Computer Graphics

UWP - Application Setup

Fernando Geraci

Requirements

Visual Studio Community 2015 with latest Update (free)

Includes Direct3D SDK and utilities already

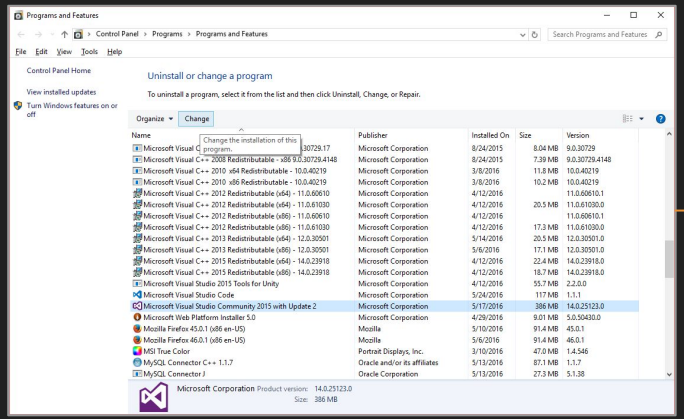
> Once installed, go to Programs and Features

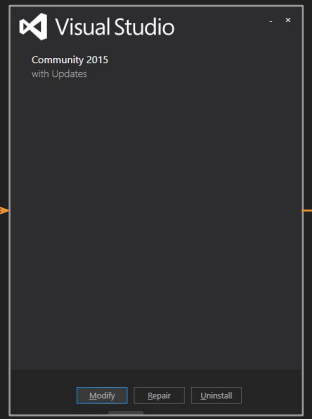
> Select Microsoft Visual Studio Community 2015 …

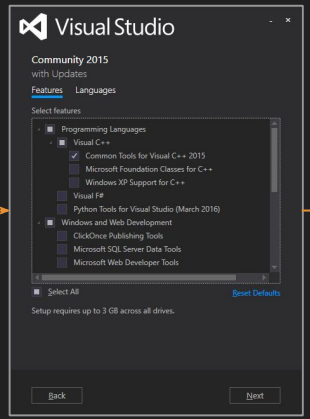
> Click on Change

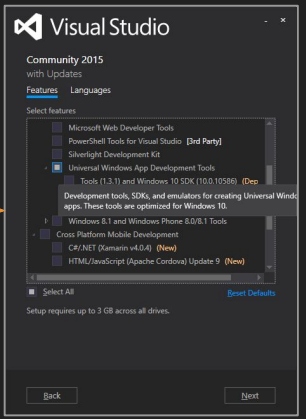
> Select Visual C++ and Universal Windows App Development Tools

> Install !

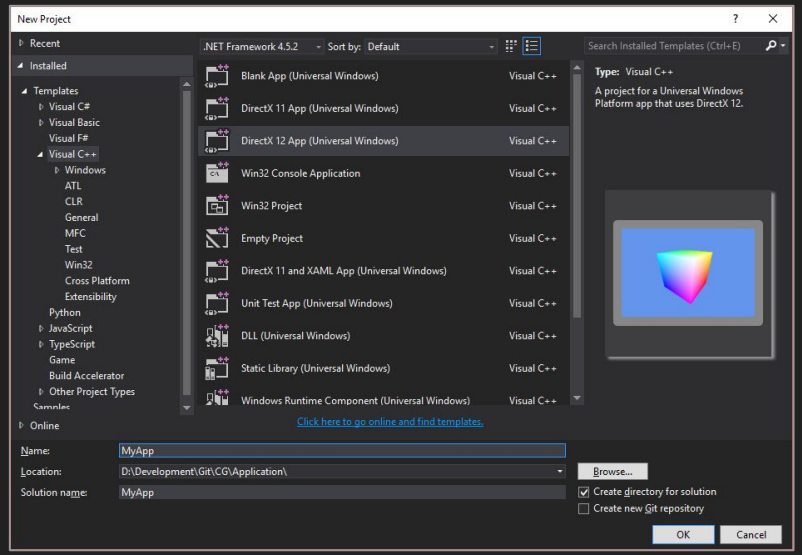


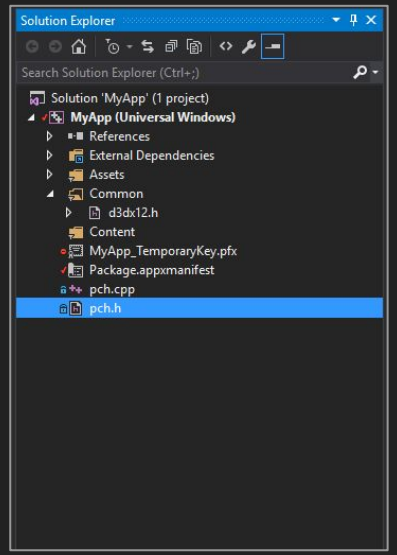






Application - Creating the Project





Open VS2015 and create a new DirectX 12 Universal Windows App.

If you create the project in a folder where you already initialized a git repository, VS will set it up for you in the IDE later on. This is great for managing the project moving forward.

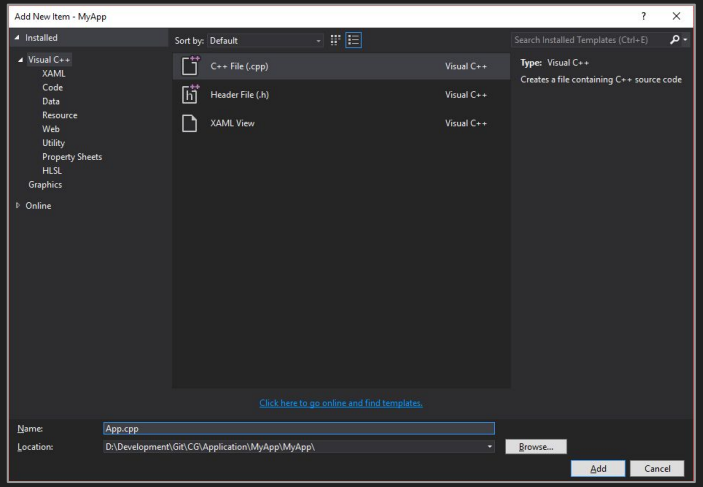
In the Solution Explorer, delete EVERYTHING but the files within ExternalDependencies, Assets and the ones displayed in the picture.

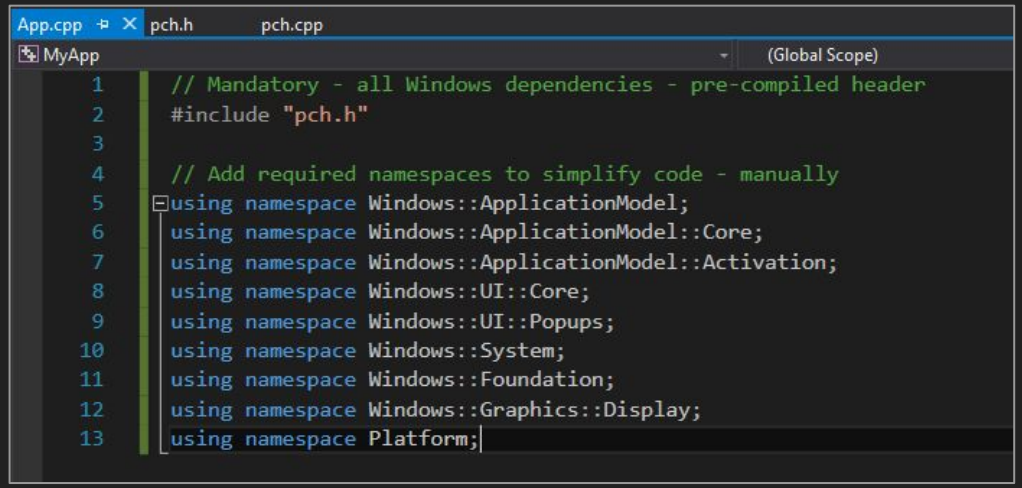
pch.h are pre-compiled headers. There, you can do project wide includes, note that D3D libs are already in place.
Recall that d3dx12.h is an utilities header file provided by D3D12, don't get rid of this.

Application - Creating the Project

Add a single .cpp file names App.cpp - here we will code everything required for a UWP application to work.

Once created, declare many of the namespaces we will use in the next steps.





Application - App Setup

App.cpp | pch.h | pch.cpp

MyApp | App

```
// Mandatory - all Windows dependencies - pre-compiled header
#include "pch.h"

// Add required namespaces to simplify code - manually
using namespace Windows::ApplicationModel;
using namespace Windows::ApplicationModel::Core;
using namespace Windows::ApplicationModel::Activation;
using namespace Windows::UI::Core;
using namespace Windows::UI::Popups;
using namespace Windows::System;
using namespace Windows::Foundation;
using namespace Windows::Graphics::Display;
using namespace Platform;

ref class App sealed : public IFrameworkView {
private:
    CoreWindow^ m_CoreWindow;
protected:
    property CoreWindow^ MainWindow {
        CoreWindow^ get() {
            return this->m_CoreWindow;
        }
    }
public:
    // Inherited via IFrameworkView
    virtual void Initialize(CoreApplicationView^ AppView) {}
    virtual void SetWindow(CoreWindow^ wnd) {}
    virtual void Load(Platform::String ^entryPoint) {}
    virtual void Run() {}
    virtual void Uninitialize() {}
};
```

d3dx12.h | App.cpp | pch.h | pch.cpp

MyApp | AppSource

```
using namespace Windows::UI::Popups;
using namespace Windows::System;
using namespace Windows::Foundation;
using namespace Windows::Graphics::Display;
using namespace Platform;

ref class App sealed : public IFrameworkView {
private:
    CoreWindow^ m_CoreWindow;
protected:
    property CoreWindow^ MainWindow {
        CoreWindow^ get() {
            return this->m_CoreWindow;
        }
    }
public:
    // Inherited via IFrameworkView
    virtual void Initialize(CoreApplicationView^ AppView) {}
    virtual void SetWindow(CoreWindow^ wnd) {}
    virtual void Load(Platform::String ^entryPoint) {}
    virtual void Run() {}
    virtual void Uninitialize() {}
};

ref class AppSource sealed : IFrameworkViewSource {
public:
    virtual Windows::ApplicationModel::Core::IFrameworkView ^ CreateView() {
        return ref new App();
    }
};
```

The main application is an instance of an IFrameworkView class. This will have many abstract “life-cycle” and call backs methods which need to be implemented. We will complete these as we go. For now, just declare them as shown in the picture.

Secondly, add below the App class yet another one which implements the IFrameworkViewSource interface. This only needs the public CreateView function.

Application - App Setup

```
15  ref class App sealed : public IFrameworkView {
16  private:
17      CoreWindow^ m_CoreWindow;
18  protected:
19      property CoreWindow^ MainWindow {
20          CoreWindow^ get() {
21              return this->m_CoreWindow;
22          }
23      }
24  public:
25      // Inherited via IFrameworkView
26      virtual void Initialize(CoreApplicationView^ AppView) {}
27      virtual void SetWindow(CoreWindow^ wnd) {}
28      virtual void Load(Platform::String ^entryPoint) {}
29      virtual void Run() {}
30      virtual void Uninitialize() {}
31  };
32
33  ref class AppCreator sealed : IFrameworkViewSource {
34  public:
35      virtual Windows::ApplicationModel::Core::IFrameworkView ^ CreateView() {
36          return ref new App();
37      }
38  };
39
40  [MTAThread]
41  int main(Array<String^>^ args) {
42      CoreApplication::Run(ref new AppCreator());
43      return 0;
44  }
```

Finally, implement the main function.

Note we are marking the main function as a Multi Threaded Apartment application.

Good, at this point, you should be able to build and run the application.

This will result in a quick splash screen which closes automatically with a message:

MyApp.exe has exited … code 0.

All good! Now we will start making it functional.

Application - App Setup

```
// Inherited via IFrameworkView
virtual void Initialize(Windows::ApplicationModel::Core::CoreApplicationView ^AppView)
{
    /* Register Window's event handlers */
    AppView->Activated += ref new TypedEventHandler     // pass OnActivate function address as the handler for the window's creation
                                                        // to the Activated event handler - match its parameters with the generic <T,Ts>
        <CoreApplicationView^, IActivatedEventArgs^>(this, &App::OnActivated);

    /* Application's state - CoreApplication::<event> handlers */
    CoreApplication::Suspending += ref new EventHandler
        <SuspendingEventArgs^>(this, &App::OnSuspending);

    CoreApplication::Resuming += ref new EventHandler
        <Object^>(this, &App::OnResuming);

    CoreApplication::Exiting += ref new EventHandler
        <Object^>(this, &App::OnExiting);
}
```

1. Initialize()

Register application's life-cycle event handler.

2. Implement the handlers for the life-cycle events.

```
/* Application - top level app's state handlers */

void OnActivated(CoreApplicationView^ CoreAppView, IActivatedEventArgs^ Args) {
    CoreWindow^ coreWnd = CoreWindow::GetForCurrentThread();
    coreWnd->Activate();
}

void OnSuspending(Object^ pSender, SuspendingEventArgs^ args) {

}
void OnResuming(Object^ pSender, Object^ args) {

}
void OnExiting(Object^ pSender, Object^ args) {

}
```

Application - App Setup

```
virtual void SetWindow(Windows::UI::Core::CoreWindow ^window)
{

    /* Set up Properties */
    this->m_CoreWindow = window;

    /* Windows event handlers */
    window->Closed += ref new TypedEventHandler
        <CoreWindow^, CoreWindowEventArgs^>(this, &App::OnWindowClosed);

    window->SizeChanged += ref new TypedEventHandler
        <CoreWindow^, WindowSizeChangedEventArgs^>(this, &App::OnWindowResized);

    /* IO events handlers */

}
```

3. SetWindow()

Register window's event handlers.

4. Implement the handlers for the window

```
void OnWindowClosed(CoreWindow^ pWnd, CoreWindowEventArgs^ args) {

}

void OnWindowResized(CoreWindow^ pWnd, WindowSizeChangedEventArgs^ args) {
    UINT width = args->Size.Width;
    UINT height = args->Size.Height;
}
```

Application - App Setup

```
virtual void Load(Platform::String ^entryPoint) { }

virtual void Run();

virtual void Uninitialize() {}
```

5. Load, Run and Uninitialize

For now, Load and Uninitialize will be empty

We will define Run here but implemented outside the class definition.

```
protected:

    property CoreWindow^ MainWindow {
        CoreWindow^ get() {
            return this->m_CoreWindow;
        }
    }

    property bool IsWindowClosed;
```

6. Add a window control property

We will use this to set flags on whether the app's main window has been closed.

Application - App Setup

```
void App::Run() {
    while (!(this->IsWindowClosed)) {

        /* Different types of Application's events processing
        ProcessUntilQuit <- is a blocking infinite loop until the Closed event is triggered. This is fully event drive rather than giving control to loop.
        alternative > coreWnd->Dispatcher->ProcessEvents(CoreProcessEventsOption::ProcessUntilQuit); << infinite loop, not good.
        We want to be in control of the processing loop, so instead, we will use ProcessAllIfPresent
        */
        this->MainWindow->Dispatcher->ProcessEvents(CoreProcessEventsOption::ProcessAllIfPresent);

    }
}
```

7. Finally, we implement our app's main Run method. This will go and listen for input until the main window has been closed.

Application - Result

At this point, you should see:

We are ready to start coding our application in it!

All the action will happen in the Run() method. Which is:

```
Run
     while(!windowClosed) {

          processInput;

          if(timer_tick) {
               MyApp.Update();
          }
     }
}
```