Q-learning 6
Samuel Rosas
Castor Köhler
Mikko Hemilä
Joni Tuppurainen

Project plan

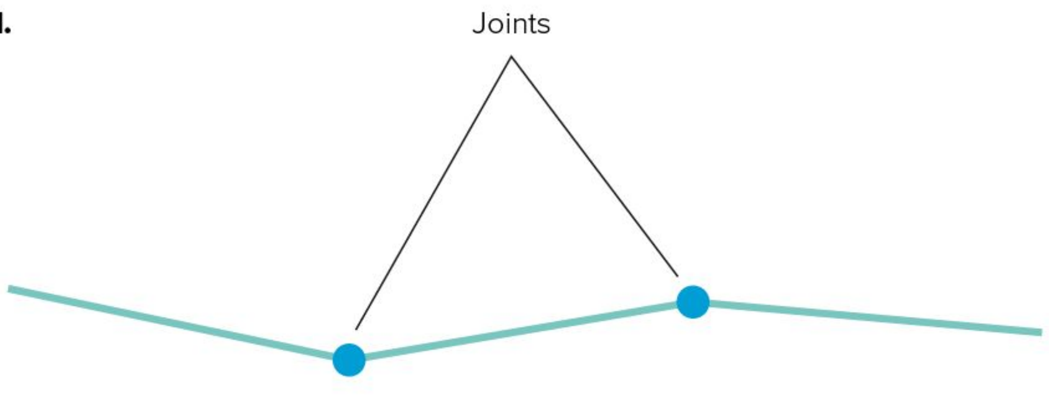
17.11.2017

SCOPE

The scope of this project is largely defined by the requirements for the project. As an absolute minimum, a basic implementation, that covers all the minimal project requirements, should be done. Our group aims to also complete the optional requirements, therefore the scope of the project should include these tasks.

The minimum scope requires that the program will have several bots that try to perform a certain task, and the bots' actions will be controlled by a Q-learning algorithm. In this minimum scope, the bot could be as simple as having only two movable joints, connected to rods, a worm of sorts. (See figure 1.) The bot could manipulate the angle of the nodes, and learn to achieve its goal this way. The bot's objective will be to move forward on a flat surface. Some basic physics should be implemented so that the bots can move. The bots will be rendered as basic shapes, and not much effort should be put into the graphics.

1.

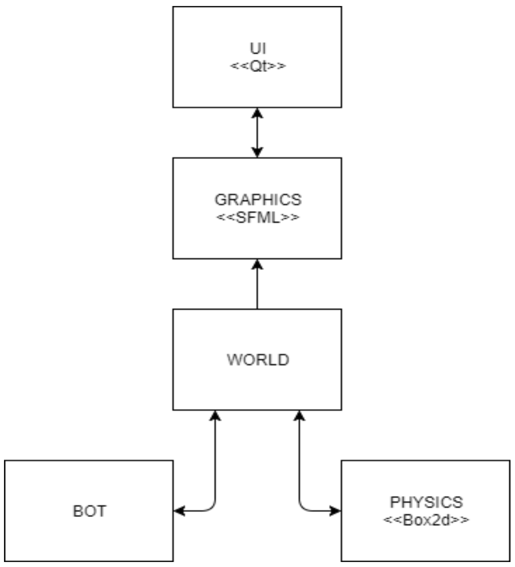


The above paragraph discusses the bare minimum of what should be done, but the scope also includes more advanced features. As per the project description, saving progress to a file, fast forwarding, and knowledge sharing are included in the scope of the project for our team. Saving and reading should be done to a text file. The save file doesn't have to be human readable, as long as it stores the current state of the bot, and can be read back into the program. There should be UI elements, for example buttons, to access the save and load features. Fast forwarding should also have an UI-element, as should knowledge sharing.

Included in the scope is also tests for all the components of the project. Testing is discussed more thoroughly later in the project plan.

This describes the scope of the project. When these features have been implemented, the team can discuss on further additions to the project, such as making more advanced bots with additional joints, or defining new, more complex objectives for the bots.

PROGRAM ARCHITECTURE



The architecture of the program will be rather straightforward. Explanation for the relations and different parts given below.

Bot

The bot part is basically the brain and body of the program. It will contain the learning matrixes and learning algorithms for each as well as handle its body parts and their properties. It will only have interaction with one other part of the program and that is the world. It will only say to the world what action it will take and then the world will reply to it with the result of the action.

World

The world is the part of the program that holds the map in which the bots move in and also has information of e.g. the bots' positions. It will also be responsible for the communication between the bots and the physics engine. This means that it will among other things relay actions that the bots do to the physics engine and relay how the bots' movements affected them from the physics engine to the bots.

Physics engine

The physics engines responsibility in the program is quite self explanatory, it will take care of the physics in the program. It won’t keep track of different bots but instead will be asked questions from the world and answer them according to how its laws of physics work.

Graphics

The graphics part will handle the drawing of the bots and their movements. It will communicate mostly with the World and the UI. The communication with the world will be about where the bots are and which states they are in so that the graphic part can picture them correctly. The communication with the UI is because SFML will be drawing its content into the Qt interface.

UI

The UI part of the program will handle the user interface of the program, in other words, it will draw the different buttons and menus in the program.

PRELIMINARY SCHEDULE

Week 45	Project plan and other planning done
Week 46	Setting up development environment and starting implementation with ui and physics as well as models for skeleton.
Week 47	Implementing the actual q-learning.
Week 48	Project should be more or less complete, this week is mainly polishing and possibly adding some extra features.
Week 49	Finishing project

ROLES IN THE GROUP

We will split the project into different parts and have different persons have the main responsibility for them. This way every team member can focus on their part and learn properly about their respective technologies. Here are the responsibilities:

Box2D - ‘Physics expert’ Mikko
Graphics / SFML - ‘Graphics and UI expert’ Samuel

Bot Brains - ‘Expert’ Joni
Bot Skeleton - ‘Expert’ Castor

DESIGN RATIONALE

We will include a world object in our structure to communicate between all of the classes and gather up all the essential functions in one location, such functions being able to give the coordinates of a bot, give a pointer to a bot, tell the geometry of the map the bots move in and so on. Dominant reason for using one object also being the ability for different members of the team to easily find the functions they need and not needing to dive into someone else's code that might haphazardly change.

Since we are provided with a comprehensive tool, Box2D, we will try to utilize it to its maximum potential as learning to use multiple libraries takes time and effort. The graphics of our program will be implemented using SFML provided Box2D doesn't have it's own tools to achieve the goals we desire. UI will be implemented using Qt in case the former defect repeats itself. The intricacies of the use of these libraries will be uncovered as we progress in the project. It is likely that our team's prior experience in using Qt and SFML will emphasize using them.

On account of our prototyping conversation, the Q-learning algorithm will be inclined to use a 36 state system regarding the direction any of the limbs of our three part worm face, meaning we split the 360 degrees a rod can face to 10 degree steps. Rotating the rod towards one of those states would be considered an action. We settled at 36 directions to keep a compromise between the size of the action / state matrix and the amount of detail the bot has in its movement. It is doubtless this may change as we learn more about different implementations of Q-learning.

The bot will resemble a worm since we didn't want to rebuild a cart that propels itself forward using a hand. We will also be able to increase the amount of joints the worm has with ease.

Conceivably the Q-learning reward algorithm we build will punish the bots for not doing anything to prevent them from going into a state of apathy where they aren't inclined to move forward and they remain stationary while joyfully jiggling their appendages. Moving forward will naturally reward the bots as that is the goal of our demonstration.

We will use Google's style guide when writing C++ as it has been refined to encompass everything essential in the language, the rules within it have gone through the judgement of being valid for thousands of engineers and the style is required to be compatible for most of the C++ community.

CODING CONVENTIONS

We aim to follow google's style guide: https://google.github.io/styleguide/cppguide.html. We collected the most important style choices here below.

- The use of the 'using' directive is forbidden. (Also demanded by course.)
- Code should be in a *namespace*.
- Use a *struct* only for passive objects that carry data; everything else is a *class*.
- Class data members must be *private*, unless they are *static const*.
- Functions should be kept short and focused. There is no hard limit, but if functions exceeds 40 lines some thought should be made to see if it can be split.
- Function parameters passed by reference must be labeled const.
- Exceptions should be used unlike in google style guide.
- Use prefix form (++i) of the increment and decrement operators with iterators and other template objects.
- Use const whenever it makes sense. With C++11, constexpr is a better choice for some uses of const.
- Avoid defining macros, especially in headers; prefer inline functions, enums, and const variables.
- Use 0 for integers, 0.0 for reals, nullptr for pointers, and '\0' for chars.
- Prefer sizeof(varname) to sizeof(type).
- Use auto to avoid type names that are noisy, obvious, or unimportant.
- Naming:
  - Names should be descriptive; avoid abbreviation.
  - Filenames should be all lowercase and can include underscores (_) or dashes (-).
  - Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass, MyExcitingEnum.
  - The names of variables (including function parameters) and data members are all lowercase, with underscores between words.
  - Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case.
  - Regular functions have mixed case (CamelCase); accessors and mutators may be named like variables.
- Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used.
- In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for.

- Each line of text in your code should be at most 80 characters long.
- Use only spaces, and indent 2 spaces at a time.
- Return type on the same line as function name, parameters on the same line if they fit.
- Prefer no spaces inside parentheses. The if and else keywords belong on separate lines.
- Brackets should start at the end of line and end after newline as in following:

    -
      ```
      if (boolean) {
        ...
      }
      ```
- Use common sense and *BE CONSISTENT*.

TESTING / QUALITY ASSURANCE

The project is quite small, so we're not planning to implement any advanced testing environments. Google Test will be used to write basic unit tests for most of the code. Parts that are challenging to test with unit tests, such as UI, will be tested through manual testing, since it's usually easy to spot errors by just observing what is on the screen.

All team members write their own tests for their own code. No single member should be burdened with doing all the testing.

As for quality assurance, the team will do peer reviews and follow good coding practice to make sure code quality stays high. Git will be useful in fulfilling these goals. Features will be developed in their own branches, and will be added to the master branch as a pull request. Another person will review the code before (possibly) merging to the master.