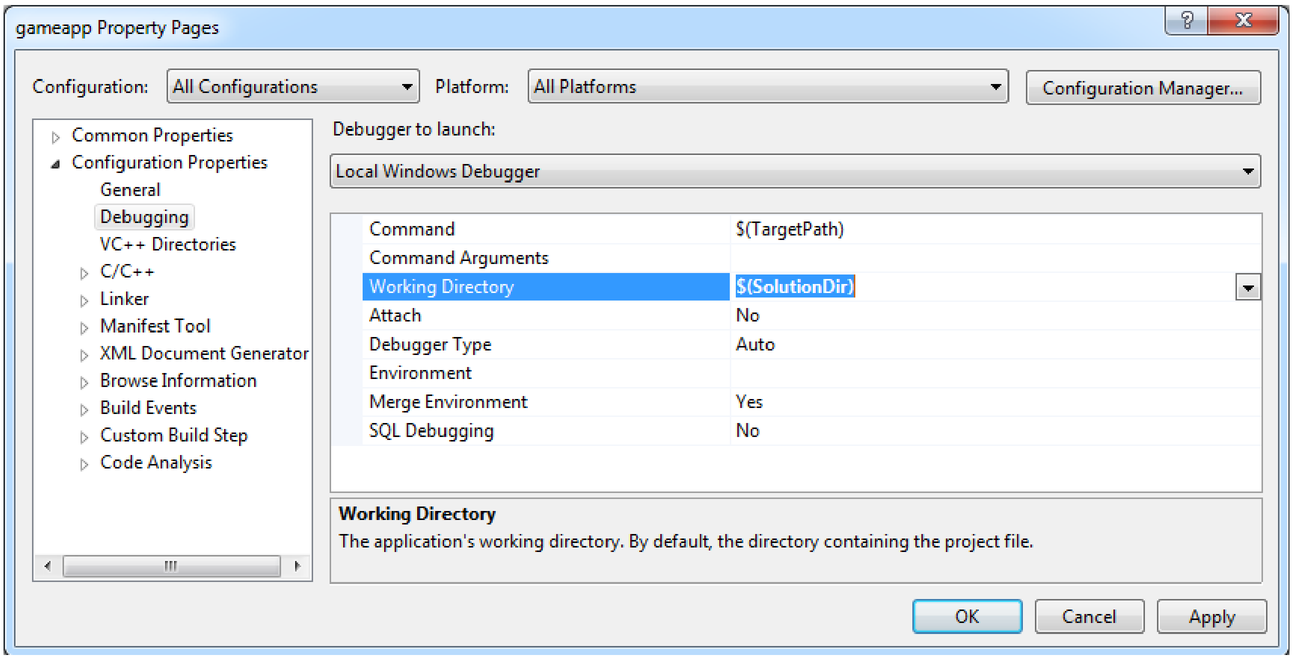
1. Preparations

Today is the first time our engine will actually load some data! This is why the application's working directory needs to be set appropriately. Otherwise, the needed files will not be found.

To double-check this, open the properties of the *GameApp*-Project and make sure that the working directory is set to *$(SolutionDir)*, as shown in the following screenshot:



Configuration: All Configurations
Platform: All Platforms
Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	
Working Directory	$(SolutionDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No

Working Directory
The application's working directory. By default, the directory containing the project file.

2. Timer

Implement all methods of the classes `Timer` and `PointInTime`. You will find the definitions of the classes in the source files *include/gep/Timer.h* and *src/gep/Timer.cpp*.

a) The `Timer` should be implemented using the `Win32` functions `QueryPerformanceCounter` and `QueryPerformanceFrequency`. The idea is that a `Timer` object remembers the current high-resolution counter value and the performance frequency upon creation, and performs all time calculations based on these initial parameters.

b) As the name suggests, the class `PointInTime` represents a given point in time. Its constructor gets a `Timer` as parameter, which can be used to calculate the respective time value.

Hint 1: You will have to add several data members to the classes `Timer` and `PointInTime`.

Hint 2: A few additional functions for easy time comparison would actually suit the class `PointInTime` quite well.

Hint 3: There is a unit test for the timer classes that you can use to try out your implementation.

3. Update Framework

Completely implement the class `UpdateFramework`, which is defined in *include/gepimpl/subsystems/updateFramework.h* and *src/gep/subsystems/updateFramework.cpp*. For your convenience, all functions you need to complete contain a `TODO`-comment and eventually some instructions.

a) Systems of the game application, which require periodic servicing, should be able to register at the `UpdateFramework` in order to receive respective callbacks. That's what the functions `registerUpdateCallback(...)` and `deregisterUpdateCallback(...)` are for. Implement these functions accordingly.

b) The actual game loop should be put into the function `run()`. The loop runs until eventually the function `stop()` is called.

It has to perform the following tasks in every iteration:

- Measure the time delta of the preceding frame
- Make the callbacks for all registered game systems
- Update the `ResourceManager`, `RendererExtractor`, and `Renderer`

Note: The `UpdateFramework` will be extended in order to support multi-threading in a later exercise. This is why all update-related procedures are bundled there.

4. GameApp

All that's left to do now is to implement a small game application!

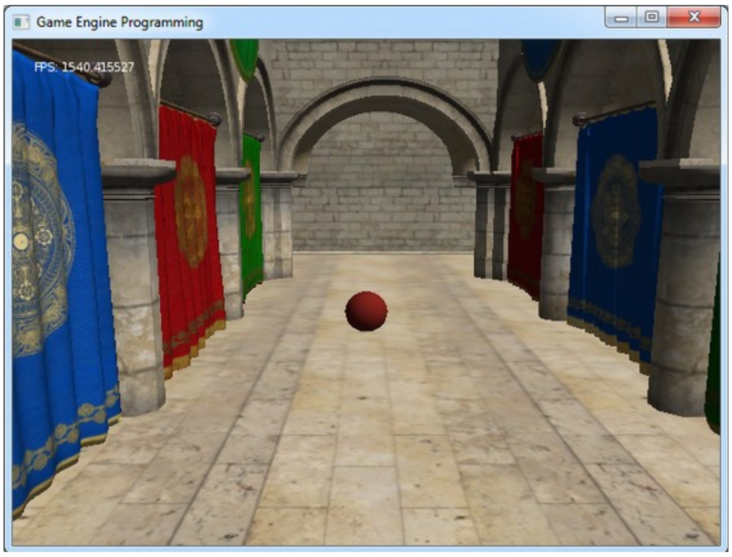
a) Get rid of all the `TODO's` in *gameapp.cpp* by implementing the respective functionalities.

b) Replace the dummy code in the function `main(...)` with real initialization and update code, in order to get the game running.

Hint 1: You will need `std::bind` in order to register the various callbacks. Try to find out how this works by yourself.

Hint 2: Don't forget to handle the eventually thrown exceptions in the main function properly.

When you're all done, you should see something like this, when you start your *GameApp*. → → →


Game Engine Programming
FPS: 1540.415527