Common Mistakes

From OpenGL.org

Quite a few websites show the same mistakes and the mistakes presented in their tutorials are copied and pasted by those who want to learn OpenGL. This page has been created so that newcomers understand GL programming a little better instead of working by trial and error.

Contents

There are also other articles explaining common mistakes:

- Common Mistakes in GLSL
- Unexpected Results you can get when using OpenGL
- Mistakes related to measuring Performance
- Common Mistakes when using deprecated functionality.

Extensions and OpenGL Versions

One of the possible mistakes related to this is to check for the presence of an extension, but instead using the corresponding core functions. The correct behavior is to check for the presence of the extension if you want to use the extension API, and check the GL version if you want to use the core API. In case of a core extension, you should check for both the version and the presence of the extension; if either is there, you can use the functionality.

The Object Oriented Language Problem

In an object-oriented language like C++, it is often useful to have a class that wraps an OpenGL object. For example, one might have a texture object that has a constructor and a destructor like the following:

```
MyTexture::MyTexture(const char *pfilePath)
{
  if(LoadFile(pfilePath)==ERROR)
        return;
  textureID=0;
  glGenTextures(1, &textureID);
  //More GL code...
}

MyTexture::~MyTexture()
{
  if(textureID)
        glDeleteTextures(1, &textureID);
}
```

There is a large pitfall with doing this. OpenGL functions do not work unless an OpenGL context has been created and is active within that thread. Thus, glGenTextures will do nothing before context creation, and glDeleteTextures will do nothing after context destruction. The latter problem is not a significant concern since OpenGL contexts clean up after themselves, but the former is a problem.

This problem usually manifests itself when someone creates a texture object at global scope. There are several potential solutions:

1. Do not use constructors/destructors to initialize/destroy OpenGL objects. Instead, use member functions of these classes for these purposes. This violates RAII principles, so this is not the best course of action.
2. Have your OpenGL object constructors throw an exception if a context has not been created yet. This requires an addition to your context creation functionality that tells your code when a context has been created and is active.
3. Create a class that owns all other OpenGL related objects. This class should also be responsible for creating the context in its constructor.

OOP and hidden binding

There's another issue when using OpenGL with languages like c++. Consider the following function:

```
void MyTexture::TexParameter(GLenum pname, GLint param)
{
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexParameteri(GL_TEXTURE_2D, pname, param);
}
```

The problem is that the binding of the texture is hidden from the user of the class. There may be performance implications for doing repeated binding of objects (especially since the API may not seem heavyweight to the outside user). But the major concern is correctness; the bound objects are *global state*, which a local member function now has changed.

This can cause many sources of hidden breakage. The safe way to implement this is as follows:

```
void MyTexture::TexParameter(GLenum pname, GLint param)
{
    GLuint boundTexture = 0;
    glGetIntegerv(GL_TEXTURE_BINDING_2D, (GLint*) &boundTexture);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexParameteri(GL_TEXTURE_2D, pname, param);
    glBindTexture(GL_TEXTURE_2D, boundTexture);
}
```

Note that this solution emphasizes correctness over *performance*; the `glGetIntegerv` call may not be particularly fast.

A more effective solution is to use the EXT_direct_state_access (http://www.opengl.org/registry/specs/EXT/direct_state_access.txt) extension, where available:

```
void MyTexture::TexParameter(GLenum pname, GLint param)
{
    glTextureParameteri(textureID, GL_TEXTURE_2D, pname, param);
}
```

Texture upload and pixel reads

You create storage for a Texture and upload pixels to it with `glTexImage2D` (or similar functions, as appropriate to the type of texture). If your program crashes during the upload, or diagonal lines appear in the resulting image, this is because the alignment of each horizontal line of your pixel array is not multiple of 4. This typically happens to users loading an image that is of the RGB or BGR format (for example, 24 BPP images), depending on the source of your image data.

Example, your image width = 401 and height = 500. The height is irrelevant; what matters is the width. If we do the math, 401 pixels x 3 bytes = 1203, which is not divisible by 4. Some image file formats may inherently align each row to 4 bytes, but some do not. For those that don't, each row will start exactly 1203 bytes from the start of the last. OpenGL's row alignment can be changed to fit the row alignment for your image data. This is done by calling `glPixelStorei(GL_UNPACK_ALIGNMENT, #)`, where # is the alignment you want. The default alignment is 4.

And if you are interested, most GPUs like chunks of 4 bytes. In other words, `GL_RGBA` or `GL_BGRA` is preferred when each component is a byte. `GL_RGB` and `GL_BGR` is considered bizarre since most GPUs, most CPUs and any other kind of chip don't handle 24 bits. This means, the driver converts your `GL_RGB` or `GL_BGR` to what the GPU prefers, which typically is BGRA.

Similarly, if you read a buffer with `glReadPixels`, you might get similar problems. There is a `GL_PACK_ALIGNMENT` just like the `GL_UNPACK_ALIGNMENT`. The default alignment is again 4 which means each horizontal line must be a multiple of 4 in size. If you read the buffer with a format such as `GL_BGRA` or `GL_RGBA` you won't have any problems since the line will always be a multiple of 4. If you read it in a format such as `GL_BGR` or `GL_RGB` then you risk running into this problem.

The `GL_PACK/UNPACK_ALIGNMENT`s can only be 1, 2, 4, or 8. So an alignment of 3 is not allowed.

Image precision

You *can* (but it is not advisable to do so) call `glTexImage2D(GL_TEXTURE_2D, 0, X, width, height, 0, format, type, pixels)` and you set X to 1, 2, 3, or 4. The X refers to the number of components (`GL_RED` would be 1, `GL_RG` would be 2, `GL_RGB` would be 3, `GL_RGBA` would be 4).

It is preferred to actually give a real image format, one with a specific internal precision. If the OpenGL implementation does not support the particular format and precision you choose, the driver will internally convert it into something it does support.

OpenGL versions 3.x and above have a set of required image formats that all conforming implementations must implement.

We should also state that it is common to see the following on tutorial websites:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels);
```

Although GL will accept `GL_RGB`, it is up to the driver to decide an appropriate precision. We recommend that you be specific and write `GL_RGB8`:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels);
```

This means you want the driver to actually store it in the R8G8B8 format. We should also state that most GPUs will up-convert `GL_RGB8` into `GL_RGBA8` internally. So it's probably best to steer clear of `GL_RGB8`. We should also state that on some platforms, such as Windows, `GL_BGRA` for the pixel upload format is preferred.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

This uses `GL_RGBA8` for the internal format. `GL_BGRA` and `GL_UNSIGNED_BYTE` (or `GL_UNSIGNED_INT_8_8_8_8` is for the data in pixels array. The driver will likely not have to perform any CPU-based conversion and DMA this data directly to the video card. Benchmarking shows that on Windows and with nVidia and ATI/AMD, that this is the optimal format.

Depth Buffer Precision

When you select a pixelformat for your window, and if you ask for a depth buffer, the depth buffer is typically stored as 16 bit integer or 24 bit integer or 32 bit integer.

It seems to be a common misconception that the depth buffer is stored as floating point but this is false. The depth values that fall into the clip region are typically from 0.0 to 1.0. The depth value is converted to an integer and finally written to the depth buffer by the GPU.

The conversion probably goes like this:

```
if(depth_buffer_precision == 16)
{
   writeToDepthBuffer = depthAsFloat * 65535;
}
else if(depth_buffer_precision == 24)
{
   writeToDepthBuffer = depthAsFloat * 16777215;
}
else if(depth_buffer_precision == 32)
{
   writeToDepthBuffer = depthAsFloat * 4294967295;
}
```

Furthermore, the case of 24 bit is a undesirable format. GPUs prefer chunks of 32 bit so the depth buffer will be padded with 8 bits that go unused. This is called the D24X8 format.

If you ask for a stencil buffer, you should ask for 8 bit so that the GPU would allocate what is called a D24S8 buffer which is 24 bit integer for the depth and 8 bit integer for stencil.

Now that the misconception about depth buffers being floating point is resolved, what is wrong with this call?

```
glReadPixels(0, 0, width, height, GL_DEPTH_COMPONENT, GL_FLOAT, mypixels);
```

The GL driver will copy the depth buffer from the graphics card and it will use the CPU to convert it to floating point values. This is better :

```
if(depth_buffer_precision == 16)
{
  GLushort mypixels[width*height];
  glReadPixels(0, 0, width, height, GL_DEPTH_COMPONENT, GL_UNSIGNED_SHORT, mypixels);
}
else if(depth_buffer_precision == 24)
{
  GLuint mypixels[width*height];    //There is no 24 bit variable, so we'll have to settle for 32 bit
  glReadPixels(0, 0, width, height, GL_DEPTH_COMPONENT, GL_UNSIGNED_INT_24_8, mypixels);  //No upconversion.
}
else if(depth_buffer_precision == 32)
{
  GLuint mypixels[width*height];
  glReadPixels(0, 0, width, height, GL_DEPTH_COMPONENT, GL_UNSIGNED_INT, mypixels);
}
```

If you have a depth/stencil format, you can get the depth/stencil data this way:

```
GLuint mypixels[width*height];
glReadPixels(0, 0, width, height, GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, mypixels);
```

There are a number of depth and depth/stencil internal formats available, with varying bitdepths and accuracy.

Creating a complete texture

What's wrong with this code?

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

The texture won't work because it is incomplete. The default `GL_TEXTURE_MIN_FILTER` state is `GL_NEAREST_MIPMAP_LINEAR`. And because OpenGL defines the default `GL_TEXTURE_MAX_LEVEL` to be 1000, OpenGL will expect there to be mipmap levels defined. Since you have only defined a single mipmap level, OpenGL will consider the texture incomplete until the `GL_TEXTURE_MAX_LEVEL` is properly set, or the `GL_TEXTURE_MIN_FILTER` parameter is set to not use mipmaps.

Better code would be to use texture storage functions (if you have OpenGL 4.2 or ARB_texture_storage (http://www.opengl.org/registry/specs/ARB/texture_storage.txt)) to allocate the texture's storage, then upload with `glTexSubImage2D`:

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width, height);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

This creates a texture with a single mipmap level, and sets all of the parameters appropriately. If you wanted to have multiple mipmaps, then you should change the `1` to the number of mipmaps you want. You will also need separate `glTexSubImage2D` calls to upload each mipmap.

If that is unavailable, you can get a similar effect from this code:

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

Again, if you use more than one mipmaps, you should change the `GL_TEXTURE_MAX_LEVEL` to state how many you will use (minus 1. The base/max level is a half-open range), then perform a `glTexImage2D` (note the lack of "Sub") for each mipmap.

Automatic mipmap generation

Mipmaps of a texture can be automatically generated with the `glGenerateMipmap` function. OpenGL 3.0 or greater is required for this function (or the extension GL_ARB_framebuffer_object). The function works quite simply; when you call it for a texture, mipmaps are generated for that texture:

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexStorage2D(GL_TEXTURE_2D, num_mipmaps, GL_RGBA8, width, height);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

```
glGenerateMipmap(GL_TEXTURE_2D);  //Generate num_mipmaps number of mipmaps here.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

If texture storage is not available, you can use the older API:

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
glGenerateMipmap(GL_TEXTURE_2D);  //Generate mipmaps now!!!
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

> **Warning:** It has been reported that on some ATI drivers, glGenerateMipmap(GL_TEXTURE_2D) has no effect unless you precede it with a call to glEnable(GL_TEXTURE_2D) in this particular case. Once again, to be clear, bind the texture, glEnable, then glGenerateMipmap. This is a bug and has been in the ATI drivers for a while. Perhaps by the time you read this, it will have been corrected. (glGenerateMipmap doesn't work on ATI as of 2011)

Legacy Generation

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

OpenGL 1.4 is required for support for automatic mipmap generation. `GL_GENERATE_MIPMAP` is part of the texture object state and it is a flag (`GL_TRUE` or `GL_FALSE`). If it is set to `GL_TRUE`, then whenever texture level 0 is updated, the mipmaps will all be regenerated.

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

In GL 3.0, `GL_GENERATE_MIPMAP` is deprecated, and in 3.1 and above, it was removed. So for those versions, you must use `glGenerateMipmap`.

gluBuild2DMipmaps

Never use this. Use either `GL_GENERATE_MIPMAP` (requires GL 1.4) or the `glGenerateMipmap` function (requires GL 3.0).

glGetError

Why should you check for errors? Why you should call glGetError()?

```
```

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);   //Requires GL 1.4. Removed from GL 3.1 and above.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

The code doesn't call glGetError(). If you were to call `glGetError`, it would return `GL_INVALID_ENUM`. If you were to place a `glGetError` call after each function call, you will notice that the error is raised at `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR)`. The magnification filter can't specify the use of mipmaps; only the minification filter can do that.

Checking For Errors When You Compile Your Shader

Always check for errors when compiling/linking shader or program objects.

Creating a Cubemap Texture

It's best to set the wrap mode to `GL_CLAMP_TO_EDGE` and not the other formats. Don't forget to define all 6 faces else the texture is considered incomplete. Don't forget to setup `GL_TEXTURE_WRAP_R` because cubemaps require 3D texture coordinates.

Example:

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAX_LEVEL, 0);
//Define all 6 faces
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face0);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face1);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face2);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face3);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face4);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA8, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, pixels_face5);
```

If you want to auto-generate mipmaps, you can use any of the aforementioned mechanisms. OpenGL will not blend over multiple textures when generating mipmaps for the cubemap leaving visible seams at lower mip levels. Unless you enable seamless cubemap texturing.

Texture edge color problem

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

Never use `GL_CLAMP`; what you intended was `GL_CLAMP_TO_EDGE`. Indeed, `GL_CLAMP` was removed from core GL 3.1+, so it's not even an option anymore.

> **Note:** If you are curious as to what `GL_CLAMP` used to mean, it referred to blending texture edge texels with

border texels. This is different from GL_CLAMP_TO_BORDER, where the clamping happens to a solid border color. The GL_CLAMP behavior was tied to special border texels. Effectively, each texture had a 1-pixel border. This was useful for having more easily seamless texturing, but it was never implemented in hardware directly. So it was removed.

Updating a texture

To change texels in an already existing 2d texture, use `glTexSubImage2D`:

```
glBindTexture(GL_TEXTURE_2D, textureID);    //A texture you have already created storage for
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

`glTexImage2D` creates the storage for the texture, defining the size/format and removing all previous pixel data. `glTexSubImage2D` only modifies pixel data within the texture. It can be used to update all the texels, or simply a portion of them.

To copy texels from the framebuffer, use `glCopyTexSubImage2D`.

glBindTexture(GL_TEXTURE_2D, textureID); //A texture you have already created storage for
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, width, height); //Copy current read buffer to texture

Note that there is a `glCopyTexImage2D` function, which does the copy to fill the image, but also defines the image size, format and so forth, just like `glTexImage2D`.

Render To Texture

To render directly to a texture, without doing a copy as above, use Framebuffer Objects.

> **Warning:** NVIDIA's OpenGL driver has a known issue with using incomplete textures. If the texture is not texture complete, the FBO itself will be considered GL_FRAMEBUFFER_UNSUPPORTED, or will have GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT. This is a driver bug, as the OpenGL specification does not allow implementations to return either of these values simply because a texture is not yet complete. Until this is resolved in NVIDIA's drivers, it is advised to make sure that all textures have mipmap levels, and that all `glTexParameteri` values are properly set up for the format of the texture. For example, integral textures are not complete if the mag and min filters have any LINEAR fields.

Depth Testing Doesn't Work

First, check to see if the Depth Test is active. Make sure that `glEnable` has been called and an appropriate `glDepthFunc` is active. Also make sure that the `glDepthRange` matches the depth function.

Assuming all of that has been set up correctly, your framebuffer may not have a depth buffer at all. This is easy to see for a Framebuffer Object you created. For the Default Framebuffer, this depends entirely on how you created your OpenGL Context.

For example, if you are using GLUT, you need to make sure you pass GLUT_DEPTH to the `glutInitDisplayMode` function.

No Alpha in the Framebuffer

If you are doing Blending and you need a destination alpha, you need to make sure that your render target has one. This is easy to ensure when rendering to a Framebuffer Object. But with a Default Framebuffer, it depends on how you created your OpenGL Context.

For example, if you are using GLUT, you need to make sure you pass `GLUT_ALPHA` to the `glutInitDisplayMode` function.

glFinish and glFlush

Use `glFlush` if you are rendering to the front buffer of the Default Framebuffer. It is better to have a double buffered window but if you have a case where you want to render to the window directly, then go ahead.

There are a lot of tutorial website that suggest you do this:

```
glFlush();
SwapBuffers();
```

This is unnecessary. The SwapBuffer command takes care of flushing and command processing.

The `glFlush` and `glFinish` functions deal with synchronizing CPU actions with GPU commands.

In many cases, explicit synchronization like this is unnecessary. The use of Sync Objects can make it necessary, as can the use of arbitrary reads/writes from/to images.

As such, you should only use `glFinish` when you are doing something that the specification specifically states will not be synchronous.

glDrawPixels

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

For good performance, use a format that is directly supported by the GPU. Use a format that causes the driver to basically to a memcpy to the GPU. Most graphics cards support `GL_BGRA`. Example:

```
glDrawPixels(width, height, GL_BGRA, GL_UNSIGNED_BYTE, pixels);
```

However, it is recommened that you use a texture instead and just update the texture with `glTexSubImage2D`, possibly with a buffer object for async transfer.

GL_DOUBLE

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

glLoadMatrixd, glRotated and any other function that have to do with the double type. Most GPUs don't support GL_DOUBLE (double) so the driver will convert the data to GL_FLOAT (float) and send to the GPU. If you put GL_DOUBLE data in a VBO, the performance might even be much worst than immediate mode (immediate mode means glBegin, glVertex, glEnd). GL doesn't offer any better way to know what the GPU prefers.

Slow pixel transfer performance

To achieve good Pixel Transfer performance, you need to use a pixel transfer format that the implementation can directly work with. Consider this:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixels);
```

The problem is that the pixel transfer format `GL_RGBA` may not be directly supported for `GL_RGBA8` formats. On certain platforms, the GPU prefers that red and blue be swapped (`GL_BGRA`).

If you supply `GL_RGBA`, then the driver may have to do the swapping for you which is slow. If you do use `GL_BGRA`, the call to pixel transfer will be much faster.

Keep in mind that for the 3rd parameter, it must be kept as `GL_RGBA8`. This defines the *texture's* image format; the last three parameters describe how your pixel data is stored. The image format doesn't define the order stored by the texture, so the GPU is still allowed to store it internally as BGRA.

Note that `GL_BGRA` pixel transfer format is only preferred when uploading to `GL_RGBA8` images. When dealing with other formats, like `GL_RGBA16`, `GL_RGBA8UI` or even `GL_RGBA8_SNORM`, then the regular `GL_RGBA` ordering may be preferred.

On which platforms is `GL_BGRA` preferred? Making a list would be too long but one example is Microsoft Windows. Note that with GL 4.3 or ARB_internalformat_query2 (http://www.opengl.org/registry/specs/ARB/internalformat_query2.txt), you can simply ask the implementation what is the preferred format with `glGetInternalFormativ(GL_TEXTURE_2D, GL_RGBA8, GL_TEXTURE_IMAGE_FORMAT, 1, &preferred_format)`.

Swap Buffers

A modern OpenGL program should always use double buffering. A modern OpenGL program should also have a depth buffer.

Render sequence should be like this:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
RenderScene();
SwapBuffers(hdc);  //For Windows
```

The buffers should *always* be cleared. On much older hardware, there was a technique to get away without clearing the scene, but on even semi-recent hardware, this will actually make things *slower*. So always do the clear.

The Pixel Ownership Problem

If your windows is covered or if it is partially covered or if window is outside the desktop area, the GPU might not render to those portions. Reading from those areas may likewise produce garbage data.

This is because those pixels fail the "pixel ownership test". Only pixels that pass this test have valid data. Those that fail have undefined contents.

If this is a problem for you (note: it's only a problem if you need to read data back from the covered areas), the solution is to render to a Framebuffer Object and render to that. If you need to display the image, you can blit to the Default Framebuffer.

Selection and Picking and Feedback Mode

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

A modern OpenGL program should not use the selection buffer or feedback mode. These are not 3D graphics rendering features yet they have been added to GL since version 1.0. Selection and feedback runs in software (CPU side). On some implementations, when used along with VBOs, it has been reported that performance is lousy.

A modern OpenGL program should do color picking (render each object with some unique color and glReadPixels to find out what object your mouse was on) or do the picking with some 3rd party mathematics library.

Point and line smoothing

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

Users notice that on some implementation points or lines are rendered a little different then on others. This is because the GL spec allows some flexibility. Consider this:

```
glPointSize(5.0);
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glEnable(GL_POINT_SMOOTH);
RenderMyPoints();
```

On some hardware, the points will look nice and round; on others, they will look like squares.

On some implementations, when you call glEnable(GL_POINT_SMOOTH) or glEnable(GL_LINE_SMOOTH) and you use shaders at the same time, your rendering speed goes down to 0.1 FPS. This is because the driver does software rendering. This would happen on AMD/ATI GPUs/drivers.

glEnable(GL_POLYGON_SMOOTH)

This is not a recommended method for anti-aliasing. Use Multisampling instead.

Color Index, The imaging subset

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

Section 3.6.2 of the GL specification talks about the imaging subset. glColorTable and related operations are part of this subset. They are typically not supported by common GPUs and are software emulated. It is recommended that you avoid it.

If you find that your texture memory consumption is too high, use texture compression. If you really want to use paletted color indexed textures, you can implement this yourself a texture and a shader.

Bitfield enumerators

Some OpenGL enumerators represent bits in a particular bitfield. All of these end in _BIT (before any extension suffix). Take a look at this example:

```
glEnable(GL_BLEND | GL_DRAW_BUFFER); // invalid
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); // valid
```

The first line is wrong. Because neither of these enumerators ends in _BIT, they are not bitfields and thus should not be OR'd together.

By contrast, the second line is perfectly fine. All of these end in _BIT, so this makes sense.

Triple Buffering

You cannot control whether a driver does triple buffering. You could try to implement it yourself using a FBO. But if the driver is already doing triple buffering, your code will only turn it into quadruple buffering. Which is usually overkill.

Paletted textures

Support for the EXT_paletted_texture (http://www.opengl.org/registry/specs/EXT/paletted_texture.txt) extension has been dropped by the major GL vendors. If you really need paletted textures on new hardware, you may use shaders to achieve that effect.

Shader example:

```
//Fragment shader
#version 110
uniform sampler2D ColorTable;     //256 x 1 pixels
uniform sampler2D MyIndexTexture;
varying vec2 TexCoord0;

void main()
{
  //What color do we want to index?
  vec4 myindex = texture2D(MyIndexTexture, TexCoord0);
  //Do a dependency texture read
  vec4 texel = texture2D(ColorTable, myindex.xy);
  gl_FragColor = texel;   //Output the color
}
```

`ColorTable` might be in a format of your choice such as `GL_RGBA8`. ColorTable could be a texture of 256 x 1 pixels in size.

`MyIndexTexture` can be in any format, though `GL_R8` is quite appropriate (`GL_R8` is available in GL 3.0). `MyIndexTexture` could be of any dimension such as 64 x 32.

We read `MyIndexTexture` and we use this result as a texcoord to read `ColorTable`. If you wish to perform palette animation, or simply update the colors in the color table, you can submit new values to `ColorTable` with `glTexSubImage2D`. Assuming that the color table is in `GL_RGBA` format:

```
glBindTexture(GL_TEXTURE_2D, myColorTableID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 1, GL_BGRA, GL_UNSIGNED_BYTE, mypixels);
```

Texture Unit

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

When multitexturing was introduced, getting the number of texture units was introduced as well which you can get with:

```
int MaxTextureUnits;
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &MaxTextureUnits);
```

You should not use the above because it will give a low number on modern GPUs.

In old OpenGL, each texture unit has its own texture environment state (glTexEnv), texture matrix, texture coordinate generation (glTexGen), texcoords (glTexCoord), clamp mode, mipmap mode, texture LOD, anisotropy.

Then came the programmable GPU. There aren't texture units anymore. Today, you have texture image units (TIU) which you can get with:

```
int MaxTextureImageUnits;
glGetIntegerv(GL_MAX_TEXTURE_IMAGE_UNITS, &MaxTextureImageUnits);
```

A TIU just stores the texture object's state, like the clamping, mipmaps, etc. They are independent of texture coordinates. You can use whatever texture coordinate to sample whatever TIU.

Note that each shader stage has its own max texture image unit count. `GL_MAX_TEXTURE_IMAGE_UNITS` returns the count for fragment shaders only. Each shader has its own maximum number of texture image units. The number of image units across *all* shader stages is queried with `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`; this is the limit of the number of textures that can be bound at any one time. And this is the limit on the image unit to be passed to functions like `glActiveTexture` and `glBindSampler`.

For most modern hardware, the image unit count will be at least 8 for most stages. Vertex shaders used to be limited to 4 textures on older hardware. All 3.x-capable hardware will return at least 16 for *each* stage.

In summary, shader-based GL 2.0 and above programs should use `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` only. The number of texture coordinates should likewise be ignored; use generic vertex attributes instead.

Disable depth test and allow depth writes

In some cases, you might want to disable depth testing and still allow the depth buffer updated while you are rendering your objects. It turns out that if you disable depth testing (`glDisable(GL_DEPTH_TEST)`), GL also disables writes to the depth buffer. The correct solution is to tell GL to ignore the depth test results with `glDepthFunc(GL_ALWAYS)`. Be careful because in this state, if you render a far away object last, the depth buffer will contain the values of that far object.

glGetFloatv glGetBooleanv glGetDoublev glGetIntegerv

You find that these functions are slow.

That's normal. Any function of the glGet form will likely be slow. nVidia and ATI/AMD recommend that you avoid them. The GL driver (and also the GPU) prefer to receive information in the up direction. You can avoid all glGet calls if you track the information yourself.

y-axis

Almost everything in OpenGL uses a coordinate system, such that when X goes right, Y goes up. This includes pixel transfer functions and texture coordinates.

For example, `glReadPixels` takes the x and y position. The y-axis is considered from the bottom being 0 and the top being some value. This may seem counter intuitive to some who are used to their OS having the y-axis being inverted (your window's y axis is top to bottom and your mouse's coordinates are y axis top to bottom). The solution is obvious for the mouse: `windowHeight - mouseY`.

For textures, GL considers the y-axis to be bottom to top, the bottom being 0.0 and the top being 1.0. Some people load their bitmap to GL texture and wonder why it appears inverted on their model. The solution is simple: invert your bitmap or invert your model's texcoord by doing 1.0 - v.

glGenTextures in render function

It seems as if some people create a texture in their render function. Don't create resources in your render function. That goes for all the other `glGen` function calls as well. Don't read model files and create VBOs with them in your render function. Try to allocate resources at the beginning of your program. Release those resources when your program terminates.

Worst yet, some create textures (or any other GL object) in their render function and never call `glDeleteTextures`. Every time their render function gets called, a new texture is created without releasing the old one!

Bad znear value

Warning: This section describes legacy OpenGL APIs that have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0). It is recommended that you not use this functionality in your programs.

Some users use `gluPerspective` or `glFrustum` and pass it a znear value of 0.0. They quickly find that z-buffering doesn't work.

You can't have a znear value of 0.0 or less. If you were to use 0.0, the 3rd row, 4th column of the projection matrix will end up being 0.0. If you use a negative value, you would end up with wrong rendering results on screen.

Both znear and zfar need to be above 0.0. `gluPerspective` will not raise a GL error. `glFrustum` will generate a `GL_INVALID_VALUE`.

As for `glOrtho`, yes you can use negative values for znear and zfar.

The vertex transformation pipeline explains how vertices are transformed.

Bad Array Size

We are going to give this example with GL 1.1 but the same principle applies if you are using VBOs or any other feature from a future version of OpenGL.

What's wrong with this code?

```
GLfloat vertex[] = {0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0};
GLfloat normal[] = {0.0, 0.0, 1.0};
GLfloat color[] = {1.0, 0.7, 1.0, 1.0};
GLushort index[] = {0, 1, 2, 3};
glVertexPointer(3, GL_FLOAT, sizeof(GLfloat)*3, vertex);
glNormalPointer(GL_FLOAT, sizeof(GLfloat)*3, normal);
glColorPointer(4, GL_FLOAT, sizeof(GLfloat)*4, color);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_SHORT, index);
```

The intent is to render a single quad, but your array sizes don't match up. You have only 1 normal for your quad while GL wants 1 normal per vertex. You have one RGBA color for your quad while GL wants one color per vertex. You risk crashing your system because the GL driver will be reading from beyond the size of your supplied normal and color array.

This issue is also explained in the FAQ.