Pixel Buffers in OpenGL

Andrew Van Pernis
Clemson University

A brief discussion of the extensions to OpenGL to support the use of off-screen rendering buffers.

Key Words and Phrases: pixel buffer, OpenGL, off-screen rendering

1. INTRODUCTION

Modern graphics hardware has developed to include large amounts of dedicated memory. Furthermore, memory dedicated to the framebuffer is often much more than is necessary for the pixel resolutions on currently available display devices. In addition, many algorithms for enhancing the appearance of scenes rendered with OpenGL require multiple rendering passes. The basic format of these multiple pass algorithms can be found in Algorithm 1. The efficiency of these multiple

Algorithm 1 A basic multiple pass OpenGL algorithm.

```
for all passes in algorithm do
  render pass to framebuffer
  read back framebuffer and store
end for
combine stored passes and display
```

pass algorithms would be greatly enhanced if multiple off-screen framebuffers were available, ideally so that there was one buffer per pass plus one for display. Pixel buffers have been introduced as an extension to OpenGL to satisfy this need for additional off-screen rendering buffers by taking advantage of the extra memory available to the framebuffer.

At least two extensions are available for providing pixel buffers in OpenGL, WGL_ARB_PBUFFER and GLX_SGIX_PBUFFER. WGL_ARB_PBUFFER is an extension which adds pixel buffers to OpenGL on platforms running Microsoft Windows. GLX_SGIX_PBUFFER does the same for those running X windows. Both extensions provide similar functionality. Complete descriptions of the extensions can be found in the OpenGL Extension Registry [SGI 2003].

2. PIXEL BUFFERS IN X WINDOWS

Creating a pixel buffer for use with X windows requires two OpenGL extensions, GLX_SGIX_FBCONFIG and GLX_SGIX_PBUFFER. GLX_SGIX_FBCONFIG is an extension to the X windows support for OpenGL. It provides expanded capabilities

E-mail: arakel@vr.clemson.edu
Web: http://www.vr.clemson.edu/~arakel

for describing OpenGL drawable regions. GLX_SGIX_PBUFFER defines the data structures and functions used to create, access, and destroy pixel buffers.

2.1 Using GLX_SGIX_FBCONFIG

When using OpenGL with X windows, one of the first steps towards creating a GLX drawable is to select an appropriate visual for the drawable. The standard way to do that is with the command `glXChooseVisual` as can be seen in Listing 1. The GLX_SGIX_FBCONFIG extension expands the visual selection process slightly. Listing 2 shows the functions used for selecting an appropriate visual when using GLX_SGIX_FBCONFIG.

Listing 1. Standard method for selecting a visual in X windows.

```
XVisualInfo *selected_visual;
selected_visual = glXChooseVisual(display, screen, attributes);
```

Listing 2. GLX_SGIX_FBCONFIG method for selecting a visual.

```
//Get a list of frame buffer configurations
GLXFBConfigSGIX *available_configs = NULL;
int num_configs = 0;
available_configs = glXChooseFBConfigSGIX(display, screen,
    attributes, &num_configs);

//Choose a visual based on the frame buffer configuration
XVisualInfo *selected_visual;
if (num_available_configs > 0)
  selected_visual = glXGetVisualFromFBConfigSGIX(display,
      available_configs[0]);
```

A new datatype, `GLXFBConfigSGIX`, is added by GLX_SGIX_FBCONFIG which stores information describing the format, type and size of the buffers associated with a GLX drawable. The function `glXChooseFBConfigSGIX` returns a sorted list of `GLXFBConfigSGIX`s that match the attributes passed as the third parameter. If the third parameter is set to `NULL`, then the function returns a list of all of the `GLXFBConfigSGIX`s available for the given screen. The number of valid `GLXFBConfigSGIX`s is stored in the forth parameter as well. Since the `GLXFBConfig` that best matched the requested attributes is the first item in the list returned by `glXChooseFBConfigSGIX`, it is then passed as the second parameter to the function `glXGetVisualFromFBConfigSGIX`. Any of the `GLXFBConfigSGIX`s could be used, but it is rare that a programmer would want anything other than the one that best matched the attributes requested.

The main contribution of the GLX_SGIX_FBCONFIG extension is allowing a programmer to request visuals with a wider variety of attributes than the standard method (i.e. using `glXChooseVisual`). Furthermore, it allows other extensions, such as GLX_SGIX_PBUFFER, to request visuals with attributes that those extensions have added to OpenGL.

2.2 Using GLX_SGIX_PBUFFER

In order to create a pixel buffer, an appropriate frame buffer configuration must be selected using the GLX_SGIX_FBCONFIG extension described in 2.1. Choosing a frame buffer configuration for the pixel buffer requires specifying a list of desired attributes. This list of attributes is defined in the code as an array of integer values. An example set of attributes can be seen in Listing 3. The first pair of values in the attribute list specify that the desired frame buffer configuration is for a pixel buffer. The remaining pairs are standard values that would be used for determining any type of frame buffer configuration. Note that double-buffering is not required, since the contents of the pixel buffer will not be displayed directly.

Listing 3. Example set of desired frame buffer attributes.

```
const int ATTRIBUTES_32BPP[] = {
  GLX_DRAWABLE_TYPE_SGIX, GLX_PBUFFER_BIT_SGIX,
  GLX_RENDER_TYPE_SGIX, GLX_RGBA_BIT_SGIX,
  GLX_DOUBLEBUFFER, GL_FALSE,
  GLX_RED_SIZE, 8,
  GLX_GREEN_SIZE, 8,
  GLX_BLUE_SIZE, 8,
  GLX_ALPHA_SIZE, 8,
  None
};
```

After an appropriate frame buffer configuration has been selected, a pixel buffer can be created with a call to `glXCreateGLXPbufferSGIX`. Listing 4 shows how to create a pixel buffer using the attributes specified above. An additional list of attributes is passed to the pixel buffer creation function as an array of integers. Currently, two values are used to define pixel buffer attributes. The boolean value `GLX_PRESERVED_CONTENTS_SGIX` specifies whether or not the contents of the pixel buffer should be maintained when a resource conflict occurs. The other value, `GLX_LARGEST_PBUFFER_SGIX`, is used to get the largest available pixel buffer when the requested size is not available.

Listing 4. Pixel buffer creation example.

```
const int PIXEL_BUFFER_ATTRIBUTES[] = {
  GLX_PRESERVED_CONTENTS_SGIX,
  None
};
int num_configs = 0;
int width = 640, height = 480;
GLXFBConfigSGIX *available_configs = glXChooseFBConfigSGIX(
    display, screen, ATTRIBUTES_32BPP, &num_configs);
GLXPbuffer pixel_buffer = glXCreateGLXPbufferSGIX(display,
    available_configs[0], width, height,
    PIXEL_BUFFER_ATTRIBUTES);
```

An OpenGL context, in the form of a `GLXContext`, must be created for the pixel buffer in order to use it. GLX_SGIX_FBCONFIG provides a function for context cre-

ation using a GLXFBConfigSGIX. The main difference between glXCreateContext-WithConfigSGIX and the standard glXCreateContext is that the context created by glXCreateContextWithConfigSGIX can be used to render to any compatible GLXDrawable, whereas glXCreateContext can only be used to create contexts for windows or GLXPixmaps. An example of context creation for a pixel buffer can be seen in Listing 5. Also note, the third parameter to glXCreateContextWith-ConfigSGIX allows the programmer to specify a lists of contexts that will share display lists with the newly created context. Contexts that share display lists also share texture objects (except for texture object 0). By sharing display lists and texture object with the main display window the contents of a pixel buffer can be copied directly into a texture using one of the glCopyTexImage* or gl-CopyTexSubImage* commands. In order to switch between contexts, the command glXMakeContextCurrent is used. The function glXGetCurrentContext can check what the current context is. It is important to remember when using pixel buffers, OpenGL commands apply only to the current context, which must be set explicitly.

Listing 5. Creating a GLXContext for a pixel buffer.

```
GLXContext _pixel_buffer_context =
    glXCreateContextWithConfigSGIX(display, available_configs
    [0], GLX_RGBA_TYPE_SGIX, window_context, GL_TRUE);
```

There are several other commands which are used to manipulate pixel buffers. The first, glXQueryGLXPbufferSGIX, is used to obtain information about the pixel buffer. The width, height and i.d. of the associated GLXFBConfigSGIX can found with glXQueryGLXPbufferSGIX, as well as the status of either the attributes used when creating a pixel buffer. Two related commands, glXSelectEventSGIX and glXGetSelectedEventSGIX, are used along with a single GLX event, GLX_BUFFER_-CLOBBER_MASK_SGIX, to determine when a pixel buffer has been invalidated by a resource conflict. The final command is glXDestroyGLXPbufferSGIX, which is used to eliminate pixel buffer that are no longer needed. An example of pixel buffer destruction can be seen in Listing 6. As can be seen in the example, the context associated with a pixel buffer should be destroyed before the pixel buffer itself.

Listing 6. Destroying a pixel buffer.

```
glXDestroyContext(display, pixel_buffer_context);
pixel_buffer_context = NULL;
glXDestroyGLXPbufferSGIX(display, pixel_buffer);
```

3. PIXEL BUFFER SUPPORT IN UBER

Uber's pixel buffer wrapper class is called pixel_buffer and is contained within the gl namespace. Because it is simply a wrapper to provide object-oriented functionality for pixel buffers, the pixel_buffer class contains an associated pixel buffer, which is not automatically created when a pixel_buffer object is created. Instead, a call to the member function init is used to create the associated pixel buffer. Subsequent calls to init will not create additional pixel buffers. The pixel buffer attached to the pixel_buffer object must be destroyed through the shutdown

method, before another pixel buffer can be created by `init`. Thus, a programmer desiring multiple pixel buffers should create multiple `pixel_buffer` objects.

The `init` method has been overloaded to provide the programmer with multiple ways of creating a pixel buffer. The simplest requires four parameters; a width and height for the pixel buffer, the desired bits for pixel assuming use of RGBA values, and a list of contexts to share. A second version of `init` is similar to the first but allows the programmer to specify a list of desired attributes for the pixel buffer as well. Listing 7 demonstrates the creation of a `pixel_buffer` object and the use of the `init` method. Note that, `init` returns a boolean value declaring whether or not a pixel buffer was successfully allocated for the `pixel_buffer` object. If a pixel buffer could not be created, the method `get_error` can be used to obtain an error message for further explanation.

Listing 7. Creating a pixel buffer in Uber.

```
const int PIXEL_BUFFER_SIZE = 512;
gl::pixel_buffer p_buffer
if (!p_buffer.init(PIXEL_BUFFER_SIZE, PIXEL_BUFFER_SIZE, 32,
    NULL)){
  std::cerr << "ERROR: " << p_buffer.get_error() << std::endl;
  return 2;
}
```

Once a pixel buffer has successfully created, the method, `activate`, is used to switch to the context associated with that pixel buffer. Because copying the contents of a pixel buffer to a texture for use in the display window is the most common case for using a pixel buffer, an Uber `texture` object is created to go along with each `pixel_buffer` object after it has been initialized. This `texture` object can be obtained with the method, `get_texture`. Listing 8 demonstrates using the `texture` object that is created by a `pixel_buffer` object. Finally, the `shutdown` method can be used to destroy the pixel buffer associated with a `pixel_buffer` object.

Listing 8. Using an Uber texture with a pixel buffer.

```
p_buffer.get_texture().bind();
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR
    );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0,
    PIXEL_BUFFER_SIZE, PIXEL_BUFFER_SIZE, 0);
```

4. CONCLUSION

Pixel buffers are a fairly simple extension to OpenGL that provide the programmer with a great deal of flexibility for implementing advanced rendering algorithms.

Although two platform dependent extensions exist to add pixel buffer support, the differences between the extensions are not significant. This allows the extensions to be wrapped inside of class within the Uber library.

REFERENCES

SGI. 1993-2003. OpenGL extension registry. http://oss.sgi.com/projects/oglsample/registry/.

Shreiner, D., Ed. 2000. *OpenGL Reference Manual*, Third ed. Addison-Wesley. The Official Reference Document to OpenGL, Version 1.2.

Woo, M., Neider, J., Davis, T., and Shreiner, D. 1999. *OpenGL Programming Guide*, Third ed. Addison-Wesley. The Official Guide to Learning OpenGL, Version 1.2.