to accomplish this, the tests allow a large amount of discrepancy. Users are cautioned not to expect identical images from different OpenGL implementations, but it is reasonable for them to expect that an implementation which diverges greatly from other implementations or does not include required features will fail the tests. The tests aim for a common ground. For example: different OpenGL implementations may render lines using different algorithms. However, an implementation will not conform if x-major lines do not have the correct number of pixels per column in them.

2.0 Parts.

The Toolkit or TK:

OpenGL was intended to be implemented in a way which is independent of a windowing system. In order to facilitate this, the conformance tests run through a simple windowing system interface. The TK defines this interface. Only the TK should need any real porting work (very minimal windowing concepts).

Coverage tests:

The coverage test set is comprised of three executables: covgl, covglu and covglx, corresponding to the OpenGL, the OpenGL Utility Library and the GLX, respectively. Coverage tests indicate if all required API routines within an implementation may be called with all enumerated parameters. These tests do not indicate if the functions are working correctly, just that they exist. At the end of a coverage test, the implementation receives a passes or fail report.

Primitive testing:

This conformance test renders four basic primitives, points, lines, triangles and quads using 11 rendering modes. It does not test for rendering correctness, just that no errors occur (crashes or internal error).

Conformance tests:

This conformance test set is comprised of two executables: conform and conformx, corresponding to the OpenGL and GLX, respectively. Conformance tests examine OpenGL functionality.

3.0 How to run the tests.

The conformance tests can be run either standalone or through scripts.

Conformance test script:

conform.x - this script assumes that a GLX server exists. This script runs conform, conformx, covgl, covglu, covglx, and primtest. It queries GLX for available visuals. This script is used for official testing.

For convenience and/or testing purposes, we provide the following two scripts:

conform.base1 - this script runs conform, covgl, covglu and primtest. It queries GLX for available visuals.

conform.base2 - this script runs conform, covgl, covglu and primtest. It runs four visual types, RGBA single buffer with stencil, depth and accumulation buffers, RGBA double buffer with stencil, depth and accumulation buffers, Color index single buffer with stencil, depth and accumulation buffers and color index double buffer with stencil, depth and accumulation buffers.

Conformance test standalone:

Each of the individual tests in the conformance test suite may be run individually. They can be used in this fashion to determine why a test failed an implementation. The conformance test suite has a set of command line arguments. The coverage tests only have adjustable verbosity level while the conformance tests have a wide variety of option (setting random number seed, verbosity, etc.). To get a complete listing of available command line option, execute each test with the -h option.

Execution Time:

The tests suite takes approximately 14 hours to run on an Indigo Entry. Run time will depend not only upon the implementation and machine, but also upon the number of visuals exported for OpenGL.

4.0 The TK (ToolKit).

Since OpenGL may run on a number of different platforms, we need a way of making the conformance tests (conform, conformx, covgl, covglu, covglx, primtest) and samples easy to port. In order to accomplish this, we have developed an interface (the TK) to separate the application from the windowing system. The interface follows a basic windowing system philosophy but is not particular to any windowing system. This interface consists of a non-changeable API and a library containing windowing system specific code. All of the conformance tests have been written using this interface, so porting them only involves coding the windowing system specific library.

To port the windowing system specific library to a particular windowing system, a "shell" must first be written. We define a shell as the collection of core functions needed by an application for it to run properly under a particular windowing system. This shell should divided into three parts, each with an API entry point. There should be an initialization part (which will be called by tkNewWindow), the main execution part (which will be called by tkExec) and a termination part (which will be called by tkCloseWindow).

The main execution part is the most important. Once the application is done initializing itself, it will pass control to this code. The job of this code is to perform all the necessary background work needed to keep the application running. On most systems, this involves handling system and event messages. It should deal with window-specific event messages internally and pass TK defined event messages to the application (the code must translate the window-specific event messages into TK equivalents).

An application's basic flow of execution is as follows:

1) Application startup.

2) Application requests a window. Perform window-specific initialization.

3) Application initialization.

4) Application relinquishes control to window-specific main event loop procedure, supplying it with an event handling procedure.

5) The window-specific event loop procedure continuously gets window-specific event messages. It handles those not defined for the TK. It translates those which are defined for TK into TK event messages and pass these to the application's event handling procedure.

6) The application's event handling procedure processes the TK event messages accordingly.

All of the TK's API calls must exist for the applications to compile, however, not all of the functions need be implemented. The samples require almost all of the functions, whereas the basic conformance tests require only **tkNewWindow**, **tkCloseWindow, tkExec** and **tkQuit**.

The data structure definitions are in tk.h, and the X Windowing System implementation of this routine is in tkx.c.

4.1 TK functions.

tkCloseWindow: closes a window.

void tkCloseWindow(void);

tkDrawFont: draws a specified string.

long tkDrawFont(char *fontName, long x, long y, char *string, long len);

returns: 0 (fail) or 1 (success)

tkGet: requests window-specific information.

long tkGet(long item, void *data);

item - enums.

TK_CITORGB - convert pixel from color index to RGB format.

TK_COLORMAPSIZE - colormap size.

TK_MOUSELOC - mouse x and y location.

TK_RGBCOLORMAP - RGB colormap.

TK_SCREENIMAGE - reads framebuffer through window system.

TK_VISUALIDS - visual IDs available.

TK_WINDLOC - window upper left corner.

data - pointer to storage area for requested info.

for TK_CITORGB, TK_ColorRec.

for TK_COLORMAPSIZE, TK_ColorMapSizeRec.

for TK_MOUSELOC, TK_MouseLocRec.

for TK_RGBCOLORMAP, TK_RGBColorMapRec.

for TK_SCREENIMAGE, TK_ScreenImageRec.

for TK_VISUALIDS, TK_VisualIDsRec.

for TK_WINDLOC, TK_WindowLocRec.

returns: (none)

tkExec: installs application's event handling procedure.

void tkExec(long (*Func)(TK_EventRec *ptr));

func - event handling function.

ptr - TK_EventRec pointer.

typedef struct _TK_EventRec {

long event;

TK_EVENT_EXPOSE.

TK_EVENT_CONFIG.

TK_EVENT_KEY.

TK_EVENT_MOUSEDOWN.

TK_EVENT_MOUSEUP.

TK_EVENT_DRAW.

long data[4];

} TK_EventRec;

returns: 1 (continue) or 0 (exit event loop)

tkImageLoad: opens a file and reads raw image data.

TK_ImageRec *tkImageLoad(char *fileName);

returns: TK_ImageRec pointer.

typedef struct _TK_ImageRec {

long sizeX, sizeY;

unsigned char *data;

} TK_ImageRec;

tkLoadFont: loads a specified font and creates OpenGL display lists for each character.

long tkLoadFont(char *fontName, long *width, long *height);

returns: 0 (fail) or 1 (success)

tkNewWindow: performs machine dependent initialization and opens a window.

long tkNewWindow(TK_WindowRec *window);

typedef struct TK_WindowRec {

char name[100];

long x, y;

long width, height;

long type;

TK_WIND_REQUEST - info contains window mask.

TK_WIND_VISUAL - info contains a visual id.

long info;

TK_WIND_RGB or TK_WIND_CI.

TK_WIND_DB or TK_WIND_SB.

TK_WIND_STENCIL.

TK_WIND_Z.

TK_WIND_AUX.

TK_WIND_ACCUM.

long eventMask;

TK_EVENT_EXPOSE.

TK_EVENT_CONFIG.

TK_EVENT_KEY.

TK_EVENT_MOUSEDOWN.

TK_EVENT_MOUSEUP.

TK_EVENT_DRAW.

} TK_WindowRec;

returns: 0 (fail) or 1 (success)

tkPostEvent: posts an event. It holds only one event - the last one posted.

void tkPostEvent(TK_EventRec *ptr);

tkQuit: terminates a program.

void tkQuit(void);

tkSet: Used to set machine specific information.

void tkSet(long item, void *data);

item- enums.

TK_RGBCOLORMAP - sets up a full color map.

TK_GREYSCALEMAP - sets up a grey scale color map.

TK_FOGMAP - sets up a color index colormap for use with fog.

TK_CITORGB - stores RGB color.

data - pointer to storage area for requested info.

for TK_RGBCOLORMAP, TK_RGBColorMapRec.

for TK_RGBGREYSCALEMAP, (none).

for TK_FOGMAP, TK_FogRec.

for TK_CITORGB, TK_ColorRec.

tkSwapBuffers: swaps display buffer if double-buffering is supported and enabled.

void tkSwapBuffers(void);

5.0 Coverage.

Coverage testing consists of 2 parts: **coverage tests** and **primitive testing**.

5.1 Coverage tests.

Coverage tests are composed of three executables: **covgl**, **covglu**, and **covglx**.

covgl - verify functions in the OpenGL library.

covglu - verify functions in the GL Utility Library.

covglx - verify functions in GLX library.

The fundamental principle of operation behind all these tests is the same. Each test contains all required API calls for the particular library it is trying to test. If any of the calls accepts a set of enumerated values, that routine will be repeatedly called with each of these enumerated values in turn. The other parameters are set to benign values. These tests do not ascertain if the routines perform correctly. They merely tests if all the routines exists and that all enumerated parameters are handled. The tests will print “passed” or “failed” after running.

5.2 Primitive testing.

Primitive testing is done by the executable **primtest**. Primtest tests an implementation’s ability to render primitives in different rendering modes. The primitives used are a point, a line, a triangle and a quad. Initially, we tried using all primitives, but the test took a prohibitively long time to run. We chose these four primitive because most other primitives are likely to be build on top of them.

Primtest uses 11 rendering modes - anti-aliasing, alpha blending, color blending, depth, dither, fog, hints, lighting, scissor, stencil and texture. Primtest does not examine the rendered output, but it does check for internal errors with the glGetError() call. It will print pass or fail after running.

6.0 Conformance tests.

The conformance tests (conform and conformx) verify the presence of features which the OpenGL specification requires as part of an OpenGL implementation. The tests also verify the correctness of the computations required by these features. The tests allow for more latitude than the OpenGL specification while still requiring consistency among different implementations.

6.1 Shell.

Conform and conformx are made up a collection of individual tests. All of these individual tests are controlled by one driver routine called the shell. The shell saves the implementation state with a glPushAttrib() call, sets up the proper Equivalent Computation Paths (ECP) for the current test, executes the test, checks for internal errors with a glGetError() call, resets the implementation state with a glPopAttrib() call and finally verifies that the implementation has indeed been reset to its default state.

Note that the ECP code may execute a test many times to cover all possible path combinations. The shell also handles administrative duties such as checking for minimum buffer depth requirements for those tests that need these, correct buffer color mode requirements, setting up random number seed between tests, etc. In addition to the many driver routines in the shell (test driver, ECP driver, etc.), there are a number of utility files. These utility files contain code that is reused in many similar tests (lighting and transform tests for example). Also, most of the standard c library functions are defined in util.h to facilitate quick routine substitution.

Several command line arguments are available to the user for customized executions.

`-1 <testfile.c>` performs the one test with the given filename.

`-d <visual ID>` execute tests with window system specify visual ID.

`-D [1-4]` execute tests with predefined visual ID. 1 and 3 are single-buffered RGB or color index visuals respectively, 2 and 4, are double buffered RGB or color index visuals respectively.

`-f <filename>` performs only those tests which are listed in the test list file. Use the -g option to create a test list file.

`-g <filename>` generates a test list file. Used in conjunction with the -f option to run only a subset of the entire test suite.

`-G <filename>` generates a test list file. Tests are commented out. Used in conjunction with the -f option to run only a subset of the entire test suite.

`-h` help.

`-p [0-4]` set ECP level. 0 = ECP disabled; 1 = all ECP on; 2 = one ECP on, all others off (each test is repeated for all ECPs); 3 one ECP off, all others on (each test is repeated for all ECPs).

`-r <seed>` execute tests with new random number seed. The default seed is 1.

`-s` disable state checking.

`-v [0-2]` set verbosity level. 0 = Minimal (pass or fail report), 1 = normal (includes test list), 2 = full (includes error messages).

`-x` force failure on visual tests. This is done by disabling the ReadScreen() call.

6.2 Acceptable errors.

The OpenGL specification allows for a large amount of difference between the behavior of different implementations. In some cases it also states explicitly how much an implementation may differ from the ideal and still be found conformant. The conformance tests use a number of specific error epsilons. These error epsilons may be sensitive to the configuration of the implementation, as in the case of color error epsilons, or they may be a fixed bound which relates to some portion the OpenGL specification, as in the case of floating point error epsilon.

The following are the smallest error epsilons used. Many tests will use these values to construct larger error margins. Each test's documentation will state if a larger error margin is used.

Floating point number zero = 1 / pow(2, 13). This value is used when two floating point numbers are compared.

Color index = 0.5 + (zero epsilon). This is used to distinguish between different color indices. It assumes little computational floating point error.

RGBA = 1 / (pow(2, number of color bits) - 1) + (zero epsilon). This number is used to distinguish between different shades of RGBA color.

Stencil plane = 1 / (pow(2, number of stencil plane bits) - 1) + (zero epsilon).

For placement of pixels, we generally allow a error of at least a half of a pixel. For specular exponents and spot exponents, we allow at least 5% error. This is small enough that a poorly implemented look-up table will fail, but it is large enough to encourage good optimizations. Fog exponent errors are expected to be small since the fog functions have small derivatives over their effective domains.

6.3 Architectural issues.

Conformance tests repeat the list of tests (from TESTLIST) for each OpenGL enabled visual, for each path level, and for both direct and remote renderers, if both are available. The test loop executes as follows:

```
for each DIRECT/INDIRECT renderer
    for each PATH LEVEL
        for each VISUAL
            execute each test in TESTLIST
```

Many tests require the existence of particular buffers, or minimum depths of given buffers to run. If a visual does not provide the minimum requirements, the test will be skipped for that visual, and the log will show a message indicating that the test is not run on that visual. This does not mean the test failed, merely that it is not appropriate for that visual.

The test mustpass.c (also referred to as the level 0 test) is run without the paths (Equivalent Computational Paths) enabled. In other words, it is executed only with default state settings.

Tests which use only one visual (Coverage Tests and Primitive Test) will first attempt to get a single-buffered visual and, if that is unavailable, use a double-buffered visual.

6.4 Conform tests.

The information printed below can also be found at the top of each file in the conformance test suite directory.

```
*************************************************************************
accum.c
Accumulation Buffer Test.

Description -
    Tests the five operations of the accumulation buffer:
    GL_LOAD, GL_ADD, GL_MULT, GL_ACCUM and GL_RETURN. Also
    checks for proper clamping of values sent to and returned
    from the buffer. AccumMaskTest() checks that the color mask
    is applied on GL_RETURN, and does not affect the other
    accumulation ops.

    For each operation, the entire color buffer and
```

accumulation buffer are cleared with a different value in each color component. The color buffer is read back so that the actual value is taken to be the current color buffer value, to minimize error later on. The value parameter is set by Random(), with a range chosen to keep the resulting buffer values mostly within range.

The operation (GL_LOAD, GL_ADD, GL_MULT, GL_ACCUM) is then repeated a number of times (currently 10) before a GL_RETURN operation is executed, and the color buffer value is compared against the expected value. The OpenGL specification says that the accumulation buffer values are in the range [-1,1] and if they go out of this range, the result is undefined. After each glAccum() call the expected value for the accumulation buffer is calculated and, if at any time it goes out of range, the result of that series of calls is ignored.

Alpha values are included in the color components that are set and read. If there is no alpha buffer, the returned alpha values are expected to alway be 1.0.

Error Explanation -

The allowable error includes two accum shades per operation. The initial color from the color buffer is taken to be the actual color read, not the color written, to eliminate that variation from the accumulated error.

****

apfunc.c
Alpha Plane Function Test.

Description -

Three points are drawn at location (0.0, 0.0), (1.0, 0.0) and (2.0, 0.0). The first point has a random alpha value between 0.0 and 0.4, the second point has an alpha value of 0.5 and the third point has a random alpha value between 0.6 and 1.0. Each alpha function is tested by setting the alpha reference value to 0.5, drawing the three points and comparing the rendered result to the expected result.

****

bclear.c
Buffer Clear Test.

Description -

For each available color buffer, a clear to both the minimum and maximum color values are performed and verified. The minimum and maximum color values for the particular color buffer are determined by the number of color bits.

Error Explanation -

The color read back from a color buffer must match the clear color (within a floating point zero epsilon only).

****

bcolor.c
Buffer Color Test.

Description -

Tests the color stability of color buffers. Since a color buffer may have limited color capacity, a point drawn with a requested color may render in a different color. However, if this rendered color is then used to draw a second point, the rendered color of the second point must match the rendered color of the first point.

Error Explanation -

Failure occurs if the rendered color from a point, when used as the request color for a second point, does not match the rendered color of this second point.

****

bcorner.c
Buffer Corner Test.

Description -

For each available color buffer, a point is drawn at each corner. The color buffer is read and examined to verify that the corner points are at the correct location.

Error Explanation -

Failure occurs if any of the corner points cannot be read correctly (implying that the color buffer’s width and height or the rendering coordinates are not correct).

****

bexact.c

Buffer Invariance Test.

Description -

Tests the primitive invariance between color buffers. A primitive is drawn in the front color buffer (which must exist) and read back. The same primitive is then drawn into any other available color buffers (back, auxiliary, stereo) and compared to the saved image. To insure thorough testing, many different primitives are used.

Error Explanation -

Failure occurs if primitives do not match pixel-exact between all available color buffers.

****

bitmap.c
Bitmap Test.

Description -

A bitmap is made. It's image pattern, size, x and y location, x and y offsets and x and y increments are randomly chosen. It is drawn and verified by checking both the rendered image and a surrounding rectangle area (for correct size). A second bitmap is made. It's image pattern, size, x and y offsets and x and y increments are randomly chosen but its x and y location is determined by the previous bitmap (initial raster position plus x and y increments). The second bitmap is drawn and verified in the same way as the first bitmap.

Error Explanation -

Failure occurs if the rendered bitmaps do not match their original bit pattern images, if the first bitmap is not rendered at the initial raster position or if the second bitmap is not rendered at the final raster position.

****

colramp.c
Color Ramp Test.

Description -

Tests the RGB color ramp. Three tests are performed on each color component (red, green and blue); monotonicity, consistency between individual color shades and total number of color shades. Given the number of bits for a color component, the minimum increment from one color shade to another can be calculated. By stepping from one color shade to the next with an increment smaller then this minimum increment, individual color shades can be isolated. The monotonicity of the color component can be checked. The granularity of the colors can be checked for consistency, and the total number of individual color shades can be counted.

Error Explanation -

Failure occurs if any of the color components is not monotonic increasing, if the color granularity is not constant over the entire color range or if the total number of individual color shades does not match the number calculated from the given number of bits per color component.

****

copypix.c
CopyPixels Test.

Description -

This test examines glCopyPixels() results for the three allowable pixel types: color, stencil and depth. In each case, it draws a reference image in the middle of the window, reads and saves this image, then uses glCopyPixels() to redraw the image offset in each of four directions (towards the corners, but overlapping the original image). It then compares the copied image with the original reference image. For depth and stencil types, the image is created so that only half of the image drawn by glCopyPixels() should be visible.

For type GL_COLOR, the test draws two rectangles, one inside the other. To get visible results for GL_STENCIL, we draw to the stencil buffer with ones to the lower half of the area, zeros to the top half. A rectangle drawn over this area with stenciling enabled should then have only its lower half visible. After copying the stencil values to the new location, a rectangle is again drawn at the new location.

Similarly, for type GL_DEPTH, we draw to the depth buffer with max values in the lower half and zeros in the upper half. The depth function is set so that a rectangle drawn over this area will be visible only in the lower half.

There is one extra step, redrawing the reference area depth values before copying them, since they were altered by drawing the rectangle over them.

The GL_DEPTH and/or GL_STENCIL sections are skipped if the corresponding buffer does not exist.

Error Explanation -

Since we are comparing the results of drawing to the same buffer twice using the same color, we expect the values to be the same, within the floating point tolerance (epsilon.zero).

****

dither.c
Dither Test.

Description -

Tests dithering by drawing rectangles of various shades with both dithering enabled and disabled, and comparing results to ensure that corresponding fragments differ by no more than one shade. Since dithering may depend upon x and y window coordinates, fragments compared must come from the same window location. For visuals with up to 256 colors, the tests uses each color. For deeper visuals, it skips over some to test up to 256 shades.

Error Explanation -

Failure occurs if the comparison between the original fragment color and the dithered fragment color varies by more then one shade of color.

****

divzero.c
Divide By Zero Test.

Description -

This tests many situations where an implementation might divide by zero. We try to assure, that division by zero does “not lead to GL interruption or termination” as specified by the OpenGL Specificaton. In two cases we test that a proper error is generated. Error generation is checked in the case of a viewport of zero width, and in the case of evaluators with a trivial domain. (The evaluator test is currently unfinished.) We currently test 24 different situations and there are more coming. These situations are summarized in the comments at the bottom of this file.

Error Explanation -

Failure occurs if an implementation interrupts or terminates during this test. Failure also occurs if a parameter value which could cause a divide by zero should generate an GL_INVALID_VALUE error and does not.

****

dlist.c
Display List Test.

Description -

Creates four display lists:

Draws a polygon, triangle, and four points, and calls list 2.
draws a polygon, triangle strip and triangle fan.
draws lines using line strip, line loop, and lines.
calls glDrawPixels() with pixel zoom.

The lists are used to test the following:

- A simple display list works, i.e. it can draw a few primitives.
- One display list can call a second display list.
- The called (second) display list can be deleted and a new display list with the same name can be created, which should be executed when the first list is called.
- The COMPILE_AND_EXECUTE mode renders upon calling glNewList. A display list can store user data.

To accomplish the above, the test does the following:

- Creates lists 1, 2, and 4 using COMPILE mode.
- Calls list 1 and tests results.
- Deletes list 2, and creates a new list 2.
- Call list 1, which calls the new list 2, tests results.
- Deletes list 1, and recreates it with COMPILE_AND_EXECUTE mode.
- Tests results both after list creation and after calling the list.
- Calls list 4 and tests results by examining corners of zoomed image.

Note that color maps are used in the DrawPixels so that the same data is used for both RGB and CI modes.

The test has calls to glGenLists, glDeleteLists, glNewList, glEndList, glCallList.

Error Explanation -

After each rendering, it samples a point from each primitive to indicate that it was drawn in the chosen color. For the glDrawPixels() call, it samples several points from the different color bands in the image to check that it was positioned and zoomed correctly.

****

drawpix.c
DrawPixels Test.

Description -

Draws a rectangle in the lower left hand corner of the window using glDrawPixels() for every pixel format and type allowed. If the pixel format is one which draws to a color buffer, then a solid rectangle appears. For alpha, stencil and depth formats, values are set so that when a rectangle is drawn, only half of the rectangle should pass the {alpha, stencil, depth} test.

Rectangles are drawn with fully saturated colors. The outcome of each glDrawPixels() is checked by examining one pixel in the interior of the upper half of the rectangle, and one in the lower half. The results should either be the color set in the initial data or the background color, for the non-passing sections of rectangles in alpha, stencil and depth tests.

The test is structured as follows:

- For each data type:
  - For each pixel format:
    - SetupData() - create raw image data.
    - ConvertCompData() - convert image data.
    - glDrawPixels().
    - TestPixRGB() or TestPixCI().
- For bitmap type:
  - For formats GL_COLOR_INDEX and GL_STENCIL_INDEX:
    - {repeat above procedure}

****

evalv.c
Evaluator Vertex Test.

Description -

An evaluator surface is rendered covering the screen and parallel to the viewport. A 2d x 4d color evaluator map is initialized. The surface is rendered with glEvalCoord() and then again with glMapMesh(GL_POINTS, ...). Single pixel points are used to render the surface as their shape and coloring is more well defined than other primitives. The results are read using glReadPixels(), and compared to the theoretical values. Some of the generated evaluator values are greater tnan 1.0. This checks if the colors are correctly clamped. If the rendered image and the theoretical values differ by more than one shade the implementation fails. For the surface itself the u, v coordinates chosen should yield x and y values exactly on pixel centers.

Error Explanation -

Failure occurs if the point rendered is more than one shade different from the internally calculated theoretical result.

****

evalv.c
Evaluator Vertex Test.

Description -

A 3d x 8d polynomial evaluator surface is set-up using the control values in the array point2f below. Cross-sections through the surface are rendered at carefully chosen grid points using glEvalCoords() or glMapMesh(). Single pixel points are used to render the sections as their shape is more well defined than other primitives. The u, v coordinates chosen yield x values exactly on pixel centers. This is possible since the evaluator is linear in x. For each column, the height, y coordinate, is compared to an internally calculated floating point number scaled and biased to correspond to screen coordinates. If the the fragment rendered is more than 2 pixels lower or higher than the ideal point the implementation fails.

Error Explanation -

Failure occurs if the point rendered is two pixels higher or lower than the internally calculated result or if a column renders with no pixels in it.

****

feedback.c
Feedback Test.

Description -

All primitives, glCopyPixels(), glDrawPixels() and glPassThrough() are rendered in feedback mode. The returned value from glRenderMode() is check to see if the correct amount of information was placed in the feedback buffer. The feedback buffer is then checked for correct information (vertex coordinates, color and texture coordinates). The test is repeated for all feedback modes.

****

fogexp.c
Fog Exponential Test.

Description -

This file contains two tests, one for the RGB and one for CI fog equations. Both the GL_FOGEXP and GL_FOGEXP2 are tested in this test. This test examines the effect which varying the z eye coordinate and the density, d, has on the color of the point. When the density value is being tested z is set to -1.0 and the third coordinate of the vertex is set to a referance value, r. When z is being examined it is set to -r and d is set to 1.0. In other tests involving the exponential function we use huge errors. This is for exponents as large as 128. In this test the exponents are bounded between 0 and 1. So error bounds are tighter.

In RGB mode the test only examines one color channel at a time. First it examines the effect of the fog weighting on the current color, and then it examines the effect on fog color. When the current color is being examined the fog color is set to black, and one channel of the current color is set to 1.0. When the fog color is being examined the current color is set to black, and one channel of the fog color is set to 1.0. In CI mode only the fog index is examined and the current index is set to 0.

The test repeatedly renders a point in the lower left corner of the window, increments r, and reads back the color of the point. Since implementations could conceivable use the distance of the object from the eye to calculate f, the point is positioned so that it's distance from the eye is equal to it’s z coordinate.

r is incremented in such a way that f, should theoretically increment uniformly. The color should therefore also increment uniformly, if we are testing the current color, and decrement uniformly, if we are testing the fog color. In the case of CI mode it will always decrement the index.

In RGB mode the observed color of the point is examined. An implementation fails if a color channel is non-zero, when it should be zero, or if it is not 1.0, when it should be 1.0. The observed values are also examined for monotonocity. An implementation fails if the results which should increment or decrement monotonically fail to do so. Finally an implementation will fail if the observed color is more than 1.5 shades from the expected color.

In CI mode the observed value is checked to be monotonically decreasing. It is confirmed to be an integer, and it is checked to reach maxIndex when the z value is equal to the fogStart value. The implementation fails if the observed index is more than 2 away from the theoretically correct index. It will also fail if the difference between two consecutive observations is larger than 2 (delta > 2).

****

foglin.c
Fog Linear Test.

Description -

This file contains two tests, one for the RGB and one for CI fog equations. Only the linear fog computation is tested in this test. This test examines the effect which varying the z eye coordinate has on the color of the point. The test repeats itself with different fog start and fog end values.

In RGB mode it only tests one color channel at a time. First it examines the effect of the fog weighting on the current color, and then it examines the effect on fog color. When the current color is being examined the fog color is set to black, and one channel of the current color is set to 1.0. When the fog color is being examined the current color is set to black, and one channel of the fog color is set to 1.0. In CI mode only the fog index is examined and the current index is set to 0.

The test repeatedly renders a point in the lower left corner of the window and reads back the color of that point. At each iteration the reference value is moved away from the fogEnd value and toward the fogStart value. The z value of the rendered point is the negative of this value. Since implementations could conceivable use the distance of the object from the eye to calculate the fog, the point is positioned so that it's distance from the eye is equal to it's z coordinate.

By uniformly incrementing the point from the fogEnd value to the fogStart value the fogWeighting, f, is uniformly incremented. This should uniformly increment the color if we are testing the current color and decrement the color if we are testing the fog color. In the case of CI mode it will always decrement the index.

In RGB mode the observed color of the point is examined. An implementation fails if a color channel is non-zero, when it should be zero, or if it is not 1.0 when it should be 1.0. The observed values are also examined for monotonocity. An implementation fails if the results which should increment/decrement monotonically fail to do so. Finally an implementation will fail if the observed color is more than 1.5 shades from the expected color.

In CI mode the observed value is checked to be monotonically decreasing. It is confirmed to be an integer, and it is checked to reach maxIndex when the z value is equal to the fogStart value. The implementation fails if the observed index is more than 2 away from the theoretically correct index. It will also fail if the difference between two consecutive observations is larger than 2 (delta > 2).

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue when it should be greater.
RGB mode: maxObserved < 1.0.
RGB mode: (observed - expected) > 1.5*epsilon.color.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 2.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_ac.c
Light - Attenuation Constant Test.

Description -

This test examines the effect which varying the Attenuation_Constants has on the color produced by lighting. CI lighting differs from RGB lighting. This file contains code for both tests. 8 lights and all 3 Attenuation_Constants are tested one at a time. The distance of the object from the origin in eye-coordinates is exactly 1.0. At each iteration the reciprocal of an Attenuation_Constant is uniformly incremented, and the red component or color index of a lit point is read and stored.

In RGB mode Ambient, Emissive, and Specular summands of the equation are set to black, and the diffuse parameters are set to fully saturated red. Successive results should differ by 1.0 / nR where nR is the number of red shades. Theoretically the differance between successive results should be constant. If successive observed differences differ by more than 0.25 / nR, the implementation is judged to differ by too much from the correct graph of the attenuation function.

In color index mode the Ambient and Diffuse indices are set to zero, and the Specular index is set to the largest color index allowed by the implementation. Successive colors should fall on integer values, should increase and should differ from the theoretical value by no more than 3 indices. The difference between successive differences should be more than 2.

Error Explanation -

In RGB mode this test issues an error if the blue or green channels are not completely “off”, or if an expected lighting result of (1, 0, 0) does not result in a fully-saturated red object. An error is also issued if a smaller Attenuation_Constant results in a less saturated red, or if the differences between the results of two successive attenuation settings differ by more than 0.25 / nR.

In color index mode this test issues an error if the value read back is not an integer, if successive values decrease, or if the difference between a value and the theoretical result is more than 3 or if two successive differences differ by more than 2.

****

l_al.c
Light - Ambient Light Test.

Description -

This lighting test analyses the effect that varying the Ambient_Light has on the color produced by lighting. This is the parameter set with glLightfv(i, GL_AMBIENT, color). A single lit point is rendered in the lower left corner of the screen, and the color of this point is read using glReadPixels().

To analyse only this one parameter, other lighting parameters are chosen so that the Emmissive, Diffuse, and Specular summands of the RGB lighting function are set to zero. The Ambient_Scene is also set to zero. The Ambient_Material parameter is set to (1, 1, 1, 1). The Attenuation and Spot parameters are left at their default values, leaving the factors involving these parameters at 1.0. This forms a restriction of the lighting function to a function of Ambient_Light f(Ambient_Light) -> color. This function is analysed for monotonicity, endpoint's values, and variation.

All color channels are tested one at a time. The implementation fails if any color channel other than the one being tested is not observed to be black. One light is enabled at a time. The first 8 individual lights are tested.

The test repeatedly renders and reads the same point, uniformly incrementing Ambient_Light so that the colors resulting from lighting should vary by a uniform amount on every 16th iterate. This controls the linearity of this restriction of the lighting function to Ambient_Light. The uniformity of this change is tested and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed restriction is also checked for monotonicity.

When Ambient_Light reaches its max/min the color is checked to insure that it is rendered with a saturation of 1.0.

Error Explanation -

Non_tested color channel not black. Ambient_Light fully saturated, but output not fully saturated. Color decreased as Ambient_Light is increased. (thisDelta - lastDelta) > 1/4 shade.

****

l_am.c
Light - Ambient Material Test.

Description -

This lighting test analyses the effect that varying Ambient_Material has on the color produced by lighting. (This parameter is set with glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ...). CI lighting differs from RGB lighting. This file contains code for both tests. A single lit point is rendered in the lower left corner of the screen and the color or color index of this point is retrieved using glReadPixels(). To analyse only this one parameter, lighting parameters are chosen so that the summands of the RGB or CI lighting equation which do not contain Ambient_Material are set to zero. In RGB mode Ambient_Scene is set to 1.0, and in both modes the individual lights are disabled.

This forms a restriction of the lighting equation to a function of Ambient_Material f(Ambient_Material) -> color. This function is analysed for monotonicity, endpoints, and variation. In CI mode a theoretical result is directly compared to the result read from the framebuffer. In RGB mode all color channels are tested one at a time. The implementation fails if any color channel which is not being tested is not zero.

The test repeatedly renders and reads the same point, uniformly increment Ambient_Material so that the colors resulting from lighting vary by a uniform amount. This helps to analyze the concavity of this restriction of the lighting function.

In RGB mode uniformity of change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Ambient_Material reaches its max the color is checked to insure that it is rendered with a saturation of 1.0.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Ambient_Material. If the theoretical value is more than 2 indices away from the observed index the test fails. The observed value must be an integer, and the difference between successive observances must be less than 2.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_ap.c
Light - Attenuation Position Test.

Description -

This test analyses the effect that varying the Light_Position has on the color produced by lighting, by way of its effect on the Attenuation factor. CI lighting differs from RGB lighting. This file contains code for both tests. To analyse only this one parameter, the Specular lighting factor is set to a completely saturated red. Other lighting parameters are chosen so that the Diffuse, Ambient, and Emissive summands of the RGB or CI lighting are set to zero. The linear and attenuation constants are set so that one is zero and the other is 1.0, and then the test is repeated with the constant's settings switched. The constant Attenuation_Constant, k0, is always 0.0.

These settings form a restriction of the lighting equation to a function of Light_Position. f(Light_Position) -> color. This function is analysed for monotonicity, endpoints, and variation. In RGB mode only the red color channel is tested. The implementation fails if any other color channel is not black. In both modes one light is enabled at a time, all individual lights are tested.

The test repeatedly renders and reads the same point at the lower left corner of the screen, decrementing the third component of the Light_Position in such a way that the attenuation factor should increase by a uniform amount. The other two components are set equal to the corresponding coordinates of the eye position of the rendered point. This controls the concavity of this restriction of the lighting function to Light_Position.

In RGB mode the uniformity of this change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Light_Position[2] reaches 1.0, the color is finally checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. If the theoretical value is more than 4 indices away from the observed index, the test fails. The observed value must be an integer, and the difference between successive observances must be less than 2.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: index not an integer.
CI mode: (observed - expected) > 4.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_as.c
Light - Ambient Scene Test.

Description -

This test analyses the effect that varying the Ambient_Scene parameter has on the color produced by lighting. This parameter is set with glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ...). It is only tested in RGB mode. A single lit point is rendered in the lower left corner of the screen, and the color of this point is retrieved using glReadPixels(). To analyse only this one parameter the Emissive color is set to (0, 0, 0, 1), and all of the lights are disabled. The Ambient_Material is set to (1, 1, 1, 1). This forms a restriction of the lighting equation to a function of the Ambient_Scene f(Ambient_scene) -> color. Except for the quantization of color this restriction should be the identity. This function is analysed for monotonicity, endpoints values, and variation. All color channels are tested one at a time. The implementation fails if any channel other than the one being tested is non-zero.

The test repeatedly renders and reads the same point, uniformly incrementing the one color channel being tested of the Ambient_Scene parameter. This controls the linearity of this restriction of the lighting function. he uniformity of change is tested and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed restricted function is also checked for monotonicity. When the Ambient_Scene reaches 1.0, the tested color channel is checked to insure that the point has been rendered with a saturation of 1.0.

Error Explanation -

other_color > 0.0.
value < lastValue.
maxObserved < 1.0.
(thisDelta - lastDelta) > 1/4 shade.

****

l_dl.c
Light - Diffuse Light Test.

Description -

This test analyses the effect that varying the Diffuse_Light parameter has on the color produced by lighting. CI lighting differs from RGB lighting. This file contains code for both tests. A single lit point is rendered in the lower left corner of the screen and the color or color index of this point is retrieved using glReadPixels(). To analyse only this one parameter, parameters are chosen so that the Attenuation factor, the SpotLight factor, the Light_Direction factor are all 1.0. In RGB mode the Specular, Emissive, and Ambient summands in the lighting equation are 0.0 and the Diffuse_Material is set to (1, 1, 1, 1). In CI mode s_prime is set to 0.0, and the Ambient_Material is set to 0.0.

This forms a restriction of the lighting equation so that the output color is equal to a multiple of the Diffuse_Light setting. In Ci mode this is done by setting all three colors of Diffuse_Light to the same value. This function's output is analysed for monotonicity, endpoints, and variation. In RGB mode all color channels are tested one at a time. The implementation fails if any color channel, besides the one being tested, is not 0.0. In both modes at most one light is enabled at a time. All individual lights are tested.

The test repeatedly renders and reads the same point, linearly incrementing Diffuse_Light. In RGB mode the uniformity of this change in observed color is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Diffuse_Light reaches 1.0, the observed color is checked to insure that it is rendered with a saturation of 1.0.

In CI mode a check is made to ensure that the observed index reaches maxIndex. The observed shade is compared to the theoretical value expected for each setting of Diffuse_Light. The observed value must be an integer, and the difference between successive observances must be less than 2. The observed index is also compared to the theoretical value expected for each setting of Diffuse_Light. If the theoretical value is more than 3 indices away from the observed index the test fails.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_dm.c
Light - Diffuse Material Test.

Description -

This test analyses the effect that varying the Diffuse_Material parameter has on the color produced by lighting. CI lighting differs from RGB lighting. This file contains code for both tests. To analyse only this one parameter, parameter values are chosen so that the Attenuation factor, the SpotLight factor, and the Light_Position factor are all 1.0. In RGB mode the Specular, Emissive, and Ambient summands in the lighting equation are 0.0, and the Diffuse_Light is set to (1, 1, 1, 1). In CI mode s_prime is set to 0.0, and the Ambient_Material is set to (0, 0, 0, 1).

This forms a restriction of the lighting equation so that the output color is equal to a multiple of the Diffuse_Material setting (color ~= Diffuse_Material). This function's output is analysed for monotonicity, endpoints, and variation. In RGB modes one light is enabled at a time. All individual lights are tested. In CI mode all three color channels of Specular_Light are set to 0.125 for the first 8 lights and all the lights are enabled.

The test repeatedly renders and reads the same point in the lower left corner of the screen, uniformly incrementing Diffuse_Material. In RGB mode the uniformity of the change in observed color is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Diffuse_Material reaches 1.0, the observed color is checked to insure that it is rendered with a saturation of 1.0. All color channels are tested one at a time, and the implementation fails if any color channel, besides the one being tested, is not 0.0.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Diffuse_Material. The observed value must be an integer, and the difference between successive observances must be less than 2. The observed index is also compared to the theoretical value expected for each setting of Diffuse_Material. If the theoretical value is more than 3 indices away from the observed index the test fails.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.

RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_dmn.c
Light - Diffuse Material Normal Test.

Description -

This file contains code for CI and RGB tests. This test analyses the effect that varying the Normal has on the color produced by lighting, by way of its effect on the Diffuse summand. To analyse only this one parameter, the Diffuse_Lighting factor is set to a completely saturated red in RGB mode and max index in CI mode. In RGB mode lighting parameters are chosen so that the Specular, Ambient, and Emissive summands of the lighting equation are set to 0.0, and the Attenuation and the Spotlight factor are both 1.0. In CI mode s_prime is 0, Diffuse_Material is maxIndex, Diffuse_Light is (1, 1, 1, 1). The SpotLight and Attenuation factors are 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Normal is incremented by a uniform amount. The other two components of Normal are set so that the total length of the Normal is 1. The Light_Position is set so that the vector, PV, from the eye position of the rendered point to the Light_Position is (0, 0, 1, 1). (Normal dot PV) = Normal[2]). The lighting equation is reduced to color = red * Normal[2] in RGB mode, or colorIndex = max index * Normal[2] in CI mode.

In RGB mode only the red color channel is tested. The implementation fails if any other color channel is not black. In both modes one light is enabled at a time, all individual lights are tested.

In RGB mode the uniformity of the change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Normal[2] reaches 1.0 the color is finally checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. If the theoretical value is more than 3 indices away from the observed index the test fails. The observed value must be an integer, and the difference between successive observances must be less than 2.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_dmp.c
Light - Diffuse Material Position Test.

Description -

This test analyses the effect that varying the Light_Position has on the color produced by lighting, by way of its effect on the Diffuse summand. CI lighting differs from RGB lighting. This file contains code for both tests. To analyse only this one parameter, the Diffuse lighting factor is set to a completely saturated red, other lighting parameters are chosen so that the Specular, Ambient, and Emissive summands of the RGB or CI lighting are set to zero, and values are chosen so the Attenuation and the Spotlight factor are both 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Light_Position is iterated by a uniform amount. The other two components of Light_Position are set so that the total length of the vector, V, which is the difference between the rendered point’s eye position and the Light_Position is 1. The Normal vector is set to (0, 0, 1) so the Light_Position factor is V[2], (Normal dot V) = V[2]). The lighting equation becomes either color = red * V[2] or colorIndex = max index * V[2].

In RGB mode only the red color channel is tested. The implementation fails if any other color channel is not black. In both modes one light is enabled at a time, all individual lights are tested.

In RGB mode the uniformity of the change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Light_Position[2] reaches 1.0 the color is finally checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. If the theoretical value is more than 3 indices away from the observed index the test fails. The observed value must be an integer, and the difference between successive observances must be less than 2.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - lastDelta) > 1/4 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_em.c
Light - Emissive Material Test.

Description -

This test analyses the effect that varying the Emissive_Material parameter has on the color produced by lighting. This parameter is set with glMaterial(GL_FRONT_AND_BACK, GL_EMISSION, ...). It is only used in RGB mode. A single lit point is rendered in the lower left corner of the screen, and the color of this point is retrieved using glReadPixels(). To analyse only this one parameter the Ambient_Material color is set to (0, 0, 0, 1), and all of the lights are disabled. This forms a restriction of the lighting equation to a function of the Emissive_Material (color = Emissive_Material). This function is analysed for monotonicity, endpoints values, and variation. All color channels are tested one at a time. The implementation fails if any channel other than the one being tested is non-zero.

The test repeatedly renders and reads the same point, uniformly incrementing the one color channel being tested of the Emissive_Material parameter. This controls the linearity of this restriction of the lighting function to the Ambient_Scene. The uniformity of change is tested and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed, restricted function is also checked for monotonicity. When the Emissive_Material reaches 1.0, the tested color channel is checked to insure that the point has been rendered with a saturation of 1.0.

Error Explanation -

other_color > 0.0.
value < lastValue.
maxObserved < 1.0.
(thisDelta - lastDelta) > 1/4 shade.

****

l_se.c
Light - Specular Exponent Test.

Description -

This test analyses the effect that varying the Normal and the Specular Exponent together have on lighting. The file contains code for both CI and RGB tests. To analyse only these parameters, the Specular_Material and Specular_Light are set to red. In CI mode the Specular_Light is set to (1, 1, 1) and the Specular_Material is set to maxIndex. Other lighting parameters are chosen so that the Diffuse, Ambient, and Emissive summands of the RGB or CI lighting are set to zero. Values are chosen so the Attenuation and the Spotlight factor are both 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Normal and the Specular Exponent are both incremented in such a way as to create a uniform change in the theoretical

Specular Exponent factor within the Specular summand. (This controlled incrementation takes place in the routine called Set() below). The other two components of the Normal are set so that the total length of the Normal is 1. The eye position and the light position are set so that the Half_Angle vector, H_hat, is (0, 0, 1), with these settings (Normal dot H_hat) = Normal[2].

In RGB mode the implementation fails if blue or green are not 0.0. In both modes one light is enabled at a time, all individual lights are tested. The observed results are also checked for monotonicity, and the change is checked for uniformity. The observed change should be less than 1 shade from the theoretical change (delta - trueDelta) < delta. When the expected value reaches 1.0 the observed color is checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. If the theoretical value is more than 5 indices away from the observed index the test fails. The observed value must be an integer, and the difference between successive observances must be less than or equal to 2.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: (thisDelta - correctDelta) > 1 shade.
RGB mode: (thisDelta - lastDelta) > 1 shade.
CI mode: (fractional index) != 0.0.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sed.c
Light - Spot Exponent And Direction Test.

Description -

This test analyses the effect that varying the Spotlight_Direction has on the color produced by lighting. This file contains code for both CI and RGB tests. To analyse only this one parameter, the RGB test uses a different analysis from that of the CI test. In RGB mode the Ambient summand is used, and in CI mode the Specular summand is used. In both modes the test repeats itself with different Spot_Exponents, and different Cutoff_Angles.

In RGB mode the Ambient_Material is set so that one color channel is fully saturated, and the other channels are set to zero. All three channels are tested. The implementation fails if any color channel not being tested is not observed to be 0.0. The Ambient_Light is set to (1, 1, 1, 1). Other lighting parameters are chosen so that the Diffuse, Specular, and Emissive summands of the RGB lighting are set to 0.0.

In CI mode the Specular_Light is set to (1, 1, 1, 1) and the Specular_Material is set to maxIndex. The Diffuse and Ambient summands are zero. Values are chosen so the Attenuation factor is 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Spotlight_Direction is incremented by a an amount sensitive to the Spotlight_Exponent being tested. The increment is controlled so that the observed resulting color would theoretically increment uniformly. (This controlled incrementing takes place in the routine called Set() below). The other two components of the Spotlight_Direction are adjusted so that the total length of the Spotlight_Direction vector is 1. The rendered point's eye position and the Spotlight_Position are set so that the vector from the Spotlight_Position to the rendered point's eye position, PV, is (0, 0, 1). With these settings (SpotDirection dot PV) = SpotDirection[2]).

In both modes if the theoretically calculated Spot_Exponent factor indicates that the Cutoff_Angle would cause the object to render black, the implementation fails if the observed color is not black. If the expected value indicates that the observed color should not be "cutoff" the implementation fails if the color is more than 5% away from the theoretical correct value. A margin around the cutoff angle of 1% is used. In both modes one light is enabled at a time, all individual lights are tested. The observed results are compared to the theoretical value. The error here is 10% for bright shades or one colorEpsilon for dark shades. This test will not run with less than 3 bits of color per channel. When Spotlight_Position[2] reaches 1.0, the color is finally checked to insure that the point has rendered with a fully saturated color. The observed results are also checked for monoticity.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. For high indices if the theoretical value is more than .05*maxIndex away from the observed index the test fails. Darker colors must be within 1 indices.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(observed - expected) > 0.1.
CI mode: index not an integer.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sen.c
Light - Specular Exponent Normal Test.

Description -

This test analyses the effect that varying the Normal has on the color produced by lighting, by way of its effect on the Specular summand. This file contains code for both CI and RGB tests. To analyse only this one parameter, the Specular_Material is set to color channel at a time, and Specular_Light is set to (1, 1, 1, 1). In CI mode the Specular_Material is set to maxIndex and the Specular_Light is set to (1, 1, 1, 1). Other lighting parameters are chosen so that the Diffuse, Ambient, and Emissive summands of the RGB or CI lighting are set to 0.0. Values are chosen so the Attenuation and the Spotlight factor are both 1.0. This test repeats itself with different Specular_Exponents (GL_SHININESS), and with different color channels.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Normal is incremented by a an amount sensitive to whichever Specular_Exponent is being tested. The increment is controlled so that the observed resulting color would theoretically increment uniformly. (This controlled incrementation takes place in the routine called Set() below). The other two components of the Normal are set so that the total length of the Normal is 1. The eye position and the light position are set so that the Half_Angle vector, H_hat, is (0, 0, 1), with these settings (Normal dot H_hat) = Normal[2].

In RGB mode the implementation fails if any color channel not being tested is not 0.0. In both modes one light is enabled at a time, all individual lights are tested. The observed results are compared to the theoretical value. The error here is 5% for bright colors and 1% for dark shades. This test will not run with less than 3 bits of color per channel. When Light_Position[2] reaches 1.0, the color is finally checked to insure that the point has rendered with a fully saturated color.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. For high indices if the theoretical value is more than 5 indices away from the observed index the test fails. Darker colors must be within 2 indices. The observed value must be an integer.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(observed - expected) > 0.05 for highly saturated colors.
RGB mode: ABS(observed - expected) > 0.01 for dark shades.
CI mode: index not an integer
CI mode: (observed - expected) > 5.0 for highly saturated colors.
CI mode: (observed - expected) > 2.0 for dark shades or shallow buffers.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sep.c
Light - Spot Exponent Position Test.

Description -

This test analyses the effect that varying the Spotlight_Direction has on the color produced by lighting. This file contains code for both CI and RGB tests. To analyse only this one parameter, the RGB test uses a different analysis from that of the CI test. In RGB mode the Ambient summand is used, and in CI mode the Specular summand is used. In both modes the test repeats itself with different Spot_Exponents and Cutoff_Angles.

In RGB mode the Ambient_Material is set so that one color channel is fully saturated, and the other channels are set to 0.0. All three channels are tested. The implementation fails if any color channel not being tested is not observed to be 0.0. The Ambient_Light is set to (1, 1, 1, 1). Other lighting parameters are chosen so that the Diffuse, Specular, and Emissive summands of the RGB lighting are set to zero.

In CI mode the Specular_Light is set to (1, 1, 1, 1) and the Specular_Material is set to maxIndex. The Diffuse and Ambient summands are 0.0. Values are chosen so the Attenuation factor is 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Spotlight_Direction is incremented by a an amount sensitive to the Spotlight_Exponent being tested. The increment is controlled so that the observed resulting color would theoretically increment uniformly. (This controlled incrementation takes place in the routine called Set() below). The other two components of the Spotlight_Direction are adjusted so that the total length of the Spotlight_Direction vector is 1. The rendered point's eye position and the Spotlight_Position are set so that the vector from the Spotlight_Position to rendered point's eye position, PV, is (0, 0, 1). With these settings (SpotDirection dot PV) = SpotDirection[2].

In both modes if the theoretically calculated Spot_Exponent factor indicates that the Cutoff_Angle would cause the object to render black, the implementation fails if the observed color is not black. If the expected value indicates that the observed color should not be "cutoff" the implementation fails if the color is more than 5% away from the theoretical correct value. A margin around the cutoff angle of 1% is used. In both modes one light is enabled at a time, all lights are individually tested. The observed results are compared to the theoretical value. The error here is 5% for bright shades and one colorEpsilon for dark shades. This test will not run with less than 3 bits of color per channel. When Spotlight_Position[2] reaches 1.0, the color is finally checked to insure that the point has rendered with a fully saturated color. The observed results are also checked for monoticity.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position. For high indices if the theoretical value is more than 0.05 * max index away from the observed index the test fails. Darker colors must be within 1 indices. The observed value must be an integer.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(value - expected) > 0.05.
RGB mode: ABS(thisDelta - lastDelta) > 1/4 shade.
CI mode: index not an integer.
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < maxIndex.
CI mode: delta > 2.0.

****

l_sl.c
Light - Specular Light Test.

Description -

This test analyses the effect that varying the Specular_Light parameter has on the color produced by lighting. CI lighting differs from RGB lighting. This file contains code for both tests. A single lit point is rendered in the lower left corner of the screen and the color or color index of this point is retrieved using glReadPixels(). To analyse only this one parameter, parameter values are chosen so that the Attenuation factor, the SpotLight factor, and the Half_Angle factor are all 1.0. In RGB mode the Diffuse, Ambient, and Emissive summands in the lighting equation set to 0.0 and the Specular_Material is set to (1, 1, 1, 1). In CI mode the Diffuse_Material is set to 0.0, and the Ambient_Material is set to 0.0.

This forms a restriction of the lighting equation so that the output color is equal to a multiple of the

Specular_Light setting. In CI mode all three colors of Specular_Light are set to the same value. This function's output is analysed for monotonicity, endpoints, and variation. In RGB mode all color channels are tested one at a time. The implementation fails if any color channel, besides the one being tested, is not 0.0. In both modes at most one light is enabled at a time. All individual lights are tested.

The test repeatedly renders and reads the same point, uniformly incrementing Specular_Light. In RGB mode the uniformity of the change in observed color is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 colorEpsilon. The observed results are also checked for monotonicity. When Specular_Light reaches 1.0, the observed color is checked to insure that it is rendered with a saturation of 1.0.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Specular_Light. The observed value must be an integer, and the difference between successive observances must be less than 2. The observed index is also compared to the theoretical value expected for each setting of Specular_Light. If the theoretical value is more than 3 indices away from the observed index the test fails.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(thisDelta - lastDelta) > 1/4 shade.
CI mode: index not an integer
CI mode: ABS(observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sleha.c
Light - Specular Light Eye Half Angle Test.

Description -

This test is different from the other light tests in that it moves the point being rendered rather than just changing the settings of lighting parameters. The test only examines

RGB lighting although a similar test could easily be written for CI mode. Most of the lighting summands are set to 0.0. The Attenuation factor and the Spotlight factor are set to 1.0. The Specular_Light is set to (1, 1, 1, 1) and the Specular_Material is set to (1, 0, 0, 1). Finally the Normal vector is set to (0, -1, 0) so the red component of the light should be equal to the y component of the point's normalized position. The color resulting from lighting is compared to the results which are observed when the same point is rendered with lighting disabled and the color of (y/length, 0, 0, 1). These values should be equal even if dithering is enabled. A difference of more than 1/64 + 1 shade results in an implementation's failure. (We allow the error of 1/64 in case the implementation uses a small lookup table for its exponential calculation.

****

l_sm.c
Light - Specular Material Test.

Description -

This test analyses the effect that varying the Specular_Material parameter values has on the color produced by lighting. This file contains code for both CI and RGB tests. To analyse only this one parameter, parameter values are chosen so that the Attenuation factor, the SpotLight factor, and the Half_Angle factor are all 1.0. In RGB mode the Diffuse, Emissive, and Ambient summands in the lighting equation are 0.0 and the Specular_Light is set to (1, 1, 1, 1). In CI mode Diffuse_Material, and Ambient_Material are set to 0.0, and values are chosen so that s_prime is equal to 1.0.

This forms a restriction of the lighting equation so that the output color is equal to the Specular_Material's setting. (color = Specular_Material). This observed color is analysed for monotonicity, endpoints, and variation.

The test repeatedly renders and reads the same point in the lower left corner of the screen, uniformly incrementing Specular_Material. In RGB mode the uniformity of the change in observed color is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Specular_Material reaches 1.0, the observed color is checked to insure that it is rendered with a saturation of 1.0. All color channels are tested one at a time, and the implementation fails if any color channel, besides the one being tested, is not 0.0. One light is enabled at a time. All individual lights are tested.

In CI mode the first 8 lights are enabled with all three Specular_Light color channels set to 0.125. A check is made to ensure that the observed index reaches maxIndex. The observed shade is compared to the theoretical value expected for each setting of Specular_Material. The observed value must be an integer, and the difference between successive observances must be less than 2. The observed index is also compared to the theoretical value expected for each setting of Specular_Material. If the theoretical value is more than 3 indices away from the observed index the test fails.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(thisDelta - lastDelta) > 1/4 shade.
CI mode: Index not a fraction
CI mode: (observed - expected) > 3.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sn.c
Light - Specular Normal Test.

Description -

This file contains code for CI and RGB tests. This test analyses the effect that varying the Normal has on the color produced by lighting, by way of its effect on the Specular summand. To analyse only this one parameter, the Specular_Lighting factor is set to a completely saturated red in RGB mode and max_index in CI mode. In RGB mode lighting parameters are chosen so that the Diffuse, Ambient, and Emissive summands of the lighting equation are set to zero and the Attenuation and the Spotlight factor are both 1.0. In CI mode Ambient and Diffuse Material Indices are set to zero so only the Specular summand is non-zero. Specular_Material is set to maxIndex, Specular_Light is (1, 1, 1, 1), and the SpotLight and

Attenuation factors are 1.0. The Specular_Exponent is always set to 1.0.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the Normal is incremented by a uniform amount. The other two components of the Normal are set so that the total length of the Normal is 1. The Light_Position is set so that the vector from the eye position of the rendered point to the Light_position is (0, 0, 1). With the viewer set at infinity. With these settings the normalized Half_Angle vector is (0, 0, 1) so (Normal dot H_hat) = Normal[2]. The lighting equation is reduced to color = red * Normal[2] in RGB mode, or colorIndex = max index * Normal[2] in CI mode.

In RGB mode only the red color channel is tested. The implementation fails if any other color channel is not black. In both modes one light is enabled at a time, all individual lights are tested.

In RGB mode the uniformity of the change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade. The observed results are also checked for monotonicity. When Normal[2] reaches 1.0 the color is finally checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches its max. The observed shade is compared to the theoretical value expected for each setting of Light_Position The allowable error includes 2 shades and and additional term in case the exponent is calculated using a small lookup table.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(thisDelta - lastDelta) > 1/4 shade.
CI mode: observed index not an integer.
CI mode: (observed - expected) > 4.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

l_sp.c
Light - Spot Position Test.

Description -

This test analyses the effect that varying the SpotLight_Position. has on the color produced by lighting, by way of its effect on the Specular summand. CI lighting differs from RGB lighting. This file contains code for both tests. To analyse only this one parameter, the Specular_Light Parameter is set to (1, 1, 1, 1). In RGB mode the Specular_Material is set to Red, and in CI mode the Specular_Material is set to maxIndex. Other lighting parameters are chosen so that the Diffuse, Ambient, and Emissive summands of the RGB or CI lighting are (0, 0, 0, 1). Similarly values are chosen so the Attenuation and the Specular_Half_Angle factor are both 1.0. The SpotLight_Exponent is always set to 1.0, and the Cutoff_Angle is 90 degrees. In RGB mode only the red color channel is tested. The implementation fails if any other color channel is not black. In both modes one light is enabled at a time, all individual lights are tested.

The test repeatedly renders and reads the same point at the lower left corner of the screen. The third component of the SpotLight_Position is uniformly incremented. The other two components of the SpotLight_Position are set so that the total length of the vector, PV, which is the difference between the SpotLight_Position and the rendered point's eye position, is 1. The Normal vector is set to (0, 0, 1) so the Spot_Position factor becomes PV[2], (Normal dot PV) = PV[2]. With these settings the lighting equation reduces to color = red * PV[2] or colorIndex = max index * PV[2].

In RGB mode the uniformity of the observed change is tested, and the implementation fails if consecutive changes are too different. (thisDelta - lastDelta) > 1/4 shade.
The observed results are also checked for monotonicity. When Light_Position[2] reaches 1.0 the color is finally checked to insure that the point has rendered with a fully saturated red.

In CI mode a check is made to ensure that the observed index reaches maxIndex. The observed shade is compared to the theoretical value expected for each setting of SpotLight_Position If the theoretical value is more than 4 indices away from the observed index the test fails.

Error Explanation -

RGB mode: other_color > 0.0.
RGB mode: value < lastValue.
RGB mode: maxObserved < 1.0.
RGB mode: ABS(thisDelta - lastDelta) > 1/4 shade.
CI mode: observed Index not an integer
CI mode: ABS(observed - expected) > 4.0.
CI mode: observedMax < numColor.
CI mode: delta > 2.0.

****

linerast.c
Line Rasterization Test.

Description -

Tests the rasterization of single width lines. A specification of the OpenGL states that there can be only one pixel per column for an X-major line and one pixel per row for a Y-major line. This test will draw a series of lines of random lengths and directions to verify this. Special code for checking near the endpoints of the line is used since the endpoints may stray up to one pixel in any direction from their calculated locations.

Error Explanation -

Failure occurs if there is more than one pixel in the cross-section of the line. Also, an error will result if an endpoint strays more than one pixel from its calculated location.

****

linestip.c
Line Stipple Test.

Description -

Tests line stipple, in particular examining whether correct fragments are hit with different factor values, whether factor is properly clamped, whether the stipple resets with a new line, and whether the stipple does not reset with connected lines. The test include a polygon drawn in line mode, checking that the stipple behaves as it would with a series of connected lines.

IMPORTANT ASSUMPTION: Even though the line specification gives a lot of leeway, we assume horizontal and vertical lines are straight. Endpoints are chosen to be on the pixel center to allow for numerical errors yet still hit the same fragment. Lines may be null-beginning or null-terminated. The test requires that first bit of the stipple pattern be 'on' to test which type of line they are. Data for loops are chosen so that the beginning of each segment hits an 'on' bit.

For all but one section of the test, all the lines are horizontal or vertical. The one section of the test which uses diagonal lines allows the full leeway provided by the line rasterization description in the OpenGL specification. The test searches for the first endpoint within one fragment of the given endpoint coordinates, and begins the stipple at the found point. From then on, it searches for one line fragment per column (for x-major lines).

****

logicop.c
Logicop Test.

Description -

Tests the functionality of logical operations. There are 16 logicop functions: GL_AND, GL_AND_INVERTED, GL_AND_REVERSE, GL_CLEAR, GL_COPY, GL_COPY_INVERTED, GL_EQUIV, GL_INVERT, GL_NAND, GL_NOOP, GL_NOR, GL_OR, GL_OR_INVERTED, GL_OR_REVERSE, GL_SET and GL_XOR. Each one is tested by choosing a random source and destination color, drawing a point using the destination color with logicop disabled, drawing the same point using the source color with logicop enabled, then reading the resulting color and comparing it to a calculated value.

****

mask.c
Mask Test.

Description -

In RGB mode a point is rendered using a mask and a color. The rendered result is read back and compared to the expected result. The implementation fails if the color read back differs from the expected color. The individual color channels of any color used are either 0.0 or 1.0. All possible combinations of color channels are tested, each using all possible masks.

In CI mode a point is rendered with the highest index. A mask is applied to this color, and the resulting color is read back. This value is compared to the mask, and if it differs the implementation fails. The masks used start at 0 and increment by 1 until they reach the highest possible index.

****

miplin.c
Mipmap Linear Test.

Description -

Tests the interpolation between mipmap levels. An eight level deep mipmap is set up, each level distinguished by a different color. A polygon is drawn with texture coordinates set so that the level used for each fragment will ramp from a starting level to the next consecutive level. The starting and ending color is checked to see that the correct levels were used. Monotonicity between each fragment texel is also checked.

****

mipsel.c
Mipmap Selection Test.

Description -

Tests the selection of levels with various magnification and minification filters. An eight level deep mipmap is set up, each level distinguished by a different color. A series of shrinking square polygons are drawn with texture coordinates set so that each polygon uses a different level of the mipmap. This is done by matching the number of fragments of a particular polygon to the number of texels in the corresponding level. The rendered polygon is read back and compared to the correct mipmap level color.

****

mstack.c
Matrix Stack Test.

Description -

Test the functionality of glPushMatrix() and glPopMatrix()
on the three matrix modes (GL_MODELVIEW_MATRIX,
GL_PROJECTION_MATRIX and GL_TEXTURE_MATRIX). Each matrix
stack is pushed and popped to their maximum depth with
unique matrices. Each matrix popped off the stack is
checked to verify stack ordering.

****

mustpass.c
Must Pass Test.

Description -
Minimum OpenGL conformance tests:
Accumulation test.
Alpha plane test.
Bitmap test.
Blend test.
Color buffer clear test.
Clip test.
Color buffer color test.
Depth buffer test.
DisplayList test.
Evaluator test.
Feedback test.
Fog test.
Homogeneous coordinate test.
Lighting test.
LogicOp test.
Pixel transfer test.
Scissor test.
Select test.
Stencil test.
Stipple test.
Texture test.
WriteMask test.

This test examines the implementation for the existance of the above features. If the implementation need not support a feature, that feature is not examined unless it is supported. Typically a primitive is rendered in such a way that if a feature is supported it will render one color, and otherwise the primitive would be a different color. An implementation need only be within 0.49% of the total range of possible colors to verify existance of a feature. (The feedbacktest is different from this).
****

polycull.c
Polygon Culling Test.

Description -

First, with culling disabled, a polygon is drawn both clockwise and counter-clockwise. The polygon should render in both cases. Culling is then enabled. With culling direction set to GL_FRONT, a polygon is drawn both clockwise and counter-clockwise. Only the polygon drawn in the clockwise direction should render. With culling direction is set to GL_BACK, a polygon is drawn both clockwise and counter-clockwise. Only the polygon drawn in the counter-clockwise direction should render.

****

polyface.c
Polygon Face Test.

Description -

Tests the functionality of glFrontFace() with glPolygonMode(). glFrontFace() and glPolygonMode() are set so that when a polygon is drawn, the frame of the polygon should render but the body should not. Both clockwise and counter-clockwise directions for glFrontFace() are checked.

Note that the corners of the polygon are skipped since the OpenGL specification allows polygons drawn in line mode to be off by one pixels. Therefore, there could be missing pixels at the polygon corners.

****

polystip.c
Polygon Stipple Test.

Description -

Tests polygon stipple, checking if correct fragments are on. To determine which fragments are in the polygon, a reference polygon is drawn with a stipple pattern that has all bits on. The same polygon is drawn with a varied stipple pattern, and each fragment is compared with the reference polygon and the stipple bit for that window coordinate.

****

pntrast.c

Point Rasterization Test.

Description -

Points of increasing size (both odd and even sizes) are drawn then checked for correctness.
Point size tested are limited to half the window size, in this case 50.

****

quadrast.c
Quad Rasterization Test.

Description -

This test renders GL_QUADS and GL_QUAD_STRIPS. It checks if those fragments which are colored the primitive's color have centers which are within the convex hull formed by the vertices, and if those fragments which are colored the background color are outside the convex hull of formed by the vertices.

Error Explanation -

Those fragments which are found to be in error are then checked to determine their distance from the nearest primitive edge. If this distance is less than 1/2 of a pixel width the error is ignored.

****

scissor.c
Scissor Test.

Description -

A scissor box is set up with a glScissor() call. A large polygon is then drawn over the box. Only the region defined by the box should render. Scissor boxes of various sizes (including zero width and height) are used to insure completeness of the test.

Error Explanation -

Failure occurs if the scissor box does not scissor to the correct size. Special attention is paid along the bottom and right edges.

****

select.c
Selection Test.

Description -

A clip volume is set up with a near z value of 10.0 and a far z value of 20.0. A primitive is first rendered outside the selection clip volume (by using a z value of 5.0), then inside the clip volume (by using a z value of 15.0). The number of selection hits returned from glRenderMode() is checked as well as the information in the select buffer. Many different primitives are used to insure thorough testing.

****

spclear.c
Stencil Plane Clear Test.

Description -

A clear to zero and a clear to one is performed on the stencil plane and verified. If stencil plane is not available, a clear to one should behave as a clear to zero.

****

spcorner.c
Stencil Plane Corner Test.

Description -

The stencil function is set to GL_NEVER, the stencil operation is set to GL_INCR and the stencil plane is cleared to zero. A point is drawn at each corner of the color buffer. Because of the stencil settings, the corners of the stencil plane should be set to one. The stencil plane is read and examined to verify that the corner points are at the correct location.

Error Explanation -

Failure occurs if any of the corner points cannot be read correctly (implying that the stencil plane’s width and height or the rendering coordinates are not correct).

****

spfunc.c
Stencil Plane Function Test.

Description -

Each stencil function is tested by clearing the stencil buffer to 1, drawing three points with different stencil values (the first point has a stencil value of 0, the second point has a stencil value of 1 and the third point has stencil value of 2) and comparing the rendered result to the expected result.

****

spop.c
Stencil Plane Operation Test.

Description -

Tests the conformance of glStencilOp(). Each stencil operation (GL_KEEP, GL_REPLACE, GL_ZERO, GL_INCR, GL_DECR, GL_INVERT) is tested by initializing the stencil plane to a known value, performing the stencil operation (by setting up the stencil and depth functions and rendering a point) and comparing the stencil plane result to an expected value. The sfail operation is tested by setting the stencil function to GL_NEVER. The dpfail operation is tested by setting the stencil function to GL_ALWAYS and the depth function to GL_NEVER. The dppass operation is tested by setting both the stencil function and depth function to GL_ALWAYS.

****

texbc.c
Texture Border Color Test.

Description -

Tests texture borders. Both border-included and no-border texture maps are tested. For a no-border texture map, the texture coordinates match the texture image. For a border-included texture map, the texture coordinates are adjusted to accommodate for the extra border pixels in the texture image. In either case, the body of the texture has no color while the texture border does. The presence of the texture border can then be verifies.

****

texdecal.c
Texture Decal Test.

Description -

Simple texture test. A polygon is drawn with a texture. The texture coordinates are set so that the texture should appear on the polygon with no distortion. The rendered polygon is read back and compared to the texture pattern.

****

trirast.c
Triangle Rasterization Test.

Description -

This test renders GL_TRIANGLES, and GL_TRIANGLE_STRIPS, and GL_TRIANGLE_FANS. It checks if those fragments which are colored the primitive's color have centers which are within the convex hull formed by the vertices, and if those fragments which are colored the background color are outside the convex hull of formed by the vertices.

Error Explanation -

Those fragments which are found to be in error are then checked to determine their distance from the nearest primitive edge. If this distance is less than 1/2 of a pixel width the error is ignored.

****

tristip.c
Triangle Stipple Test.

Description -

Tests triangle stippling, checking if correct fragments are on. To determine which fragments are in the triangle, a reference primitive is drawn with a solid stipple pattern. The same primitive is drawn with a varied stipple pattern, and each fragment is compared with the reference primitive and the stipple bit for that window coordinate.

Triangles are drawn using GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN and GL_TRIANGLES.

****

vorder.c
Vertex Order Test.

Description -

Test that triangle strips, triangle fans and quad strips preserve proper vertex order when folded back on themselves. Front and back are assigned different colors and drawn counter-clockwise. Test is repeated for each polygon mode.

The test algorithm:

- Use glLightModel to assign different colors to front and back.
- For each of GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP:
  - Generate vertices of a strip/fan which folds back on itself. Each primitive consists of approximately 16 vertices. The folded-back region is constructed so as to not obscure front-facing vertices, except on the crease edge.
- For each polygon mode (GL_POINT, GL_LINE, GL_FILL):
  - Draw the strip/fan.
  - Examine framebuffer at vertices to be sure appropriate front or back color appeared. Since vertex location may be on pixel boundaries, an allowance is made for the vertex to be off one pixel in any direction.

****

vpclamp.c
Viewport Clamp Test.

Description -

Test the behavior of glViewport() when bad values are given. Out of range and negative width and height values should be clamped. A negative width or height value should also set an error condition.

Error Explanation -

Failure occurs if the implementation fails to clamp out of range viewport values or set an error condition for a negative viewport size.

****

xform.c
Transform Test.

Description -

Tests glRotate(), glScale() and glTranslate(). A single point is drawn under various transformations and its new position checked. The order of matrix multiplication is also checked by executing pairs of rotates and translates. The test also verifies that no assumption of matrix mode GL_PROJECTION are tied to glOrtho() and glFrustum(); they should be applied with the current matrix mode. Similarly, glRotate(), glTranslate() and glScale() should not assume GL_MODELVIEW.

The sections of the test contain the following:

Tests single tranformations. In each iteration of a loop, creates either a translation, rotation, or scale call using random data then checks resulting point location.

Tests the order of matrix multiplications. Calls a translation then a scale, checks results, then calls a scale followed by a translation and checks results.

Test that glOrtho() and glFrustum() do not assume GL_PROJECTION transformation. In GL_MODELVIEW mode, make calls to glOrtho() and glFrumtum(). Check that glGet() returns tranformed GL_MODELVIEW matrix. Check that glGet() shows GL_PROJECTION is still identity. For glOrtho(), overwrite GL_PROJECTION matrix identity and verify the point drawn correctly demonstrates the GL_MODELVIEW transform.

Test that glScale(), glTranlate() and glRotate() do not assume GL_MODELVIEW transformation. In GL_PROJECTION mode, make calls to translate, rotate and scale. Check that Get returns proper GL_PROJECTION matrix. Check that Get shows GL_MODELVIEW matrix is still identity. For each transformation, overwrite GL_MODELVIEW matrix with identity and verify that the drawn point demonstrates new GL_PROJECTION matrix.

Error Explanation -

Since values may fall on pixel boundaries, and transformations have numerical error, allowance is made for pixel to be off by one in any direction from expected location.

****

xformmix.c
Matrix Stack Mixing Test.

Description -

Tests that matrix stacks can be mixed. A series of transformations are applied while the matrix mode is switched with each glPushMatrix() and glPopMatrix(). After a series of transformations, a small quad is drawn and its position is checked.

****

xformn.c
Transform Normal Test.

Description -

Tests that the normal vector is correctly multiplied by the inverse of the modelview matrix. To test this, a polygon is drawn with lighting parameters set in such a way that most values drop out and the resulting pixel color is a function of the normal vector.

In RGB mode, the equation reduces to c = max(0, n dot VP) and (n dot VP) reduces to cos(theta). In CI mode, the equation reduces to s'sm which becomes cos(theta)sm.

The polygon is rotated around various axes. Initially the light is fixed while the polygon rotates. At each rotation, the color of the fragment at the center of the window (position 0, 0, 0) is read and compared with expected results, from the equations above. In the second section, the light source rotated about an axis, always at a distance of 1.0 from the origin. This section is repeated, taking both the light and the primitive through rotations about different axes.

****

xformvp.c
Viewport Transformation Test.

Description -

Tests that viewport computations are correct. The test randomly selects values to define a viewport within the window. A point is drawn with this new viewport and its position in the window is examined.

Error Explanation -

An allowance is made for the point position to be off by one pixel in any direction.

****

zbclear.c
Depth buffer Clear Test.

Description -

A clear to zero and a clear to one is performed on the depth buffer and verified. If depth buffer is not available, a clear to one should behave as a clear to zero.

****

zbfunc.c
Depth Buffer Function Test.

Description -

Each depth function is tested by initializing the depth buffer to known values (with the depth function set to GL_ALWAYS, three points are drawn with vertices (0.0, 0.0, 0.5), (1.0, 0.0, 0.5) and (2.0, 0.0, 0.5)), drawing three points with different z values (the first point has vertex (0.0, 0.0, 0.0), the second point has vertex (1.0, 0.0, 0.5) and the third point has vertex of (2.0, 0.0, 1.0)) and comparing the rendered result to the expected result. The initialization of the depth buffer was done this way to avoid floating point inaccuracies.

****

6.5 Conformx tests.

The information printed below can also be found at the top of each file in the X conformance test suite directory.

****

font.c
Font Test.

Description -

This tests compares text produced through the window system with strings produced through display lists with UseFont(). First, the letters of the alphabet, divided into two strings, are written using the window system calls. The TK routine chooses the font and returns the name of the font used. The results are read read from the framebuffer and stored in a local buffer. The same strings are drawn using the GL window system calls, through TK, with the font name returned from the first call. The framebuffer results are also stored to a buffer. The buffers are then compared.

****

readpix.c
glReadPixel() Test.

Description -

Test glReadPixels() by comparing buffer as read by the window system with buffer read by glReadPixels(). An assymetric rectangle is drawn to check that buffer fills left to right, bottom to top.

****

7.0 Equivalent Computational Paths.

Conformance tests are designed to be modular; procedures test use a small set of OpenGL procedures. It is desirable, however, to test combinations of state settings. To those ends we have added a feature to the conformance tests referred to as **Equivalent Computational Paths (ECPs)**. Certain OpenGL rendering modes are enabled using parameters which should not effect the resulting image, yet which may force the implementation to execute through a different code path.

Consider the following scenario:

A smooth shading test runs by itself and passes. However, when it is run after a stenciling test, and the stenciling was accidentally left enabled, the smooth shading test fails. And after further debugging the stenciling test, we find that all of the fragments are being drawn correctly, but the color data assigned to those fragments is wrong.

This type of bug seems perplexing since stenciling should only affect whether a fragment is rendered or not, it should not directly affect the computation of color data.

This behavior is easily explained when we realize that implementations of the OpenGL may compute the color data for a shaded primitive in software if stenciling is enabled, while they might be able to compute the same primitive in hardware if stenciling is disabled. In other words, there is redundant functionality in the implementation.

Since the same conformance tests are used by all implementations of the OpenGL, and since we do not have specific information about the actual architecture of the software or hardware on which the OpenGL may be run, we have to anticipate circumstances or settings which might cause an implementation to switch between redundant computational modules. For any given implementation of the OpenGL, many of these switches will have no effect, and may cause the conformance tests to seem annoyingly repetitive and slow, but a few may have some affect.

Most of the ECPs are asserted by a few simple settings during the initialization of a test. For example, by setting the stencil function to GL_ALWAYS and the stencil operator to GL_KEEP, we may enable stenciling, but it will have no effect on any test which does not use the stencil function. Tests for lighting or texturing, should pass with or without this setting, even if different modules of the implementation are being used to compute the data used by lighting or texturing.

We realize that a sophisticated designer is likely to recognize that GL_ALWAYS and GL_KEEP is essentially the same as disabling stenciling, so we use a stencil ECP where the stencil function is GL_GEQUAL and the stencil plane is cleared to 0. In implementations which have no stencil buffer, this repetition of tests will be superfluous, but it is necessary to achieve a degree of uniformity across those implementations which do support stencil buffers.

Certain ECPs cannot be used with certain tests. For example, the copypix test compares two rectangles drawn in different parts of the screen, expecting them to be the same. The dithering ECP cannot be used for this test, for the dithering algorithm depends upon screen coordinates and could cause the rectangles to not match exactly. There are mechanisms built into the conformance test to prevent a test from being run with ECPs which would cause it to fail conformant implementations.

Similarly certain ECPs may not be run at the same time. For example, there is an ECP involving alpha blending, and there is an ECP involving anti-aliasing. Anti-aliasing works correctly when the source blend function is set to GL_SRC_ALPHA, and the destination blend function is GL_ONE. A blending ECP which changes these functions must not be run with the anti-aliasing ECP. The conformance test protects against this also.

7.1 Characteristics of an ECP.

We considered the following characteristics when looking for ECPs:

1) *We had to be able to specify parameters which should not affect the image*. For example, enabling stipple with a solid on pattern should not affect any of the current tests.

2) *We looked for things which would switch operations between software and hardware*. For example, rendering a polygon without stippling might be done in hardware where as rendering the same polygon with stippling might be done in software.

3) *We looked for places where computational simplifications could enhance performance*. For example, if the specular exponent is identically 1.0, there is no need to use a time consuming exponential computation.

7.2 Currently implemented ECPs.

PathAlias. Some tests will be run with antialiasing enabled as well as disabled. Note the dependency between this ECP and **PathBlend**.

PathAlpha. Tests will be run with the alpha test enabled and with alpha test disabled. Note the dependency between this ECP and **PathBlend**.

PathBlend. Blend parameters are set so that tests will be unaffected, whether or not the implementations have alpha planes. We avoid settings which will interact badly with **PathAlias** and the **PathAlpha**.

PathDepth. Settings are chosen which will not interfere with the color and primitive tests. We avoid settings which will interact badly with **PathStencil**.

PathDither. Tests are run with dither enabled or disabled. It is disabled for all lighting tests and some geometry tests. Currently, dither is disabled for a large number of tests, since the tests rely upon finding a particular color at a fragment and the dither function allows a great deal of implementation-dependent differences.

PathFog. Several fog parameters are chosen which have little or no affect on colors.

PathLogicOp. Logicops are chosen which should allow all fragments to be rendered.

PathShade. Many tests should pass if smooth shading is enabled or if it is disabled.

PathStencil. Stencil operators and functions are chosen which will allow all fragments to pass. We avoid settings which will interact badly with **PathDepth**.

PathStipple. Sets line stipple and polygon stipple with a solid pattern.

7.3 Implementation.

There are 4 ECP levels. These levels are:

Level 0 - use default settings.

Level 1 - all ECPs on.

Level 2 - one ECP on, all others off (loop through all ECPs).

Level 3 - one ECP off, all others on (loop through all ECPs).

An ECP may be thought of as a set of rendering options. Each set is described in a unique data structure corresponding to the options described above (alias, alpha, blend, depth, dither, fog, logicop, shade, stencil, stipple).

For example, a test data structure may allow the following ECPs:

```
{
      "Divide By Zero", "divzero.c",           /* name and ID */
      DivZeroExec, DivZeroExec,                /* RGB and color index tests */
      COLOR_NONE,                              /* color requirements */
      PATH_ALIAS | PATH_ALPHA | PATH_BLEND |   /* ECPs allowed */
      PATH_DEPTH | PATH_DITHER | PATH_FOG |
      PATH_LOGICOP | PATH_SHADE |
      PATH_STENCIL |PATH_STIPPLE
},
```

And the stenciling ECP data structure may have one of two entries:

```
stencilPathCustomRec stencilPath1 = {
      PATHTEST_DEFAULT                         /* use default setting */
};
```

```
stencilPathCustomRec stencilPath2 = {
       PATHTEST_CUSTOM,                          /* use non-default */
       PATHDATA_ENABLE,                          /* enable stencil */
       0,                                        /* clear value */
       0,                                        /* mask */
       GL_GEQUAL,                                /* func */
       1, 0,                                     /* ref, mask */
       GL_KEEP, GL_KEEP, GL_KEEP                 /* StencilOp */
};
```

These 2 sets are used by the ECP driver code to set up equivalent computational paths when the “Divide By Zero” test is executed.