OpenGL* Performance Tips: Textures Have Better Rendering Performance than Images

Introduction

This article discusses why using a texture rather than an image can improve OpenGL rendering performance. It is accompanied by a simple C++ application that alternates between using a texture and using an image. The purpose of this application is to show the effect on rendering performance (milliseconds per frame) when using the two techniques. 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 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

Textures Have Better Rendering Performance than Images

Use a texture rather than an image to get the best rendering performance out of OpenGL. The accompanying application demonstrates this by alternating between using a texture and a 2D image. The current performance for each (displayed in milliseconds per frame) displays in the console window, along with the number of frames per second. Pressing the spacebar rotates through the various combinations so you can compare them. When a switch is made the image animates, visually signaling the switch.

The following combinations are used when measuring textures:

- GL_TEXTURE_MAX_LEVEL
- GL_TEXTURE_BASE_LEVEL
- GL_TEXTURE_MAG_FILTER
- GL_TEXTURE_MIN_FILTER

To keep things fair when using an image, the application calculates the closest-sized image in the texture to what the texture sampling hardware is doing in the texture's mipmap chain. It then uses an image of this size. You can see the calculations in the `reshape()` function. This means that the larger the on-screen image is, the larger the image the application uses in the mipmap chain. The reverse is also true for smaller images on the screen.

Skylake Processor Graphics

The 6th generation Intel Core processors provide superior 2D and 3D graphics performance, reaching up to 1152 GFLOPS. Its multicore architecture improves performance and increases the number of instructions per clock cycle.

These 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. 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, 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 roughly 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.

The display features further offer multiplane overlay options with hardware support to scale, convert, color correct, and compose 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.

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 unpack it into a working directory.

2. Open the **lesson3_textureVsImage/lesson3.sln** file by double-clicking it to start 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 rendered using a texture or image combination, with the performance measurements in the Microsoft Visual Studio 2013 console window. Press the spacebar to switch to the next mode and see the performance differences. When switching modes the image animates to visually indicate a change. Pressing ESC exits the application.

Code Highlights

This example uses either a texture or an image. Here are the texture and image fragment shaders:

```
// Fragment shader gets output color from texture()
static std::string texFragmentShader =
    "#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 imageLoad()
static std::string imgFragmentShader =
    "#version 430 core\n"
    "\n"
    "readonly restrict layout(rgba8) uniform image2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    fragColor = imageLoad(texUnit, ivec2(texcoord * imageSize(texUnit)));\n"
    "}\n"
;
```

The combinations the application will test are stored in an array.

```
// Array of structures, one item for each option we're testing
#define I(texture, magFilter, minFilter, maxLevel, baseLevel) texture, #texture,
magFilter, #magFilter, minFilter, #minFilter, maxLevel, baseLevel
struct {
    GLuint& texture;   const char* textureStr;
    GLint   magFilter; const char* magFilterStr;
    GLint   minFilter; const char* minFilterStr;
    GLint   maxLevel;
    GLint   baseLevel;
} options[] = {
    { I(magTexture, GL_NEAREST, GL_NEAREST,                0,        0 )},
    { I(magTexture, GL_LINEAR,  GL_NEAREST,                0,        0 )},
    { I(minTexture, GL_NEAREST, GL_NEAREST,                0,        0 )},
    { I(minTexture, GL_NEAREST, GL_LINEAR,                 0,        0 )},
    { I(minTexture, GL_NEAREST, GL_NEAREST_MIPMAP_NEAREST, mipLevel, 0 )},
    { I(minTexture, GL_NEAREST, GL_NEAREST_MIPMAP_LINEAR,  mipLevel, 0 )},
```

```
    { I(minTexture, GL_NEAREST, GL_LINEAR_MIPMAP_NEAREST,  mipLevel, 0 )},
    { I(minTexture, GL_NEAREST, GL_LINEAR_MIPMAP_LINEAR,   mipLevel, 0 )},
};
```

The options used are based upon this array when it is time to draw the image. You can see how the texture is compared using GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAG_FILTER, and GL_TEXTURE_MIN_FILTER.

```
// GLUT display function.   Draw one frame's worth of imagery.
void display()
{
    // attribute-less rendering
    glClear(GL_COLOR_BUFFER_BIT);                                                   GLCHK;
    if (animating) {
        glUseProgram(texProgram);                                                   GLCHK;
        glUniform1f(texOffset, animation);                                          GLCHK;
    } else if (mode) {
        glUseProgram(texProgram);                                                   GLCHK;
        glUniform1f(texOffset, 0);                                                  GLCHK;
        glBindTexture(GL_TEXTURE_2D, options[selector].texture);                    GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, options[selector].maxLevel);
                        GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL,
                         options[selector].baseLevel);                              GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                        options[selector].magFilter);                              GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                         options[selector].minFilter);                              GLCHK;
    } else {
        glUseProgram(imgProgram);                                                   GLCHK;
        glUniform1f(imgOffset, 0);                                                  GLCHK;
        glBindImageTexture(0, minTexture, selector < 2 ? 7 : imgLevel, GL_FALSE, 0,
                           GL_READ_ONLY, GL_RGBA8);                                 GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);      GLCHK;
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
                        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 spacebar causes the application to move to the next set of combinations in the array; pressing ESC exits the application. When a new combination 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(options); skip = 0;
            cnt = start = 0;
            mode ^= 1;
            print();
            printf("%s: ", (mode ? "texture()" : "image2d()")); fflush(stdout);
        }
    }

    // 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;
            mode ^= 1;
            printf("%s: ", (mode ? "texture()" : "image2d()")); fflush(stdout);
        }
    }

    // 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 has performance advantages over using an image when rendering. This is important to graphical game developers, who strive to get the best performance possible for their games.

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.

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.