

ECE 480 Application Note

By: Jacob Hersha

4/3/15

Creating a Sequence of Media with Visual Studio

Executive Summary

Microsoft Visual Studio can be used to perform a wide variety of media processing techniques. This software allows a user to create programs to manipulate numerous forms of multimedia. Visual Studio is an Event Driven programming environment. Everything done to media is a result of a response to a message from Windows. This application note will be providing a guide of how to put different types of media into a structured sequence for the user to view. Once created media will be displayed at certain points in time defined by the user.

Keywords: Microsoft Visual Studio, Multimedia, Image Processing

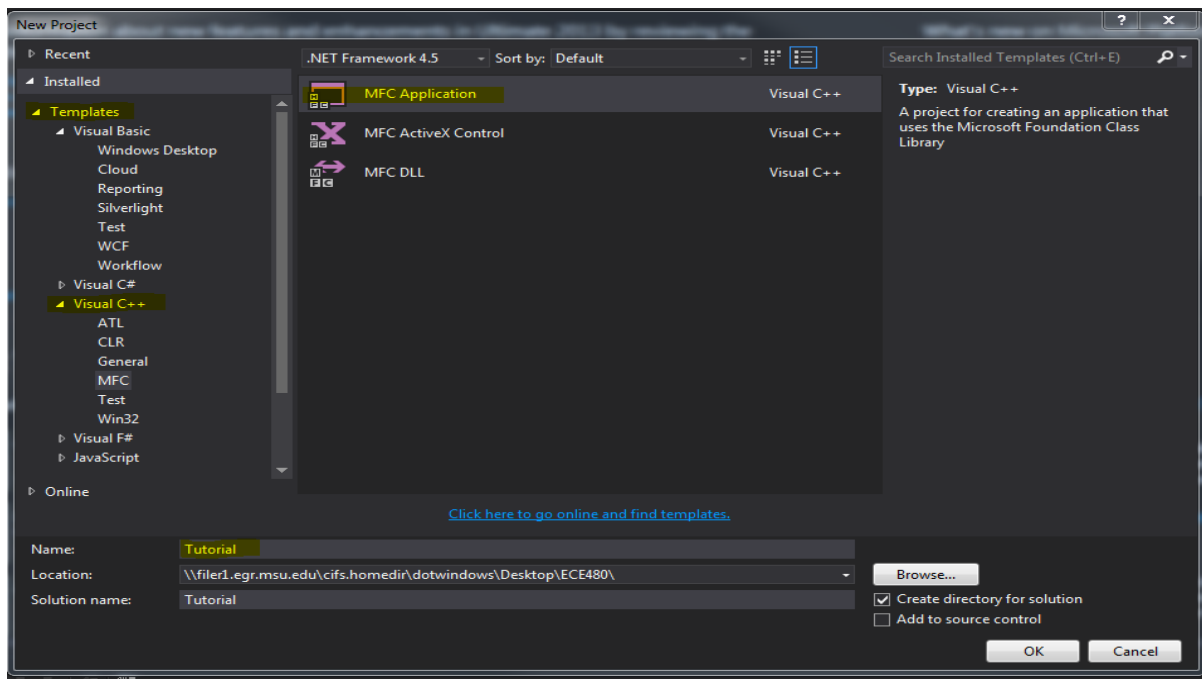
Introduction

Visual Studio is an extremely user friendly environment. Debugging a program is very simple and a testing environment is created very quickly. One of the most basic programs a user can create is a way to display media at certain points in time. There are many use cases to create a properly timed media sequence. In the scope of the design project for this team, a signal could be sent to send a message, image, or video to the user when a hot spot is detected. For the average person, a media sequence could be created to simply show a collection of media the user would like to continuously run.

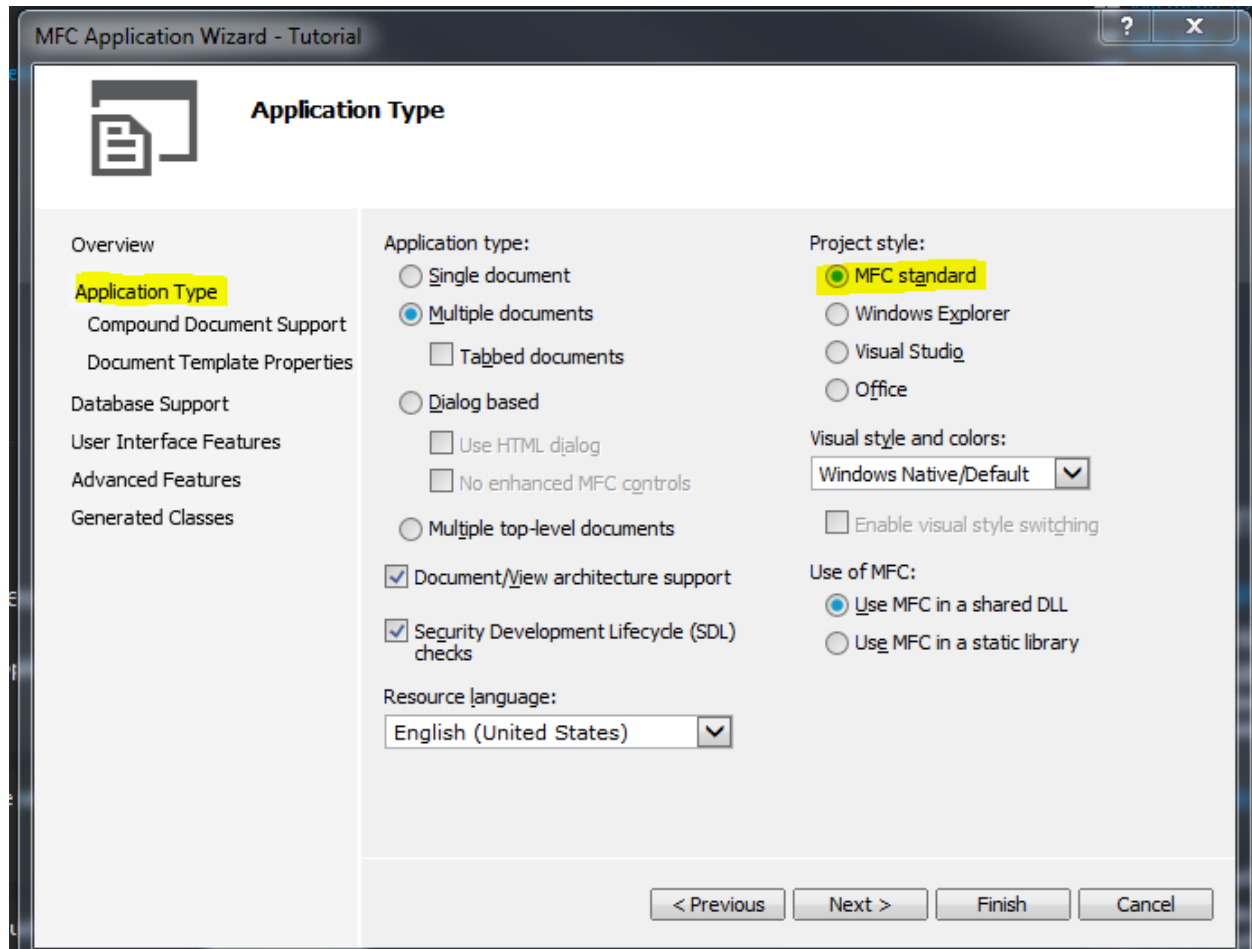
Objective

In order to keep track of what the user is doing a simple state machine must be created. This state machine will take in media and signal inputs and then properly output in ways defined by the state machine. The state machine will need to be provided “cases” in which it will perform certain tasks. A sequencer is actually simple to implement. The state machine needs to be provided media, case statements of what to do with the media, and a timer of when to switch cases. This tutorial will provide a general step-by-step tutorial on how to set this up.

To begin, a basic Microsoft Foundation Class Application (MFC) must be created to start a new project. This tutorial is demonstrating use with Visual Studio 2013 using the C++ language, but the steps will be the exact same in Visual Studio 2010. Open Visual Studio 2013 and select **File/New/Project**. In the New Project dialog box, drop down the Visual C++ tab from the Installed Templates category. Choose **MFC** and then select **the MFC Application** template. Save the project to an appropriate title and click OK.



The MFC Application Wizard will appear. When it does select **Application Type** and enable **MFC standard** as the Project style and the click Finish.



Visual Studio will create multiple classes by default for this program. You can view these classes by selecting the **Solution Explorer** tab on the right side of the screen. For this tutorial there will not be any new classes created, but if a program requires a more sophisticated layout, classes can easily be added. An important item to note is that for this tutorial the state machine will switch states by certain times we tell it to. For industry uses, such as this UAV project, we could relay a signal from the sub-module to Visual Studio to start a certain state. If autonomous detection was created then a signal could be sent to this program to change states when a signal is detected.

The first sequence in this state machine is the program will display a text saying, "Possible Hot Spot Found." When a signal is detected by the drone system created for team 14's design project, this would essentially be the first state, indicating the user a hot spot may have been found. Before this state can be created, all states of the program must be declared in the project. To do this open the **TutorialView.h** file and create add an **enum** of states with public access.

```

// TutorialView.h : interface of the CTutorialView class
//

#pragma once

class CTutorialView : public CView
{
protected: // create from serialization only
    CTutorialView();
    DECLARE_DYNCREATE(CTutorialView)

// Attributes
public:
    CTutorialDoc* GetDocument() const;
    enum States {Start, Text, Image1, Image2, Done};

```

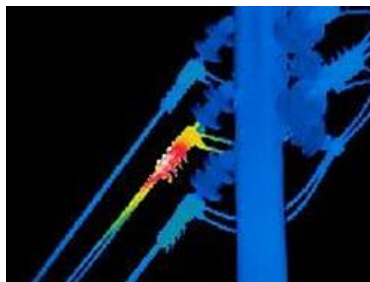
To keep track of what state the program is in, a state variable has to be added to the **TutorialView.h** file as a private member variable so it may not be altered. To declare this simply paste **States m_state;** as a private variable. After this, the state should be initialized to the first possible state, which is **Start**. This should be initialized in the constructor which is in the **TutorialView.cpp** file.

```

CTutorialView::CTutorialView()
{
    m_state = Start;
}

```

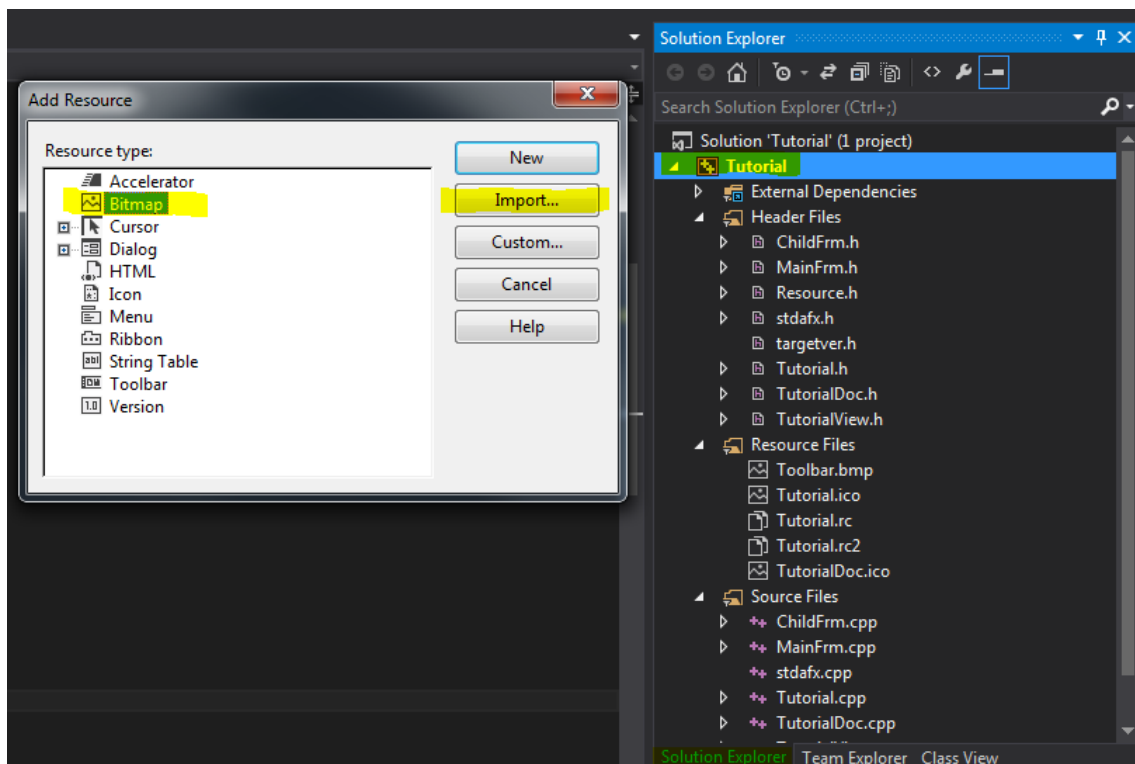
Since the states are now created we can start working with to do in each state. For this tutorial we will load in two images, but in real applications you may write a script to load in new images that are continuously coming in. These images should be added as private member variables of type **CBitmap** in our view class. The two images below will be added. The first image is called “drone” and the second “thermal.” Declaring these as variables **CBitmap m_drone** and **CBitmap m_thermal** in the header file will add them to the project.



To do this create a folder named **media** in the project directory folder created at the location specified when you created the program. Save these images into the media folder as **BMP** files.

Tutorial				
New folder				
Name	Date modified	Type	Size	
Debug	4/2/2015 3:28 PM	File folder		
ipch	4/2/2015 3:25 PM	File folder		
media	4/3/2015 9:50 AM	File folder		
Tutorial	4/2/2015 4:06 PM	File folder		
Tutorial	4/3/2015 9:36 AM	SQL Server Comp...	77,504 KB	
Tutorial	4/2/2015 3:15 PM	Microsoft Visual S...	1 KB	

Now in Visual Studio they can be added as a resource to the program. In the **Solution Explorer** right click on **Tutorial** that is under the first line of the window. Click **Add → Resource**. Then select **Bitmap** and press **Import**. Find the image you want to add and click open. Do this for any other images you want added. Next, in the constructor, load a bitmap into the program by the line `m_drone.LoadBitmap(IDB_BITMAP1);`. The name of the bitmap can be found in **Resource View** → **Tutorial.rc** → **Bitmap**. From there the bitmap names will be present.



To ensure there are no errors at this point the program should be compiled. Select **Build → Build Solution** from the menu bar. If there are no errors the window at the bottom of the screen should say

Succeeded. We can begin to deal with what each class does. Add the code below to the function `OnDraw()`. Right now this does not switch states or even have a timer but we have loaded our images into the correct states. As you can see the case `Text` has created a colored pen and a font. The text is given a pointer and the draw variable points to this font to draw the text.

```
void CTutorialView::OnDraw(CDC* pDC)
{
    CTutorialDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    if (m_firstdraw)
    {
        m_firstdraw = false;
        OnFirstDraw();
    }
    // TODO: add draw code for native data here

    CDC bmpDC;
    switch (m_state)
    {
    case Text:
    {
        CPen pen;
        pen.CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
        CFont font;
        font.CreateFont(30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, TEXT("Arial"));

        CFont *oldfont = pDC->SelectObject(&font);
        |
        pDC->TextOut(50, 50, L"Possible Hot Spot Found");

        pDC->SelectObject(oldfont); }
        break;

    case Image1:
        bmpDC.CreateCompatibleDC(pDC);
        bmpDC.SelectObject(&m_drone);
        pDC->BitBlt(0, 0, 640, 480, &bmpDC, 0, 0, SRCCOPY);
        break;

    case Image2:
        bmpDC.CreateCompatibleDC(pDC);
        bmpDC.SelectObject(&m_thermal);
        pDC->BitBlt(0, 0, 640, 480, &bmpDC, 0, 0, SRCCOPY);
        break;

    default:
        // Display nothing
        break;
    }
}
```

As you can probably notice is the variable `m_firstdraw`. You should create a function `OnFirstDraw()` in the view class to put things in the program must be done when the program starts. For this program we simply want to just get the time of day and start the timer.

```
void CTutorialView::OnFirstDraw()
{
```

```

        m_starttime = timeGetTime();
        OnTimer(1);
    }

```

This brings the tutorial to starting the timer. First a timer function can be created. This is simple in Visual Studio as you can select **Class View** and select the **CTutorialView** class. There is a **message** button below to press. Scroll until **WM_TIMER** is found and click the arrow to select **Add OnTimer**. This function is now added to the class. Add the code below to the **OnTimer** function and initialize a private variable **m_timer** to be equal to zero in the constructor.

```

void CTutorialView::OnTimer(UINT_PTR nIDEvent)
{
    // Kill any existing timer
    if (m_timer)
    {
        KillTimer(m_timer);
        m_timer = 0;
    }

    // Force redraw whenever the timer fires
    Invalidate();

    // This will keep track of the relative time
    // to the next state.
    DWORD nexteventtime;

    switch (m_state)
    {
    case Start:

        m_state = Text;
        nexteventtime = 2000;
        break;

    case Text:
        m_state = Image1;

        nexteventtime = 8000;
        break;

    case Image1:
        m_state = Image2;
        nexteventtime = 12000;
        break;

    case Image2:
        m_state = Done;
        break;
    }

    // Only reset the timer if we are not done
    if (m_state != Done)
    {
        DWORD currenttime = timeGetTime();
        DWORD nexttime = m_starttime + nexteventtime;
        DWORD tillnext = nexttime > currenttime ? nexttime - currenttime : 0;

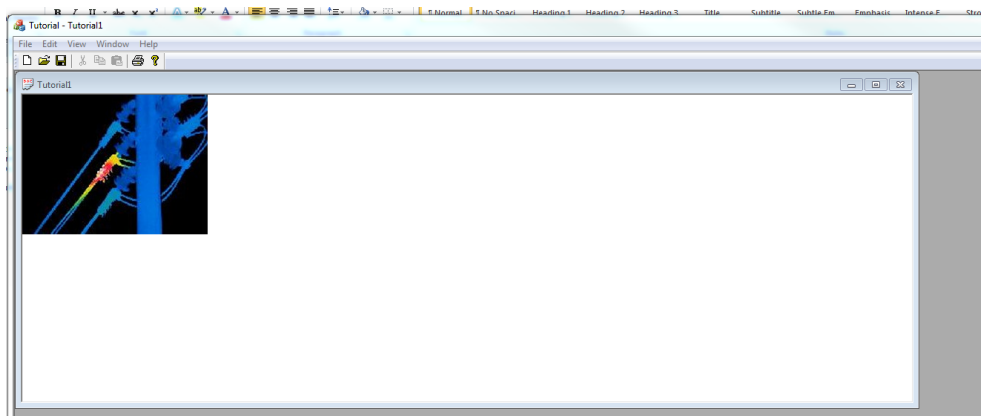
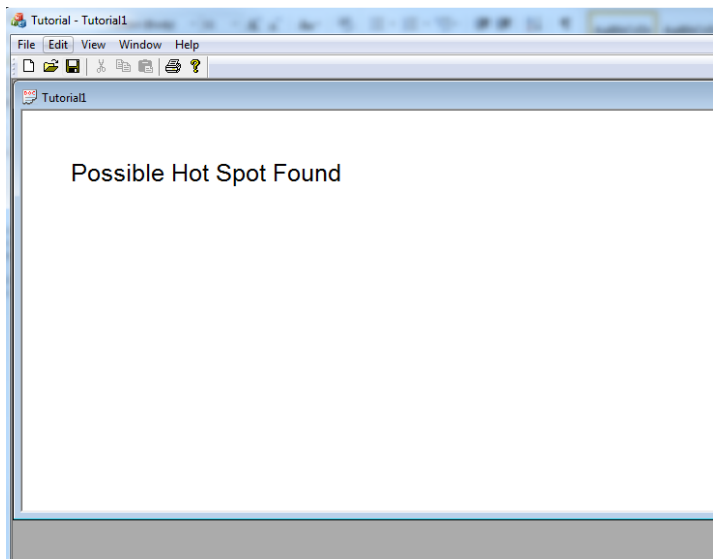
        m_timer = SetTimer(1, tillnext, NULL);
    }
}

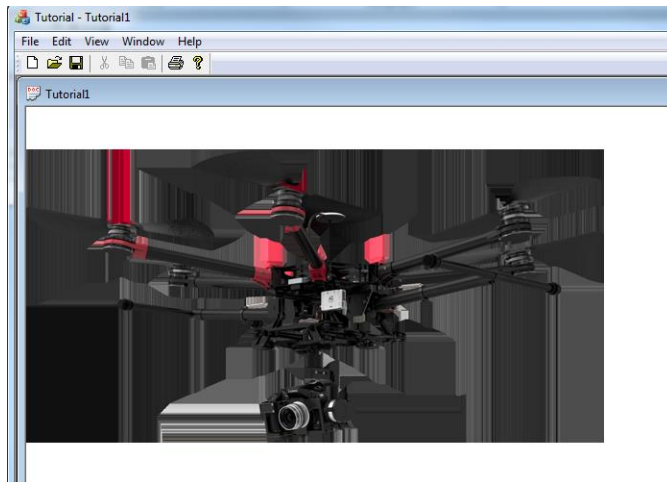
```

This function simply takes each state and defines which state should follow. Each state is given a certain time in milliseconds which will determine how long the state runs for. The `Invalidate()` function simply clears the screen. This would be useful in the event of a continuous data coming in and possibly wanting to clear everything in order to allow other media to come in. The most important aspect of this function is

the time. **timeGetTime()** uses the Windows operating system and returns the actual system time. This way the program can be altered to switch states after certain events defined by the user happens. This function only works if the library **mmsystem.h** is included in the project.

Compile the program to test for any errors. For this tutorial an error is likely common if a function or member variable that was created was not declared in the header file. Once the program compiles it can now be ran. To do this select **Debug → Start without debugging**. A new testing window will appear and the program will run. For this program the text that a hot spot was found will show for 2 seconds, the hot spot image will show for 6 seconds, and the drone image will be displayed by 4 more seconds. Images of this sequence in the window are below.





The original drone image is transparent so it gets distorted in this program. There are certainly ways to manipulate pixels to deal with transparent images, but this tutorial will not explore those options. This program can be extremely useful. Ideally, a script would be created to load in images of hot spots automatically when a possible hot spot is detected. This script would also be continuously running and restart each time a hot spot is detected to display media sequence over again.

Conclusion

This is an extremely basic Visual Studio program. Visual Studio can be used to manipulate video and image frames for a wide variety of purposes. For this tutorial case, a user at a ground station monitoring hot spots would find this very useful in knowing when a potential hot spot is detected and what it looks like. Visual Studio is a very powerful tool and with enough knowledge of media processing, one can perform numerous applications with a wide variety of media. There are also many tutorials available online for those interested in performing other media processing programs.