A Sample ECS Architecture

By Christian Sagel

Foreword

In this project I will be demonstrating how to compose a game engine framework using an Entity-Component System (ECS) architecture, where every class ultimately derives from a base *Object* class. Variants of this architecture are used in game engines such as the DigiPen Zero Engine and Unity.

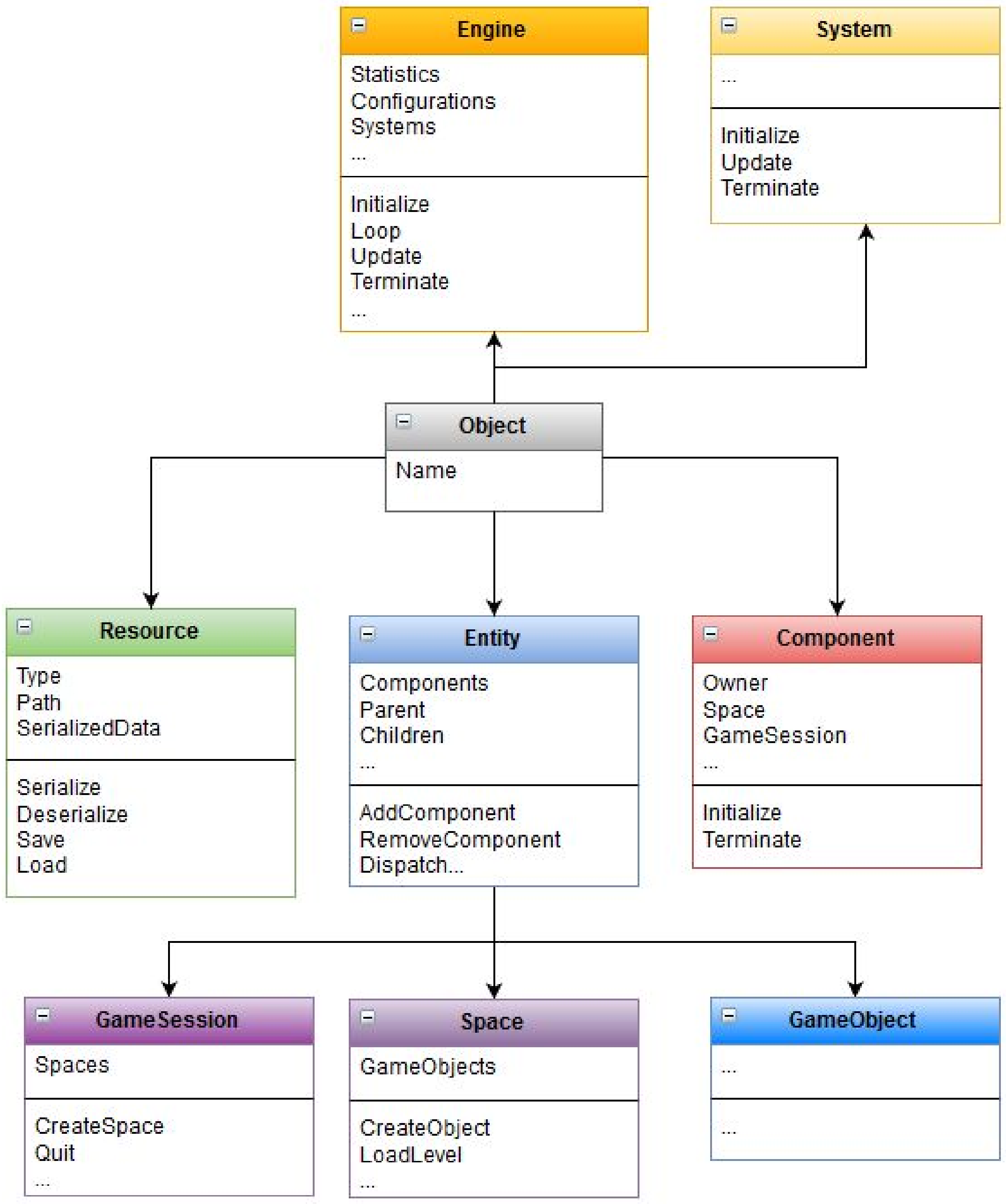
Table of Contents

Introduction
Libraries
Precompiled Header
Tracing

Introduction

The main strength behind this architecture is a coherent conceptualization of how the engine is configured, also allowing us to specify common methods, for example serialization routines which derived classes can utilize.

Class Diagram

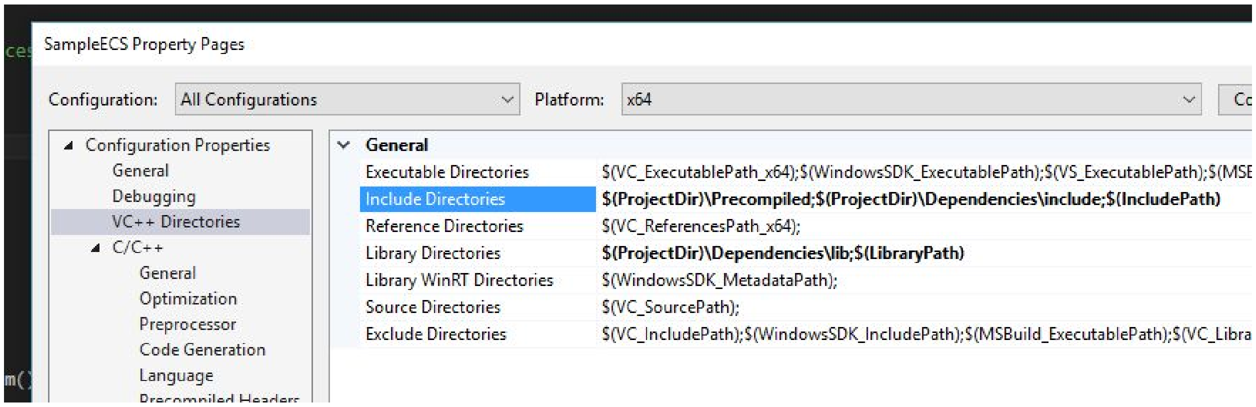


Let's review the main classes used in this architecture:

- **Object**: The base class, providing a name and an unique identifier (ID).
- **Engine**: The main class encapsulating the engine's functions, such as initialization and termination routines. The engine, which has a container of all systems, is itself a singleton and often accessed for one system to talk to another at runtime.
- **System**: A system provides one branch of functionality to the engine, such as providing graphics rendering, audio playback, content management, object creation (Factory), etc.
- **Resource**: The resource-derived classes encapsulate the functionality and management of common resources such as images, sound clips, level data, etc.
- **Entity**: The entity is the main composition class which forms the basis for the structure of how game logic is driven. An entity's logic and function depends on the composition of its components. Its derived classes are the ones that compose the structure of a game. (GameObject, Space, GameSession)
- **Component**: The lynchpin behind the architecture. A component is an object which not only holds data but also drives logic through the methods it provides. A component is always provided initialization and update methods which are often the starting point for it to take action.
- **GameSession**: The singleton entity which owns every other entity in a given game instance. A GameSession is composed of one or more spaces.
- **Space**: The entity which acts as a logical space and container of GameObjects, which are commonly loaded through user-created levels (Levels are basically serialized containers of GameObjects) or instantiated at run-time (By active components).
- **GameObject**: The entities owned by Spaces and which act as the actors in a game.

Libraries

Since it is very common for projects to include external headers and libraries, Visual Studio provides a way for them to be easily accessible by your project. This is configured in the *VC++ Directories* column.



SampleECS Property Pages

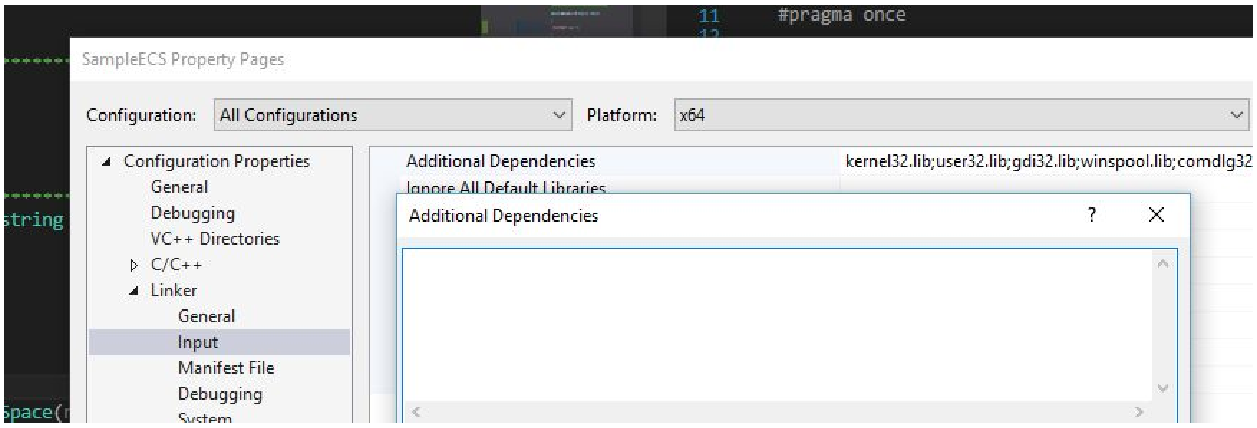
Configuration: All Configurations Platform: x64

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers

General	
Executable Directories	$(VC_ExecutablePath_x64);$(WindowsSDK_ExecutablePath);$(VS_ExecutablePath);$(MSE
Include Directories	**$(ProjectDir)\Precompiled;$(ProjectDir)\Dependencies\include;$(IncludePath)**
Reference Directories	$(VC_ReferencesPath_x64);
Library Directories	**$(ProjectDir)\Dependencies\lib;$(LibraryPath)**
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);$(MSBuild_ExecutablePath);$(VC_Libra

Note the line *$(ProjectDir)*. Since Visual Studio has been said to be a glorified makefile, it has its own unique keywords, all starting with a $ prefix. By Using the *ProjectDir*, this allows the project to be portable since it will be using a dynamic directory starting from the project's root folder rather than fixed (as in the root starting from your hard-drive)

In addition to including to providing access to headers and library files, we will also need to *link* specific static libraries when needed. This is done in the Linker's *Input* tab.



SampleECS Property Pages

Configuration: All Configurations Platform: x64

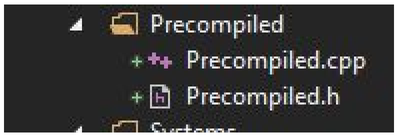
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System

Additional Dependencies	kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32
Ignore All Default Libraries	

Do pay the following in mind. You often will end up linking different libraries for Debug and Release configurations (Debug ones being larger and slower usually). So make sure you set the proper configuration.

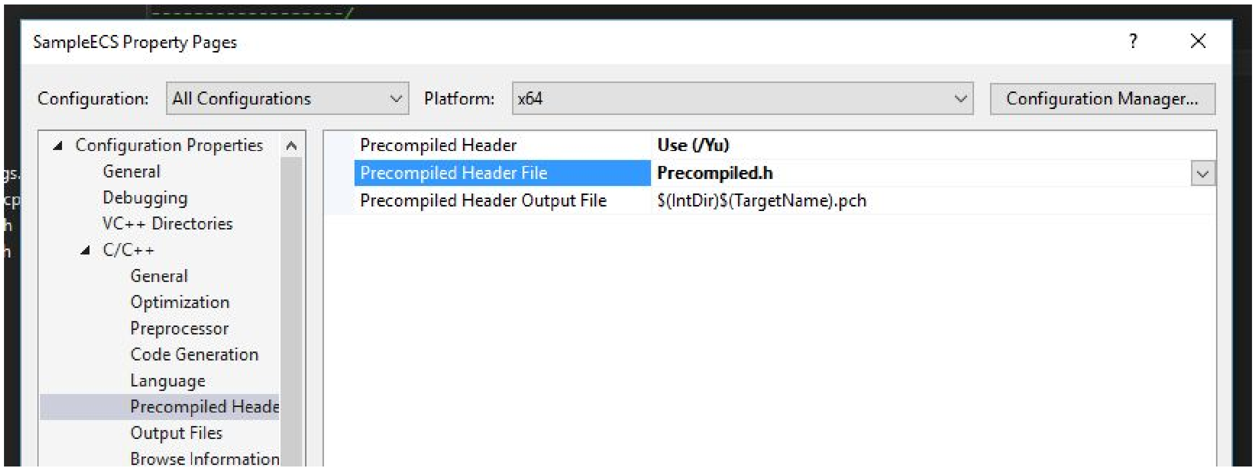
Precompiled Header

First, we will create a folder for our project to hold the precompiled header (and the accompanying compilation unit .cpp):



```
Precompiled
  Precompiled.cpp
  Precompiled.h
```

Next, let's configure the project to use it for *every* configuration.

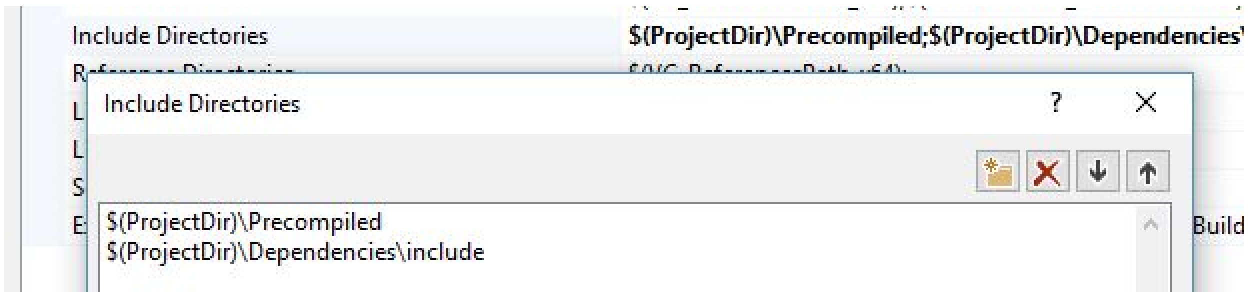


SampleECS Property Pages

Configuration: All Configurations   Platform: x64

Precompiled Header	**Use (/Yu)**
Precompiled Header File	**Precompiled.h**
Precompiled Header Output File	$(IntDir)$(TargetName).pch

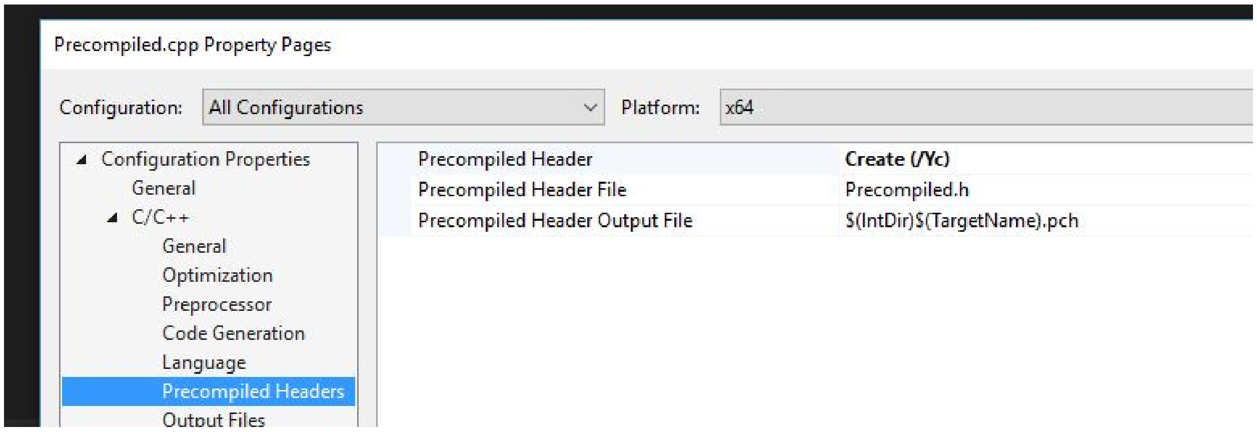
However as it is, it won't be able to find it since we haven't added it to the project's include paths:



Include Directories   **$(ProjectDir)\Precompiled;$(ProjectDir)\Dependencies**

```
$(ProjectDir)\Precompiled
$(ProjectDir)\Dependencies\include
```

Now right-click on Precompiled.cpp and examine its properties. On the C/C++ column, in the Precompiled headers tab we will mark it as the file creating the precompiled header with the *Create /yu* flag.

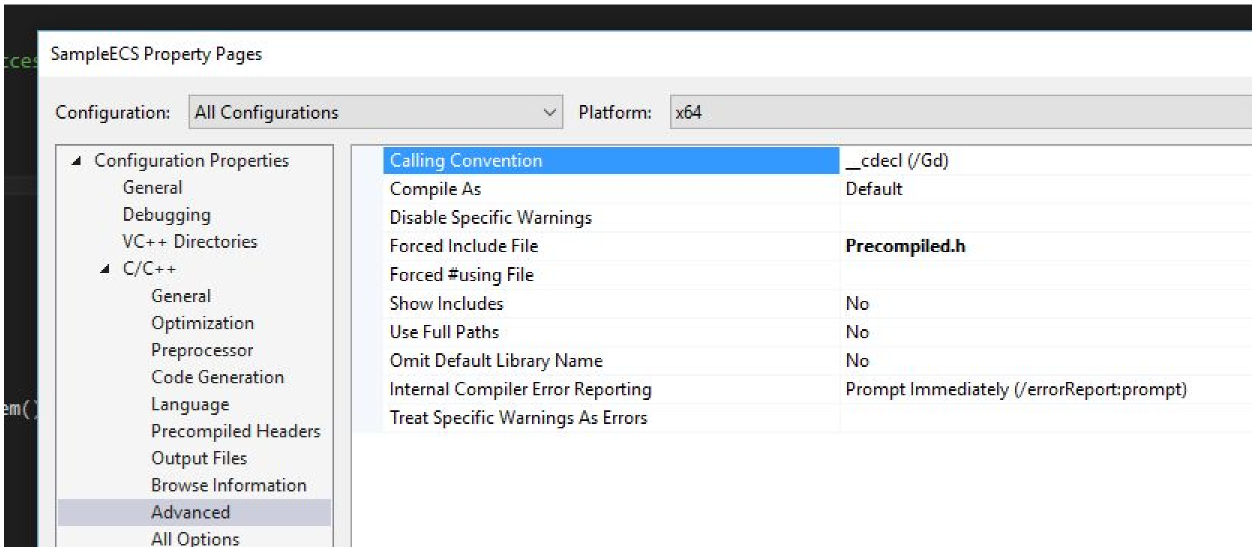

Precompiled.cpp Property Pages

Configuration: All Configurations Platform: x64

- Configuration Properties
  - General
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files

Precompiled Header	**Create (/Yc)**
Precompiled Header File	Precompiled.h
Precompiled Header Output File	$(IntDir)$(TargetName).pch

One last, optional step. If you don't want to have to manually include "Precompiled.h" at the top of every file, in the Advanced tab in the project properties you can set a forced include file. We will be setting it to Precompiled.h.


SampleECS Property Pages

Configuration: All Configurations Platform: x64

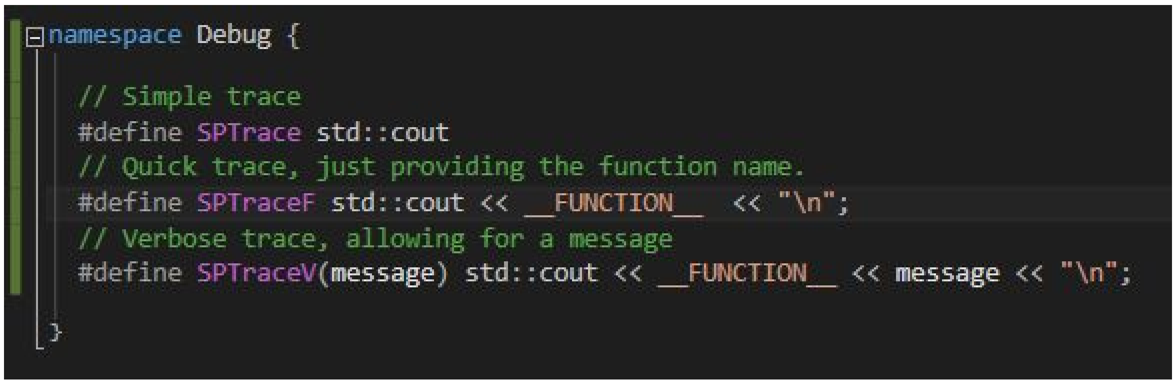
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options

Calling Convention	__cdecl (/Gd)
Compile As	Default
Disable Specific Warnings	
Forced Include File	**Precompiled.h**
Forced #using File	
Show Includes	No
Use Full Paths	No
Omit Default Library Name	No
Internal Compiler Error Reporting	Prompt Immediately (/errorReport:prompt)
Treat Specific Warnings As Errors	

Note: Make sure the Precompiled file is properly included in the VC++ Directories if you are having issues with it not finding the files!

Tracing

Properly logging your operations sequentially in order will pay dividends when you need to debug your code. It is not uncommon to provide macros or functions to quickly do this. I have gone ahead and wrote a few macros for you to get started. A more robust implementation can be found on another sample project, *Tracer*.

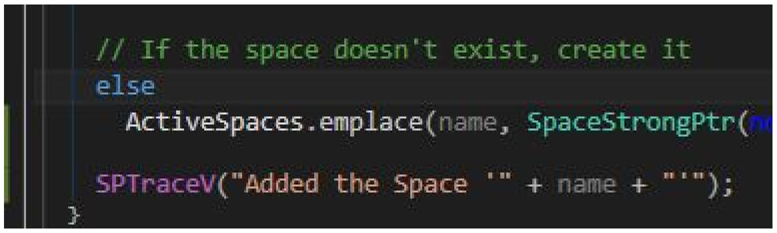


```
namespace Debug {

    // Simple trace
    #define SPTrace std::cout
    // Quick trace, just providing the function name.
    #define SPTraceF std::cout << __FUNCTION__  << "\n";
    // Verbose trace, allowing for a message
    #define SPTraceV(message) std::cout << __FUNCTION__ << message << "\n";

}
```

Example:



```
      // If the space doesn't exist, create it
      else
        ActiveSpaces.emplace(name, SpaceStrongPtr(

      SPTraceV("Added the Space '" + name + "'");
    }
```