

6Score: _____ Section: _____ Date: _____ Name: _____

ECE 2036 Laboratory Assignment 6

Due Date: Section B – Tuesday Dec 2, C – Wednesday Dec 3

Final day for all TA in lab demos is Dec 4!

A [thread](#) is just a sequence of code ready to execute. In C++, this can be thought of as just a special function to run in parallel with the main program. Only programs with multiple threads can run code faster using multiple processor cores working in parallel. The default C++ program runs on only one processor core even on multicore processors. Unfortunately, threading ideas and multicore processors are newer than C++. This means that they are not supported in the standard C++ language features. Some newer languages such as C# and Java have such support. There are however, OS API calls that enable C++ to support multiple threads. Since they are OS specific, such code only runs on one OS and compiler. This lab will use Windows and Visual Studio. Students with MACs can run a Windows partition on their MAC or use one of the three ECE PC labs. Thread programs could be converted to run on MACs by changing these special APIs to the ones for a MAC and/or other compilers, but it would take several extra hours or days of work for this lab. Linux also has a special C++ thread library called [Pthreads](#) with similar APIs. *Pthreads* can run on several common OSes, but a special library may need to be installed or linked.

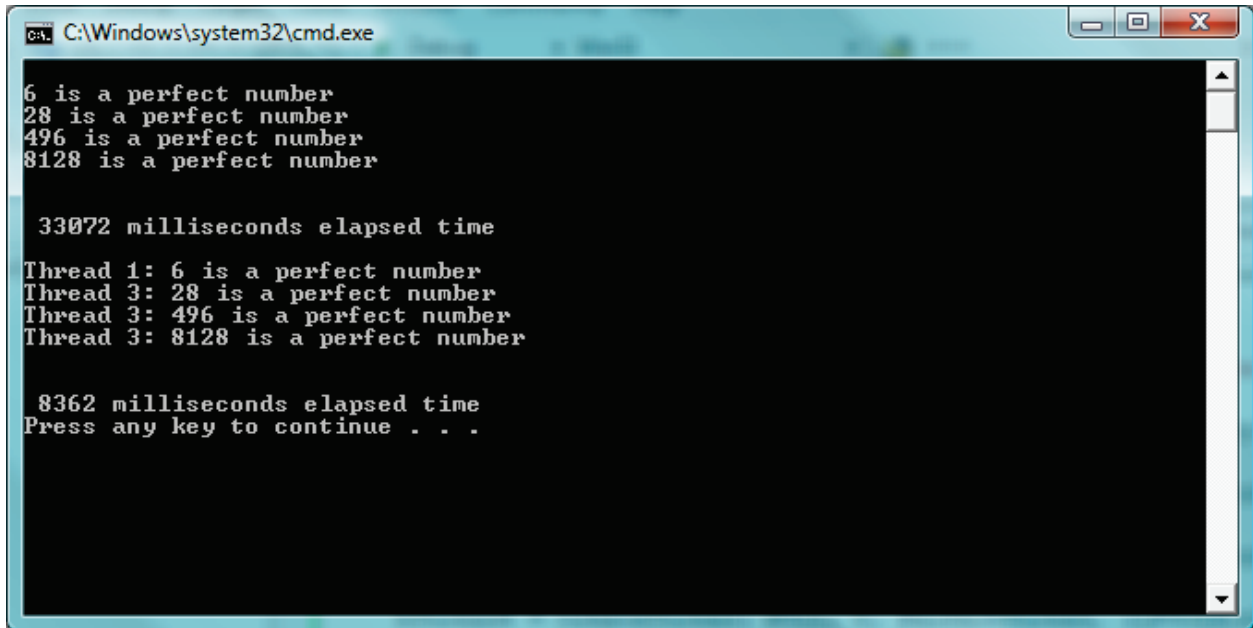
In this lab, you will use C/C++ and the Windows OS to compare the performance of a program with and without threads. The program computes the Mandelbrot set. Read <http://warp.povusers.org/Mandelbrot/> for background and an explanation of the C++ code used for the fractal computations. Two versions will be timed and compared. One will use a standard application without any thread creation. The second version will use four created threads each computing the color for different pixels. The main application will also need to wait for all of the threads to complete using an OS synchronization primitive called a semaphore and read a timer to determine execution time. Be sure to use the same algorithm on both the threaded and non-threaded version so that the timing and multicore speedup comparison (i.e. threads vs. no threads) makes more sense.

On processors with multiple cores, such as the newest PCs in the ECE Klaus PC labs that have two cores you should be able to demonstrate close to a linear speedup, since this CPU bound problem has the ideal properties needed. So you could see close to a 2X speedup on a dual core processor using two threads. It can be split up evenly among processors, does very minimal I/O, and only has minimal shared data. The processor cores do share the same memory. The point here is to balance the computational load for maximum speedup. Be sure to shutdown other applications when taking timing measurements as other applications running and even the cache will impact timing. On the ECE virtual lab servers (and not a local PC) this might not be possible and timing may be a bit more random depending on what else the servers are doing. The OS is always running other tasks and threads, so timing can vary quite a bit each time it runs. If you have access to a four core processor, you could see a speedup of almost 4X. With an i7 processor with 8 threads you should see some additional speedup beyond 4 due to [hyperthreading](#) (two different threads can run on one core on the i7 and reduce pipeline stalls a bit, but not 2X faster). There are even some new processors used in high-end servers with 16 cores. Servers can limit the number of cores per user.

A very similar example program that searches for perfect numbers is attached along with additional information on the various synchronization APIs available in the Windows OS. The example program shows how to create threads, time execution, and use semaphores. A source code file is for the example available at <http://users.ece.gatech.edu/~hamblen/3055/course/perfect.cpp> or you could cut and paste the code that follows. In VS create a "WIN32 Console Application" when creating the project, so that *printf* can be used in a basic text or console window (i.e., no complex GUI is needed). If you have or move to a newer version of VS you may need to convert the project files to the new version, but the C source code will be the same. This example uses [semaphores](#), but in the lab assignment [events](#) must be used instead of semaphores. A summary of the other OS

synchronization API primitives including events in Windows is provided on the last page of this assignment. Each thread sends (sets) an event to indicate that it has finished its computations in the lab assignment code.

Here is a C example with and without threads that searches for Perfect numbers. According to the ancient Greeks, a perfect number is an integer equal to the sum of its integer divisors with no remainders. 6 for example is evenly divisible by 3,2, and 1, and $1+2+3=6$. There are not many perfect numbers.

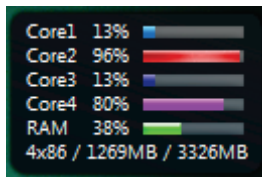


```
C:\Windows\system32\cmd.exe
6 is a perfect number
28 is a perfect number
496 is a perfect number
8128 is a perfect number

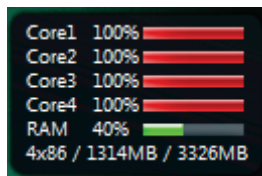
33072 milliseconds elapsed time
Thread 1: 6 is a perfect number
Thread 3: 28 is a perfect number
Thread 3: 496 is a perfect number
Thread 3: 8128 is a perfect number

8362 milliseconds elapsed time
Press any key to continue . . .
```

Timed results above are from one thread version and then four threads searching for perfect numbers. Note that a near linear speedup of nearly 4X is obtained using four threads on a four core processor.



Here is a display of how busy the cores are in Windows on a Quad core processor. Without threads, only one core is totally busy cranking through numbers. Some other OS tasks appear to be running some of the time on the other cores.



When running four threads, all cores are busy on a Quad Core Processor.

The code for this example follows. It shows how to create and start running threads using *CreateThread* and synchronize them using the Windows OS APIs and semaphore synchronization primitives. Each thread signals that its computations are complete by releasing the semaphores. When all threads have released it, the computations have completed. The *GetTickCount* API reads a millisecond timer and this value used to compute the code execution time. *Sleep(x)* delays *x* milliseconds. Note that this execution time can run a couple of minutes!!!

```

// Perfect.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include "Windows.h"
#include "LIMITS.h"
static HANDLE Thread_semaphore;
int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE hThread1;
    DWORD dwThread1ID = 0;
    HANDLE hThread2;
    DWORD dwThread2ID = 0;
    HANDLE hThread3;
    DWORD dwThread3ID = 0;
    HANDLE hThread4;
    DWORD dwThread4ID = 0;
    INT nParameter = 1;
    DWORD WINAPI PerfectThread (LPVOID lpArg);
    DWORD time_count;
    unsigned long i, j, sum;
    printf("\n");
    time_count = GetTickCount();
    // count primes using only one thread
    for (j=2; j<=100000; j++)
    {
        sum = 0;
        for (i=1; i<j; i++)
        {
            if ((j % i) == 0) sum = sum + i;
        }
        if (sum == j) printf("%d is a perfect number\n", j);
    }
    time_count = GetTickCount() - time_count;
    printf("\n\n %d milliseconds elapsed time\n\n\r", time_count);
    // now do it with four threads
    // setup a semaphore for synchronization of threads with an initial value of 0 (waits on 0)
    Thread_semaphore = CreateSemaphore(NULL, 0, 4, TEXT("Thread_Done"));
    time_count = GetTickCount();
    hThread1 = CreateThread( NULL, 0, PerfectThread, (LPVOID)nParameter, 0, &dwThread1ID);
    nParameter++;
    hThread2 = CreateThread( NULL, 0, PerfectThread, (LPVOID)nParameter, 0, &dwThread2ID);
    nParameter++;
    hThread3 = CreateThread( NULL, 0, PerfectThread, (LPVOID)nParameter, 0, &dwThread3ID);
    nParameter++;
    hThread4 = CreateThread( NULL, 0, PerfectThread, (LPVOID)nParameter, 0, &dwThread4ID);
    // waits for all four threads to signal done with release semaphore
    WaitForSingleObject(Thread_semaphore, INFINITE);
    WaitForSingleObject(Thread_semaphore, INFINITE);
    WaitForSingleObject(Thread_semaphore, INFINITE);
    WaitForSingleObject(Thread_semaphore, INFINITE);
    time_count = GetTickCount() - time_count;
    printf("\n\n %d milliseconds elapsed time\n\n\r", time_count);
    CloseHandle(hThread1);
    CloseHandle(hThread2);
    CloseHandle(hThread3);
    CloseHandle(hThread4);
    return 0;
}

// code for each of the threads to execute
DWORD WINAPI PerfectThread (LPVOID lpArg) {
INT threadnumber = (INT) lpArg;
unsigned long i, j, sum;
for (j=1 + threadnumber; j<=100000; j=j+4)
{
    sum = 0;
    for (i=1; i<j; i++)
    {
        if ((j % i) == 0) sum = sum + i;
    }
    if (sum == j) printf("Thread %d: %d is a perfect number\n\n\r", threadnumber, j);
}
ReleaseSemaphore(Thread_semaphore, 1, NULL);
return 0;
}

```

(70%) Part I **Add Threads** - A code example is provided below which computes the Mandelbrot set and stores the result in a 2-D array, *Image*, that contains data used for pixel colors (color is based on the number of iterations in the algorithm). This code is based on the code found at the website <http://warp.povusers.org/Mandelbrot/>, but it is modified to store the number of iterations for each pixel in the 2D array and the maximum number of iterations has been increased for more colors.

```
//image size in pixels and array setup
const unsigned ImageHeight=600;
const unsigned ImageWidth=800;
int Image[ImageWidth][ImageHeight];

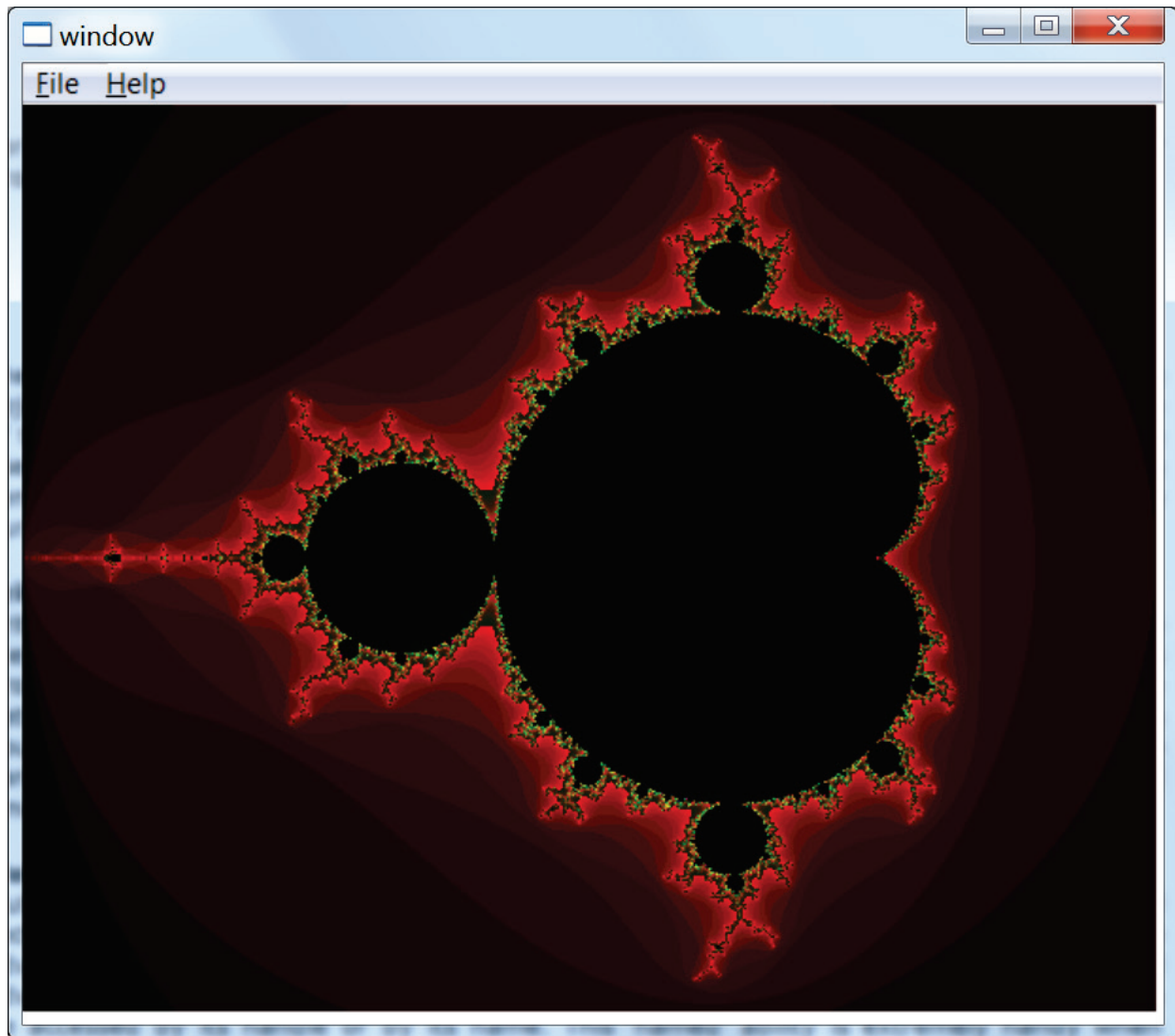
void Mandelbrot(){
double MinRe = -2.0;
double MaxRe = 1.0;
double MinIm = -1.2;
double MaxIm = MinIm+(MaxRe-MinRe)*ImageHeight/ImageWidth;
double Re_factor = (MaxRe-MinRe)/(ImageWidth-1);
double Im_factor = (MaxIm-MinIm)/(ImageHeight-1);
unsigned MaxIterations = 4096;

for(unsigned y=0; y<ImageHeight; ++y)
{
double c_im = MaxIm - y*Im_factor;
for(unsigned x=0; x<ImageWidth; ++x)
{
double c_re = MinRe + x*Re_factor;
double Z_re = c_re, Z_im = c_im;
bool isInside = true;
int niterations=0;
for(unsigned n=0; n<MaxIterations; ++n)
{
double Z_re2 = Z_re*Z_re, Z_im2 = Z_im*Z_im;
if(Z_re2 + Z_im2 > 4)
{
isInside = false;
niterations = n;
break;
}
Z_im = 2*Z_re*Z_im + c_im;
Z_re = Z_re2 - Z_im2 + c_re;
}
Image[x][y] = niterations;
}
}
}
```

Replace the perfect function from the earlier code example with this new Mandelbrot code, run this program and note the execution time printed out for a single thread (i.e., the main program). Using thread and synchronization ideas from the previous example with four threads convert the Mandelbrot calculations to an application with four threads (each thread runs the Mandelbrot function for different pixels). The main change will be in the function at the end that runs on each thread. Note that each thread gets a different thread number and *threadnumber* is used so that each thread computes a different result (using different data – in this case different pixels). This can be handled with a simple change (very similar to the perfect example) in the first “for” loop in the Mandelbrot C++ code. Note that the *CreateThread* API starts running a new thread and that semaphore APIs (*ReleaseSemaphore*

and *WaitForSingleObject*) are needed to make it wait for all four threads to finish. Don't leave out *ReleaseSemaphore* or it will wait forever.

(20%) For part II **Add Graphics in a GUI** - The MandelBrot set will be displayed in a Windows GUI application. Assuming your program works, it should look like the following image. The Window below was manually resized to clear out some white space for a nicer picture.



MandelBrot Set Pseudocolor Image

The project for Part II must be initially setup in VS as a *WIN32 Application (GUI)* and not a console application. VS generates hello world GUI code in *myproject.cpp* with a default window that can contain graphics (unlike the earlier text only WIN32 console application). A basic hello world window is around 4 pages of C++ code. This example could also be setup to run faster using multiple threads similar to part I, but with the limited time we have for this lab only one thread (i.e., the main program) will be used to keep it simple. A Windows application paints the color image in a *WM_Paint* case in the windows callback function (near the end of the *myproject.cpp* source code). It is called when the window first starts up and whenever an I/O event (mouse or keyboard action) occurs inside the window area. A Windows OS API, *SetPixel*, can be used to color individual pixels. So the Mandelbrot

code needs to compute the pixel colors in the 2D array first, and then draw the color image in a 2D For loop with *SetPixel* calls. Something like the code below will need to be added inside the paint case in the windows callback function:

```
case WM_PAINT:
    Mandelbrot();
    // Compute data for image pixel color array
    hdc = BeginPaint(hWnd, &ps);
    // now Paint the computed image in the Window
    for(unsigned y=0; y<ImageHeight; ++y)
    {
        for(unsigned x=0; x<ImageWidth; ++x)
        {
            int color=Image[x][y];
//color each pixel using a pseudocolor based on the number of iterations
            SetPixel(hdc, x, y, RGB((color & 0xF)*16,
                ((color >>4) & 0xF)*16, ((color >>8) & 0x0F)*16));
        }
    }
    EndPaint(hWnd, &ps);EndPaint(hWnd, &ps);
    break;
```

Note that the code above uses a function call to run the Mandelbrot calculations to fill the 2D array. Place the new variable declarations and the *Mandelbrot* function code (that you will need to add – the single thread code in part I based on code from the website at <http://warp.povusers.org/Mandelbrot/>), just before the callback function (near end of code). It is a bit ugly to pass and return 2D arrays using functions in C++, so just make the 2D array a variable that is global and make the Image size a constant something like this from the earlier code:

```
//image size in pixels and image array setup
const unsigned ImageHeight=600;
const unsigned ImageWidth=800;
int Image[ImageWidth][ImageHeight];
```

(10%) **Add User Input** – A left mouse button click will be used to provide user input to zoom into a region closer and redraw the image. A Windows callback function is called when the window first starts up and whenever an I/O event (mouse or keyboard action) occurs inside the window area. I/O works a bit differently in a Windows application. Only the OS can control which Window is active and know exactly which Window that an I/O event belongs to. So unlike code with “*cin*” and “*cout*” where an application just requests I/O directly from a device, in Window GUIs the OS tells the Windows application program that an I/O event has occurred for this Window by calling the Windows callback function (each application has to have one, look for LRESULT CALLBACK in code to find it). The callback function has a switch statement to decide (decode) which message has been sent.

In the case of a mouse left key click, the message sent to the callback function is WM_LBUTTONDOWN. So to handle a mouse left click event, a new case must be added to the outer switch statement in the callback function that checks for the different messages to handle. The code for each message in the case statement responds to the decoded message, in this case it needs to redraw a zoomed in image of the MandelBrot set using updated max and min view region values in complex plane for the MandelBrot function. WM_LBUTTONDOWN is already defined as a constant in a Windows include file, so it can just be used in the case statement code to be added.

The code in the MandelBrot function calculation needs to zoom in on a center point in the complex plane. The center point is a constant and the region around it gets smaller each time the left mouse button is clicked resulting in the image being redrawn. So the MandelBrot function code needs to be modified to zoom in around this center point (it recomputes the Max and Min complex plane values to zoom in before the iteration loop starts that determines each pixels color). A zoom factor of around 2.0, 3.0, or 4.0 seems appropriate for each mouse click.

There are a couple options on exactly how to implement the zoom feature in this lab. A “static” variable could be used in the MandelBrot function to remember and change the zoom setting (for new Max and Min region calculations each time using zoom factor) and/or new arguments could be passed to the MandelBrot function. Start out with an initial display view of something near -2.0 to +2.0 for both the real and complex values. This displays the typical picture shown of the bounds of the regions of interest in the MandelBrot set.

The code in the WM_LBUTTONDOWN needs to recompute the MandelBrot set and display it using setpixel calls just like the earlier case setup for WM_PAINT that displayed the initial difference. Most of the code needed can be “cut” and “pasted” into this new case, but there is one important change needed in addition to the zoom calculations. Only WM_PAINT uses the *BeginPaint* and *EndPaint* calls. Replace the *BeginPaint* line with “*hdc = GetDC(hWnd);*”. Replace the *EndPaint* line with “*ReleaseDC(hWnd, hdc);*”. These functions are required by Windows to prevent more than one thread accessing the drawing object at the same time (and messing things up). This solves another synchronization issue with multiple processes and threads by using a special API call just for this purpose.

There are several places in the MandelBrot set (near a black edge region) where it is possible to zoom in by hundreds of orders of magnitude and continue to obtain complex images that are interesting. Here is an interesting zoom center point to use in the lab (all digits are needed!):

$$0.28693186889504513 + 0.014286693904085048 i$$

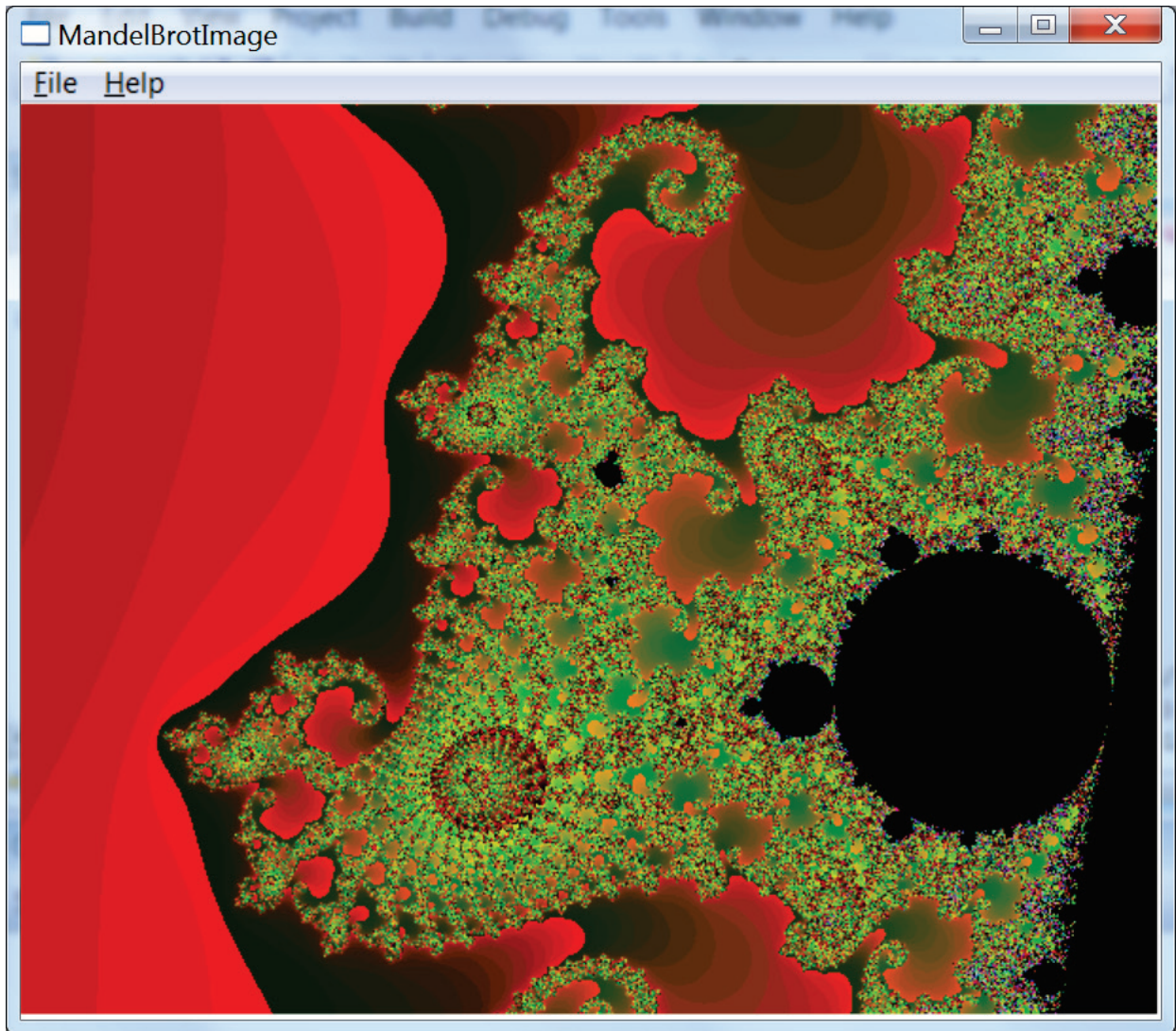
At this center point, it should be possible to zoom in a dozen or more times before the image turns one color. This happens because even with double precision floating point the calculations and constants are not accurate enough after a few orders of magnitude of zooming in on the critical center point. Zooming further in requires even more accuracy in floating point numbers and calculations than the 14 or so decimal places a C++ double provides. There are some special packages that can do this, but the calculations really start to slow down and even more iterations may be needed for each pixel. Some zooming videos on the web have used several months of computer time.

It takes several seconds to do the computations for each left mouse button click. Large black regions take more computation time since those pixels hit the maximum number of iterations in the innermost loop of the calculations. So extreme patience is needed on mouse clicks, and they need to be done slowly while watching the image. Windows is busy with calculations and does not always buffer up all of mouse clicks while the image is being redrawn. There are ways that this issue could be fixed, but they are a bit too complex for a simple class lab assignment.

(2.5%) **Extra Credit** Add code so that a right mouse button-click zooms back out in addition to the left click zooming in.

Ideas for Further Work

Very complex images can be found on the boundaries of the Mandelbrot set. There are a lot of [other interesting areas](#) to explore in the Mandelbrot set and by changing the initial values of the complex constants, the program can plot them. Several are listed in table 1. With a bit more time and effort, it would be possible to add more features to allow the user to move around and zoom into these areas using the keyboard or mouse. A new C++ complex class could also be used with operating overloading for complex numbers, but it would likely run just a bit slower and is probably overkill for this simple example with only one equation using complex variables. Global variables shared between threads that are both read and written must have a mutual exclusion lock using synchronization primitives for correct operation. Fortunately, these somewhat simple calculations for the lab do not need such variables. Code that reads or modifies shared global variables is called a critical section. There are also faster ways to draw images than *SetPixel*, but they are a bit complicated.



Mandelbrot set for $\text{MinRe} = -0.75104$, $\text{MaxRe} = -0.7408$, and $\text{MinIm} = 0.10511$.

Table 1: Other Interesting Areas to Examine

| MinRe | MaxRe | MinIm |
|------------|------------|-----------|
| -0.19920 | -0.12954 | 1.10148 |
| -0.95 | -0.88333 | 0.23333 |
| -0.713 | -0.4082 | 0.49216 |
| -1.781 | -1.764 | 0.0 |
| -0.74758 | -0.74624 | 0.10671 |
| -0.74591 | -0.74448 | 0.11196 |
| -0.745538 | -0.745054 | 0.112881 |
| -0.7454356 | -0.7454215 | 0.1130037 |
| -0.7454301 | -0.7454289 | 0.1130076 |

Windows Synchronization APIs

The Windows OS offers several other synchronization primitives, sometimes called waitable objects. These are mutexes, semaphores, events, and critical sections. With the exception of the faster user mode critical section, the others are all kernel objects and each can be used to synchronize both processes and threads. The previous code example contained a semaphore. Semaphores can be used on both processes and threads.

Critical sections

Critical sections are user mode synchronization objects provided by the system. Because they operate in user mode, they are fast. Unfortunately, user-mode objects can't cross process boundaries, so critical sections won't work for synchronization tasks that run between different processes. Although critical sections can't be used across process boundaries, they are very useful for in-process synchronization needs and are able to handle most simple synchronization tasks. Critical sections are handy to protect data shared among threads.

Mutexes

Mutexes are kernel mode synchronization objects. As such, they are slower than critical sections because it takes time to switch from user mode to kernel mode, but they have the advantage that they can operate across process boundaries. Mutexes offer an additional advantage over Critical sections in that they can be 'named objects' whereby the mutex can be accessed by its handle or by its name. This 'named' ability is extremely handy when using the same mutex from different processes (and not threads only). An alternate to using a named mutex is to create the mutex in one process and then use the `DuplicateHandle` function to allow its use in another process. This approach can incur additional overhead because it requires the handle value and process id from the first process to be passed to the second process.

Events

Events are not really used to protect shared data per se, but are used to signal when an action has occurred. For example, if T1 changes some data, it is useful for T1 to signal T2 when the data has changed. Because of events' signaling ability, they are very useful. A common mistake with developers new to multithreaded programming is to use a 'time based' operation rather than event to wait. Use `CreateEvent` to initialize and `SetEvent` to signal an event. These events are frequently used with the `WaitFor` family of functions: `WaitForSingleObject`, `WaitForMultipleObjects`, and `MsgWaitForMultipleObjects`. Events are very important when performing multithreaded programming.

Table 1: MS Windows OS Critical Section, Mutex, and Semaphore Functions

| Operation | Synchronization Object APIs | | | |
|------------------|----------------------------------------|-------------------------------------------------------------------------|-------------------------------|--------------------------|
| | Critical Section | Mutex | Semaphore | Event |
| Initialize | <code>InitializeCriticalSection</code> | <code>CreateMutex</code> | <code>CreateSemaphore</code> | <code>CreateEvent</code> |
| Lock or Wait | <code>EnterCriticalSection</code> | <code>WaitForSingleObject</code> or <code>WaitForMultipleObjects</code> | | |
| Unlock or Signal | <code>LeaveCriticalSection</code> | <code>ReleaseMutex</code> | <code>ReleaseSemaphore</code> | <code>SetEvent</code> |
| Close | <code>DeleteCriticalSection</code> | <code>CloseHandle</code> | | |

For more information on these APIs, their arguments, and example code, search using the function name in the VS on-line help or search for "synchronization functions".