Avoiding 16 Common OpenGL Pitfalls

Every software engineer who has programmed long enough has a war story about some insidious bug that induced head scratching, late night debugging, and probably even schedule delays. More often than we programmers care to admit, the bug turns out to be self-inflicted. The difference between an experienced programmer and a novice is knowing the good practices to use and the bad practices to avoid so those self-inflicted bugs are kept to a minimum.

A programming interface pitfall is a self-inflicted bug that is the result of a misunderstanding about how a particular programming interface behaves. The pitfall may be the fault of the programming interface itself or its documentation, but it is often simply a failure on the programmer's part to fully appreciate the interface's specified behavior. Often the same set of basic pitfalls plagues novice programmers because they simply have not yet learned the intricacies of a new programming interface.

You can learn about the programming interface pitfalls in two ways: The hard way and the easy way. The hard way is to experience them one by one, late at night, and with a deadline hanging over your head. As a wise main once explained, "Experience is a good teacher, but her fees are very high." The easy way is to benefit from the experience of others.

This is your opportunity to learn how to avoid 19 software pitfalls common to beginning and intermediate OpenGL programmers. This is your chance to spend a bit of time reading now to avoid much grief and frustration down the line. I will be honest; many of these pitfalls I learned the hard way instead of the easy way. If you program OpenGL seriously, I am confident that the advice below will make you a better OpenGL programmer.

If you are a beginning OpenGL programmer, some of the discussion below might be about topics that you have not yet encountered. This is not the place for a complete introduction to some of the more complex OpenGL topics covered such as mipmapped texture mapping or OpenGL's pixel transfer modes. Feel free to simply skim over sections that may be too advanced. As you develop as an OpenGL programmer, the advice will become more worthwhile.

1. Improperly Scaling Normals for Lighting

Enabling lighting in OpenGL is a way to make your surfaces appear more realistic. Proper use of OpenGL's lighting model provides subtle clues to the viewer about the curvature and orientation of surfaces in your scene.

When you render geometry with lighting enabled, you supply normal vectors that indicate the orientation of the surface at each vertex. Surface normals are used when calculating diffuse and specular lighting effects. For example, here is a single rectangular patch that includes surface normals:

```
glBegin(GL_QUADS);

glNormal3f(0.181636,-0.25,0.951057);

glVertex3f(0.549,-0.756,0.261);

glNormal3f(0.095492,-0.29389,0.95106);

glVertex3f(0.288,-0.889,0.261);

glNormal3f(0.18164,-0.55902,0.80902);

glVertex3f(0.312,-0.962,0.222);

glNormal3f(0.34549,-0.47553,0.80902);

glVertex3f(0.594,-0.818,0.222);

glEnd();
```

The *x*, *y*, and *z* parameters for each
`glNormal3f`
call specify a direction vector. If you do the math, you will find that the length of each normal vector above is essentially 1.0. Using the first
`glNormal3f`
call as an example, observe that:

$\texttt{sqrt}(0.181636^2 + -0.25^2 + 0.951057^2)$
»
 1.0

For OpenGL's lighting equations to operate properly, the assumption OpenGL makes by default is that the normals passed to it are vectors of length 1.0.

However, consider what happens if before executing the above OpenGL primitive,
`glScalef`
is used to shrink or enlarge subsequent OpenGL geometric primitives. For example:

```
glMatrixMode(GL_MODELVIEW);

glScalef(3.0, 3.0, 3.0);
```

The above call causes subsequent vertices to be enlarged by a factor of three in each of the *x*, *y*, and *z* directions by scaling OpenGL's modelview matrix.
`glScalef`
can be useful for enlarging or shrinking geometric objects, but you must be careful because OpenGL transforms normals using a version of the modelview matrix called the *inverse transpose modelview* matrix. Any enlarging or shrinking of vertices during the modelview transformation *also* changes the length of normals.

Here is the pitfall: Any model view scaling that occurs is likely to mess up OpenGL's lighting equations. Remember, the lighting equations assume that normals have a length of 1.0. The symptom of incorrectly scaled normals is that the lit surfaces appear too dim or too bright depending on whether the normals enlarged or shrunk.

The simplest way to avoid this pitfall is by calling:

```
glEnable(GL_NORMALIZE);
```

This mode is not enabled by default because it involves several additional calculations. Enabling the mode forces OpenGL to normalize transformed normals to be of unit length before using the normals in OpenGL's lighting equations. While this corrects potential lighting problems introduced by scaling, it also slows OpenGL's vertex processing speed since normalization requires extra operations, including several multiplies and an expensive reciprocal square root operation. While you may argue whether this mode should be enabled by default or not, OpenGL's designers thought it better to make the default case be the fast one. Once you are aware of the need for this mode, it is easy to enable when you know you need it.

There are two other ways to avoid problems from scaled normals that may let you avoid the performance penalty of enabling
`GL_NORMALIZE`
. One is simply to not use
`glScalef`
to scale vertices. If you need to scale vertices, try scaling the vertices before sending them to OpenGL. Referring to the above example, if the application simply multiplied each
`glVertex3f`
by 3, you could eliminate the need for the above
`glScalef`
without having the enable the
`GL_NORMALIZE`
mode.

Note that while
`glScalef`
is problematic, you can safely use
`glTranslatef`
and
`glRotatef`
because these routines change the modelview matrix transformation without introducing any scaling effects. Also, be aware that
`glMatrixMultf`
can also be a source of normal scaling problems if the matrix you multiply by introduces scaling effects.

The other option is to adjust the normal vectors passed to OpenGL so that after the inverse transpose modelview transformation, the resulting normal will become a unit vector. For example, if the earlier
`glScalef`
call tripled the vertex coordinates, we could correct for this corresponding thirding effect on the transformed normals by pre-multiplying each normal component by 3.

OpenGL 1.2 adds a new
`glEnable`
mode called
`GL_RESCALE_NORMAL`
that is potentially more efficient than the
`GL_NORMALIZE`
mode. Instead of performing a true normalization of the transformed normal vector, the transformed normal vector is scaled based on a scale factor computed from the inverse modelview matrix's diagonal terms.
`GL_RESCALE_NORMAL`
can be used when the modelview matrix has a uniform scaling factor.

2. Poor Tessellation Hurts Lighting

OpenGL's lighting calculations are done *per-vertex*. This means that the shading calculations due to light sources interacting with the surface material of a 3D object are only calculated at the object's vertices. Typically, OpenGL just interpolates or *smooth shades* between vertex colors. OpenGL's per-vertex lighting works pretty well except when a lighting effect such as a specular highlight or a spotlight is lost or blurred because the effect is not sufficiently sampled by an object's vertices. Such under-sampling of lighting effects occurs when objects are coarsely modeled to use a minimal number of vertices.

Figure 1 shows an example of this problem. The top left and top right cubes each have an identically configured OpenGL spotlight light source shining directly on each cube. The left cube has a nicely defined spotlight pattern; the right cube lacks any clearly defined spotlight pattern. The key difference between the two models is the number of vertices used to model each cube. The left cube models each surface with over 120 distinct vertices; the right cube has only 4 vertices.

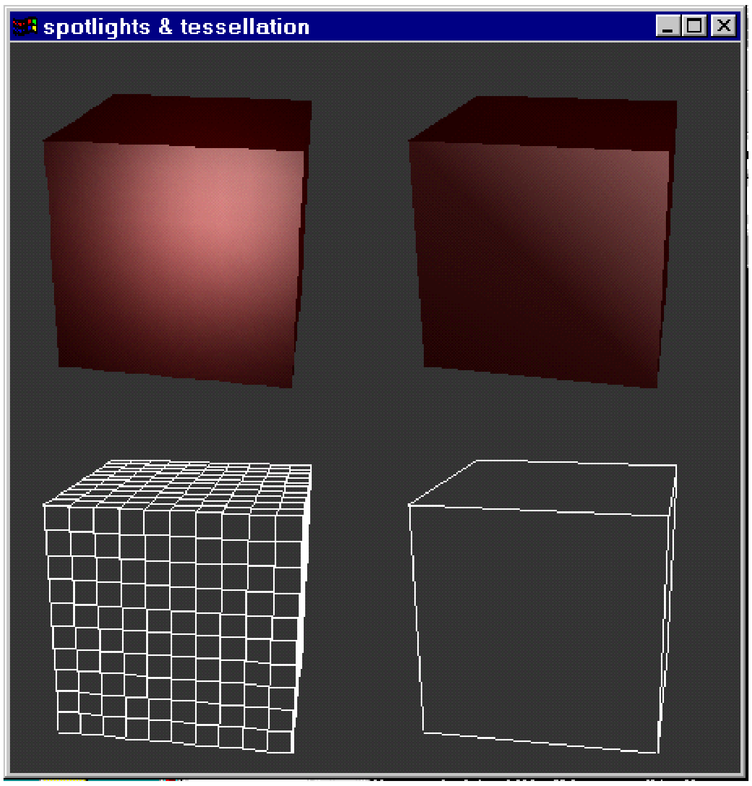

Figure 1: Two cubes rendered with identical OpenGL spotlight enabled.

(The lines should all be connected but are not due to resampling in the image above.)

At the extreme, if you tessellate the cube to the point that each polygon making up the cube is no larger than a pixel, the lighting effect will essentially become per-pixel. The problem is that the rendering will probably no longer be interactive. One good thing about per-vertex lighting is that you decide how to trade off rendering speed for lighting fidelity.

Smooth shading between lit vertices helps when the color changes are gradual and fairly linear. The problem is that effects such as spotlights, specular highlights, and non-linear light source attenuation are often not gradual. OpenGL's lighting model only does a good job capturing these effects if the objects involved are reasonably tessellated.

Novice OpenGL programmers are often tempted to enable OpenGL's spotlight functionality and shine a spotlight on a wall modeled as a single huge polygon. Unfortunately, no sharp spotlight pattern will appear as the novice intended; you probably will not see any spotlight affect at all. The problem is that the spotlight's cutoff means that the extreme corners of the wall where the vertices are specified get *no* contribution from the spotlight and since those are the only vertices the wall has, there will be no spotlight pattern on the wall.

If you use spotlights, make sure that you have sufficiently tessellated the lit objects in your scene with enough vertices to capture the spotlight effect. There is a speed/quality tradeoff here: More vertices mean better lighting effects, but also increases the amount of vertex transformation required to render the scene.

Specular highlights (such as the bright spot you often see on a pool ball) also require sufficiently tessellated objects to capture the specular highlight well.

Keep in mind that if you use more linear lighting effects such as ambient and diffuse lighting effects where there are typically not *sharp* lighting changes, you can get good lighting effects with even fairly coarse tessellation.

If you do want *both* high quality and high-speed lighting effects, one option is to try using multi-pass texturing techniques to texture specular highlights and spotlight patterns onto objects in your scene. Texturing is a per-fragment operation so you can correctly capture per-fragment lighting effects. This can be involved, but such techniques can deliver fast, high-quality lighting effects when used effectively.

3. Remember Your Matrix Mode

OpenGL has a number of 4 by 4 matrices that control the transformation of vertices, normals, and texture coordinates. The core OpenGL standard specifies the modelview matrix, the projection matrix, and the texture matrix.

Most OpenGL programmers quickly become familiar with the modelview and projection matrices. The modelview matrix controls the viewing and modeling transformations for your scene. The projection matrix defines the view frustum and controls the how the 3D scene is projected into a 2D image. The texture matrix may be unfamiliar to some; it allows you to transform texture coordinates to accomplish effects such as projected textures or sliding a texture image across a geometric surface.

A single set of matrix manipulation commands controls all types of OpenGL matrices:
`glScalef`
,
`glTranslatef`
,
`glRotatef`
,
`glLoadIdentity`
,
`glMultMatrixf`
, and several other commands. For efficient saving and restoring of matrix state, OpenGL provides the
`glPushMatrix`
and
`glPopMatrix`
commands; each matrix type has its own a stack of matrices.

None of the matrix manipulation commands have an explicit parameter to control which matrix they affect. Instead, OpenGL maintains a current matrix mode that determines which matrix type the previously mentioned matrix manipulation commands actually affects. To change the matrix mode, use the
`glMatrixMode`
command. For example:

```
glMatrixMode(GL_PROJECTION);

/* Now update the projection matrix. */

glLoadIdentity();

glFrustum(-1, 1, -1, 1, 0.0, 40.0);

glMatrixMode(GL_MODELVIEW);

/* Now update the modelview matrix. */

glPushMatrix();

  glRotatef(45.0, 1.0, 1.0, 1.0);

  render();

glPopMatrix();
```

A common pitfall is forgetting the current setting of the matrix mode and performing operations on the wrong matrix stack. If later pre assumes the matrix mode is set to a particular state, you both fail to update the matrix you intended and screw up whatever the actual current matrix is.

If this can trip up the unwary programmer, why would OpenGL have a matrix mode? Would it not make sense for each matrix manipulation routine to also pass in the matrix that it should manipulate? The answer is simple: lower overhead. OpenGL's design optimizes for the common case. In real programs, matrix manipulations occur more often than matrix mode changes. The common case is a sequence of matrix operations all updating the same matrix type. Therefore, typical OpenGL usage is optimized by controlling which matrix is manipulated based on the current matrix mode. When you call
`glMatrixMode`
, OpenGL configures the matrix manipulation commands to efficiently update the current matrix type. This saves time compared to deciding which matrix to update every time a matrix manipulation is performed.

In practice, because a given matrix type does tend to be updated repeatedly before switching to a different matrix, the lower overhead for matrix manipulation more than makes up for the programmer's burden of ensuring the matrix mode is properly set before matrix manipulation.

A simple program-wide policy for OpenGL matrix manipulation helps avoid pitfalls when manipulating matrices. Such a policy would require any premanipulating a matrix to first call
`glMatrixMode`
to always update the intended matrix. However in most programs, the modelview matrix is manipulated quite frequently during rendering and the other matrices change considerably less frequently overall. If this is the case, a better policy is that routines can assume the matrix mode is set to update the modelview matrix. Routines that need to update a different matrix are responsible to switch back to the modelview matrix after manipulating one of the other matrices.

Here is an example of how OpenGL's matrix mode can get you into trouble. Consider a program written to keep a constant aspect ratio for an OpenGL-rendered scene in a window. Maintaining the aspect ratio requires updating the projection matrix whenever the window is resized. OpenGL programs typically also adjust the OpenGL viewport in response to a window resize so the code to handle a window resize notification might look like this:

```
void

doResize(int newWidth, int newHeight)

{

 GLfloat aspectRatio = (GLfloat)newWidth / (GLfloat)newHeight;

  glViewport(0, 0, newWidth, newHeight);

  glMatrixMode(GL_PROJECTION);

  glLoadIdentity();

  gluPerspective(60.0, aspectRatio, 0.1, 40.0);

  /* WARNING: matrix mode left as projection! */

}
```

If this code fragment is from a typical OpenGL program,
`doResize`
is one of the few times or even only time the projection matrix gets changed after initialization. This means that it makes sense to add to a final
`glMatrixMode(GL_MODELVIEW)`
call to
`doResize`
to switch back to the modelview matrix. This allows the window's redraw code safely assume the current matrix mode is set to update the modelview matrix and eliminate a call to

`glMatrixMode`
. Since window redraws often repeatedly update the modelview matrix, and redraws occur considerably more frequently than window resizes, this is generally a good approach.

A tempting approach might be to call
`glGetIntegerv`
to retrieve the current matrix mode state and then only change the matrix mode when it was not what you need it to be. After performing its matrix manipulations, you could even restore the original matrix mode state.

This is however almost certainly a bad approach. OpenGL is designed for fast rendering and setting state; retrieving OpenGL state is often considerably slower than simply setting the state the way you require. As a rule,
`glGetIntegerv`
and related state retrieval routines should only be used for debugging or retrieving OpenGL implementation limits. They should *never* be used in performance critical code. On faster OpenGL implementations where much of OpenGL's state is maintained within the graphics hardware, the relative cost of state retrieval commands is considerably higher than in largely software-based OpenGL implementations. This is because state retrieval calls must stall the graphics hardware to return the requested state. When users run OpenGL programs on high-performance expensive graphics hardware and do not see the performance gains they expect, in many cases the reason is invocations of state retrieval commands that end up stalling the hardware to retrieve OpenGL state.

In cases where you do need to make sure that you restore the previous matrix mode after changing it, try using
`glPushAttrib`
with the
`GL_TRANSFORM_BIT`
bit set and then use
`glPopAttrib`
to restore the matrix mode as needed. Pushing and popping attributes on the attribute stack can be more efficient than reading back the state and later restoring it. This is because manipulating the attribute stack can completely avoid stalling the hardware if the attribute stack exists within the hardware. Still the attribute stack is not particularly efficient since all the OpenGL transform state (including clipping planes and the normalize flag) must also be pushed and popped.

The advice in this section is focused on the matrix mode state, but pitfalls that relate to state changing and restoring are common in OpenGL. OpenGL's explicit state model is extremely well suited to the stateful nature of graphics hardware, but can be an unwelcome burden for programmers not used to managing graphics state. With a little experience though, managing OpenGL state becomes second nature and helps ensure good hardware utilization.

The chief advantage of OpenGL's stateful approach is that well-written OpenGL rendering code can minimize state changes so that OpenGL can maximize rendering performance. A graphics- interface that tries to hide the inherently stateful nature of well-designed graphics hardware ends up either forcing redundant state changes or adds extra overhead by trying to eliminate such redundant state changes. Both approaches give up performance for convenience. A smarter approach is relying on the application or a high-level graphics library to manage graphics state. Such a high-level approach is typically more efficient in its utilization of fast graphics hardware when compared to attempts to manage graphics state in a low-level library without high-level knowledge of how the operations are being used.

If you want more convenient state management, consider using a high-level graphics library such as Open Inventor or IRIS Performer that provide both a convenient programming model and efficient high-level management of OpenGL state changes.

4. Overflowing the Projection Matrix Stack

OpenGL's
`glPushMatrix`
and
`glPopMatrix`
commands make it very easy to perform a set of cumulative matrix operations, do rendering, and then restore the matrix state to that before the matrix operations and rendering. This is very handy when doing hierarchical modeling

during rendering operations.

For efficiency reasons and to permit the matrix stacks to exist within dedicated graphics hardware, the size of OpenGL's various matrix stacks are limited. OpenGL mandates that all implementations *must* provide at least a 32-entry modelview matrix stack, a 2-entry projection matrix stack, and a 2-entry texture matrix stack. Implementations are free to provide larger stacks, and
`glGetIntergerv`
provides a means to query an implementation's actual maximum depth.

Calling
`glPushMatrix`
when the current matrix mode stack is already at its maximum depth generates a
`GL_STACK_UNDERFLOW`
error and the responsible
`glPushMatrix`
is ignored. OpenGL applications guaranteed to run correctly on all OpenGL implementations should respect the minimum stack limits cited above (or better yet, query the implementation's true stack limit and respect that).

This can become a pitfall when software-based OpenGL implementations implement stack depth limits that exceed the minimum limits. Because these stacks are maintained in general purpose memory and not within dedicated graphics hardware, there is no substantial expense to permitting larger or even unlimited matrix stacks as there is when the matrix stacks are implemented in dedicated hardware. If you write your OpenGL program and test it against such implementations with large or unlimited stack sizes, you may not notice that you exceeded a matrix stack limit that would exist on an OpenGL implementation that only implemented OpenGL's mandated minimum stack limits.

The 32 required modelview stack entries will not be exceeded by most applications (it can still be done so be careful). However, programmers should be on guard not to exceed the projection and texture matrix stack limits since these stacks may have as few as 2 entries. In general, situations where you actually need a projection or texture matrix that exceed two entries are quite rare and generally avoidable.

Consider this example where an application uses two projection matrix stack entries for updating a window:

```
void

renderWindow(void)

{

  render3Dview();

  glPushMatrix();

    glMatrixMode(GL_MODELVIEW);

    glLoadIdentity();

    glMatrixMode(GL_PROJECTION);

    glPushMatrix();

      glLoadIdentity();

      gluOrtho2D(0, 1, 0, 1);

      render2Doverlay();

    glPopMatrix();

  glPopMatrix();

  glMatrixMode(GL_MODELVIEW);
}
```

The window renders a 3D scene with a 3D perspective projection matrix (initialization not shown), then switches to a simple 2D orthographic projection matrix to draw a 2D overlay.

Be careful because if the
`render2Doverlay`
tries to push the projection matrix again, the projection matrix stack will overflow on some machines. While using a matrix push, cumulative matrix operations, and a matrix pop is a natural means to accomplish hierarchical modeling, the projection and texture matrices rarely require this capability. In general, changes to the projection matrix are to switch to an entirely different view (not to make a cumulative matrix change to later be undone). A simple matrix switch (reload) does not need a push and pop stack operation.

If you find yourself attempting to push the projection or texture matrices beyond two entries, consider if there is a simpler way to accomplish your manipulations that will not overflow these stacks. If not, you are introducing a latent interoperability problem when you program is run on high-performance hardware-intensive OpenGL implementations that implement limited projection and texture matrix stacks.

5. Not Setting All Mipmap Levels

When you desire high-quality texture mapping, you will typically specify a mipmapped texture filter. Mipmapping lets you specify multiple levels of detail for a texture image. Each level of detail is half the size of the previous level of detail in each dimension. So if your initial texture image is an image of size 32x32, the lower levels of detail will be of size 16x16, 8x8, 4x4, 2x2, and 1x1. Typically, you use the
`gluBuild2DMipmaps`
routine to automatically construct the lower levels of details from you original image. This routine re-samples the original image at each level of detail so that the image is available at each of the various smaller sizes.

Mipmap texture filtering means that instead of applying texels from a single high-resolution texture image, OpenGL automatically selects from the best pre-filtered level of detail. Mipmapping avoids distracting visual artifacts that occur when a distant textured object under-samples its associated texture image. With a mipmapped minimization filter enabled, instead of under-sampling a single high resolution texture image, OpenGL will automatically select the most appropriate levels of detail.

One pitfall to be aware of is that if you do not specify every necessary level of detail, OpenGL will silently act as if texturing is not enabled. The OpenGL specification is very clear about this: "If texturing is enabled (and
`TEXTURE_MIN_FILTER`
is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays 0 through *n* is incomplete, based on the dimensions of array 0, then it is as if texture mapping were disabled."

The pitfall typically catches you when you switch from using a non-mipmapped texture filter (like
`GL_LINEAR`
) to a mipmapped filter, but you forget to build complete mipmap levels. For example, say you enabled non-mipmapped texture mapping like this:

```
glEnable(GL_TEXTURE_2D);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 3, width, height, GL_RGB, GL_UNSIGNED_BYTE, imageData);
```

At this point, you could render non-mipmapped textured primitives. Where you could get tripped up is if you naively simply enabled a mipmapped minification filter. For example:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

The problem is that you have changed the minification filter, but not supplied a complete set of mipmap levels. Not only do you not get the filtering mode you requested, but also subsequent rendering happens as if texture mapping were not even enabled.

The simple way to avoid this pitfall is to use
`gluBuild2DMipmaps`
(or
`gluBuild1DMipmaps`
for 1D texture mapping) whenever you are planning to use a mipmapped minification filter. So this works:

```
glEnable(GL_TEXTURE_2D);

glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);

gluBuild2DMipmaps(GL_TEXTURE_2D, depth, width, height, GL_RGB, GL_UNSIGNED_BYTE, imageData);
```

The above code uses a mipmap filter and uses
`gluBuild2DMipmaps`
to make sure all the levels are populated correctly. Subsequent rendering is not just textured, but properly uses mipmapped filtering.

Also, understand that OpenGL considers the mipmap levels incomplete not simply because you have not specified all the mipmap levels, but also if the various mipmap levels are inconsistent. This means that you must consistently specify border pixels and each successive level must be half the size of the previous level in each dimension.

6. Reading Back Luminance Pixels

You can use OpenGL's
`glReadPixels`
command to read back rectangular regions of a window into your program's memory space. While reading back a color buffer as RGB or RGBA values is straightforward, OpenGL also lets you read back luminance values, but it can a bit tricky to get what you probably expect. Retrieving luminance values is useful if you want to generate a grayscale image.

When you read back luminance values, the conversion to luminance is done as a simple addition of the distinct red, green, and blue components with result clamped between 0.0 and 1.0. There is a subtle catch to this. Say the pixel you are reading back is 0.5 red, 0.5 green, and 0.5 blue. You would expect the result to then be a medium gray value. However, just adding these components would give 1.5 that would be clamped to 1.0. Instead of being a luminance value of 0.5, as you would expect, you get pure white.

A naive reading of luminance values results in a substantially brighter image than you would expect with a high likelihood of many pixels being saturated white.

The right solution would be to scale each red, green, and blue component appropriately. Fortunately, OpenGL's pixel transfer operations allow you to accomplish this with a great deal of flexibility. OpenGL lets you scale and bias each component separately when you send pixel data through OpenGL.

For example, if you wanted each color component to be evenly averaged during pixel read back, you would change OpenGL's default pixel transfer state like this:

```
glPixelTransferf(GL_RED_SCALE,0.3333);

glPixelTransferf(GL_GREEN_SCALE,0.3334);

glPixelTransferf(GL_BLUE_SCALE,0.3333);
```

With OpenGL's state set this way,
`glReadPixels`
will have cut each color component by a third before adding the components during luminance conversion. In the

previous example of reading back a pixel composed of 0.5 red, 0.5 green, and 0.5 blue, the resulting luminance value is 0.5.

However, as you may be aware, your eye does not equally perceive the contribution of the red, green, and blue color components. A standard linear weighting for combining red, green, and blue into luminance was defined by the National Television Standard Committee (NTSC) when the US color television format was standardized. These weightings are based on the human eye's sensitivity to different wavelengths of visible light and are based on extensive research. To set up OpenGL to convert RGB to luminance according to the NTSC standard, you would change OpenGL's default pixel transfer state like this:

```
glPixelTransferf(GL_RED_SCALE, 0.299);

glPixelTransferf(GL_GREEN_SCALE, 0.587);

glPixelTransferf(GL_BLUE_SCALE, 0.114);
```

If you are reading back a luminance version of an RGB image that is intended for human viewing, you probably will want to use the NTSC scale factors.

Something to appreciate in all this is how OpenGL itself does not mandate a particular scale factor or bias for combining color components into a luminance value; instead, OpenGL's flexible pixel path capabilities give the application control. For example, you could easily read back a luminance image where you had suppressed any contribution from the green color component if that was valuable to you by setting the green pixel transfer scale to be 0.0 and re-weighting red and blue appropriately.

You could also use the biasing capability of OpenGL's pixel transfer path to enhance the contribution of red in your image by adding a bias like this:

```
glPixelTransferf(GL_RED_BIAS, 0.1);
```

That will add 0.1 to each red component as it is read back.Please note that the default scale factor is 1.0 and the default bias is 0.0. Also be aware that these same modes are not simply used for the luminance read back case, but *all* pixel or texture copying, reading, or writing. If you program changes the scales and biases for reading luminance values, it will probably want to restore the default pixel transfer modes when downloading textures.

7. Watch Your Pixel Store Alignment

OpenGL's pixel store state controls how a pixel rectangle or texture is read from or written to your application's memory. Consider what happens when you call
`glDrawPixels`
. You pass a pointer to the pixel rectangle to OpenGL. But how exactly do pixels in your application's linear address space get turned into an image?

The answer sounds like it should be straightforward. Since
`glDrawPixels`
takes a width and height in pixels and a format (that implies some number of bytes per pixel), you could just assume the pixels were all packed in a tight array based on the parameters passed to
`glDrawPixels`
. Each row of pixels would immediately follow the previous row.

In practice though, applications often need to extract a sub-rectangle of pixels from a larger packed pixel rectangle. Or for performance reasons, each row of pixels is setup to begin on some regular byte alignment. Or the pixel data was read from a file generated on a machine with a different byte order (Intel and DEC processors are little-endian; Sun, SGI, and Motorola processors are big-endian).

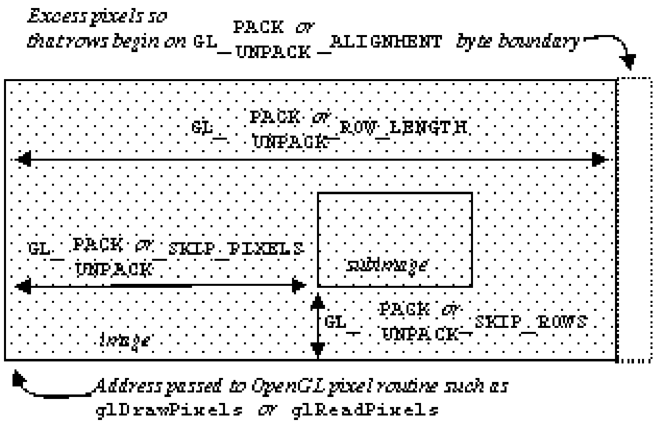
Excess pixels so that rows begin on GL_PACK or UNPACK_ALIGNMENT byte boundary
GL_PACK or UNPACK_ROW_LENGTH
GL_PACK or UNPACK_SKIP_PIXELS
subimage
GL_PACK or UNPACK_SKIP_ROWS
image
Address passed to OpenGL pixel routine such as glDrawPixels or glReadPixels

Figure 2: Relationship of the image layout pixel store modes.

So OpenGL's pixel store state determines how bytes in your application's address space get unpacked from or packed to OpenGL images. Figure 2 shows how the pixel state determines the image layout. In addition to the image layout, other pixel store state determines the byte order and bit ordering for pixel data.

One likely source of surprise for OpenGL programmers is the default state of the
`GL_PACK_ALIGNMENT`
and
`GL_UNPACK_ALIGNMENT`
values. Instead of being 1, meaning that pixels are packed into rows with no extra bytes between rows, the actual default for these modes is 4.

Say that your application needs to read back an 11 by 8 pixel area of the screen as RGB pixels (3 bytes per pixel, one byte per color component). The following
`glReadPixels`
call would read the pixels:

```
glReadPixels(x, y, 11, 8, GL_RGB, GL_UNSIGNED_BYTE, pixels);
```

How large should the pixels array need to be to store the image? Assume that the
`GL_UNPACK_ALIGNMENT`
state is still 4 (the initial value). Naively, your application might call:

```
pixels = (GLubyte*) malloc(3 * 11 * 8); /* Wrong! */
```

Unfortunately, the above code is wrong since it does not account for OpenGL's default 4-byte row alignment. Each row of pixels will be 33 bytes wide, but then each row is padded to be 4 byte aligned. The effective row width in bytes is then 36. The above
`malloc`
call will not allocate enough space; the result is that
`glReadPixels`
will write several pixels beyond the allocated range and corrupt memory.

With a 4 byte row alignment, the actual space required is not *simply BytesPerPixel* × *Width* × *Height*, but instead *((BytesPerPixel* × *Width + 3) >> 2) << 2)* × *Height*. Despite the fact that OpenGL's initial pack and unpack alignment state is 4, most programs should not use a 4 byte row alignment and instead request that OpenGL tightly pack and unpack pixel rows. To avoid the complications of excess bytes at the end of pixel rows for alignment, change OpenGL's row alignment state to be "tight" like this:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

glPixelStorei(GL_PACK_ALIGNMENT, 1);
```

Be extra cautious when your program is written assuming a 1 byte row alignment because bugs caused by OpenGL's initial 4 byte row alignment can easily go unnoticed. For example, if such a program is tested only with images and textures of width divisible by 4, no memory corruption problem is noticed since the test images and textures result in a tight row packing. And because lots of textures and images, by luck or design, have a width divisible by 4, such a bug can easily slip by your testing. However, the memory corruption bug is bound to surface as soon as a customer tries to load a 37 pixel width image.

Unless you *really* want a row alignment of 4, be sure you change this state when using pixel rectangles, 2D and 1D textures, bitmaps, and stipple patterns. And remember that there is a distinct pack and unpack row alignment.

8. Know Your Pixel Store State

Keep in mind that your pixel store state gets used for textures, pixel rectangles, stipple patterns, and bitmaps. Depending on what sort of 2D image data you are passing to (or reading back from) OpenGL, you may need to load the pixel store unpack (or pack) state.

Not properly configuring the pixel store state (as described in the previous section) is one common pitfall. Yet another pitfall is changing the pixel store modes to those needed by a particular OpenGL commands and later issuing some other OpenGL commands requiring the original pixel store mode settings. To be on the safe side, it is usually a good idea to save and restore the previous pixel store modes when you need to change them.

Here is an example of such a save and restore. The following code saves the pixel store unpack modes:

```
GLint swapbytes, lsbfirst, rowlength, skiprows, skippixels, alignment;

/* Save current pixel store state. */

glGetIntegerv(GL_UNPACK_SWAP_BYTES, &swapbytes);

glGetIntegerv(GL_UNPACK_LSB_FIRST, &lsbfirst);

glGetIntegerv(GL_UNPACK_ROW_LENGTH, &rowlength);

glGetIntegerv(GL_UNPACK_SKIP_ROWS, &skiprows);

glGetIntegerv(GL_UNPACK_SKIP_PIXELS, &skippixels);

glGetIntegerv(GL_UNPACK_ALIGNMENT, &alignment);

/* Set desired pixel store state. */

glPixelStorei(GL_UNPACK_SWAP_BYTES, GL_FALSE);

glPixelStorei(GL_UNPACK_LSB_FIRST, GL_FALSE);

glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);

glPixelStorei(GL_UNPACK_SKIP_ROWS, 0);

glPixelStorei(GL_UNPACK_SKIP_PIXELS, 0);

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Then, this code restores the pixel store unpack modes:

```
/* Restore current pixel store state. */

glPixelStorei(GL_UNPACK_SWAP_BYTES, swapbytes);

glPixelStorei(GL_UNPACK_LSB_FIRST, lsbfirst);

glPixelStorei(GL_UNPACK_ROW_LENGTH, rowlength);
glPixelStorei(GL_UNPACK_SKIP_ROWS, skiprows);

glPixelStorei(GL_UNPACK_SKIP_PIXELS, skippixels);

glPixelStorei(GL_UNPACK_ALIGNMENT, alignment);
```

Similar code could be written to save and restore OpenGL's pixel store pack modes (change
`UNPACK`
to
`PACK`
in the code above).

With OpenGL 1.1, the coding effort to save and restore these modes is simpler. To save, the pixel store state, you can call:

```
glPushClientAttrib(GL_CLIENT_PIXEL_STORE_BIT);
```

Then, this code restores the pixel store unpack modes:

```
glPopClientAttrib(GL_CLIENT_PIXEL_STORE_BIT);
```

The above routines (introduced in OpenGL 1.1) save and restore the pixel store state by pushing and popping the state using a stack maintained within the OpenGL library.

Observant readers may wonder why
`glPushClientAttrib`
is used instead of the shorter
`glPushAttrib`
routine. The answer involves the difference between OpenGL client-side and server-side state. It is worth clearly understanding the practical considerations that surround the distinction between OpenGL's server-side and client-side state.

There is not actually the option to use
`glPushAttrib`
to push the pixel store state because
`glPushAttrib`
and
`glPopAttrib`
only affects the server-state attribute stack and the pixel pack and unpack pixel store state is client-side OpenGL state.

Think of your OpenGL application as a client of the OpenGL rendering service provided by the host computer's OpenGL implementation.

The pixel store modes are *client-side state*. However, most of OpenGL's state is server-side. The term server-side state refers to the fact that the state actually resides within the OpenGL implementation itself, possibly within the graphics hardware itself. Server-side OpenGL state is concerned with how OpenGL commands are rendered, but client-side OpenGL state is concerned with how image or vertex data is extracted from the application address space.

Server-side OpenGL state is often expensive to retrieve because the state may reside only within the graphics hardware. To return such hardware-resident state (for example with
`glGetIntegerv`
) requires all preceding graphics commands to be issued before the state is retrievable. While OpenGL makes it possible to read back nearly all OpenGL server-side state, well-written programs should always avoid reading back OpenGL server-side state in performance sensitive situations.

Client-side state however is not state that will ever reside only within the rendering hardware. This means that using
`glGetIntegerv`
to read back pixel store state is relatively inexpensive because the state is client-side. This is why the above code that

explicitly reads back each pixel store unpack mode can be recommended. Similar OpenGL code that tried to save and restore server-side state could severely undermine OpenGL rendering performance.

Consider that whether it is better to use
`glGetIntegerv`
and
`glPixelStorei`
to explicitly save and restore the modes or whether you use OpenGL 1.1's
`glPushClientAttrib`
and
`glPopClientAttrib`
will depend on your situation. When pushing and popping the client attribute stack, you do have to be careful not to overflow the stack. An advantage to pushing and popping the client attribute state is that both the pixel store and vertex array client-side state can be pushed or popped with a single call. Still, you may find that only the pack or only the unpack modes need to be saved and restored and sometimes only one or two of the modes. If that is the case, an explicit save and restore may be faster.

9. Careful Updating that Raster Position

OpenGL's raster position determines where pixel rectangles and bitmaps will be rasterized. The
`glRasterPos2f`
family of commands specifies the coordinates for the raster position. The raster position gets transformed just as if it was a vertex. This symmetry makes it easy to position images or text within a scene along side 3D geometry. Just like a vertex, the raster position is logically an *(x,y,z,w)* coordinate. It also means that when the raster position is specified, OpenGL's modelview and projection matrix transformations, lighting, clipping, and even texture coordinate generation are all performed on the raster position vertex in exactly the same manner as a vertex coordinate passed to OpenGL via
`glVertex3f`
.

While this is all very symmetric, it rarely if ever makes sense to light or generate a texture coordinate for the raster position. It can even be quite confusing when you attempt to render a bitmap based on the current color and find out that because lighting is enabled, the bitmap color gets determined by lighting calculations. Similarly, if you draw a pixel rectangle with texture mapping enabled, your pixel rectangle may end up being modulated with the single texel determined by the current raster texture coordinate.

Still another symmetric, but generally unexpected result of OpenGL's identical treatment of vertices and the raster position is that, just like a vertex, the raster position can be clipped. This means if you specify a raster position outside (even slightly outside) the view frustum, the raster position is clipped and marked "invalid". When the raster position is invalid, OpenGL simply discards the pixel data specified by the
`glBitmap`
,
`glDrawPixels`
, and
`glCopyPixls`
commands.

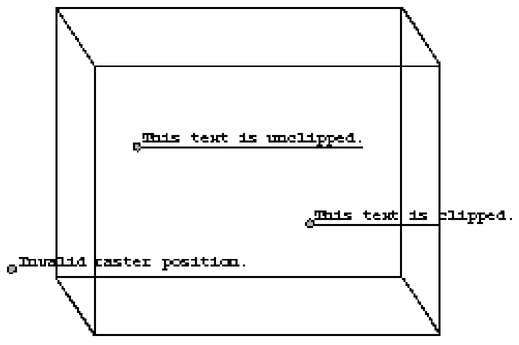

Figure 3: The enclosing box represents the view frustum and viewport. Each line of text is preceded by a dot indicating where the raster position is set before rendering the line of text. The dotted underlining shows the pixels that will actually be rasterized from each line of text. Notice that none of the pixels in the lowest line of text are rendered because the line's raster position is invalid.

Consider how this can surprise you. Say you wanted to draw a string of text with each character rendered with `glBitmap` . Figure 3 shows a few situations. The point to notice is that the text renders as expected in the first two cases, but in the last case, the raster position's placement is outside the view frustum so no pixels from the last text string are drawn.

It would appear that there is no way to begin rendering of a string of text outside the bounds of the viewport and view frustum and render at least the ending portion of the string. There is a way to accomplish what you want; it is just not very obvious. The `glBitmap` command both draws a bitmap and then offsets the raster position in *relative* window coordinates. You can render the final line of text if you first position the raster position within the view frustum (so that the raster position is set valid), and then you offset the raster position by calling `glBitmap` with relative raster position offsets. In this case, be sure to specify a zero-width and zero-height bitmap so no pixels are actually rendered.

Here is an example of this:

```
glRasterPos2i(0, 0);

glBitmap(0, 0, 0, 0, xoffset, yoffset, NULL);

drawString("Will not be clipped.");
```

This code fragment assumes that the `glRasterPos2i` call will validate the raster position at the origin. The code to setup the projection and modelview matrix to do that is not show (setting both matrices to the identity matrix would be sufficient).

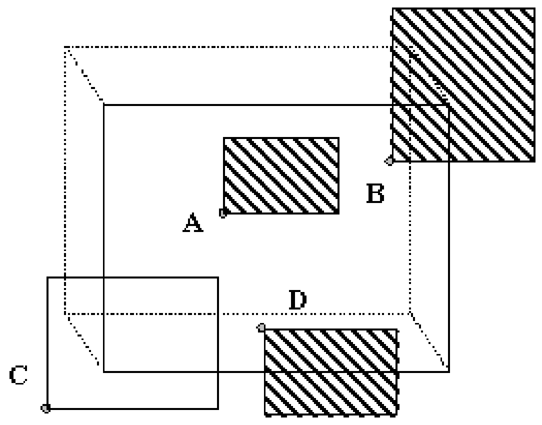

Figure 4: Various raster position scenarios. A, raster position is within the view frustum and the image is totally with the viewport. B, raster position is within the view frustum but the image is only partially within the viewport; still fragments are generated outside the viewport. C, raster position is invalid (due to being placed outside the view frustum); no pixels are rasterized. D, like case B except `glPixelZoom(1,-1)` **has inverted the Y pixel rasterization direction so the image renders top to bottom.**

10. The Viewport Does Not Clip or Scissor

It is a very common misconception that pixels cannot be rendered outside the OpenGL viewport. The viewport is often mistaken for a type of scissor. In fact, the viewport simply defines a transformation from normalized device coordinates (that is, post-projection matrix coordinates with the perspective divide applied) to window coordinates. The OpenGL specification makes no mention of clipping or culling when describing the operation of OpenGL's viewport.

Part of the confusion comes from the fact that, most of the time, the viewport is set to be the window's rectangular extent and pixels are clipped to the window's rectangular extent. But do not confuse window ownership clipping with anything the viewport is doing because the viewport *does not* clip pixels.

Another reason that it seems like primitives are clipped by the viewport is that vertices are indeed clipped against the view frustum. OpenGL's view frustum clipping does guarantee that no vertex (whether belonging to a geometric primitive or the raster position) can fall outside the viewport.

So if vertices cannot fall outside the view frustum and hence cannot be outside the viewport, how do pixels get rendered outside the viewport? Might it be an idle statement to say that the viewport does not act as a scissor if indeed you cannot generate pixels outside the viewport? Well, you *can* generate fragments that fall outside the viewport rectangle so it is not an idle statement.

The last section has already hinted at one way. While the raster position vertex must be specified to be within the view frustum to validate the raster position, once valid, the raster position (the state of which is maintained in window coordinates) can be moved outside the viewport with the glBitmap call's raster position offset capability. But you do not even have to move the raster position outside the viewport to update pixels outside of the viewport rectangle. You can just render a large enough bitmap or image so that the pixel rectangle exceeds the extent of the viewport rectangle. Figure 4 demonstrates image rendering outside the viewport.

The other case where fragments can be generated outside the viewport is when rasterizing wide lines and points or smooth points, lines, and polygons. While the actual vertices for wide and smooth primitives will be clipped to fall within the viewport during transformation, at rasterization time, the widened rasterization footprint of wide or smooth primitives may end up generating fragments outside the boundaries of the viewport rectangle.

Indeed, this can turn into a programming pitfall. Say your application renders a set of wide points that slowly wander around on the screen. Your program configures OpenGL like this:

```
glViewport(0, 0, windowWidth, windowHeight);

glLineWidth(8.0);
```

What happens when a point slowly slides off the edge of the window? If the viewport matches the window's extents as indicated by the
`glViewport`
call above, you will notice that a point will disappear suddenly at the moment its center is outside the window extent. If you expected the wide point to gradually slide of the screen, that is not what happens!

Keep in mind that the extra pixels around a wide or antialiased point are generated at rasterization time, but if the point's vertex (at its center) is culled during vertex transformation time due to view frustum clipping, the widened rasterization never happens. You can fix the problem by widening the viewport to reflect the fact that a point's edge can be up to four pixels (half of 8.0) from the point's center and still generate fragments within the window's extent. Change the
`glViewport`
call to:

```
glViewport(-4, -4, windowWidth+4, windowHeight+4);
```

With this new viewport, wide points can still be rasterized even if the hang off the window edge. Note that this will also slightly narrow your rectangular region of view, so if you want the identical view as before, you need to compensate by also expanding the view frustum specified by the projection matrix.

Note that if you really do require a rectangular 2D scissor in your application, OpenGL does provide a true window space scissor. See
`glEnable(GL_SCISSOR_TEST)`
and
`glScissor`
.

11. Setting the Raster Color

Before you specify a vertex, you first specify the normal, texture coordinate, material, and color and then only when
`glVertex3f`
(or its ilk) is called will a vertex actually be generated based on the current per-vertex state. Calling
`glColor3f`
just sets the current color state.
`glColor3f`
does not actually create a vertex or any perform any rendering. The
`glVertex3f`
call is what binds up all the current per-vertex state and issues a complete vertex for transformation.

The raster position is updated similarly. Only when
`glRasterPos3f`
(or its ilk) is called does all the current per-vertex state get transformed and assigned to the raster position.

A common pitfall is attempting to draw a string of text with a series of
`glBitmap`
calls where different characters in the string are different colors. For example:

```
          glColor3f(1.0, 0.0, 0.0); /* RED */

          glRasterPos2i(20, 15);

          glBitmap(w, h, 0, 0, xmove, ymove, red_bitmap);


glColor3f(0.0, 1.0, 0.0); /* GREEN */
```

```
glBitmap(w, h, 0, 0, xmove, ymove, green_bitmap);

/* WARNING: Both bitmaps render red. */
```

Unfortunately, `glBitmap` 's relative offset of the raster position just updates the raster position location. The raster color (and the other remaining raster state values) remain unchanged.

The designers of OpenGL intentionally specified that `glBitmap` should not latch into place the current per-vertex state when the raster position is repositioned by `glBitmap` . Repeated `glBitmap` calls are designed for efficient text rendering with mono-chromatic text being the most common case. Extra processing to update per-vertex state would slow down the intended most common usage for `glBitmap` .

If you do want to switch the color of bitmaps rendered with `glBitmap` , you will need to explicitly call `glRasterPos3f` (or its ilk) to lock in a changed current color.

12. OpenGL's Lower Left Origin

Given a sheet of paper, people write from the top of the page to the bottom. The origin for writing text is at the upper left-hand margin of the page (at least in European languages). However, if you were to ask any decent math student to plot a few points on an X-Y graph, the origin would certainly be at the lower left-hand corner of the graph. Most 2D rendering APIs mimic writers and use a 2D coordinate system where the origin is in the upper left-hand corner of the screen or window (at least by default). On the other hand, 3D rendering APIs adopt the mathematically minded convention and assume a lower left-hand origin for their 3D coordinate systems.

If you are used to 2D graphics APIs, this difference of origin location can trip you up. When you specify 2D coordinates in OpenGL, they are generally based on a lower left-hand coordinate system. Keep this in mind when using `glViewport` , `glScissor` , `glRasterPos2i` , `glBitmap` , `glTexCoord2f` , `glReadPixels` , `glCopyPixels` , `glCopyTexImage2D` , `glCopyTexSubImage2D` , `gluOrtho2D` , and related routines.

Another common pitfall related to 2D rendering APIs having an upper left-hand coordinate system is that 2D image file formats start the image at the top scan line, not the bottom scan line. OpenGL assumes images start at the bottom scan line by default. If you do need to flip an image when rendering, you can use
`glPixelZoom(1,-1)`
to flip the image in the Y direction. Note that you can also flip the image in the X direction. Figure 4 demonstrates using
`glPixelZoom`
to flip an image.

Note that
`glPixelZoom`
only works when rasterizing image rectangles with
`glDrawPixels`
or
`glCopyPixels`
. It does not work with
`glBitmap`
or
`glReadPixels`
. Unfortunately, OpenGL does not provide an efficient way to read an image from the frame buffer into memory starting with the top scan line.

13. Setting Your Raster Position to a Pixel Location

A common task in OpenGL programming is to render in window coordinates. This is often needed when overlaying text or blitting images onto codecise screen locations. Often having a 2D window coordinate system with an upper left-hand origin matching the window system's default 2D coordinate system is useful.

Here is code to configure OpenGL for a 2D window coordinate system with an upper left-hand origin where
`w`
and
`h`
are the window's width and height in pixels:

```
glViewport(0, 0, w, h);

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

glOrtho(0, w, h, 0, -1, 1);

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();
```

Note that the bottom and top parameters (the 3rd and 4th parameters) to
`glOrtho`
specify the window height as the *top* and zero as the *bottom*. This flips the origin to put the origin at the window's upper left-hand corner.

Now, you can safely set the raster position at a pixel position in window coordinates like this

```
glVertex2i(x, y);

glRasterPos2i(x, y);
```

One pitfall associated with setting up window coordinates is that switching to window coordinates involves loading both the modelview and projection matrices. If you need to "get back" to what was there before, use
`glPushMatrix`

and
`glPopMatrix`
(but remember the pitfall about assuming the projection matrix stack has more than two entries).

All this matrix manipulation can be a lot of work just to do something like place the raster position at some window coordinate. Brian Paul has implemented a freeware version of the OpenGL API called Mesa. Mesa implements an OpenGL extension called
`MESA_window_pos`
that permits direct efficient setting of the raster position without disturbing any other OpenGL state. The calls are:

```
glWindowPos4fMESA(x,y,z,w);

glWindowPos2fMESA(x,y)
```

Here is the equivalent implementation of these routines in unextended OpenGL:

```
void
glWindowPos4fMESAemulate(GLfloat x,GLfloat y,GLfloat z,GLfloat w)
{
  GLfloat fx, fy;

  /* Push current matrix mode and viewport attributes. */

  glPushAttrib(GL_TRANSFORM_BIT | GL_VIEWPORT_BIT);

    /* Setup projection parameters. */

    glMatrixMode(GL_PROJECTION);

    glPushMatrix();

      glLoadIdentity();

      glMatrixMode(GL_MODELVIEW);

      glPushMatrix();

        glLoadIdentity();

        glDepthRange(z, z);

        glViewport((int) x - 1, (int) y - 1, 2, 2);

        /* Set the raster (window) position. */

        fx = x - (int) x;

        fy = y - (int) y;

        glRasterPos4f(fx, fy, 0.0, w);

      /* Restore matrices, viewport and matrix mode. */

      glPopMatrix();

    glMatrixMode(GL_PROJECTION);

    glPopMatrix();

  glPopAttrib();

}

void
glWindowPos2fMESAemulate(GLfloat x, GLfloat y)
```

```
{
  glWindowPos4fMESAemulate(x, y, 0, 1);
}
```

Note all the extra work the emulation routines go through to ensure that no OpenGL state is disturbed in the process of setting the raster position. Perhaps commercial OpenGL vendors will consider implementing this extension.

14. Careful Enabling Color Material

OpenGL's color material feature provides a less expensive way to change material parameters. With color material enabled, material colors track the current color. This means that instead of using the relatively expensive `glMaterialfv` routine, you can use the `glColor3f` routine.

Here is an example using the color material feature to change the diffuse color for each vertex of a triangle:

```
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
glBegin(GL_TRIANGLES);
  glColor3f(0.2, 0.5, 0.8);
  glVertex3f(1.0, 0.0, 0.0);
  glColor3f(0.3, 0.5, 0.6);
  glVertex3f(0.0, 0.0, 0.0);
  glColor3f(0.4, 0.2, 0.2);
  glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

Consider the more expensive code sequence needed if `glMaterialfv` is used explicitly:

```
GLfloat d1 = { 0.2, 0.5, 0.8, 1.0 };
GLfloat d2 = { 0.3, 0.5, 0.6, 1.0 };
GLfloat d3 = { 0.4, 0.2, 0.2, 1.0 };
glBegin(GL_TRIANGLES);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,d1);
  glVertex3f(1.0, 0.0, 0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,d2);
  glVertex3f(0.0, 0.0, 0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,d3);
  glVertex3f(1.0, 1.0, 0.0);
```

```
glEnd();
```

If you are rendering objects that require frequent simple material changes, try to use the color material mode. However, there is a common pitfall encountered when enabling the color material mode. When color material is enabled, OpenGL immediately changes the material colors controlled by the color material state. Consider the following piece of code to initialize a newly create OpenGL rendering context:

```
GLfloat a[] = { 0.1, 0.1, 0.1, 1.0 };

glColor4f(1.0, 1.0, 1.0, 1.0);

glMaterialfv(GL_FRONT, GL_AMBIENT, a);

glEnable(GL_COLOR_MATERIAL);
/* WARNING: Ambient and diffuse material latch immediately to the current color. */
glColorMaterial(GL_FRONT, GL_DIFFUSE);

glColor3f(0.3, 0.5, 0.6);
```

What state will the front ambient and diffuse material colors be after executing the above code fragment? While the programmer may have intended the ambient material state to be (0.1, 0.1, 0.1, 1.0) and the diffuse material state to be (0.3, 0.5, 0.6, 1.0), that is not quite what happens.

The resulting diffuse material state is what the programmer intended, but the resulting ambient material state is rather unexpectedly (1.0, 1.0, 1.0, 1.0). How did that happen? Well, remember that the color material mode *immediately* begins tracking the current color when enabled. The initial value for the color material settings is
`GL_FRONT_AND_BACK`
and
`GL_AMBIENT_AND_DIFFUSE`
(probably not what you expected!).

Since enabling the color material mode immediately begins tracking the current color, both the ambient and diffuse material states are updated to be (1.0, 1.0, 1.0, 1.0). Note that the effect of the initial
`glMaterialfv`
is lost. Next, the color material state is updated to just change the front diffuse material. Lastly, the
`glColor3f`
invocation changes the diffuse material to (0.3, 0.5, 0.6, 1.0). The ambient material state ends up being (1.0, 1.0, 1.0, 1.0).

The problem in the code fragment above is that the color material mode is enabled before calling
`glColorMaterial`
. The color material mode is very effective for efficient simple material changes, but to avoid the above pitfall, always be careful to set
`glColorMaterial`
before you enable
`GL_COLOR_MATERIAL`
.

15. Much OpenGL State Affects All Primitives

A fragment is OpenGL's term for the bundle of state used to update a given pixel on the screen. When a primitive such as a polygon or image rectangle is rasterized, the result is a set of fragments that are used to update the pixels that the primitive covers. Keep in mind that all OpenGL rendering operations share the same set of per-fragment operations. The same applies to OpenGL's fog and texturing rasterization state.

For example, if you enabled depth testing and blending when you render polygons in your application, keep in mind that when you overlay some 2D text indicating the application's status that you probably want to disable depth testing and blending. It is easy to forget that this state also affects images drawn and copied with
`glDrawPixels`

and
`glCopyPixels`
.

You will quickly notice when this shared state screws up your rendering, but also be aware that sometimes you can leave a mode enabled such as blending without noticing the extra expense involved. If you draw primitives with a constant alpha of 1.0, you may not notice that the blending is occurring and simply slowing you down.

This issue is not unique to the per-fragment and rasterization state. The pixel path state is shared by the draw pixels (
`glDrawPixels`
), read pixels (
`glReadPixels`
), copy pixels (
`glCopyPixels`
), and texture download (
`glTexImage2D`
) paths. If you are not careful, it is easy to get into situations where a texture download is screwed up because the pixel path was left configured for a pixel read back.

16. Be Sure to Allocate Ancillary Buffers that You Use

If you intend to use an ancillary buffer such as a depth, stencil, or accumulation buffer, be sure that you application actually requests all the ancillary buffers that you intend to use. A common interoperability issue is developing an OpenGL application on a system with only a few frame buffer configurations that provide all the ancillary buffers that you use. For example, your system has no frame buffer configuration that advertises a depth buffer without a stencil buffer. So on your development system, you "get away with" not explicitly requesting a stencil buffer.

The problem comes when you take your supposedly debugged application and run it on a new fancy hardware accelerated OpenGL system only to find out that the application fails miserably when attempting to use the stencil buffer. Consider that the fancy hardware may support extra color resolution if you do not request a stencil buffer. If you application does not explicitly request the stencil buffer that it uses, the fancy hardware accelerated OpenGL implementation determines that the frame buffer configuration with no stencil but extra color resolution is the better choice for your application. If your application would have correctly requested a stencil buffer things would be fine. Make sure that you allocate what you use.

Conclusion

I hope that this review of various OpenGL pitfalls saves you much time and debugging grief. I wish that I could have simply read about these pitfalls instead of learning most of them the hard way.

Visualization has always been the key to enlightenment. If computer graphics changes the world for the better, the fundamental reason why is that computer graphics makes visualization easier.

Column Header

Coding Resources

- OpenGL SDK
- Getting Started Wiki
- OpenGL Registry
- FAQs
- GLUT & Utility Libraries
- GLUT
- Other Utility Toolkits
- GLX, GLU & DRI
- Higher Level Libraries

- Programming Language Bindings
- Sample Code & Tutorials
- Benchmarks
- Mailing Lists & News Groups
- OpenGL StackOverflow
- OpenGL ES
- OpenCL
- WebGL
- Archived Resources

Column Footer

- About OpenGL
- Privacy Policy