B16 Software Engineering Laboratory

Victor Adrian Prisacariu*
Department of Engineering Science
University of Oxford

Hilary Term 2017

1 Introduction

The lab aims to reinforce aspects of structured and object-oriented programming covered in half of the B16 Software Engineering course, as well as demystify the use of an integrated development environment. You will design and implement some C++ classes, bring them together into a full application. You will explore important notions in object-oriented programming, such as inheritance and interfaces. You will learn how to include and link to your program third-party libraries and how to use such a library to generate graphical output, including animations.

The lab is designed to take place in the Software Lab, Windows section, using the Visual Studio 2012 development environment. The latest version, Visual Studio 2015 Community, can be downloaded for free from `http://www.visualstudio.com` to use with your own computer. The code can also be used with the Linux (GCC and GNU Make) and Mac OS X (Xcode) development environments, which can be obtained free of charge by anyone.

As all lab files are copied to the local drive C:, they will be deleted when you log out. Please save your work before you log out by, for example, saving it to the cloud.

1.1 General instructions

The lab notes contain a refresher/introduction of several concepts of C/C++ programming. The lab is divided in a number of **Tasks**, asking you to understand and answer theoretical questions, experiment with the programming environment, and create your own code and programs. Annotate in your **log book** answers to the questions, empirical observations about the outcome of the experiments, and any note taken during the development of your programs.

Once you are done coding and are satisfied with the results, print the source code of your programs as well as a snapshot of the program results. In the case your program generates a graphical output, press Alt + PrtSc (print screen) to capture a snapshot of the current window. The snapshot is saved to the clipboard. You can then paste it into Microsoft Office and save it to disk or print it.

1.2 Getting started

In order to get started with your Windows workstation in the Software Lab, please follow these instructions:

1. Before you log on, make sure that you will log on to the `ENG` domain (this should be shown in the last text box of the login dialog). Then log on using your lab username and password. If successful, you should land in Windows.

*This laboratory is derived from a former lab of Frank Wood, which in turn was derived from one by Andrea Vedaldi, which in turn was derived from one by Prof. Ian Reid and Dr. E. Sommerlade :).

2. Use Windows Explorer find `Documentation` folder under the `Computer` group in the left column browser. Open the `Documentation` folder and then the folder `B16` therein. Find the `install.bat` and double click it. This will copy the `b16-lab` package to your local `Documents` folder (note: this is *neither* the `Home Area` folder, *nor* the `Documentation` folder). Before you leave the installation window, take note of the installation location (something like `C:\Users\eng1234\Documents\b16-lab`). In the rest of the notes, we will refer to this path as `<b16-lab-path>`.[1]

3. Use Windows Explorer to navigate to the `b16-lab` package just installed. You can access this folder from the left browser (`Libraries` → `Documents` → `b16-lab`), or by pasting `<b16-lab-path>` into Windows Explorer address bar (see above). Inside you will find a copy of these notes in PDF format (`<b16-lab-path>\b16-lab.pdf`).

4. Inside `<b16-lab-path>` locate the *Visual Studio Solution* file `<b16-lab-path>\b16-lab.sln`. Double click it to open it in the Visual Studio Integrated Development Environment (IDE).

5. If this is the first time you start Visual Studio, it will ask you to select an *environment setting*. Since we are going to program in C++, select *Visual C++ Development Settings* and then press the button *Start Visual Studio*. **Wait for Visual Studio to start**: the first time it will take a few minutes! Once the setup is done, you should see a window similar to Figure 1.

Tip: If at any point during the lab you wish to **discard your changes and start over** you can run `install.bat` again. Note, however, that this will delete all of your changes!

2 Program development

In contrast to MATLAB, C and C++ are compiled languages: before being executed by the computer, the C/C++ code has first to be transformed into an executable format. First, a **compiler** translates each **source file** (a text file with extension `.cpp` containing the C/C++ code) into an **object file** (a file with extension `.o` containing the corresponding machine instructions). Then, a **linker** combines the objects files, as well as a number of third-party **libraries** (extension `.lib`), into a single executable program (extension `.exe`).

The traditional (some say archaic) way of performing these steps is to use a simple text editor of preferred flavour to create the input source code files, and then to manually type commands into a shell or command line interface to run the compiler and linker:

```
CC -c module1.cpp -o module1.o
CC -c module2.cpp -o module2.o
LINK module1.o module2.o -llib1 -llib2 -o program.exe
```

Here the first two lines invoke the compiler on two hypothetical source files named `module1.cpp` and `module2.cpp`, and the last line combines the outputs of the compiler an two libraries `lib1` and `lib2` into a single program fittingly called `program.exe`. The iteration of editing, compiling, linking and fixing errors is called a single development cycle, and can be repeated until you are happy with the results.

As you can imagine, invoking these steps over and over to correct your own programming mistakes is tedious, and many modern operating systems provide a **integrated development environment** (IDE) in which this can be done fairly painlessly. An IDE usually comprises an editor, a compiler, a linker, and a debugger (to find your mistakes in said executable).

On Windows, the most common IDE is **Microsoft Visual Studio**. Some useful features of Visual Studio are:

[1] If you run this lab on your machine, download and unpack the lab package from http://www.robots.ox.ac.uk/~victor/teaching/labs/b16/.

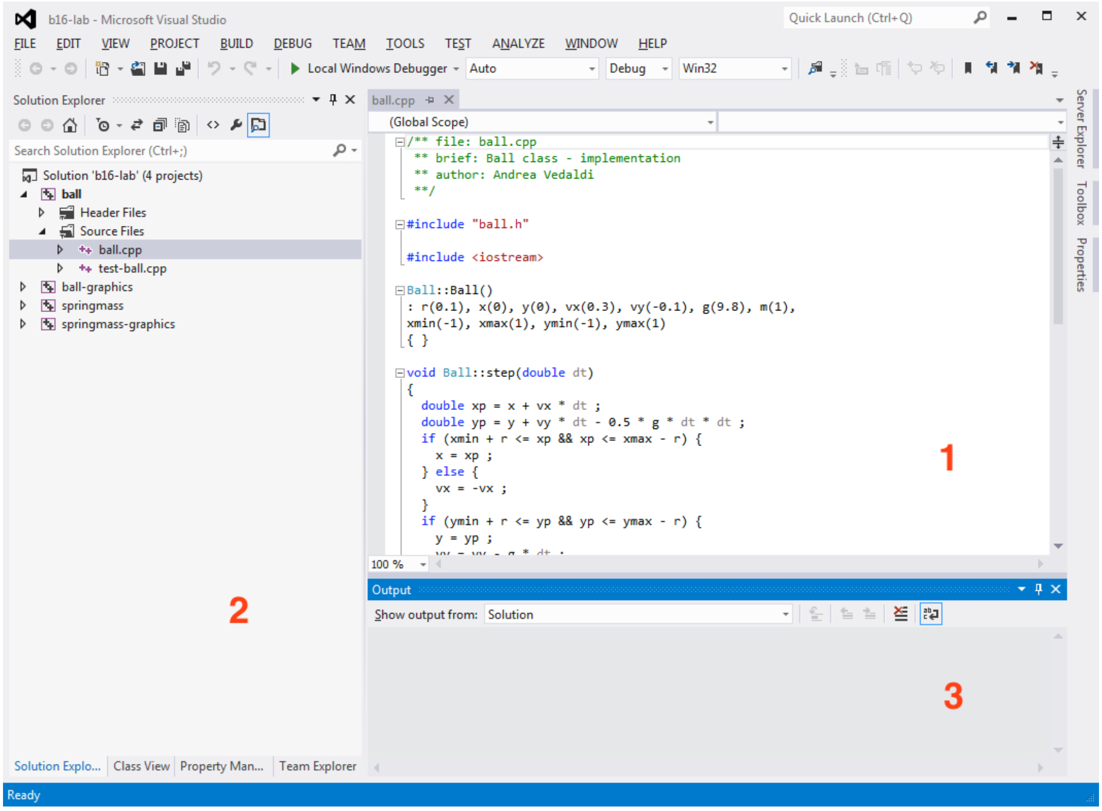

Figure 1: A screenshot of the Visual Studio integrated development environment. (1) The integrated editor (2) The *Project Explorer* shows all projects contained in the solution, and their accompanying files. (3) The output window for error messages.

1. A graphical debugging tool. Like any debugger, this tool enables you to step through a compiled program line by line, examine the contents of variables at each stage, trap errors, and set **breakpoints** where continuous execution will be paused. The debugger is accessed through the Visual Studio GUI and is integrated in the text editor, which makes its use particularly convenient.

2. A way of organising multiple related source files into a **project**, and multiple projects in a **solution**. Projects enable compilation, linking and generation of an executable without having to type all the command line gibberish; one nice related feature is that compiler errors appear as clickable links which then transport you right to the offending bit of source code. Within a solution, you can have several projects which share similar code.

3. It has an **online help** to obtain useful information such as descriptions of the functions contained in system libraries. You can access this help by pressing `F1` and search for a specific term, or by highlighting text in your source code or error messages and then pressing `F1`. The online help then searches for the highlighted text in the documentation automatically.

The IDE then shows a list of projects which correspond to the first assignments in this course, and should look similar to the screenshot in Figure 1. The project you are currently working on is highlighted in bold in the solution explorer. To enable a different project, highlight this project's name and select *Set as StartUp project* from the *Project* menu (or, in short, *Project* → *Set as StartUp project*). Among other things, this tells the debugger to start the program associated with the active project.

3 A simple program

A small application to perform a dynamic simulation (a ball falling and bouncing under gravity) has been written for you. We will know investigate in detail how this program works and learn about object classes, their use in constructing a program, and compiling and debugging the latter.

3.1 An overview

The bouncing ball application is implemented in the project named `ball`. This projects contains four source files: `simulation.h`, `ball.h`, `ball.cpp` and `test-ball.cpp`. You can view these files by clicking first on the `ball` project, which shows subfolders *Header Files* and *Source Files* (see Figure 1). In each of these you will find required files which open in the editor environment if double-clicked.

Start by inspecting the *header file* `ball.h`. As you will remember from lectures, a header file contains a set of related function declarations and class definitions, constituting a *module*. By packing this information together in a `.h` file, it is easy to *include* it in any source code file that may need to use the module. In this example, the `ball.h` header file defines of a single class, called `Ball`. This is the core of the bouncing ball simulation.

Examine now the way `Ball` class has been coded. The first thing you will notice are three member functions:[2] `Ball()`, `step()` and `display()` (we occasionally use the `()` suffix to emphasise that a name refers to a function). These functions serve the following purposes:

- `Ball()` is a *constructor*, a function used to *initialise* objects of class `Ball` when they are *instantiated* (created) in the program. In this case, the constructor specifies the initial position and velocity of the ball, among other things.

- `display()` prints a line of text containing the position of the ball. In practice, this is a pair of numbers x and y separated by a space. When the program runs, this output will be printed to a *console* or *terminal*. You can identify the console as a window with a black background that Windows opens when the program runs.

- `step(dt)` advances the simulation by `dt` seconds by taking the current state of the ball and integrating the differential equations of the motion to obtain the state `dt` seconds later. The integrator uses Euler method: if $\mathbf{g} \in \mathbb{R}^2$ is the gravity vector, $\mathbf{x}(t)$ the position of the ball at time t, and $\mathbf{v}(t)$ the velocity, the update is

$$\mathbf{x}(t+dt) = \mathbf{x}(t) + \mathbf{v}(t)\,dt + \frac{1}{2}\mathbf{g}\,dt^2,$$
$$\mathbf{v}(t+dt) = \mathbf{v}(t) + \mathbf{g}\,dt.$$

Still looking at the file `ball.h`, notice that, after declaring these three member functions, the class defines a number of *data members*, variables of type `double` storing the position, velocity, etc. of the bouncing ball simulation.

Task 1: Understand the implementation of the member functions. Look now at the `ball.cpp` file, containing the *implementation* of the member functions `Ball()`, `display()` and `step()`, and answer the following questions:

- What is the initial state of the ball when the object is created (instantiated)?

- How does `display()` prints information on the screen?

[2] A *member function* is a function that "belongs" to a class and can operate on instances of that class. In other object-oriented languages member functions are known as *methods*.

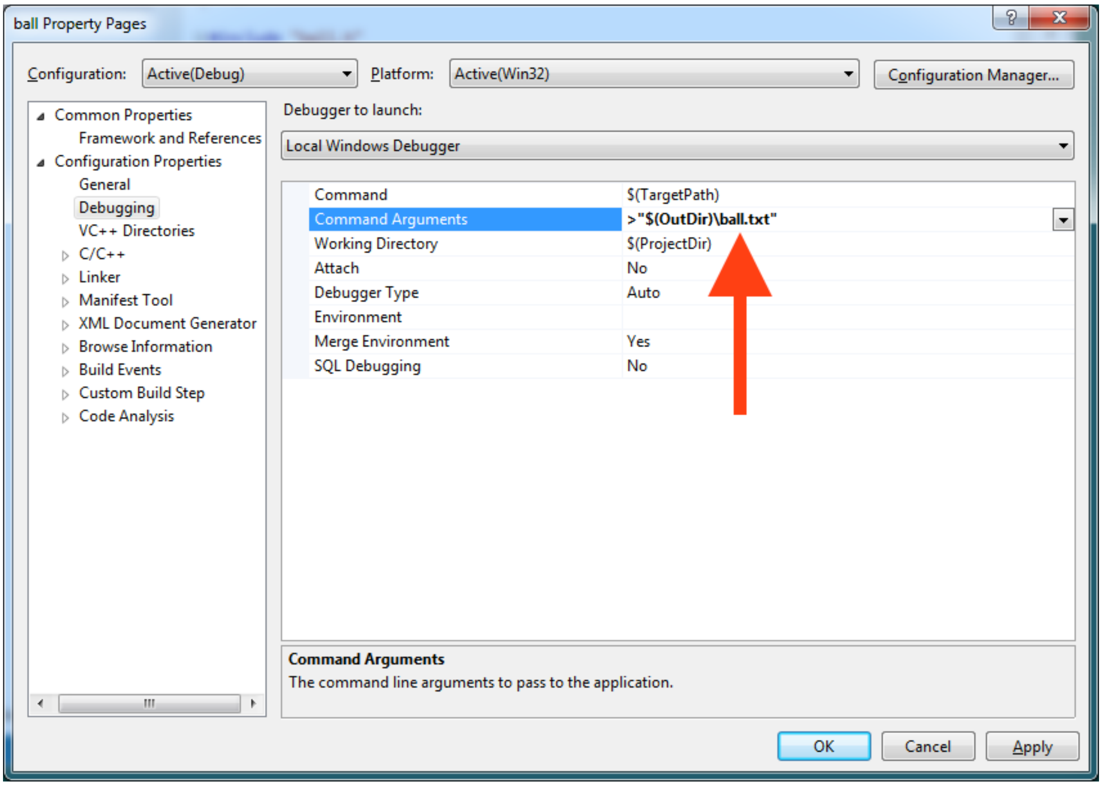

Figure 2: Slightly hidden, but there: The command line and redirection options. Observe the > symbol before the file name.

- The ball is constrained to bounce in the box $[-1, 1] \times [-1, 1]$. How is this handled by the integrator implemented in `step()`? Is the total (kinetic + potential) energy of the ball conserved by this integrator?

Now look at `test-ball.cpp` and answer the following question:

- What do you expect the output of your program to be?

Task 2: Running the program. The last question can be answered empirically, by compiling and running the program. In order to do so:

- Make sure that `ball` is the active project (see Sect. 2).
- Select *Start Without Debugging* in the *Debug* menu (*Debug* → *Start Without Debugging*, or *Ctrl+F5*).

This will automatically compile and link the program to create and run the executable `ball.exe` for you. A console window will pop up, displaying the output of the program.[3] Does it look reasonable?

3.2 Redirecting the output and visualising it in MATLAB

By inspecting a sequence of numbers it may not be obvious whether the program output is reasonable or not. This can be answered more easily by viewing the output graphically. Later

[3]If you select *Start Debugging* instead of *Start Without Debugging*, the program will run, but the console window will close immediately after the program ends, giving you the impression that nothing happened. If the problem persists, see Appendix C.

in the lab we will incorporate this functionality directly into the program; for now we will use a "trick" to export the program output to MATLAB and use the latter to do the plotting.

Task 3: Use redirection to save the textual output to a file. A way to save the textual output of the ball.exe program to a file is to *redirect* the output from the console to a file on disk. This is a convenient trick as it does not require to change the program at all.

To use redirection, open the project properties (from the menu select *Project → Properties* or press *Alt+F7*) and navigate to the *Configuration Properties → Debugging* submenu (see Figure 2). Redirection is specified by appending to the *Command arguments* the > symbol followed by the desired file name, e.g. > "$(OutDir)\ball.txt". Here the macro $(OutDir) is replaced automatically by VisualStudio with the directory used to store the executable. This directory is by default <b16-lab-path>\Debug.

Run the program again (*Start Without Debugging*). This time the output window is empty because the output has been redirected to the ball.txt file. Locate the file with Windows Explorer at <b16-lab-path>\Debug\ball.txt (in the same directory you will find the executable ball.exe too). Double click the file to visualise it in a text editor.

Task 4: Load the output in MATLAB and visualise it. Now that the data is the ball.txt file, it is easy to load it in MATLAB and plot it. Start MATLAB and issue the instructions

```
>> cd '<b16-lab-path>\Debug'
% OR: drag the Debug directory from Windows Explorer into MATLAB.
>> load ball.txt
>> plot(ball(:,1),ball(:,2))
```

Does the output seem reasonable?

3.3 Public, private, and protected class members

Now that you have a grasp of the basic structure of the program, we will investigate several details of the implementation. Inspect again the file ball.h and notice the following:

- The *declaration* class Ball consists of the word class followed by the name of the class being defined. The declaration tells the compiler that Ball is the name of a class. In this case, the complete declaration class Ball : public Simulation tells the compilers something more: the Ball class inherits from the Simulation class (more on this later).

- In this particular example, the declaration is immediately followed by the class *definition*, enclosed within { } and terminated with a semicolon ;. The definition is the blueprint of the class and the compiler requires it to know: (i) which operations (*member functions*) can be applied to objects of the class, and (ii) the list of *data members* comprising the class, so that the correct amount of memory can be allocated to store objects of the class. Note that the definition is divided into two parts, public and protected.

- In this example, the public section of the definition contains the member functions Ball(), step() and display(). Together, these three member functions embody what a *user* can do with objects of the class (namely, create a new object, advance the simulation, and display the state). They have public scope, and can therefore be called by any part of the program.

- In contrast, the scope of anything in the protected section is limited to functions that are member of the class or one of its descendants. In the Ball example, the protected section defines several *data members* of type double, representing the state of the ball (2D position and velocity) as well as other parameters of the simulation. Since these are protected, only Ball(), step() and display() can access them.

- Class members can also be declared in the protected section. In this case, they can only be accessed by members of the class, while class descendants are excluded.

Task 5: Member functions and separation of concerns. By isolating data members in the protected part of the class definition, the representation of the data (protected data members) can be separated from the operations that apply to it (public member functions). Answer the following questions:

- How should the class be changed so that a user could be able to get and set the position of the ball?
- The member functions of a class are often said to encode its "behaviour". Can you find a practical example demonstrating why separating the data representation from its behaviour is useful?

3.4 Implementing the class by defining the member functions

While the class `Ball` is defined in `ball.h`, the member functions are declared but not defined there. Their definition is found in the `ball.cpp` file. This file should be self-explanatory, except perhaps for the following facts:

- The **compiler directive** `#include "ball.h"` causes the compiler to insert on the fly the content of the `ball.h` file into the `ball.cpp` file. In this manner, the definition of the `Ball` class, contained in `ball.h`, is known during the compilation of `ball.cpp`. Similar directives can be used to include and the use system libraries. An example is the directive `#include <iostream>` which includes the `iostream` module of the C++ Standard Template Library (STL). The latter module provides input output functionalities (`std::cout` and similar).[4]
- All member function definitions are prefixed with the string `Ball::`. This tells the compiler that the function begin defined is a member of the class `Ball` as opposed to a "global" function.
- The constructor `Ball::Ball()` is defined as

```
Ball::Ball()
: r(0.1), x(0), y(0), vx(0.3), vy(-0.1), g(9.8), m(1),
xmin(-1), xmax(1), ymin(-1), ymax(1)
{ }
```

  The long list of parenthesised expressions after the : symbol is known as an *initialisation list*. Its purpose is to construct (initialise) the various data members, for example by choosing an initial value. A similar effect could be obtained by assigning the variables explicitly in the constructor body:

```
Ball::Ball() {
  r = 0.1 ;
  x = 0 ;
  // etc.
}
```

  For elementary data types like `double` using an initialisation list or multiple assignments is roughly equivalent; however, if a data member is an instance of a user defined class, the assignment operator `=` may not be defined for it. In this case, the only option is to initialise the class in the initialisation list, as the latter can use the class constructor (always defined) instead of the assignment operator `=`. We will see an example of this fact later.

[4] Note the slightly different syntax: when the filename is enclosed in quotes, as in `#include "ball.h"`, the file `ball.h` is searched in the same directory of the file `ball.cpp` which is including it. When the filename is enclose in the angular brackets, as in `#include <iostream>`, the file `iostream` is searched in a number of system directories which contain library header files.

- The statement `std::cout<<x<<" "<<y<<std::endl ;` in the implementation of `display()` deserves some further attention. The operator `x<<y` is meant to pass the object `y` as input to the object `x`. Here, the receiving object is `std::cout`, an *output stream* (it is defined in the included file `<iostream>`). `std::cout` dumps its input to the system console or terminal, and thus can be used to print information on the screen. The symbol `std::endl` represents an end-of-the-line character.

Task 6: Understanding constructors. The constructor `Ball()` does not allow the user of the class to *choose* a custom initial position or velocity of the ball. Design an alternative constructor that would allow to do so upon class instantiation.

3.5 Using a class by instantiating it and calling its member functions

At this stage you should have a good understanding of the class `Ball`. It is now time to put it to work. To this end, inspect the source code in `test-ball.cpp`.

After including the `ball.h` header file, `test-ball.cpp` defines a function called `main`. This is the default *entry point* of the program, i.e. the function that the operating system calls when the program is launched. Arguments from the operating system are passed to the program through the variables `argc` (the number of arguments) and `argv` (the arguments as an array of strings), making a program a black box to the operating system as a function definition is to a programmer. The `main` function returns to the operating system an error code, in this case the constant `0` (i.e. no error). In the implementation of `main` we note the following:

- The variable `dt` is defined to contain the simulation step size. By defining this as `const`, the compiler knows that it is an hard coded parameter that cannot be changed during the execution of the program.

- The statement `Ball ball ;` defines a local variable `ball` of type `Ball`. The compiler automatically creates an *instance* of class `Ball`, calling implicitly the constructor `Ball()` to initialise the object. The variable `ball` can then be used to refer to this instance. This variable has a *local scope*, i.e. it is visible only in the context of `main()`.

- The class members are accessed via the . (dot) operator. For example, `ball.step(dt)` calls the `step` member function in `ball`, advancing the simulation by the specified amount. Data members can be accessed as well; for example `ball.x` would access the variable `x` in `Ball()`. However, `x` was declared a `protected` member of the class, so in practice this causes a compilation error.

- The `for` loop gets a fixed number (100) of iterations. At each iteration, the member functions `step` and `display` are called to advance the simulation and dump the location (x, y) of the ball on the screen.

Task 7: Programming principles. Answer the following questions:

- What is the distinction between a class declaration, definition, implementation, and instantiation?

3.6 Compiling, testing, and debugging the program

Now that you have a full understanding of the source code, we will investigate in more detail how to compile, run, and debug the code.

Task 8: Compiling the program. In order to compile the program:

- Select the `ball` project in the Visual Studio environment and set it as the StartUp project to select that project as the active one ($\rightarrow$ *Project* $\rightarrow$ *Set as StartupProject*). The name of the project should be rendered bold.

- Build the project (*Build* → *Build ball*).

This runs the C++ compiler and linker on your program. After a few seconds translation will be complete and should leave you with a summary of the process in the error window:

```
1>ball - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Navigate once more to the output directory `<b16-lab-path>\Debug` with Windows Explorer; you will see that a **executable file** `ball.exe` has been created. This is your program, ready to be launched (try double clicking on it!). As we have seen before, the program can be run from the IDE by selecting *Debug* → *Start Without Debugging*, or *Ctrl+F5*).

Task 9: Try the debugger. The **debugger** component of Visual Studio is very useful. It enables a programmer to step through a program as it is running, pausing, examining variable values, etc. To run the debugger on your executable `ball`:

- Place a **breakpoint** on the line where the Ball object is instantiated. Similar to MATLAB, you click left to the start of the line. A red dot should appear.
- Select *Debug* → *Start* or hit *F5*.
- If you have changed the source, a small window will pop up, asking you whether you want to build it. This is a good idea. Keeping this window from appearing, too, and you can tick the little *Do not show this dialog again* box.
- Placing the cursor over the icons either on the debugger toolbar will tell you briefly what each does. The most important are summarised below (see Figure 3):
  - **Stop at:** insert a breakpoint at the location of the cursor in the source code. Program execution will stop at this point giving you the chance to examine variables, etc.
  - **Step into:** execute the current line of code, but if this involves a function call, then stop at the first line of the function.
  - **Step over:** execute the current line of code then stop again.
  - **Start/Continue:** start debugging or continue execution from the current location (until another breakpoint is met).

You can also halt the program mid-flow by hitting Control-Break, and this will bring you into the debugger at the current line of execution.

Try out the various options. In particular try to discover how to display the value of a variable (e.g. `ball`). Once you have done this, step through the program one line at a time examining how it changes during the `for` loop. Also try inserting a breakpoint and continuing execution to observe the program stopping at the breakpoint.

3.7 Inheritance and virtual functions

Throughout the lab, we will not only define brand new classes, but also *derive* new ones form existing ones through the mechanism of *inheritance*. Inheritance allows to *specialise* a given class by extending or customising it for a given purpose. Consider the following example:

```cpp
#include<iostream>
class Person {
public:
  void wakeUp() { std::cout<<"Good morning!"<<std::endl; }
} ;
class Worker : public Person {
public:
  void goToWork() { std::cout<<"Driving."<<std::endl; }
} ;
```

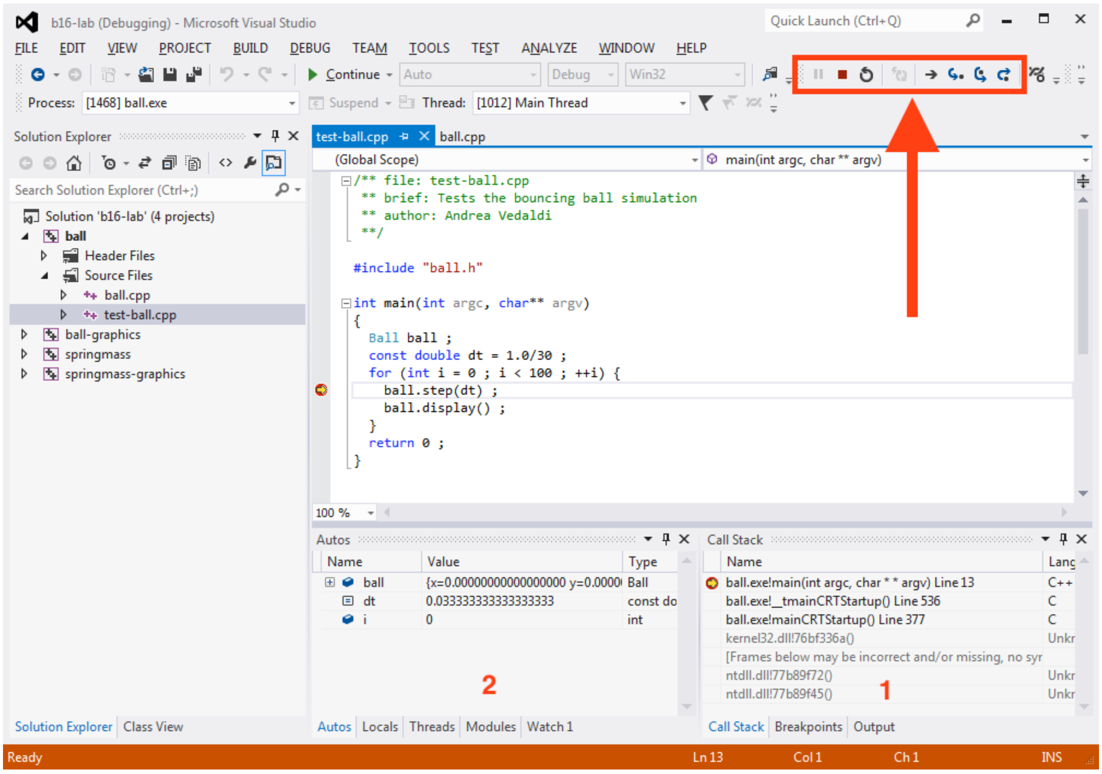

Figure 3: The Debugger toolbar (see arrow) pops up only during debugging. (1) shows the variables currently in scope, and (2) shows the current call stack, which can be navigated by double clicks, similarly to the `dbup` and `dbdown` commands in MATLAB. The red dot marking one of the line in the editor is the location of the breakpoint, and the yellow arrow the current location (PC) in the program. To reach this particular state, the *step into* icon was pressed once after the program stopped at the breakpoint.

Here the derived class `Worker` has two member functions: `goToWork`, as given in its definition, and `wakeUp`, inherited from the base class `Person`. An object of type `Worker` is *also* of type `Person`, so it can be used whenever `Person` can. For example:

```
void ringBell(Person &aPerson) {
  aPerson.wakeUp() ;
}
void main(int argc, char** argv) {
  Worker aWorker ;
  ringBell(aWorker) ; // prints "Good morning!"
}
```

In addition of *adding* new members, inheritance can be used to *override* existing ones, modifying the behaviour of the base class. For example:

```
class Person {
public:
  void wakeUp() { std::cout<<"Good morning!"<<std::endl; }
  void goToWork() { std::cout<<"No way!"<<std::endl; }
} ;
class Worker : public Person {
public:
  void goToWork() { std::cout<<"Driving."<<std::endl; }
} ;
```

In this case, a `Worker` modifies the behaviour of the more generic `Person` by redefining the function `goToWork`. However, the old behaviour is *still* associated to objects of class `Worker` when seen as `Person`. For instance:

```
void timeToGo(Person &aPerson) {
  aPerson.goToWork() ;
}
int main(int argc, char** argv) {
  Worker aWorker ;
  timeToGo(aWorker) ; // prints "No way!"
  aWorker.goToWork() ; // prints "Driving."
  return 0 ;
}
```

What happens is that the `timeToGo` function accepts a `Worker` as input, but treats it exactly as it was of type `Person`. This can be modified by defining the functions in the base class to be *virtual*:

```
#include<iostream>
class Person {
public:
  virtual void wakeUp() { std::cout<<"Good morning!"<<std::endl; }
  virtual void goToWork() { std::cout<<"No way!"<<std::endl; }
};
class Worker : public Person {
public:
  void goToWork() { std::cout<<"Driving."<<std::endl; }
};
void timeToGo(Person &aPerson) {
  aPerson.goToWork() ;
}
int main(int argc, char** argv) {
  Worker aWorker ;
  timeToGo(aWorker) ; // prints "Driving."
  aWorker.goToWork() ; // prints "Driving."
  return 0 ;
}
```

Now the new definition of the `goToWork` member function is used when the object is referred to as a `Worker` or as a `Person`.

3.8 Interfaces

A particularly important application of virtual member functions is the definition of *interfaces*. An interface is an "empty class" that specifies a set of operations (virtual member functions) without providing *any* implementation. Defining an implementation for those functions is left to any particular class that derives the interface.

Task 10: Understanding interfaces. The class `Ball` (see `ball.h`) inherits from the interface `Simulation`, whose definition (see `simulation.h`) is

```
class Simulation {
public:
  virtual void step(double dt) = 0 ;
  virtual void display() = 0 ;
} ;
```

Thus `Simulation` *declares* two virtual member functions `step()` and `display()`. The `= 0` suffix tells the compiler that it should not expect `Simulation` to provide a *definition* of these functions. Rather, this is left to the compiler. Answer the following questions:

- Consider the function

```
void run(Simulation & s, double dt) {
  for (int i = 0 ; i < 100 ; ++i) { s.step(dt) ; s.display() ; }
}
```

  Can you replace the for loop in the `main` function defined in `test-ball.cpp` with the instruction `run(ball, dt)`?

- What would happen if `Simulation` did not declare `step` and `display` to be virtual? Would `run` still work as expected? If not, what would happen?

4 Building your own application

The main task of the lab is to create another more complicated dynamic simulation; a mass-spring system consisting of two masses and one spring (though we would like to bear in mind how this could be extended to a system of several masses and springs). A skeleton project `springmass` has been created for you to give you an idea of how the classes and files might be organised. You will need to decide what responsibilities the classes will have, and how they will interact. Before you proceed, do not forget to make the `springmass` project active (*Set as StartUp Project*).

Task 11: Complete the Mass class. Start by completing the implementation `Mass` class by implementing the member functions `getEnergy()` and `step()`. The following definition is given to you in `springmass.h`:

```
class Mass
{
public:
  Mass() ;
  Mass(Vector2 position, Vector2 velocity, double mass, double radius) ;
  void setForce(Vector2 f) ;
  void addForce(Vector2 f) ;
  Vector2 getForce() const ;
  Vector2 getPosition() const ;
  Vector2 getVelocity() const ;
  double getMass() const ;
  double getRadius() const ;
  double getEnergy(double gravity) const ;
  void step(double dt) ;

protected:
  double xmin ;
  double xmax ;
  double ymin ;
  double ymax ;

  Vector2 position ;
  Vector2 velocity ;
  Vector2 force ;
  double mass ;
  double radius ;
} ;
```

The `Mass` class is an improved version of `Ball`, where:

- `setForce(f)` and `addForce(f)` set/accumulate a force `f` in the internal force accumulator variable `force`.

- `step(dt)` updates the position and velocity of the based on the current value of `force` and the other parameters, similarly to the `step` function member of `Ball`.

Differently from `Ball`, the class is making use of the `Vector2` class, also defined in `springmass.h`, in order to simplify basic operations on 2-dimensional vectors. Note that `Vector2` is provided for the sake of the example only; in practice, you would use a fully-fledged off-the-shelf C++ math library instead, like Eigen (`http://eigen.tuxfamily.org`).

Most member functions of `Mass` have already been defined for you in `springmass.cpp`. Now open that file and complete the implementation by writing the code of the `step` function.

Task 12: Complete the Spring class. Once you are satisfied with your implementation of `Mass`, move to the implementation of `Spring`. In writing this implementation it is helpful to think how `Spring` can be used in combination with `Mass` in the simulation.

- Start by the fact that a spring exerts two opposite forces $\mathbf{F}_1$ and $\mathbf{F}_2 = -\mathbf{F}_1$ at its two extrema $\mathbf{x}_1$ and $\mathbf{x}_2$, where

  $$\mathbf{F_1} = k(l - l_0)\mathbf{u}_{12}, \quad l = \|\mathbf{x_2} - \mathbf{x_1}\|, \quad \mathbf{u}_{12} = \frac{1}{l}(\mathbf{x_2} - \mathbf{x_1}).$$

  Thus, the class `Spring` should have at the very least a member `double naturalLength` to store l_0 and a member `double stiffness` to store k.

- Add also a damping factor d (`double damping`). Similar to stiffness, damping also exerts two opposite forces $\mathbf{F}_2 = -\mathbf{F}_1$ along the spring, but in this case the forces are proportional to the elongation velocity:

  $$\mathbf{F_1} = d\,\mathbf{v}_{12}, \quad \mathbf{v}_{12} = ((\mathbf{v}_2 - \mathbf{v}_1) \cdot \mathbf{u}_{12})\mathbf{u}_{12}$$

  where $\mathbf{u}_{12}$ is defined as before and $\cdot$ denotes the inner product (see `dot(Vector2,Vector2)`).

- Note that the `Spring` constructor takes two pointers to the masses connected to its extrema:

```
Spring::Spring(Mass * mass1, Mass * mass2, double length, double stiffness, double
    damping)
: mass1(mass1), mass2(mass2), naturalLength(length), stiffness(stiffness), damping(
    damping)
{ }
```

  Add suitable data members `mass1` and `mass2` to be able to store this information in the class.

- Implement the member function `getForce()` to compute the spring force $\mathbf{F}_1$ (note that the force $\mathbf{F}_2$ is the opposite). Since `Spring` contains pointers to the masses, it can use the `Mass` member functions `getPosition()` and `getVelocity()` to obtain the information required of this calculation.

Task 13: Design and implement SpringMass. Once `Mass` and `Spring` are complete, it is time to implement the simulation by constructing a class `SpringMass` that represents the simulated "world". In this case you are expected to work out part of the design of a class itself.

The world contains two masses and a spring connecting them (or, more in general, n masses and m springs). A minimal solution, therefore, must store `Mass mass1`, `Mass mass2`, and `Spring spring` as data member of `SpringMass`. Similarly to the `Ball` class encountered earlier, furthermore, `SpringMass` inherits from the `Simulation` class and must implement the `step()` and `display()` methods.

- Start by implementing the interface necessary to setup the simulation. One way to do this is to create two masses and a spring in the main program and then pass the latter to an instance of `SpringMass` by pointer or reference. Design and implement suitable methods in `SpringMass` or, alternatively, redesign the constructor to be able to initialise the simulation in this manner.

- Implement the method `step()` to advance the simulation. To do this:
  - Go through both masses and set the initial force to be gravity (or zero) by using the `setForce` function of mass.
  - Examine the spring and compute the forces $\mathbf{F}_1$ and $\mathbf{F}_2$ that it applies to the connected masses. Accumulate these forces to the respective masses by using the `addForce` function of `Mass` (by doing so, the spring forces will be summed to gravity).
  - Go through all masses again and update their state (position and velocity) by calling `step()` on them.
- Define the method `display()` to dump the sate of all mass locations.
- **Optional.** Think whether your design would easily extend to the case of more than two masses and more than one spring in the simulation. If not, what could you change?

Task 14: Initialise and run the simulation. Modify the file `test-springmass.cpp` to run your simulation. Write code to setup a new `SpringMass` instances with two masses and one spring (or a more complex configuration) using the interface that you just designed and implemented.

Task 15: Redirect the program output to a file and visualise it in MATLAB. Try the same trick we used for the `Ball` simulation in order to redirect your program output to a file, load it in MATLAB, and visualise it. Note that in this case more than one mass per line of output should be included.

5 Graphics

The dynamic simulation you have created would be much more interesting if it could be graphically viewed in "real-time" rather than as a static plot. Displaying graphics on the screen requires interfacing to the graphics subsystem built in your machine. At the core of this subsystem there usually is a dedicated *graphical processor unit* (GPU), a specialised hardware component that can execute graphical operations, such as drawing a line, very efficiently.

Rather than coding the GPU directly, programmers normally use an abstraction known as a *device driver*, that lives in the operating system. Most graphic drivers support one or more standard interfaces, made available to the C/C++ programmer as libraries. In this manner a program can work without modification with GPU of different vendors (e.g. nVidia, ATI, Intel, or PowerVR).

The graphics library we will use is called *OpenGL*, an industry standard implemented in many environments (e.g. Windows, Mac OS X, Linux, iOS and Android). By using OpenGL, a program can use the powerful graphic accelerators built into modern computers and mobile gadgets. Although we will focus on simple 2D drawing capabilities, OpenGL allow accessing all the features required to generate complex 3D graphics as well. You can view the OpenGL manual by going to `http://www.opengl.org/sdk/docs/man2/` in Internet Explorer.

In general, it is very common to rely on system and third-party libraries when coding complex programs. Knowing how to use them optimally is a key programming skill. This is not always a simple matter. For example, using OpenGL requires dealing with several details, such as opening a window to display the output, handling events such as a user resizing the window, and updating the window content at a regular interval to generate an animation. We shall approach this problem in an organised manner, using a few classes and inheritance to help us along.

5.1 Understanding graphics and animations

OpenGL is a fairly complex library. In order to help you getting started with it, the lab package provides a `graphics.h` module. This module contains a class `Figure` that simplifies

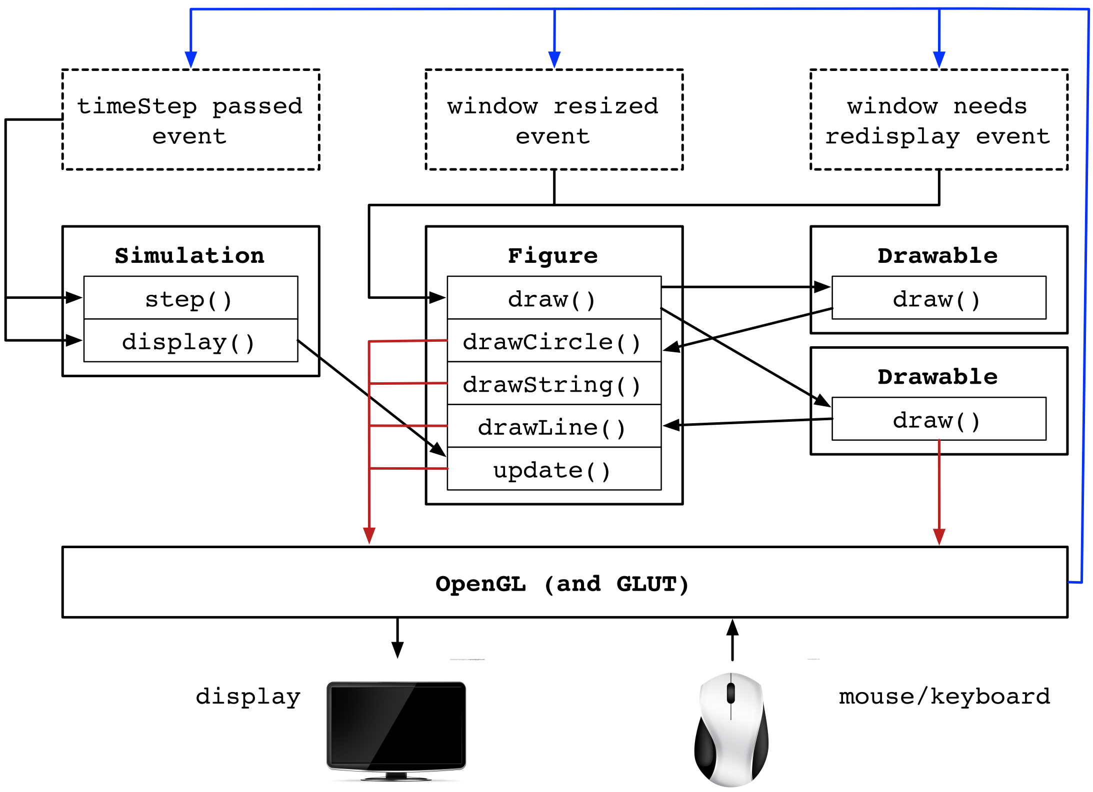

Figure 4: Overview of the graphical subsystem. `Simulation`, `Figure` and `Drawable` show instances of objects of the corresponding classes. In practice, the same object can implement two or more classes by multiple inheritance.

creating figures and drawing into them. Furthermore, the lab package contains an example project `ball-graphics` that demonstrates using `Figure` to visualise the bouncing ball simulation in real time.

Task 16: Understand how to open a window and draw in it. Make the `ball-graphics` project active (*Set as StartUp Project*). Then navigate in Visual Studio to the file `graphics.h` file and double click to inspect it. This module does a few things:

- It includes the OpenGL header files (`glut.h`). This means that, by including `graphics.h`, your program will have access to the OpenGL functions as well. GLUT is an utility library, companion to OpenGL, which handles windows and events — more on this later.
- It declares a function `run()` that takes as input an object of type `Simulation` and a `timeStep`. This function never returns; instead, it enters a loop that handles the stream of events from the graphical user interface (e.g. a window being resized). It also runs the simulation by repeatedly calling the member functions `step` (to advance the simulation) and `display` (to update a corresponding figure).
- It defines a class `Figure` which simplifies opening a window and drawing in it.

The class `Figure` handles the basic workflow of displaying a window on screen and updating its content during an animation. It also provides a few functions to draw basic primitives that are simpler to use than OpenGL directly (but you *can* still use OpenGL with it). The workflow is summarised in Figure 4 and described next.

Upon creation of an instance, the class `Figure` opens a new window on the screen. It also tells OpenGL to call the class member function `draw()` whenever the window content must be drawn on screen. OpenGL decides to (re)draw the content in response to events such as: the window is first displayed on the screen, the window is resized by the user, or the window is occluded and then displayed again as the user manipulates other windows. While most of these events are caused by the user, your program can also request the window to be redrawn, for example because the content of the figure should be updated to reflect the new status of the simulation. Explicitly requesting a window update is done by calling the `update()` function of `Figure`.

Given that OpenGL calls `draw()` in `Figure` to draw the window, how does `Figure` knows *what* should be drawn? By default, the `draw()` function in `Figure` starts by painting a white background and a 10×10 reference grid. It then goes through a list of other drawable objects that the user has "attached" to the figure. Each such object inherits from the class `Drawable` and as such has its own `draw()` member function, different from the one of `Figure`. An object is attached to the figure list by using the `addObject` function of `Figure`. To summarise, you must: define an object of class `Drawable`, define its function `draw()` (this does the actual work), and add an instance of that object to the figure by calling `addObject`.

This leaves us with the design of the function `draw()` in the `Drawable` object. In order to paint something on the screen, your program must use OpenGL function calls. While this is not particularly difficult, `Figure` provides functions that simplify drawing basic primitives: `drawString`, `drawCircle` and `drawLine`. The meaning and usage of these functions should be obvious from their declaration in `graphics.h`. It is however very instructive to inspect their implementation in `graphics.cpp` to know how to call OpenGL directly instead. You can then refer to OpenGL reference manual or, preferably, to one of the many tutorials available online to go past (see for example http://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_Introduction.html).

Task 17: Understand how to generate an animation. So far, it was shown how to use `Figure` and `Drawable` objects to draw figures on the screen. The last piece of the puzzle is to understand how to generate an animation by iteratively calling the simulation code and redrawing the window content. This is done by calling the function `run` declared in `graphics.h`. This function takes a `Simulation` object implementing the `step` and `display` functions. It also takes a time step `timeStep` as a parameter and tries to call the `step` and `display` functions with that frequency. Here `step` will advance the simulation as before; however, `display` must be changed to update the figure rather than printing a message on the screen. As seen above, the latter is done by calling `figure.update()`.

Task 18: Understand how this is implemented by using inheritance. Once how to solve the previous two tasks is clear, open `test-ball-graphics.cpp` and understand how this has been done by defining a `BallDrawable` class, inheriting from `Ball` and `Drawable`. This is called **multiple inheritance**, and can be used to create "hybrid" classes which combine more than one parent classes. Look in particular at the implementation of the member functions `draw()` and `update()`. Compare this with Figure 4 and answer the following questions:

1. Which objects are of type `Simulation` and which of type `Drawable`?
2. Which objects are creating an instance of `Figure` and when?
3. Which objects are added as `Drawable` to the figure and when?
4. Which objects implement respectively: `display`, `draw`, and `update`?
5. Which different purposes do these functions serve?

Note also that the very first line of the `main` function in `test-ball-graphics.cpp` contains a call to `glutInit()`. This function initialises GLUT and OpenGL, making them ready for use. Do not forget to use it in your code later!

Task 19: Compile and run code using the graphics library. Visual Studio must be instructed to *link* your program to OpenGL before it can be compiled correctly. In fact, this example code uses an additional utility library known as GLUT (`http://www.opengl.org/documentation/specs/glut/spec3/spec3.html`) which adds to OpenGL functions to manipulate windows and user events. Linking to GLUT is sufficient, as the latter causes OpenGL to be linked as well.

A further complication is that, differently from OpenGL, GLUT is *not* part of the standard Windows SDK. This means that it has to be downloaded and installed as a *third part library*. In practice, your `<b16-lab-path>` folder already contains a copy of GLUT (`<b16-lab-path>\support\GLUT`). Furthermore, the project `ball-graphics` is already setup to correctly link to the latter, so compiling and running the program should work without a problem (try this now). If you are curious, the process is described in some detail in Appendix A.

5.2 Adding a graphical display to the mass-spring simulation

Now that you have seen how to add a graphical display to `Ball`, it should be straightforward to do so for `SpringMass`. Do not forget to make the `springmass-graphics` project active.

Task 20: Create a graphic version of SpringMass. In the `test-springmass-graphics.cpp` file, derive a `SpringMassDrawable` class from `SpringMass` in a manner similar to the way `BallDrawable` was derived from `Ball` (see `test-ball-graphics.cpp`). In particular, override the `display()` function to draw all the masses and all the springs (as simple sticks connecting the masses). Then run the simulation physically by using the `run` function.

5.3 Further extensions

If all this was too easy and you have time left, try the following:

Task 21: Compute and display the energy of the spring-mass system. Design and add `getEnergy()` member functions to your classes for the masses and the springs. Then modify the `step()` function of the `SpringMass` simulation to compute and remember the energy of the system, by summing the energy of all masses and springs. Make this value available to the user by adding a function `getEnergy()` to this class as well. Finally, extend the `draw()` method of `SpringMassDrawabe` to overlay this information on the figure (use the `drawString` function of `Figure` to do so). Then answer the following questions:

- What happens to the energy as the simulation progresses? Why?
- Now set the damping factor of the spring to zero. What happens now to the energy? Is this result correct? If not, what may be the problem?

Task 22: Generalise the mass-spring simulation to multiple masses. Extend the `SpringMass` simulation to handle multiple masses and springs, with an arbitrary topology. This requires handling a *graph* of springs connecting the different masses, but it is otherwise a rather simple task.

Task 23: Extend the simulation to a 3D world. The physics extends trivially to the 3D case by adding a third coordinate. Develop a `Vector3` version of `Vector2`, then modify the `Spring`, `Mass`, and `SpringMass` simulation to handle the third coordinate.

Task 24: Visualize a 3D simulation. OpenGL makes it very easy to draw three dimensional graphics as well. Understand how to do so from the WWW, staring from the tutorial and documentation pointers include before, and then using Google to fill in the remaining information. Note: most programmers find the WWW an invaluable tool to extend their knowledge, sometimes preferable to books. Learn how to use this tool best!

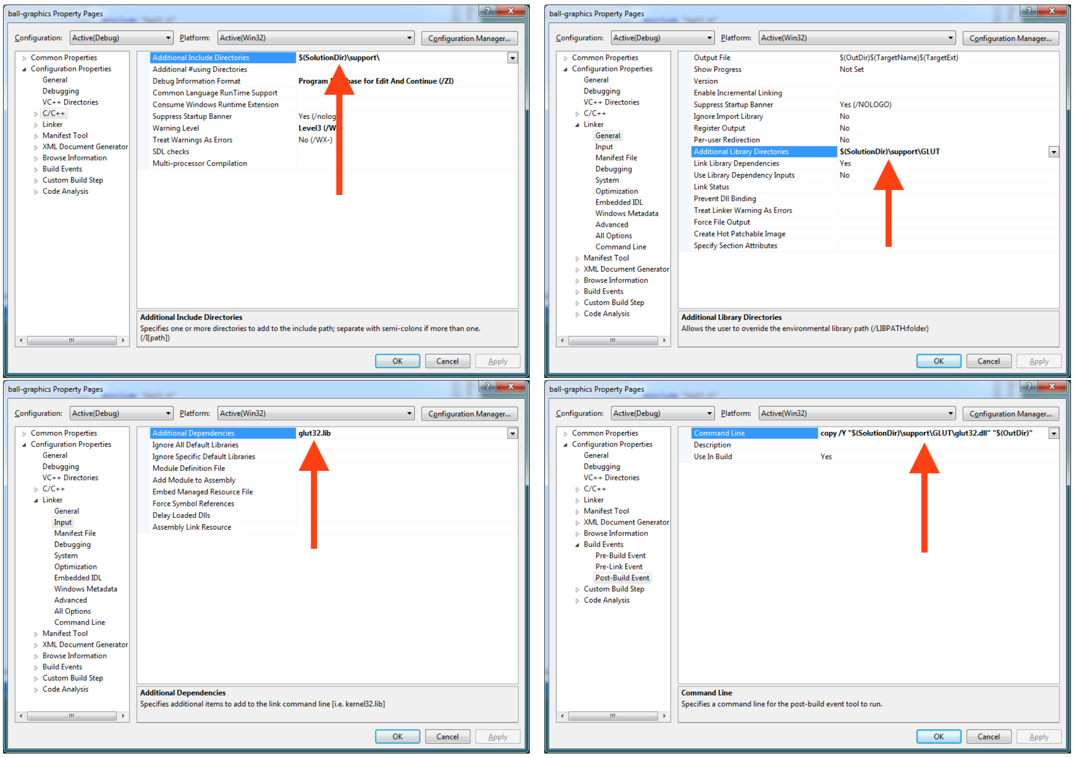

Figure 5: Setting up Visual Studio to link to GLUT. In writing order: setting up the include search path, the linker search path, the linked libraries, and the post-build rule to copy GLUT to the output directory. See the text for details.

A Configuring Visual C to link to a third-party library (GLUT)

This appendix shows hot to configure Visual Studio to do link to a third party library using GLUT as an example. In this example, the following steps must be taken (Figure 5):

- Add GLUT to the `#include` search path, so that the compiler may correctly execute the statement `#include <GLUT/glut.h>` (see `graphics.h`). This is done by adding the directory `$(SolutionDir)\support` to *Project* → *Properties* → *Configuration Properties* → *C/C++* → *Additional Include Directories*. Here `$(SolutionDir)` is a variable that is automatically expanded by Visual Studio to the directory containing the solution, which is in this case equal to `<b16-lab-path>`.

- Add the directory containing the GLUT library to the linker search path, so that `glut32.lib` can be successfully linked to your program. This directory is `$(SolutionDir)\support\GLUT` and should be added to *Project* → *Properties (Alt + F7)* → *Configuration Properties* → *Linker* → *Additional Library Directories*.

- Add `glut32.lib` to the list of libraries to be linked in *Project* → *Properties (Alt + F7)* → *Configuration Properties* → *Linker* → *Input* → *Additional Dependencies*. Note that there are seemingly two libraries in the `<b16-lab-path>\support\GLUT\` folder: `glut32.lib` and `glut32.dll`. The former is needed by the linker during compilation; the latter is the actual library which is *dynamically loaded* and attached to your program by the operating system when the executable is run. This is known as a *dynamically linked library* (DLL) for this reason.

- Finally, the GLUT DLL must be copied to a place that can be found by the operating system (Windows) when the executable is run. While there are several standard places where such DLLs are usually stored, here we do not want to permanently add software to the Windows machine in the laboratory! The alternative is to copy `glut32.dll` in the *same directory of the executable* `ball-graphics.exe`. As you should know by know, this directory is `<b16-lab-path>\Debug`, or simply `$(OutDir)$` using a Visual Studio macro. In order to copy GLUT to this directory after the compilation is complete, we will add a *post-build rule*, in this case the command

```
copy /y $(SolutionDir)\support\GLUT\glut32.dll $(OutDir)\
```

  This string should be added to *Project* → *Properties (Alt + F7)* → *Configuration Properties* → *Build Events* → *Post-Build Event* → *Command Line*.

B References

- General programming questions
  - Google. Seriously.
  - Stack Exchange `http://stackexchange.com` and, in particular, Stack Overflow `http://stackoverflow.com`.
- OpenGL
  - OpenGL 2 reference manual: `http://ewww.opengl.org/sdk/docs/man2/`.
  - GLUT refernce manual: `http://www.opengl.org/documentation/specs/glut/spec3/spec3.html`.
  - OpenGL tutorial on 2D graphics: `http://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_Introduction.html`.
  - OpenGL tutorial on 3D graphics: `http://www.cs.uccs.edu/~ssemwal/indexGLTutorial.html`.
- When all else fails:
  - Demonstrator.

C Troubleshooting

- **Disappearing console.** Differently from Visual Studio 2008, in Visual Studio 2010 even when running with *Start Without Debugging* the console may disappear quickly after the program terminates. In this case set *Project* → *Properties* → *Configuration Properties* → *Linker* → *System* → *SubSystem* to `Console (/SUBSYSTEM:CONSOLE)` (Figure 6).

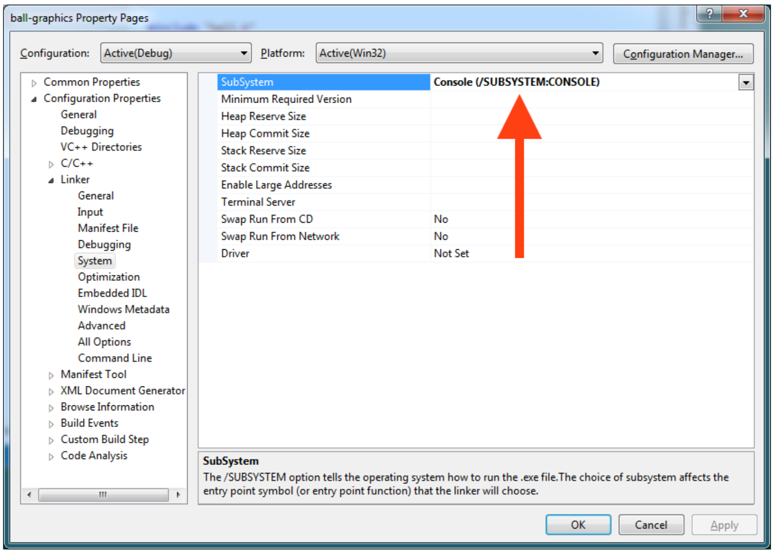

Figure 6: The fix for a disappearing console.