COMP 8551 - Assignment 2

1. [50 marks] Start with the sample code given in class for x86/x64 assembly language. In one of them, the windows application reads in a colour image and a “kernel” image. The kernel image is “blended” with the original image according to the following:
   ```
   pBlendImg[i][j] = pOrigImg[i][j] * blendFac + pKernelImg[i][j] * (1 - blendFac);
   ```

   where pOrigImg is the original image, pKernelImg is the kernel image, pBlendImg is the resulting image, and blendFac is the blending factor. Describe how the blitBlend function works for each of the three simMode’s. Include descriptions of how each line of code with the intrinsics and asm instructions works.

   Which mode runs the fastest and second fastest, and why?

   The SIMD_EMMX mode, written almost entirely in assembly, is fastest. This mode Blends images 16 pixels at a time, giving it an advantage over the serial mode. It does this blending by iterating over the source and destination images, and unpacking their color values for R, G, and B into separate registers. Then, it does the blending formula in the document, using the alpha channel of the source image as the blending factor to simulate transparency. To finally bring it all together, it packs the color values from the source and destination images back into each other, and since they were interleaved with 0s, their color data for the 16 pixels each loop can be combined and drawn onto the destination image.

   The SIMD_NONE serial mode is the slowest. It also follows the same formula as the other modes, but uses regular C++, and importantly, only iterates across the source image one pixel at a time. As such, it is noticeably slower than the other two modes.

   The SIMD_EMMX_INTRINSICS serial mode is the second fastest. It works in a very similar way to the fastest EMMX mode, also iterating by 16 pixels, except it is written in C++ and uses some 'expensive' function calls that have unneeded overhead, at least when compared to the EMMX mode. It also defines several variables and their register spaces that aren’t needed or used. Because of this, it is slightly slower than the mode written in pure assembly that performs only the needed operations.

2. [30 marks] The code uses the alpha channel of the kernel image as the value for blendFac. Modify the code in both cases to use the alpha channel of the destination image for the blending factor instead.

   This appears to cause the bubbles to lose their blending, which makes sense. Using the alpha channel of the destination image to blend the source image doesn't really work because the destination image is supposed to be opaque, (at least with the tank example) so it's not going to have alpha values that make the code do much blending.

3. [10 marks] Write a simple function that takes an object as an argument by value. Write code to call the function and show the disassembly code associated with the function call. Now write the same function, but have it take the argument by reference, pass the object by reference and show the new disassembly code. What is the difference?

   The disassembly code associated with the function call is as follows(int simpleFunc(5)):

```
    int x = 5;
0045F6A5  mov          dword ptr [x],5
    int y = simpleFunc(x);
0045F6AC  mov          eax,dword ptr [x]
0045F6AF  push         eax
0045F6B0  call         simpleFunc (045D301h)
0045F6B5  add          esp,4
0045F6B8  mov          dword ptr [y],eax
```

```
//Part 3
int simpleFunc(int x) {
009B75C0  push        ebp    ≤ 3ms elapsed
009B75C1  mov         ebp,esp
009B75C3  sub         esp,0C0h
009B75C9  push        ebx
009B75CA  push        esi
009B75CB  push        edi
009B75CC  mov         edi,ebp
009B75CE  xor         ecx,ecx
009B75D0  mov         eax,0CCCCCCCCh
009B75D5  rep stos    dword ptr es:[edi]
009B75D7  mov         ecx,offset _B2F704DE_main@cpp (0A86303h)
009B75DC  call        @__CheckForDebuggerJustMyCode@4 (097CFA0h)
  return x * x;
009B75E1  mov         eax,dword ptr [x]
009B75E4  imul        eax,dword ptr [x]
}
```

When we pass by reference:

```
    int y = simpleFunc(x);
0010F6B6  lea          eax,[x]
0010F6B9  push         eax
0010F6BA  call         simpleFunc (010D2FCh)
0010F6BF  add          esp,4
0010F6C2  mov          dword ptr [y],eax
```

```
//Part 3
int simpleFunc(int& x) {
001475C0  push         ebp    ≤ 2ms elapsed
001475C1  mov          ebp,esp
001475C3  sub          esp,0C0h
001475C9  push         ebx
001475CA  push         esi
001475CB  push         edi
001475CC  mov          edi,ebp
001475CE  xor          ecx,ecx
001475D0  mov          eax,0CCCCCCCCh
001475D5  rep stos     dword ptr es:[edi]
001475D7  mov          ecx,offset _B2F704DE_main@cpp (0216303h)
001475DC  call         @__CheckForDebuggerJustMyCode@4 (010CFA0h)
    return x * x;
001475E1  mov          eax,dword ptr [x]
001475E4  mov          ecx,dword ptr [x]
001475E7  mov          eax,dword ptr [eax]
001475E9  imul         eax,dword ptr [ecx]
}
```

In the first function call, the assembly casts the contents of x as a dword ptr into register EAX, but with the pass by reference, it can use lea to simply pass the existing address of the variable into EAX.

4. [10 marks] Write a simple function that takes no arguments and performs a simple math function. Show the disassembly code associated with the function. Now make the function inline and show the new disassembly code. What is the difference?

```
// Part 4
void simpleMathFunc(){
002D78B0  push        ebp
002D78B1  mov         ebp,esp
002D78B3  sub         esp,0E4h
002D78B9  push        ebx
002D78BA  push        esi
002D78BB  push        edi
002D78BC  lea         edi,[ebp-24h]
002D78BF  mov         ecx,9
002D78C4  mov         eax,0CCCCCCCCh
002D78C9  rep stos    dword ptr es:[edi]
002D78CB  mov         ecx,offset _B2F704DE_main@cpp (03A6303h)
002D78D0  call        @__CheckForDebuggerJustMyCode@4 (029CFA0h)
    int a = 5;
002D78D5  mov         dword ptr [a],5
    int b = 5;
002D78DC  mov         dword ptr [b],5
    int c = a * b;
002D78E3  mov         eax,dword ptr [a]
002D78E6  imul        eax,dword ptr [b]
002D78EA  mov         dword ptr [c],eax
}
```

```
inline void simpleMathFunc() {
00797750  push        ebp
00797751  mov         ebp,esp
00797753  sub         esp,0E4h
00797759  push        ebx
0079775A  push        esi
0079775B  push        edi
0079775C  lea         edi,[ebp-24h]
0079775F  mov         ecx,9
00797764  mov         eax,0CCCCCCCCh
00797769  rep stos    dword ptr es:[edi]
0079776B  mov         ecx,offset _B2F704DE_main@cpp (0866303h)
00797770  call        @__CheckForDebuggerJustMyCode@4 (075CFAAh)
    int a = 5;
00797775  mov         dword ptr [a],5
    int b = 5;
0079777C  mov         dword ptr [b],5
    int c = a * b;
00797783  mov         eax,dword ptr [a]
00797786  imul        eax,dword ptr [b]
0079778A  mov         dword ptr [c],eax
}
```

There is no difference in this case.

However, making a function inline can cause it to take up additional registers since the function body gets inserted at the point of function call, thereby, requiring additional registers for those variables. This can only be noticed, if there are lots of variables being added by the function call, resulting in additional overhead.