OpenGL* Performance Tips: Swap Frame Buffer Objects (FBO) Instead of Surfaces Using a Single FBO

Introduction

This article discusses how to improve OpenGL* performance by swapping Frame Buffer Objects (FBO) instead of using a single FBO and swapping surfaces. Swapping is useful when making multiple changes to a rendered image, such as switching color, depth, or stencil attachments. The recommendation is to use dedicated FBOs for each set in use, rather than sharing an FBO among all attachments. Switching an entire FBO is more efficient than switching individual surfaces one at a time.

The article is accompanied by a C++ example application that shows the results of not using and using the suggested technique. While this article refers to graphical game developers, the concepts apply to all applications that use OpenGL 4.3 and higher. The sample code is written in C++ and is designed for Windows* 8.1 and Windows® 10 devices.

Requirements

The following are required to build and run the example application:

- A computer with a 6th generation Intel® Core™ processor (code-named Skylake)
- OpenGL 4.3 or higher
- Microsoft Visual Studio* 2013 or newer

Swap FBOs Instead of Using a Single FBO and Swapping Surfaces

This article deals with multipass rendering or rendering multiple targets. Many of today's visual effects in modern graphic games require multiple passes to render. For example, shadowing requires at least two passes: the first pass creates the shadow data (map or volume), and the second one uses that data in the scene. An additional pass is required if you want post-processing effects like screen space ambient occlusion. Add in effects like blurring and the number of passes continues to increase.

The data for each of these passes are supplied using FBOs. You can use a single FBO and just alter the data attachments as needed, but it is more efficient to create a separate FBO for each dataset and switch between FBOs. Switching an entire FBO is more efficient than switching individual attachments one at a time.

The example application displays an image rendered using both an FBO reused multiple times with different data and with separate FBOs. The current performance for each approach is displayed in a console window in milliseconds-per-frame and number of frames per second. Pressing the spacebar toggles between the two methods so you can compare the two approaches. When switching, the application animates the image as a visual indicator of the change.

6th Generation Intel® Core™ Processor Graphics

6th generation Intel Core processors provide superior two- and three-dimensional graphics performance, reaching up to 1152 GFLOPS. Its multicore architecture improves performance and increases the number of instructions per clock cycle.

6th generation Intel Core processors offer a number of all-new benefits over previous generations and provide significant boosts to overall computing horsepower and visual performance. Sample enhancements include a GPU that, coupled with the CPU's added computing muscle, provides up to 40 percent better graphics performance over prior Intel® Processor Graphics. 6th generation Intel Core processors have been redesigned to offer higher-fidelity visual output, higher-resolution video playback, and more seamless responsiveness for systems with lower power usage. With support for 4K video playback and extended overclocking, it is ideal for game developers.

GPU memory access includes atomic min, max, and compare-and-exchange for 32-bit floating point values in either shared local memory or global memory. The new architecture also offers a performance improvement for back-to-back atomics to the same address. Tiled resources include support for large, partially resident (sparse) textures, and buffers. Reading unmapped tiles returns zero, and writes to them are discarded. There are also new shader instructions for clamping LOD and obtaining operation status. There is now support for larger texture and buffer sizes. For example, you can use up to 128k × 128k × 8B mipmapped 2D textures.

Bindless resources increase the number of dynamic resources a shader may use, from about 256 to 2,000,000 when supported by the graphics API. This change reduces the overhead associated with updating binding tables and provides more flexibility to programmers.

Execution units (EUs) have improved native 16-bit floating-point support as well. This enhanced floating-point support leads to both power and performance benefits when using half precision.

Display features further offer multiplane overlay options with hardware support to scale, convert, color correct, and composite multiple surfaces at display time. Surfaces can additionally come from separate swap chains using different update frequencies and resolutions (for example, full-resolution GUI elements composited on top of up-scaled, lower-resolution frame renders) to provide significant enhancements.

Its architecture supports GPUs with up to three slices (providing 72 EUs). This architecture also offers increased power gating and clock domain flexibility.

Building and Running the Application

Follow these steps to compile and run the example application.

1. Download the ZIP file containing the source code for the example application, and then unpack it into a working directory.

2. Open the **lesson5_fboSwitching/lesson5.sln** file in Microsoft Visual Studio 2013.

3. Select <*Build*>/<*Build Solution*> to build the application.

4. Upon successful build you can run the example from within Visual Studio.

Once the application is running, a main window opens and you will see an image. The console window shows what method was used to render it and the current milliseconds-per-frame and number of frames-per-second. Pressing the spacebar toggles between the two methods and compares the performance difference. Pressing ESC exits the application.

Code Highlights

The application sets four different textures, first using a single FBO and then with separate FBOs. The textures are created in the initialization phase.

```
// configure texture unit
   glActiveTexture(GL_TEXTURE0);                                                   GLCHK;
   glUniform1i(texUnit, 0);                                                        GLCHK;

   // create and configure the textures
   glGenTextures(1, &texture);                                                     GLCHK;
   glBindTexture(GL_TEXTURE_2D, texture);                                          GLCHK;
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);                   GLCHK;
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);                   GLCHK;
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);              GLCHK;
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);              GLCHK;
```

The FBOs are also created during initialization.

```
   // create the fbos
   glGenFramebuffers(3, fbo);                                                      GLCHK;
   glGenRenderbuffers(2, rb);                                                      GLCHK;
   for (int i = 0; i < 2; ++i) {
       glBindRenderbuffer(GL_RENDERBUFFER, rb[i]);                                 GLCHK;
       glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, 640, 480);                 GLCHK;
   }
   glViewport(0, 0, 640, 480);                                                     GLCHK;
   glBindRenderbuffer(GL_RENDERBUFFER, 0);                                         GLCHK;
   for (int i = 0; i < 3; ++i) {
       glBindFramebuffer(GL_FRAMEBUFFER, fbo[i]);                                  GLCHK;
       glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                                 GL_RENDERBUFFER, rb[i % 2]);                      GLCHK;
       if (GL_FRAMEBUFFER_COMPLETE != glCheckFramebufferStatus(GL_FRAMEBUFFER))
           __debugbreak();
   }
```

Called once for each screen refresh, the `display()` method first checks whether we are switching between the two methods (that is, animating) and then whether to update the image with multiple FBOs or a single FBO. We arbitrarily use the last FBO when using a single FBO.

```
void display()
{
    static unsigned cnt; ++cnt;

    // attributeless rendering
    glClear(GL_COLOR_BUFFER_BIT);                                                 GLCHK;
    glBindTexture(GL_TEXTURE_2D, texture);                                        GLCHK;
    if (!animating) {
        glViewport(0, 0, 640, 480);                                               GLCHK;
        glUniform1f(offset, 0.f);                                                 GLCHK;
        if (options::FBO == options[selector].option) {
            glBindFramebuffer(GL_FRAMEBUFFER, fbo[cnt % 2]);                      GLCHK;
        } else if (options::SURFACE == options[selector].option) {
            glBindFramebuffer(GL_FRAMEBUFFER, fbo[2]);                       GLCHK;
            glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                                      GL_RENDERBUFFER, rb[cnt % 2]);         GLCHK;
        }
    } else {
        glViewport(0, 0, w, h);                                              GLCHK;
        glUniform1f(offset, animation);                                      GLCHK;
    }
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);                                   GLCHK;
    if (!animating) {
        glReadBuffer(GL_COLOR_ATTACHMENT0);                                  GLCHK;
        glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);                           GLCHK;
        glBlitFramebuffer(0, 0, 640, 480, 0, 0, w, h, GL_COLOR_BUFFER_BIT,
                          GL_LINEAR);                                        GLCHK;
        glBindFramebuffer(GL_FRAMEBUFFER, 0);                                GLCHK;
    }
    glutSwapBuffers();
}
```

Each time a video frame is drawn, the performance output is updated in the console and the application checks whether the spacebar or ESC has been pressed. Pressing the spacebar causes a switch from multiple to single FBO or back; pressing ESC exits the application. When switching, the performance measurements are reset and the image animates as a visual indicator that something changed. If no key was pressed, the next frame is rendered.

```
// GLUT idle function.  Called once per video frame.  Calculate and print timing
// reports and handle console input.
void idle()
{
    // Calculate performance
    static unsigned __int64 skip;  if (++skip < 512) return;
    static unsigned __int64 start; if (!start &&
!QueryPerformanceCounter((PLARGE_INTEGER)&start))              __debugbreak();
    unsigned __int64 now;  if (!QueryPerformanceCounter((PLARGE_INTEGER)&now))
                                  __debugbreak();
    unsigned __int64 us = elapsedUS(now, start), sec = us / 1000000;
    static unsigned __int64 animationStart;
    static unsigned __int64 cnt; ++cnt;

    // We're either animating
    if (animating)
    {
        float sec = elapsedUS(now, animationStart) / 1000000.f; if (sec < 1.f) {
            animation = (sec < 0.5f ? sec : 1.f - sec) / 0.5f;
        } else {
            animating = false;
            selector = (selector + 1) % options::nOPTS; skip = 0;
            cnt = start = 0;
            print();
        }
    }

    // Or measuring
    else if (sec >= 2)
    {
        printf("frames rendered = %I64u, uS = %I64u, fps = %f,
               milliseconds-per-frame = %f\n", cnt, us, cnt * 1000000. / us,
               us / (cnt * 1000.));
        if (swap) {
            animating = true; animationStart = now; swap = false;
        } else {
            cnt = start = 0;
        }
    }

    // Get input from the console too.
    HANDLE h = GetStdHandle(STD_INPUT_HANDLE); INPUT_RECORD r[128]; DWORD n;
    if (PeekConsoleInput(h, r, 128, &n) && n)
        if (ReadConsoleInput(h, r, n, &n))
            for (DWORD i = 0; i < n; ++i)
                if (r[i].EventType == KEY_EVENT && r[i].Event.KeyEvent.bKeyDown)
                    keyboard(r[i].Event.KeyEvent.uChar.AsciiChar, 0, 0);

    // Ask for another frame
    glutPostRedisplay();
}
```

Closing

Multipass rendering is required to obtain the desired effects expected in today's graphic games, but there are ways to minimize the effects of multiple passes. One key way is to create separate FBOs for each dataset and use the appropriate FBO depending upon the dataset, rather than reuse the same FBO over and over. Switching an entire FBO is more efficient than switching individual attachments one at a time.

By combining this technique with the advantages of the 6th generation Intel Core processors, graphic game developers can ensure their games perform the way they were designed.

References

An Overview of the 6th generation Intel® Core™ processor (code-named Skylake)

Graphics API Developer's Guide for 6th Generation Intel® Core™ Processors

About the Author

Praveen Kundurthy works in the Intel® Software and Services Group. He has a master's degree in Computer Engineering. His main interests are mobile technologies, Microsoft Windows, and game development.