CS 251: Code Warrior: Makefiles and Box2D: Inlab

- Due: 8:10 PM 30/08
- Please write (only if true) the honor code. If you used any source (person or thing) explicitly state it. You can find the honor code on Piazza.

Overview

Small projects – those with a couple of modules – are easy to manage. Developers can recompile them easily by calling the compiler directly, passing source files as arguments. This simple approach fails when a project gets too complex; with many source files it becomes necessary to have a tool that allows the developer to manage the project easily and efficiently. That's where makefiles come handy.

Talking of large projects, there will be times when you can't do everything from scratch. You'll use some of the existing software, and build on top of that. Ideally we would take something like Firefox or the Linux kernel and mess with it, but we would need a lot of other concepts. So We're going to take something you all know and love – physics. We will use an open source physics simulation library called Box2D, and create some nice simulations for ourselves. As you go along, think about the amount of work it'd have taken you had you started from scratch. Oh, Box2D was used to create Angry Birds at some point.

Makefiles are used in Box2D, so that's one reason we go through task B. But if you get stuck, and spent more than 45 minutes, move on to Section B

A. Makefiles

Makefiles are a simple way to organize code compilation. This exercise does not even scratch the surface of what is possible using make, but is a starter's guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

Start with downloading `dependency1.cpp, dependency1.h`, `dependency2.cpp, dependency2.h` and `main.cpp` from the usual support directory. This represents a typical main program, some functional code in separate files, and an include file, respectively. You can assume that all the files are in the same directory.

Without a makefile, you would compile them as `g++ main.cpp dependency1.cpp dependency2.cpp`. And then, the typical approach to the test/modify/debug cycle is to use the 'up' arrow in a terminal to go back to your last compile command so you don't have to type it each time – especially once you've added a few more .cpp files to the mix. But if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best.

Perhaps a script solves that problem?

0. Write a script `compile.sh` to compile the files and create the executable **out**.

1. For this part of the task, we are going to write the simplest possible makefile to achieve our step wise learning approach. Create (a) a target **out** (with its dependencies), and (b) add `out` to the dependency list of the special target **all**. Now running `make` or `make out` should create the executable `out`. You should have 3 lines in your makefile (and `out` is a short for `output`, nothing else, please).

   Question: What's the difference – if any – between the makefile approach, and the script approach?

   Answer the question in the file `answers.txt`. Clearly mention what are the targets, dependency list and their role. Also briefly explain how the makefile achieves this. Name this as `makefile_1`. (How do you use this file with this new name?)

2. `makefile_1` was expected to be the simplest, not the best makefile you can write (as you might have observed) Create `makefile_2`, which ensures that only source files that are modified are recompiled. In other words, source files that are not modified should NOT be compiled. Write targets to compile each of the source files into (the so-called), object files and a target **out** to link them all together so as to get the executable `out`. Run `make` to compile everything once. Now make a small change to `main.cpp`. Run `make` again to generate the executable. What changed, and what did not change? what commands got executed? Speculate briefly how a makefile achieves this? Answer this in your answers.txt file.

3. For this part to achieve the learning objectives, you should start all over and get the timing right. (Seriously start all over as follows). Start with a clean situation in which you have no object files, only the source files, and the makefile in step 2. Run make exactly twice and make sure you understand what happened. What was the output of the second make?

   Now add a target `clean` to your makefile, which removes all the additional files that you generated on running `make`, the object files and the final executable. You should use the `rm -i` command to remove files. Ensure that it works by doing the following

   ```
   make   # what was the output?
   make clean # what was the output?
   ```

   Now type the command `make`, then `./out`, wait for a few seconds (this is important) and then `make clean`. Did you expect this behaviour? We didn't. Why exactly is this happening? Now edit your makefile to fix this. In the end, we should have a makefile which doesn't have **any (and all)** such issues. Answer in answers.txt your observation, the reason for that and how the fix handles it. Name this as `makefile_3`.

B. Box2D

Like we mentioned earlier, Box2D is a physics simulation engine. It provides a lot of essential components that you can use to make cool demonstrations, simulate environments, and even make a game like Angry Birds. We won't go that fancy for this lab, but you'll get some idea of what using a nice open source library can do for your projects. You'll also learn how to work with large pieces of codes, and more specifically, code by other people. Here is a nice tutorial on Box2D that could come handy.

Note: We are using a slightly older version of Box2D meant for the Linux platform.

0. Download the distribution `Box2D.tgz` from the support URL. Untar it, and cd to the top level directory. From here on, assume that all addresses are relative to the base folder.

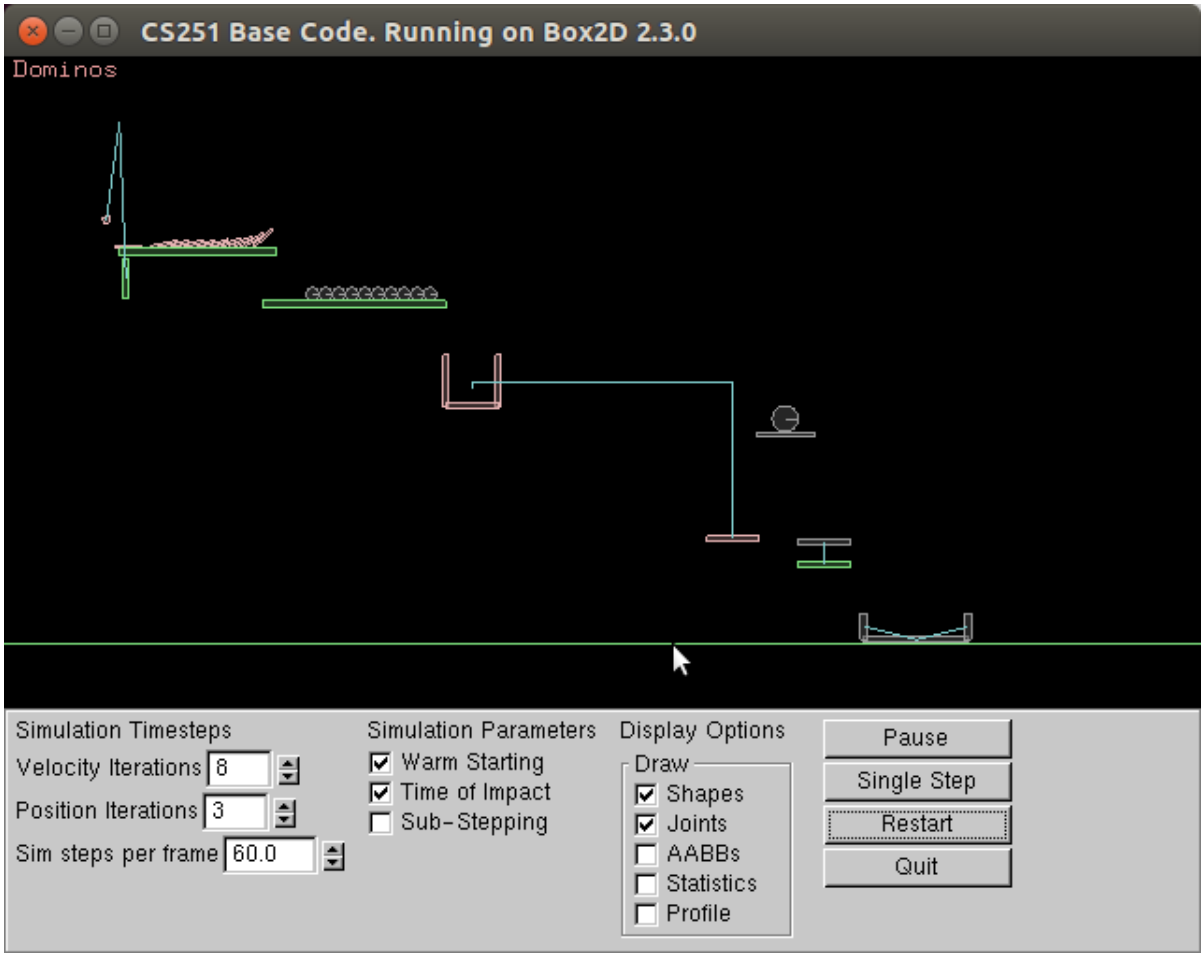
1. Follow the instructions that you'll find in the Instructions.txt file inside the `src` directory. At the end of this, (this will take some time) you'd have installed Box2D. You can use the features provided by this to run physics simulations.

   You will find one example in the src folder. After compilation (using make!) the executable will be found in the bin. Execute it, and check out the simulation. Tweak around with the GUI, and see what changing the parameters does to the simulation.

2. Go to ./external/include/Box2D. Here you'll see the features that are provided by the Box2D software. Check out some of these files, especially Box2D.h, ./Dynamics/* and ./Dynamics/Joints/*, . Each of these objects will have some attributes and procedures in their definition. You can also find them in the online Box2D manual, e.g., here

3. For this task, we'll take the given demo (dominos), and make some changes to it. We want to get rid of the see-saw, and replace it with a slab, joined firmly to a spring-like (elastic) object. So when the ball drops, it should fall somewhere on this slab, and bounces off, with the spring

being compressed. Of course we don't want the ball to go out of our display, so create a bucket on the ground to catch the ball after that. A sample executable is provided to you in the support folder. You need not replicate the same. You can create your own spring-slab system. One easy way of doing it could be to constrain the relative motion of the 2 slabs in a smart way to behave like a spring. You are allowed to change the parameters of any object in the scene, but all the objects in the scene except see-saw should be present in the final setup. It could be something like this (run the executable inlab in the support folder)



4. **[Challenge]** You can create a nice simulation, but it can be a bit boring to watch the entire thing unfold till the end, unless you have a role to play in it. So let's try out some controls for our new objects. Check out `callbacks.cpp` to understand how keyboard and mouse events are handled. We want to control the slab + spring mechanism by pressing and holding the 'W' and 'S' keys. The keys will pull the slab in the corresponding direction, stretching and compressing the spring respectively. Also, now that the ball can go to different places (thanks to the change in force), our stationary bucket might not be enough to catch it every time. Pressing 'A' and 'D' keys move the bucket in the corresponding direction.

Submission Guidelines

1. When you submit, please document individual percentages such as Student 1: 80%, Student 2:100%, Student 3:10% in the readme.txt. In this example, the second student will get full marks (10/10) and the first student will receive 8/10.

2. Do include a readme.txt (telling us whatever you want to tell me). (The reflection essay is not required for inlab, but is required for outlab.) Do include group members (name, roll number),

group number, honour code, citations etc.

3. The folder and its compressed version should both be named `lab05_groupXY_inlab` for example folder should be named `lab05_group07_inlab` and the related `tar.gz` should be named `lab05_group07_inlab.tar.gz`.

4. The submission folder should contain 2 subfolders taskA and taskB.

5. The subfolder taskA contains 4 files makefile_1, makefile_2, makefile_3 and answers.txt. Also write the answers for subtask 1,2 and 3 in your answers.txt. Ensure that they are indexed correctly.

6. The subfolder taskB should contain the src directory of your project. If you have changed/added any other files add them to the subfolder and mention the instructions that need to be followed for handling them in instructions.txt. Submit only one src directory for both extra credit as well as sub task 3

7. Your submission folder should look something like this:

```
lab05_groupXY_inlab
├── taskA
│   ├── makefile_1
│   ├── makefile_2
│   ├── makefile_3
│   └── answers.txt
├── taskB
│   ├── src
│   │   ├── callbacks.cpp
│   │   ├── dominos.cpp
│   │   └── ...
│   ├── other files changed/added, if any
│   └── instructions.txt, if needed
└── readme.txt
```

How We will Grade You [30 Marks]

Extra credit points are available to you only if you get the basics right.

1. Task A [**15 Marks**]
    - `makefile_1` : 1 Marks
    - Sub-task 1 reason : 2 Marks
    - `makefile_2` : 3 Marks
    - Sub-task 2 reason : 3 Marks
    - `makefile_3` : 3 Marks
    - Sub-task 3 reason : 3 Marks
2. Task B [**15 Marks**]
    - Spring + Slab : 10 Marks
    - Bucket : 5 Marks
    - Challenge: No Marks