PART 7: “Debugging”

Introduction

This episode covers *debugging*. We will see how an application can be halted at any time or at a specific line and how we can inspect variables to verify that everything works as intended. Debugging can be a complex process but we’ll limit ourselves to the basics of breakpoints and following program flow, while keeping an eye on variables.

Getting the stuff you need

As usual, we start with a fresh copy of the template. Extract the package to a new directory (say, `c:\my_projects\debugging`) and load up the .sln file. Remove the ‘hello world’ code in the `Tick` function.

Debug Mode

Visual Studio projects typically can run in two modes: *debug* and *release*. You can select this mode in the drop-down box that should be next to your menu bar (if you followed the instructions in the first episode of this tutorial). By default, debug mode is selected.

So what’s the difference?

Well, *debug* generates larger, slower programs. To verify this: build your program (hit F7), go to the folder where you unpacked the template, and then to the `Debug` folder. The executable here (Template.exe) is 48Kb. You can’t run it here by the way; it won’t be able to find .dll files and assets. However if you copy it (just the .exe) to the folder where the .sln and .vcxproj are, it will work fine. Now do the same in *release* mode. Interestingly enough, the template puts the .exe right next to the .sln this time (and not in the Release folder); this will be fixed in Tmpl_2017-02. ☺ As you can see, the release .exe is only 29Kb.

Why is it so much smaller?

The answer is *debugging*. The debug executable contains extra information, lacks certain optimizations, and adds a number of checks (e.g., array boundary checks). It will also initialize variables to 0. The release executable on the other hand lacks the checks and variable initializations. As a result, it runs faster and is smaller.

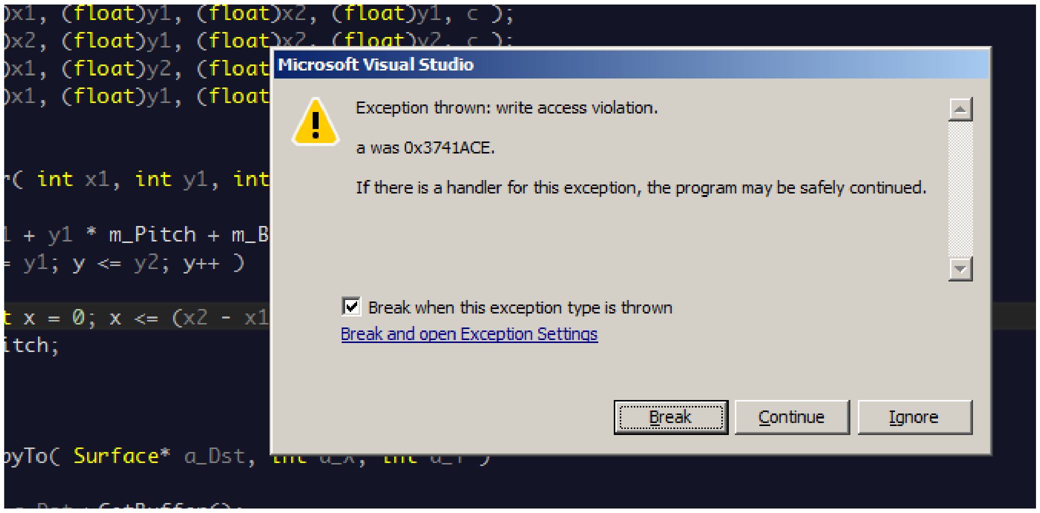
Fail

Insert the following code in a fresh template:

```
Surface image( "assets/ball.png" );

// -----------------------------------------------------------
// Main application tick function
// -----------------------------------------------------------
void Game::Tick( float deltaTime )
{
   // clear the graphics window
   screen->Clear( 0 );
   // draw a grid
   for( int x = 15; x < 800; x += 16 )
   {
      for( int y = 6; y < 512; y += 12 )
      {
         Pixel p = image.GetBuffer()[x / 16 + (y / 12) * 50];
         int red = p & 0xff0000;
         int green = p & 0x00ff00;
         int blue = p & 0x0000ff;
         screen->Bar( x, y, x + 12, y + 2, red );
         screen->Bar( x, y + 4, x + 12, y + 6, green );
         screen->Bar( x, y + 8, x + 12, y + 10, blue );
      }
   }
}
```

What it is supposed to do is a surprise, but for now one thing is clear: it doesn’t do it:



```
)x1, (float)y1, (float)x2, (float)y1, c );
)x2, (float)y1, (float)x2, (float)y2, c );
)x1, (float)y2, (float
)x1, (float)y1, (float

r( int x1, int y1, int

1 + y1 * m_Pitch + m_B
= y1; y <= y2; y++ )

t x = 0; x <= (x2 - x1
itch;

pyTo( Surface* a_Dst, int a_X, int a_Y )
```

Microsoft Visual Studio

Exception thrown: write access violation.

a was 0x3741ACE.

If there is a handler for this exception, the program may be safely continued.

☑ Break when this exception type is thrown

Break and open Exception Settings

Now what?

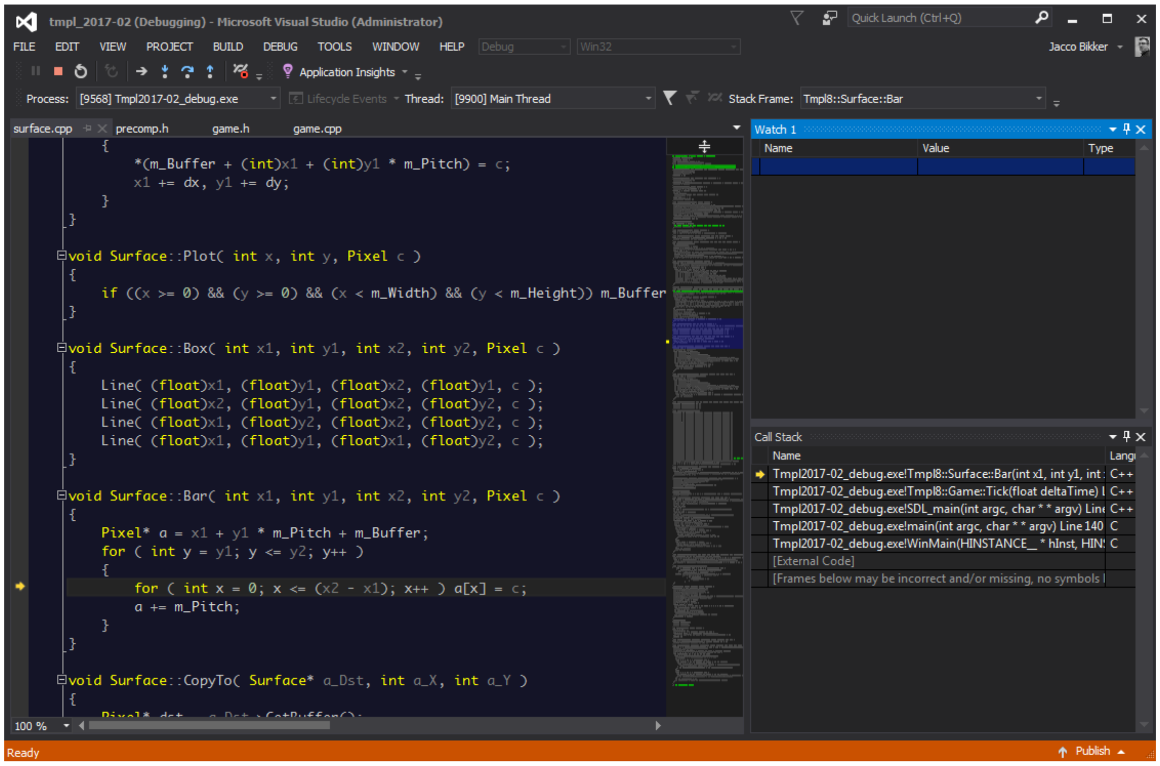
After you click ‘Break’ you will be looking at some template code: function Surface::Bar, with a little yellow arrow pointing at a line that apparently caused a problem. Time to figure out what the problem is.

Gathering Intel

Since Visual Studio uses a distinct screen layout when debugging (look at that ugly icon bar at the top!), we may just as well make ourselves at home:

- In the menu bar, click on DEBUG, select Windows, then Call Stack. Drag the call stack window to the right of the screen.
- Likewise, go to DEBUG / Windows and add Watch / Watch 1. Drag this to the right of the screen as well, so the call stack and watch 1 both take up half of the vertical space.

You now have a layout as in the screenshot.


tmpl_2017-02 (Debugging) - Microsoft Visual Studio (Administrator)
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TOOLS WINDOW HELP
Debug
Win32
Jacco Bikker
Application Insights
Process: [9568] Tmpl2017-02_debug.exe
Lifecycle Events
Thread: [9900] Main Thread
Stack Frame: Tmpl8::Surface::Bar
surface.cpp
precomp.h
game.h
game.cpp
Watch 1
Name
Value
Type
Call Stack
100 %
Ready
Publish

```
        {
            *(m_Buffer + (int)x1 + (int)y1 * m_Pitch) = c;
            x1 += dx, y1 += dy;
        }
    }

void Surface::Plot( int x, int y, Pixel c )
{
    if ((x >= 0) && (y >= 0) && (x < m_Width) && (y < m_Height)) m_Buffer
}

void Surface::Box( int x1, int y1, int x2, int y2, Pixel c )
{
    Line( (float)x1, (float)y1, (float)x2, (float)y1, c );
    Line( (float)x2, (float)y1, (float)x2, (float)y2, c );
    Line( (float)x1, (float)y2, (float)x2, (float)y2, c );
    Line( (float)x1, (float)y1, (float)x1, (float)y2, c );
}

void Surface::Bar( int x1, int y1, int x2, int y2, Pixel c )
{
    Pixel* a = x1 + y1 * m_Pitch + m_Buffer;
    for ( int y = y1; y <= y2; y++ )
    {
        for ( int x = 0; x <= (x2 - x1); x++ ) a[x] = c;
        a += m_Pitch;
    }
}

void Surface::CopyTo( Surface* a_Dst, int a_X, int a_Y )
{
```

Name	Lang
Tmpl2017-02_debug.exe!Tmpl8::Surface::Bar(int x1, int y1, int	C++
Tmpl2017-02_debug.exe!Tmpl8::Game::Tick(float deltaTime)	C++
Tmpl2017-02_debug.exe!SDL_main(int argc, char * * argv) Line	C++
Tmpl2017-02_debug.exe!main(int argc, char * * argv) Line 140	C
Tmpl2017-02_debug.exe!WinMain(HINSTANCE__ * hInst, HIN	C
[External Code]	
[Frames below may be incorrect and/or missing, no symbols	

The call stack is our first useful bit of information. It tells us where we are (a bit verbose: `Tmpl2017-02_debug.exe!Tmpl8::Surface::Bar`) and how we got there (via `Game::Tick`). Double-click on `Game::Tick` to see which line of code jumped to the Bar method.

Let's go back to `Surface::Bar` to see what's going on. In the Watch window, type (under 'Name'): x1. You will immediately see the value of parameter x1, which was passed to `Surface::Bar` from the Tick function. Add y1 as well. It's value is 514… And that's outside the screen, which is 800x512 pixels. Is that a problem? Yes it is: a chunk of memory was reserved for those pixels, and we're writing outside that chunk. There could be anything there, and therefore the program crashes.

Fixing this is easy: in the Tick function, make sure we don't loop all the way to 512 over y; let's loop to 480 or so instead.

Tracing

Obviously you don't have to wait for a crash to freeze your program mid-flight. Move your cursor to line 35 in `game.cpp` and hit F9. The result is a red sphere in front of the line. Now run the program again. It will halt at the line you marked. This is called a *breakpoint*, and you can have as many in your program as you like.

Once your program is halted you can restart it using F5, or step through it one line at a time by pressing F10. When you encounter a function call (such as `screen->Bar(…)`), you can either step over it using F10, or into it, using F11. And, if you don't feel like remembering those function keys: there's icons on the icon bar for the same functionality.

Conditional Breakpoints

Imagine you want to halt the program when a certain condition is met, e.g. when the x coordinate exceeds 500.

One way to do this is by setting a condition for the breakpoint. Hover over the breakpoint, then click the gears to get to the settings for the breakpoint. Here you can add such a condition. Or, keep it simple, and add the condition to your code:

```
if (x > 500)
{
   int w = 0;
}
```

The `int w = 0` statement is obviously irrelevant: it is just there to be able to place a breakpoint inside the if-statement.

You can safely leave bits of debugging code like these: they will not affect application performance once you switch back to release mode.

The reason for this is interesting: in release mode, the compiler does its best to optimize your code. It will notice that the value of w is never used, so it doesn't actually produce code in your .exe for it. That means in turn that there is no code inside the if-scope, so that is removed as well. In debug mode this code is of course there, but this mode is not for performance anyway.

Trust No One

Some things are not supposed to happen. Perhaps you assume no one will draw boxes outside the screen. And still it happens. A great way to catch problems like that is using asserts.

Add the following line inside the inner for-loop:

```
assert( y < 400 );
```

Note: you need the 2017-02 template for this to work.

Obviously this assertion is going to fail. When it does, the program will halt.
It is a good habit to add assertions like these (well, slightly more sane assertions perhaps) to your program: they are only included in debug mode, and they add a fair bit of safety for those unforeseen situations.

Assignment

Debugging is something you need to get used to; once you are familiar with the process you will not be able to live without it. It's a bit hard to give you a debugging assignment, so instead:

- Fix the Surface::Bar function so that it doesn't crash when off-screen coordinates are specified.
- Replace the ball.png image with an image you found on the internet (i.e., not an image that is already in the assets folder). Modify the size of the 'leds' so that the full image fits on the screen.

END OF PART 7

Next part: "Addresses"