1

P-buffers and FBOs

This document describes in more detail how to configure and use P-buffers and Frame Buffer Objects, that we did not have space to include in the book.
P-buffers are not the modern way of rendering into off-screen storage, use FBOs for that. FBOs are the currently recommended way of rendering to a texture. They are a very useful way of doing interesting effects.

1.1 The P buffer and Framebuffer Objects

It was mentioned in the book that in addition to the front and back output buffers OpenGL may offer a few additional buffers, AUX0 AUX1 etc. however, many display adapters do not support these buffers. Even if the AUXn buffers are supported they have a major limitation in that they cannot exceed the current size of the output window, more specifically, they cannot exceed the size of the underlying *drawing surface*, an output window is just a system dependent view of the drawing *surface*, OpenGL renders onto a drawing surface attached to a window.. This is a major problem if you want to render a scene at a resolution that is greater than the desktop size. It is also a major problem if you want to render to a texture (an image map) to be used later. (for example, an image may be required at a resolution of twice the maximum window size, and include an automatically generated mip-map, taking its resolution even higher. Being able to render to very large off-screen buffers of any size has advantages, it is used in OpenFX (see chapter **??**).

The need for off-screen output buffers was recognized early in the development of OpenGL and provision was made to create off screen buffers to hold the output of the fragment processor, these were called *Pixel Buffers*, or just P-buffers. P-buffers behave in a similar way to the normal drawing surface and initializing them requires the specification of a pixel format and the creation of device and drawing contexts specifically for the off-screen buffer This process is platform dependent, but once created there is no difference in rendering into a P-buffer or rendering directly to a visible display surface.

P-buffers are considers as old technology, because each buffer has its own OpenGL context and switching between contexts is slow, and they don't allow rendering directly to a texture. Each P-buffer has its own depth, stencil and AUX buffer but the depth buffer cannot be shared among several P-buffers. To make a texture based on the output of a P-buffer it is necessary to read the P-buffer contents into a normal RAM buffer an then pass the pixels back into the GPU, this slows down the whole process, so this is where the FBOs come in. FBOs were designed so that firstly, textures could be created by directly rendering into texture memory, and secondly, several FBO may be used with a single OpenGL context, thus avoiding a time consuming context switch. A FBO is actual a short structure of pointers to various video memory buffers, it is these buffers that have to be allocated with an FBO is created.

During the use of a FBO the memory buffers are write-only and their contents cannot be accessed. However if we create and attach another type of buffer (called a *RenderBufffer*) to the FBO then the contents of the FBO can be returned to the CPU at any time. RenderBuffer images and textures can be shared among FBOs, for example the depth buffer can be shared and this saves memory. FBOs allow several ways of switching between rendering targets. For example:

- Multiple FBOs: A separate FBO is created for each texture to be rendered to. Switching from one to the other using `BindFramebuffer():` is much faster than using `wglMakeCurrent();`.
- Single FBO with multiple texture attachments: The textures should have the same format and size, function `FramebufferTexture();` is used to switch between them.
- Single FBO with multiple textures used as color output buffers: Use `glDrawBuffer()`; the switch rendering between different color buffers.

In sections 1.1.1 and 1.1.2 we will develop short template codes that show the steps that need to be taken to use a P-buffer, and an FBO with an attached renderbuffer. Because of the platform dependent nature of these steps we will focus our examples on a Windows host. You can see a comprehensive example of P-buffer and FBO usage in the code of OpenFX, and this will be discussed further in chapter ??;

1.1.1 Using P-buffers

The code discussed in this section is taken from examples that execute on a Windows PC. The creation and initialization of a P-buffer is platform-specific, in the case of a Windows host this requires that a number of the Windows specific OpenGL functions are called (Windows specific OpenGL functions have names that begin `wgl.....`). When it comes to rendering into a P-buffer it is necessary to have a record of which buffer and which rendering contexts to use. In our example this information is recorded in a small structure and a couple of global variables:

```
struct PBuffer        {
  HPBUFFERARB  hPBuffer; // P-buffer pointer
  HDC   hDC;  // buffer's device context
  HGLRC  hGLRC; // buffer's rendering context
} m_PBuffer;
// copy of main window's device context
static HDC m_hDCWnd;
// copy of main window's render context
static HGLRC m_hGLRCWnd;
```

To create a P-buffer, its pixel format and buffer requirements must be specified by calling function `.wglChoosePixelFormat()`, the buffer is created by calling `wglCreatePbuffer(..)` and the rendering context created by calling `wglCreateContext(...)` the code to achieve this is shown in listing1.1. and the most commonly used buffer properties are defined through the settings in listing 1.1. listing1.1.

```
BOOL InitPBuffer(void){
 int iPixelFormat;
 HDC hPrimaryDevCtx = wglGetCurrentDC();
 unsigned int iFormatCount;

 int pPixFmtRequirements[] = {... 0};
 int pBufRequirements [] = {... 0 };

 wglChoosePixelFormat(
   hPrimaryDevCtx, // Device context to query
  (const int*) pPixFmtRequirements, // integer parameters
   NULL, // List of float parameters
   1, // Number of formats to return
   &iPixelFormat, // The outputted pixel format
   &iFormatCount ); // Number of outputted formats
 if( iFormatCount == 0 )return FALSE;

// Now that we have a device context and a pixel format, we can
 // create the P-Buffer.
 m_PBuffer.hPBuffer = wglCreatePbuffer(
 hPrimaryDevCtx, // Window device context
  iPixelFormat, // Chosen pixel format
  RENDERBUFFER_WIDTH, // P-Buffer width
  RENDERBUFFER_HEIGHT, // P-Buffer height
  pBufRequirements ); // No extra attributes
 if( m_PBuffer.hPBuffer == NULL )return FALSE;

 // record buffer information.
 m_PBuffer.hDC = wglGetPbufferDC( m_PBuffer.hPBuffer );
 m_PBuffer.hGLRC = wglCreateContext( m_PBuffer.hDC );
 return TRUE;
}
```

Listing 1.1: Create a P-buffer.

To render into a P-buffer it is sufficient to activate the OpenGL context created for the P-buffer and then execute the same drawing commands, in the same way as would be done for a surface attached to an ordinary desktop window, see listing 1.3 for typical code to achieve this.

When the P-buffer is no longer required it shoud be destroyes with the code.

```
wglDeleteContext( m_PBuffer.hGLRC );
wglReleasePbufferDCARB( m_PBuffer.hPBuffer, m_PBuffer.hDC );
wglDestroyPbufferARB( m_PBuffer.hPBuffer );
```

1.1.2 Using FBOs

Before a FBO can be used as a target for rendered output it has to be created, and the buffers that it uses allocated and configured. The code explained in this section will exemplify how to use a FBO as a target for an output image rather

```
int pPixFmtRequirements[] = {
 WGL_DRAW_TO_PBUFFER, TRUE, // Enable buffer
 WGL_SUPPORT_OPENGL, TRUE, // Associate with OpenGL
 WGL_DOUBLE_BUFFER, FALSE, // Single-buffer
 WGL_RED_BITS, 8, // 8 bits of red
 WGL_GREEN_BITS, 8, // 8 bits of green
 WGL_BLUE_BITS, 8, // 8 bits of bluem
 WGL_DEPTH_BITS, 24, // 16 bits depth buffer
 WGL_ACCUM_BITS, 16, // 16 bit accumulation buffer
 WGL_STENCIL_BITS, 1,
0 // End of list
int pBufRequirements [] = {
 // P-Buffer will bind to a 2D texture
 WGL_TEXTURE_TARGET, WGL_TEXTURE_2D,
 // P-Buffer will bind to a texture with format RGBA
 WGL_TEXTURE_FORMAT, WGL_TEXTURE_RGBA,
 0
};
```

Listing 1.2: P-buffer parameters. Constants that begin `WGL_..` are defined for the Windows platform and become available to programs through the OpenGL header files.

```
// activate the P-buffer context
wglMakeCurrent( m_PBuffer.hDC, m_PBuffer.hGLRC );
// render the scene
RenderScene(..)  // P

//If desired, read back the pixels data
// from the P-buffer
glReadPixels(0,0,
 RENDERBUFFER_WIDTH,RENDERBUFFER_HEIGHT, // image size
 GL_RGBA,GL_UNSIGNED_BYTE, // format
 RenderBufferCopy); // buffer to receive image

// switch back to the window's context
wglMakeCurrent( m_hDCWnd, m_hGLRCWnd );
```

Listing 1.3: Drawing into the P-buffer, and optionally reading the pixel image back from the P-buffer into the application's memory space.

than as an internally generated texture for use in an image map in another part of the program. As a target for a hidden surface 3D image, the FBO will need to have a color buffer and a depth buffer attached to it. The full example code can be found on our website, whilst here in the text, only the key constructs are highlighted in listings 1.4 to 1.6. The FBOs and its attached buffers are identified using integer handles identified by the following global variables:

```
GLuint g_frameBuffer;
GLuint g_depthRenderBuffer;
GLuint g_colorRenderBuffer ;
```

The dimensions of the image size that will be created are defined by variables `ImageWidth, ImageHeight`. As this is a Windows application some of the plat-

```
..
// Set up Framebuffer for normal rendering
glGenFramebuffers(1, &g_frameBuffer); // The FBO
// A colour buffer output buffer
glGenRenderbuffers(1, &g_colorRenderBuffer );
glBindRenderbuffer(GL_RENDERBUFFER,g_colorRenderBuffer );
glRenderbufferStorage( GL_RENDERBUFFER,
  GL_RGBA8, // 8 bit each for R G B colors and Alpha
  ImageWidth, ImageHeight );
// A depth buffer
glGenRenderbuffers(1,&g_depthRenderBuffer );
glBindRenderbuffer(GL_RENDERBUFFER,g_depthRenderBuffer );
glRenderbufferStorage(GL_RENDERBUFFER,
  GL_DEPTH_COMPONENT24,  // 24 bit depth resolution
  ImageWidth,ImageHeight);
status = glCheckFramebufferStatus( GL_FRAMEBUFFER );
switch( status ) {
 case GL_FRAMEBUFFER_COMPLETE :
   //.. everything OK
  break;
 case GL_FRAMEBUFFER_UNSUPPORTED :
  //.. fail
  break;
 default: return NULL;
}
...
```

Listing 1.4: Setting-up a FBO for normal rendering with attached color and depth renderbuffers.

form specific `wgl..` library functions are used to build the FBOs. Listing 1.4 presents the code to generate the FBO and renderbuffers that have the required properties. A depth buffer with 24 bits will give depth resolution of one part in about 16 million. Once an FBO has been set up it can be targeted for the output from any OpenGL drawing commands, there is no need for an interactive display window to be active. And it is possible to read the pixel data from the FBO back into the application's memory, and hence write them into an image file. Listing 1.5 illustrates the steps involved in activating the FBO as a target for a stream of drawing commands. The final listing in this sequence (listing 1.6) illustrates the mechanism for releasing the resources allocated to the FBO.

```
// Activate FBO (by binding it)
glBindFramebuffer( GL_FRAMEBUFFER , g_frameBuffer );
// attach the color renderbuffer to current FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER , GL_COLOR_ATTACHMENT0 ,
GL_RENDERBUFFER , g_colorRenderBuffer );
// attach the depth renderbuffer to the FBO
glFramebufferRenderbuffer( GL_FRAMEBUFFER , GL_DEPTH_ATTACHMENT ,
GL_RENDERBUFFER , g_depthRenderBuffer );
// clear the output buffer
glViewport( 0, 0, ImageWidth, ImageHeight );
glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
glClear(GL_COLOR_BUFFER_BIT);
// render the view (Normal OpenGL function calls)
RenderCameraView();
glFlush();
//
// Do anything here with the image, for example read the pixels
// into CPU memory buffer
/// glReadPixels(0,0, ImageWidth,ImageHeight,GL_RGBA,GL_UNSIGNED_BYTE,
/// PointerToImageBufferInCPU);
//
// unbind the FBO as rendering target
glBindFramebuffer( GL_FRAMEBUFFER , 0 );

..
```

Listing 1.5: Draw into an FBO.

```
// delete all the buffers and release GPU memory
glDeleteFramebuffers(1, &g_frameBuffer );
glDeleteRenderbuffers(1, &g_depthRenderBuffer );
glDeleteRenderbuffers(1, &g_colorRenderBuffer );
```

Listing 1.6: Releasing the graphics resources used by the FBO and renderbuffers.

Bibliography