OpenGL* Performance Tips: Use Native Formats for Best Rendering Performance

Introduction

Game developers often use OpenGL to handle the rendering chores for graphics-intensive games. OpenGL is an application programming interface for efficiently rendering two- and three-dimensional vector graphics. It is available on most platforms.

This article demonstrates how using the proper texture format can improve OpenGL performance—in particular, using native texture formats will give game developers the best OpenGL performance. Accompanying the article is a C++ example application that shows the effects on rendering performance when using a variety of texture formats. Note that while this article refers to concepts relevant to graphical game developers, these 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 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

Skylake Processor Graphics

The 6th generation Intel Core processors, also known as Skylake, 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.

The 6th gen Intel Core processors offer a number of 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. The 6th gen 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, Skylake 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 x 128k x 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 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, which are well worth taking advantage of.

Lesson 2: Use Native Texture Formats for Best Rendering Performance

Anyone who works with OpenGL is familiar with working with textures. However, not all textures are created equal—some texture formats have better rendering performance than other formats. Using formats that are native to the hardware means that the texture can be used "as is," avoiding an unnecessary conversion.

This lesson shows the impact of different formats—the example cycles through a variety of different texture formats as it renders an image in a window. For each format the current performance is displayed in milliseconds-per-frame, along with the number of frames-per-second. Pressing the spacebar rotates to the next texture in the list so you can see which formats work best on your hardware.

The example uses the following formats. This list is based on the list of OpenGL "Required Formats" at https://www.opengl.org/wiki/Image_Format:

- GL_RGBA8
- GL_RGBA16
- GL_RGBA16F
- GL_RGBA32F
- GL_RGBA8I
- GL_RGBA16I
- GL_RGBA32I
- GL_RGBA8UI
- GL_RGBA16UI
- GL_RGBA32UI
- GL_RGB10_A2
- GL_RGB10_A2UI
- GL_R11F_G11F_B10F
- GL_SRGB8_ALPHA8
- GL_RGB8
- GL_RGB16
- GL_RGBA8_SNORM
- GL_RGBA16_SNORM
- GL_RGB8_SNORM
- GL_RGB16_SNORM
- GL_RGB16F
- GL_RGB32F
- GL_RGB8I
- GL_RGB16I
- GL_RGB32I
- GL_RGB8UI
- GL_RGB16UI
- GL_RGB32UI
- GL_SRGB8
- GL_SGB9_ES

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 **lesson2a_textureformat/lesson2a.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 Microsoft Visual Studio 2013 console window displays the type of texture used to render the image and its performance measurements. Press the spacebar to change to the next texture format. Press ESC to exit the application.

Code Highlights

The code for this example is straightforward, but there are a few items to highlight.

First, we are going to need three different fragment shaders, based upon the texture being used. One shader will get its output color from a normalized texture, another is designed for non-normalized, signed integer textures, and the last is for non-normalized, unsigned integer textures.

```
// Fragment shader gets output color from a normalized texture
static std::string fragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform sampler2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    fragColor = texture(texUnit, texcoord);\n"
    "}\n"
;

// Fragment shader gets output color from a non-normalized signed integer texture
static std::string ifragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform isampler2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    fragColor = vec4(texture(texUnit, texcoord))/255.0;\n"
    "}\n"
;

// Fragment shader gets output color from a non-normalized unsigned integer texture
static std::string ufragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform usampler2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    fragColor = vec4(texture(texUnit, texcoord))/255.0;\n"
    "}\n"
;
```

These shaders are compiled and prepared.

```
    // compile and link the shaders into a program, make it active
    vShader  = compileShader(vertexShader, GL_VERTEX_SHADER);
    fShader  = compileShader(fragmentShader, GL_FRAGMENT_SHADER);
    ifShader = compileShader(ifragmentShader, GL_FRAGMENT_SHADER);
    ufShader = compileShader(ufragmentShader, GL_FRAGMENT_SHADER);
    program  = createProgram({ vShader, fShader });
    iprogram = createProgram({ vShader, ifShader });
    uprogram = createProgram({ vShader, ufShader });
```

The collection of textures is stored in an array that contains the texture and which program to use.

```
// Array of structures, one item for each option we're testing
#define F(x,y,z) x, y, #x, 0, z
static struct {
    GLint fmt, type;
    const char* str;
    GLuint obj, &pgm;
} textures[] = {
    F(GL_RGBA8,           GL_RGBA,         program),
    F(GL_RGBA16,          GL_RGBA,         program),
    F(GL_RGBA8_SNORM,     GL_RGBA,         program),
    F(GL_RGBA16_SNORM,    GL_RGBA,         program),
    F(GL_RGBA16F,         GL_RGBA,         program),
    F(GL_RGBA32F,         GL_RGBA,         program),
    F(GL_RGBA8I,          GL_RGBA_INTEGER, iprogram),
    F(GL_RGBA16I,         GL_RGBA_INTEGER, iprogram),
    F(GL_RGBA32I,         GL_RGBA_INTEGER, iprogram),
    F(GL_RGBA8UI,         GL_RGBA_INTEGER, uprogram),
    F(GL_RGBA16UI,        GL_RGBA_INTEGER, uprogram),
    F(GL_RGBA32UI,        GL_RGBA_INTEGER, uprogram),
    F(GL_RGB10_A2,        GL_RGBA,         program),
    F(GL_RGB10_A2UI,      GL_RGBA_INTEGER, uprogram),
    F(GL_R11F_G11F_B10F,  GL_RGBA,         program),
    F(GL_SRGB8_ALPHA8,    GL_RGBA,         program),
    F(GL_RGB8,            GL_RGBA,         program),
    F(GL_RGB16,           GL_RGBA,         program),
    F(GL_RGB8_SNORM,      GL_RGBA,         program),
    F(GL_RGB16_SNORM,     GL_RGBA,         program),
    F(GL_RGB16F,          GL_RGBA,         program),
    F(GL_RGB32F,          GL_RGBA,         program),
    F(GL_RGB8I,           GL_RGBA_INTEGER, iprogram),
    F(GL_RGB16I,          GL_RGBA_INTEGER, iprogram),
    F(GL_RGB32I,          GL_RGBA_INTEGER, iprogram),
    F(GL_RGB8UI,          GL_RGBA_INTEGER, uprogram),
    F(GL_RGB16UI,         GL_RGBA_INTEGER, uprogram),
    F(GL_RGB32UI,         GL_RGBA_INTEGER, uprogram),
    F(GL_SRGB8,           GL_RGBA,         program),
};
```

When it is time to draw the image, the correct texture and format is used based upon the texture array.

```
// GLUT display function.   Draw one frame's worth of imagery.
void display()
{
    // attribute-less rendering
    glUseProgram(textures[selector].pgm);                                   GLCHK;
    glClear(GL_COLOR_BUFFER_BIT);                                           GLCHK;
    glBindTexture(GL_TEXTURE_2D, textures[selector].obj);                   GLCHK;
    if (animating) {
        glUniform1f(offset, animation);                                     GLCHK;
    }
    else if (selector) {
        glUniform1f(offset, 0.f);                                           GLCHK;
    }
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);                                  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 is pressed. Pressing the space bar causes the application to move to the next texture in the array; pressing ESC exits the application. When a new texture is loaded 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) % _countof(textures); skip = 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 (advance) {
            animating = true; animationStart = now;  advance = 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();
}
```

Conclusion

This example demonstrated that using a texture format native to the GPU will increase performance. This simple application will help you determine the right texture for your Skylake hardware.

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

Download Code Sample

Below is the link to the code samples on Github
https://github.com/IntelSoftware/OpenGLBestPracticesfor6thGenIntelProcessor

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.

Notes

[1] March, Meghana R., “An Overview of the 6th generation Intel® Core™ processor (code-name Skylake).” March 23, 2016. https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake

Notices