# Using P-Buffers for Off-Screen Rendering in OpenGL

Chris Wynn
cwynn@nvidia.com
NVIDIA Corporation

## Overview

The OpenGL extension `WGL_ARB_pbuffer`, used in conjunction with the `WGL_ARB_pixel_format` extension, offers the ability to render to an off-screen pixel buffer. On NVIDIA-based graphics cards, the off-screen buffer is stored in video memory and this process of rendering to an off-screen buffer is accelerated in hardware.

In OpenGL applications, there are often several circumstances in which rendering to an off-screen buffer is necessary or useful. Sometimes rendering to such a buffer can be useful for creating dynamic textures for things like dynamic cube-map generation, dynamic normal-map generation, or other feedback effects such as procedural texturing and image processing. Rendering to an off-screen buffer is also useful when you want to update a texture but don't want the size of the texture to be limited by the current window size. When used in conjunction with the `WGL_ARB_render_texture` extension, pbuffers are invaluable for creating dynamic texture data. While this document does not cover the `WGL_ARB_render_texture` extension, the groundwork for using pbuffers is explained in detail. Please see future documentation for information on using this powerful extension on the NVIDIA GeForce/Quadro family of GPUs. The extensions discussed in this paper are available in currently available drivers. In addition, example applications that demonstrate the use of pbuffers are available at www.nvidia.com/Developer. These example programs illustrate one possible way to use an abstraction of a pbuffer class object to encapsulate the complexity of the pbuffer creation and management process.

## General Process

The process of using pbuffers can be separated into four phases:

1. pbuffer *extension initialization*
2. pbuffer *creation*
3. pbuffer *binding*
4. pbuffer *destruction*.

In the sections that follow, we describe each of these in detail.

## Extension initialization

In order to use pbuffers, the entry points for the `WGL_ARB_pixel_format` and `WGL_ARB_pbuffer` extensions must first be obtained. There are three steps to this process. (If you are already familiar with extension initialization in OpenGL, feel free to skip this section, but be aware that this uses WGL extensions, which are somewhat trickier than your standard OpenGL extensions.)

**Step1.** Determine if `WGL_ARB_extensions` string is available. To do this, all that is required is a `wglGetProcAddress()` call:

```
wglGetExtensionsStringARB =(PFNWGLEXTENSIONSSTRINGARBPROC)
          wglGetProcAddress( “wglGetExtensionsStringARB” );
```

If this function does not return NULL, you can continue.

**Step 2.** Acquire the extensions string and search for the sub-strings `WGL_ARB_pixel_format` and `WGL_ARB_pbuffer`:

```
extensions = (const Glubyte *)
            wglGetExtensionsStringARB( wglGetCurrentDC() );
/* Now search through the string using strstr() or any other
similar function. */
```

If the both strings are found, you can continue. If the strings are not found, make sure you are using an appropriate version of the driver.

**Step 3.** Load up the entry points for the `WGL_ARB_pixel_format` and `WGL_ARB_pbuffer` extensions:

```
#define INIT_ENTRY_POINT( funcname, type )                    \
funcname = (type) wglGetProcAddress(#funcname);               \
if ( !funcname )                                              \
    fprintf( stderr, "#funcname() not initialized\n" );



/* Initialize WGL_ARB_pbuffer entry points. */
INIT_ENTRY_POINT(
  wglCreatePbufferARB, PFNWGLCREATEPBUFFERARBPROC );
INIT_ENTRY_POINT(
  wglGetPbufferDCARB, PFNWGLGETPBUFFERDCARBPROC );
INIT_ENTRY_POINT(
  wglReleasePbufferDCARB, PFNWGLRELEASEPBUFFERDCARBPROC );
INIT_ENTRY_POINT(
  wglDestroyPbufferARB, PFNWGLDESTROYPBUFFERARBPROC );
INIT_ENTRY_POINT(
  wglQueryPbufferARB, PFNWGLQUERYPBUFFERARBPROC );


/* Initialize WGL_ARB_pixel_format entry points. */
INIT_ENTRY_POINT(
  wglGetPixelFormatAttribivARB,
  PFNWGLGETPIXELFORMATATTRIBIVARBPROC );
INIT_ENTRY_POINT(
  wglGetPixelFormatAttribfvARB,
  PFNWGLGETPIXELFORMATATTRIBFVARBPROC );
INIT_ENTRY_POINT(
  wglChoosePixelFormatARB,
  PFNWGLCHOOSEPIXELFORMATARBPROC );
```

If any of the entry points fail to initialize, ensure that you are using the correct version of the driver and that the entry points were spelled correctly. Once you have all the entry points initialized, you can now create and use a pbuffer.

## Pbuffer Creation

The process of creating a pbuffer requires a few tedious steps. Fortunately, once the creation process has been implemented successfully, it can usually be encapsulated into a function or class and the implementation reused in multiple applications.

There are five major steps required to create a pbuffer:

1. Get a valid device context
2. Find a pbuffer-capable pixel format
3. Create a pbuffer of the chosen pixel format
4. Create a device context for the pbuffer
5. Create an OpenGL rendering context associated with the pbuffer

**Step 1.** Getting a valid device context amounts to getting an ID or handle to the device that's going to be doing all the rendering work. In this case (and most cases in general), the device we want to get is the graphics accelerator device. If a window (capable of displaying OpenGL graphics on the screen) has just been created, this process is straightforward:

```
HDC hdc = wglGetCurrentDC();
```

This call gives you the handle to the graphics accelerator device.

**Step 2.** The process of determining which pixel format to use when creating a pbuffer is the most challenging part of the creation process. Basically, the purpose of this step is to tell the hardware what "kind" of pbuffer is required by your application. Essentially, you must give the hardware a list of attributes or properties that you will require of the pbuffer, and the hardware returns a list of candidate pixel formats that meet your minimum criteria. Each pixel format is identified by a unique integer value, so the returned list of formats is actually just a list of integers. In some (rare) cases it may be the case that you request a set of attributes for which the hardware is unable to come up with a suitable pixel format. In such cases, the number of elements contained in the list will be zero and you will likely need to "weaken" the requirements of the pixel buffer you are attempting to create. In general, however, this will not happen and you will be left with a handful of formats from which to choose.

Before asking the hardware for a pixel format you must first determine what type of pbuffer your application will require. As an example, let's suppose that you need to perform a certain type of off-screen rendering that requires a 32-bit color (RGBA) pbuffer with at least 24 bits of depth. You would first initialize an array that stores a set of (attribute, value) pairs and then you would make a call to the function, `wglChoosePixelFormatARB()` to determine which formats match what the hardware has available. Code listing 1 shows exactly what you might do for this example.

```
// Get ready to query for a suitable pixel format that meets our
// minimum requirements.

int   iattributes[2*MAX_ATTRIBS];
float fattributes[2*MAX_ATTRIBS];
int   nfattribs = 0;
int   niattribs = 0;

// Attribute arrays must be “0” terminated - for simplicity, first
// just zero-out the array then fill from left to right.
for ( int a = 0; a < 2*MAX_ATTRIBS; a++ )
    {
    iattributes[a] = 0;
    fattributes[a] = 0;
    }

// Since we are trying to create a pbuffer, the pixel format we
// request (and subsequently use) must be “p-buffer capable”.
iattributes[2*niattribs  ] = WGL_DRAW_TO_PBUFFER_ARB;
iattributes[2*niattribs+1] = true;
niattribs++;

// We require a minimum of 24-bit depth.
iattributes[2*niattribs  ] = WGL_DEPTH_BITS_ARB;
iattributes[2*niattribs+1] = 24;
niattribs++;

// We require a minimum of 8-bits for each R, G, B, and A.
iattributes[2*niattribs  ] = WGL_RED_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;
iattributes[2*niattribs  ] = WGL_GREEN_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;
iattributes[2*niattribs  ] = WGL_BLUE_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;
iattributes[2*niattribs  ] = WGL_ALPHA_BITS_ARB;
iattributes[2*niattribs+1] = 8;
niattribs++;

// Now obtain a list of pixel formats that meet these minimum
// requirements.
int pformat[MAX_PFORMATS];
unsigned int nformats;
if ( !wglChoosePixelFormatARB( hdc, iattributes, fattributes,
                               MAX_PFORMATS, pformat, &nformats ) )
    {
    fprintf( stderr, "pbuffer creation error:  Couldn't find a \
                      suitable pixel format.\n" );
    exit( -1 );
    }
```

**Code Listing 1. Finding a set of compatible pixel formats.**

Code listing 1, demonstrates how to build up a list of essential attributes and then request from the hardware all pixel formats that meet your requirements. Careful examination of the code reveals that an array of (attribute, value) pairs is initialized first.

The pair ( `WGL_DRAW_TO_PBUFFER_ARB, true` ) specifies that the requested pixel format(s) must be a "p-buffer capable" format.

The pair ( `WGL_DEPTH_BITS_ARB, 24` ) specifies that the requested pixel format(s) must have a minimum of 24 bits of depth buffer.

The pair ( `WGL_RED_BITS_ARB, 8` ) specifies that the requested pixel format(s) must have a minimum of 8 bits for the red color channel.

The remaining (attribute, value) pairs set a minimum of 8 bits for the green, blue, and alpha channels.

In this example, all of the attributes are specified as having integer (as opposed to floating point) attribute values. To specify floating point (attribute, value) pairs, the array `fattributes` would be filled with pairs as well.

The function, `wglChoosePixelFormat()` is the function that asks the hardware for a list of pixel formats that matches the criteria specified in the attributes list. This function returns a non-zero value if the function succeeds and creates a list of pixel format indices that matched met the criteria specified in the list of attributes. The arguments to this function are as follows:

| | |
|---|---|
| `hdc` | The handle for the device context acquired in step #1. |
| `iattributes` | A zero-terminated list of integer (attribute, value) pairs. |
| `fattributes` | A zero-terminated list of floating point (attribute, value) pairs. |
| `MAX_PFORMATS` | An integer indicating the maximum number of matching formats that should be returned. |
| `pformat` | An integer array capable of storing at least `MAX_PFORMATS` values (this is where the list of matching pixel format indices will be stored when the function returns). |
| `nformats` | An integer that will contain the actual number of matching formats when the function returns. |

While this example shows how to find a suitable pbuffer with at least 32 bits of color and 24 bits of depth, there are many other attributes that may be specified as well. Here's a sampling of some of the other more commonly used attributes:

WGL_COLOR_BITS_ARB
: The minimum number of total R, G, and B color bits.

WGL_STENCIL_BITS_ARB
: The minimum number of stencil bits.

WGL_AUX_BUFFERS_ARB
: The minimum number of auxiliary buffers.

WGL_ACCELERATION_ARB[1]
: Whether or not the pixel format must be supported in hardware.

WGL_SUPPORT_OPENGL_ARB[1]
: Whether the format must support OpenGL.

WGL_ACCUM_BITS_ARB
: The minimum number of accumulation bits (rarely used).

WGL_DOUBLE_BUFFER_ARB
: Double-buffered (very rarely used).

WGL_PIXEL_TYPE_ARB
: The type of pixel data: color-index mode or RGBA (color-index very rarely used).

For a complete list of possible attributes and their possible values, take a look at the WGL_ARB_pixel_format extension available at the OpenGL extensions registry web-site:

http://www.oss.sgi.com/projects/ogl-sample/registry/

[1] On NVIDIA-based hardware, wglChoosePixelFormatARB() will only return hardware accelerated formats that support OpenGL so there is no need to specify the WGL_ACCELERATION_ARB or WGL_OPENGL_ARB attributes. This is not necessarily the case for alternate implementations of the WGL_ARB_pixel_format extension. For portability purposes, you may want to explicitly specify those attributes in addition to any others your application may require.

**Step 3.** After determining a compatible pixel format, the next step is to create a pbuffer of the chosen format. Fortunately this step is fairly easy, as you merely select one of the formats returned in the list in step #2 and call the function:

```
HPBUFFERARB hbuffer =
     wglCreatePbufferARB( hdc, format, iwidth,
                          iheight, iattribs );
```

The arguments to this function are:

| | |
|---|---|
| `hdc` | The handle for the device context acquired in step #1. |
| `format` | The single index from the list of those returned in step #2. (Typically, the first element in the list suffices) |
| `iwidth` | The desired pixel width of the pbuffer. |
| `iheight` | The desired pixel height of the pbuffer. |
| `iattribs` | A zero-terminated list of integer (attribute, value) pairs. Only one attribute is supported by this function: WGL_PBUFFER_LARGEST_ARB If the value of this attribute is set to a non-zero value, the largest available pbuffer is allocated when the allocation of the pbuffer would otherwise fail due to insufficient resources. (The width or height of any allocated pbuffer will never exceed iwidth or iheight respectively)[2] |

The return value of this function, `hbuffer`, is a handle that identifies the created pbuffer.

**Step 4.** The next step is to create a device context for the newly created pbuffer. To do this, call the the function:

```
HDC hpbufdc = wglGetPbufferDCARB( hbuffer );
```

Where the function parameters are:

| | |
|---|---|
| `hbuffer` | The handle to the pbuffer returned in step #3. |

The return value of this function, `hpbufdc`, is the handle that pbuffer's device context.

[2] After creating a pbuffer, you may use the functions

```
wglQueryPbufferARB( hbuffer, WGL_PBUFFER_WIDTH_ARB, &h );
wglQueryPbufferARB( hbuffer, WGL_PBUFFER_WIDTH_ARB, &w );
```

to determine the dimensions of the pbuffer actually created.

**Step 5.** The final step of pbuffer creation is to create an OpenGL rendering context and associate it with the handle for the pbuffer's device context created in step #4. This is done as follows:

```
    pbufglctx = wglCreateContext( hpbufdc );
```

The argument to this function is:

| | |
|---|---|
| `hpbufdc` | The handle to the pbuffer's device context returned in step #4. |

The return value, `pbufglctx`, is a handle to the OpenGL rendering context that is now associated with the pbuffer. This value (as well as the pbuffer's device context returned in step #4) must be retained and used whenever the pbuffer's rendering context is "bound" or made current.

## Pbuffer Binding

After a pbuffer has been successfully created you can use it for off-screen rendering. To do so, you'll first need to "bind" the pbuffer, or more precisely, make its GL rendering context the current context that will interpret all OpenGL commands and state changes.

To do this, simply call:

```
    wglMakeCurrent( hpbufdc, pbufglctx );
```

Where the parameters to this function are:

| | |
|---|---|
| `hpbufdc` | The handle to the pbuffer's device context returned in step #4 of the creation process. |
| `pbufglctx` | The handle to the OpenGL rendering context acquired in step #5 of the creation process. |

This will make the pbuffer's rendering context the current rendering context.

To restore the current OpenGL rendering context to the on-screen OpenGL window, you'll need to call `wglMakeCurrent()` with the device context and OpenGL rendering context corresponding to the displayable window. (If using GLUT for your window creation management, you may instead call `glutSetWindow( id )` where `id` is the identifier for the glut window. This identifier is retrievable with the `glutGetWindow()` function call.)

## Pbuffer Destruction

When the pbuffer is no longer required, it should be "destroyed" so that the system and video memory resources associated with the off-screen buffer may be restored and made available for other tasks.

There are three steps that should be taken when destroying a pbuffer:

1. Free up the memory associated with the pbuffer's GL context
2. Release the pbuffer's device context
3. Free up the remaining memory associated with the pbuffer

These steps should be implemented with the following code:

```
wglDeleteContext( pbufglctx );
wglReleasePbufferDCARB( hbuffer, hpbufdc );
wglDestroyPbufferARB( hbuffer );
```

Where:

| | |
|---|---|
| `hpbufdc` | The handle to the pbuffer's device context acquired in step #4 of the creation process. |
| `pbufglctx` | The handle to the OpenGL rendering context acquired in step #5 of the creation process. |
| `hbuffer` | The handle to the pbuffer acquired in step #3 of the creation process. |

After these functions are completed, all the video memory associated with the buffer is restored to the video memory subsystem, and the associated host memory is restored to the system as well.

## Handling a Display Mode-Switch

The memory associated with a pbuffer is not guaranteed to remain valid when a display "mode-switch" occurs. Typically these happens during a call to ChangeDisplayMode() but may happen for other reasons as well. When this does occur, it is the programmer's responsibility to ensure that the memory associated with the pbuffer may still be used as a pbuffer. To do this, a query has been provided that can be called after a mode-switch to determine whether a pbuffer was invalidated during the mode-switch.

To query if a pbuffer is still valid after a mode-switch, the following function should be called:

```
int flag = 0;
wglQueryPbufferARB( hbuffer, WGL_PBUFFER_LOST_ARB,
                    &flag );
```

where `hbuffer` is the handle to the pbuffer.

If the value of `flag` is non-zero after this function call, the pbuffer has been lost and therefore must be destroyed and then re-created.

## Summary

This documented has presented the groundwork for the creation and management of pbuffers on Windows-based operating systems. It should be noted that hardware accelerated pbuffers, such as those available on NVIDIA-based hardware, do consume video memory. Since video memory is the most precious resource of the graphics subsytem (used for storing texture data, frame-buffer data, and sometimes geometry data), pbuffers should be used with some degree of discretion. Additionally, the size of a pbuffer should be limited to the maximum size required by a particular application and destroyed during extended periods of time when they will not be used. For additional information on pbuffers, reference the WGL_ARB_pbuffers extension available at the OpenGL extensions registry web-site:

http://www.oss.sgi.com/projects/ogl-sample/registry/