OpenGL* Performance Tips: Avoid OpenGL Calls that Synchronize CPU and GPU

Introduction

To get the highest level of performance from OpenGL* you want to avoid calls that force synchronization between the CPU and the GPU. This article covers several of those calls and describes ways to avoid using them. It is accompanied by a C++ example application that shows the effect of some of these calls on rendering performance. While this article refers to graphical game development, 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

Avoid OpenGL Calls that Synchronize CPU and GPU

OpenGL contains a variety of calls that force synchronization between the CPU and the GPU. These are called Sync Objects and are designed to synchronize the activity between the GPU and the application. Unfortunately this hurts overall performance because the CPU stalls until the GPU has completed its action. If possible, avoid these calls.

The OpenGL Foundation’s website describes Sync Objects at https://www.opengl.org/wiki/Sync_Object, but here is a summary of ways to avoid this issue:

- Avoid glReadPixels() or glFinish(), which force synchronization between the CPU and GPU. If you need to use glReadPixels() do so in conjunction with Pixel Buffer Objects.
- Use glFlush() with caution; if you must synchronize between contexts use Sync Objects instead.
- Avoid updating resources that are used by the GPU. It is better to create static resources when the application starts and not have to modify them later. Whenever possible, create vertex buffer objects as static (GL_STATIC_DRAW).
- Avoid updating resources that are used by the GPU. For example, do not call glBufferSubData/glTexImage if there are queued commands that access a given VBO/texture. Limit the chances of simultaneous read/write access to resources.
- Use immutable versions of API calls to create buffers and textures. For example, use API calls like glBufferStore() and glTexStorage*().


- Update buffers and avoid GPU/CPU synchronization issues by creating a pool of bigger buffers with `glBufferStorage()` and permanently map them with the `glMapBuffer()` function call. The application can then iterate over individual buffers with increasing offsets, providing new chunks of data.

- Use `glBindBufferRange()` for uniform buffer objects to bind new chunks of data at the current offset. For vertex buffer objects access newly copied chunks of data with `firstIndex` (for `glDrawArrays`) or `indices/baseVertex` parameters (for `glDrawElements/BaseVertex`). Increase the initial number of pools if the oldest buffer submitted for GPU consumption is still in use. Monitor the progress of the GPU by accessing the data from the buffers with `Sync Objects`.

The example application demonstrates the effects of three different OpenGL calls that cause the CPU and GPU to synchronize. The calls are `glReadPixels`, `glFlush`, and `glFinish`. These calls are compared to a non-synchronized performance. 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 cycles between the methods so you can compare the effects. When switching, the application animates the image as a visual indicator of the change.

**Intel Skylake 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.

The 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 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 (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, creating a powerful game delivery system.
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 lesson6_gpuCpuSynchronization/lesson6.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 open 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 cycles between the methods and compares the performance difference. Pressing ESC exits the application.

Code Highlights

The application uses three calls to force synchronization, as well as the unsynchronized approach. The various combinations are stored in an array that is created during the initialization phase.

```c
// Array of structures, one item for each option we're testing
#define I(x) { options:: ## x, #x }
struct options {
    enum { NONE, READPIXELS, FLUSH, FINISH, nOPTS } option;
    const char* optionStr;
} options[] {
    I(NONE),
    I(READPIXELS),
    I(FLUSH),
    I(FINISH),
};
```

To test this, the application creates a vertex and fragment shader, plus loads textures into VRAM.

```c
// compile and link the shaders into a program, make it active
vShader = compileShader(vertexShader, GL_VERTEX_SHADER);
fShader = compileShader(fragmentShader, GL_FRAGMENT_SHADER);
program = createProgram({ vShader, fShader });
offset = glGetUniformLocation(program, "offset");
texUnit = glGetUniformLocation(program, "texUnit");
gUseProgram(program);

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

// create and configure the textures
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
```
Called once for each screen refresh, the display() method checks first whether we are switching between the methods (that is, animating). If not switching, it then uses the option pointed to in the array of options. Switching from method to method walks through this array.

```cpp
void display()
{
    // attributeless rendering
    glClear(GL_COLOR_BUFFER_BIT);    GLCHK;
glBindTexture(GL_TEXTURE_2D, texture);    GLCHK;
if (animating) {
    glGenerateMipmap(GL_TEXTURE_2D);    GLCHK;
    glUniform1f(offset, animation);    GLCHK;
} else {
    glUniform1f(offset, 0.f);    GLCHK;
}
glDrawArrays(GL_TRIANGLES, 0, 3);
if (!animating)
switch (options[selector].option) {
    case options::NONE:          break;
case options::READPIXELS:    glReadPixels(0, 0, w, h, GL_RGBA, GL_UNSIGNED_BYTE, &img[0]);    GLCHK;          break;
case options::FLUSH:          glFlush();    GLCHK;          break;
case options::FINISH:         glFinish();    GLCHK;          break;
}
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 the application to move through the non-synchronizing and synchronizing calls; 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.

```cpp
// GLUT idle function. Called once per video frame. Calculate and print timing
// reports and handle console input.
void idle()
```
Closing

Depending upon the game you are developing, it may not be possible to avoid calls that cause synchronization between the CPU and the GPU, especially if your application needs to interact with the pixels on the screen in some fashion or synchronize between different contexts. In general, it is best to
avoid synchronization to get the most performance out of your system. This article has covered some of the calls that cause synchronization and suggested alternative approaches.

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.

Notices
Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel, the Intel logo, and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2016 Intel Corporation