Daniel Kim (dsk252)
Ryan Butler (rjb392)
Stephen Kim (shk77)
Jack Solon (jps386)

BatlCaml Design

System description:

Key idea: Provide users a game where they can creatively write AI code for virtual battlebots and compete against each other's designs.

Key Features:

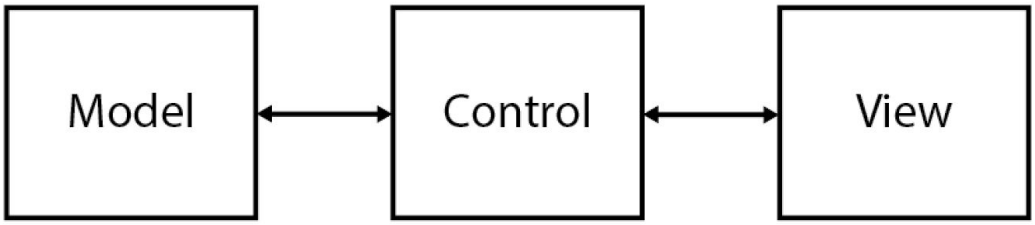
- Provide users with an OCaml API to code robots
- Design the behavior of robots using OCaml code
- Test your robot AI against preprogrammed designs, and other user-submitted designs in a simulated battle environment
- Provide users a command-line application to view the robot battle simulation
- Be able to have the program compile and run user code, rather than users being required to do this

Description:

Our project aims to design a "zero-player game" where players use an API we provide to write OCaml code that controls the behavior of virtual robots. These virtual robots would resemble tanks, and would compete by fighting each other until their opponent is destroyed. This project is inspired by RoboCode, which is written in Java. Once users write their code (using external tools), the game will then load their code and the code of a user chosen opponent. In our first prototype, the user will follow the simulation by reading text-based output displayed after every step of the simulation. If time allows, a 2D, top-down view of the battle arena will be shown to the user, where the two robots will be placed in a random location and orientation. They will then execute the code that the user provided, causing the robots to fight. Whichever robot remains alive is the winner. The aspect of the game that will be interesting to users is refining the AI code to develop more complex and successful robots.

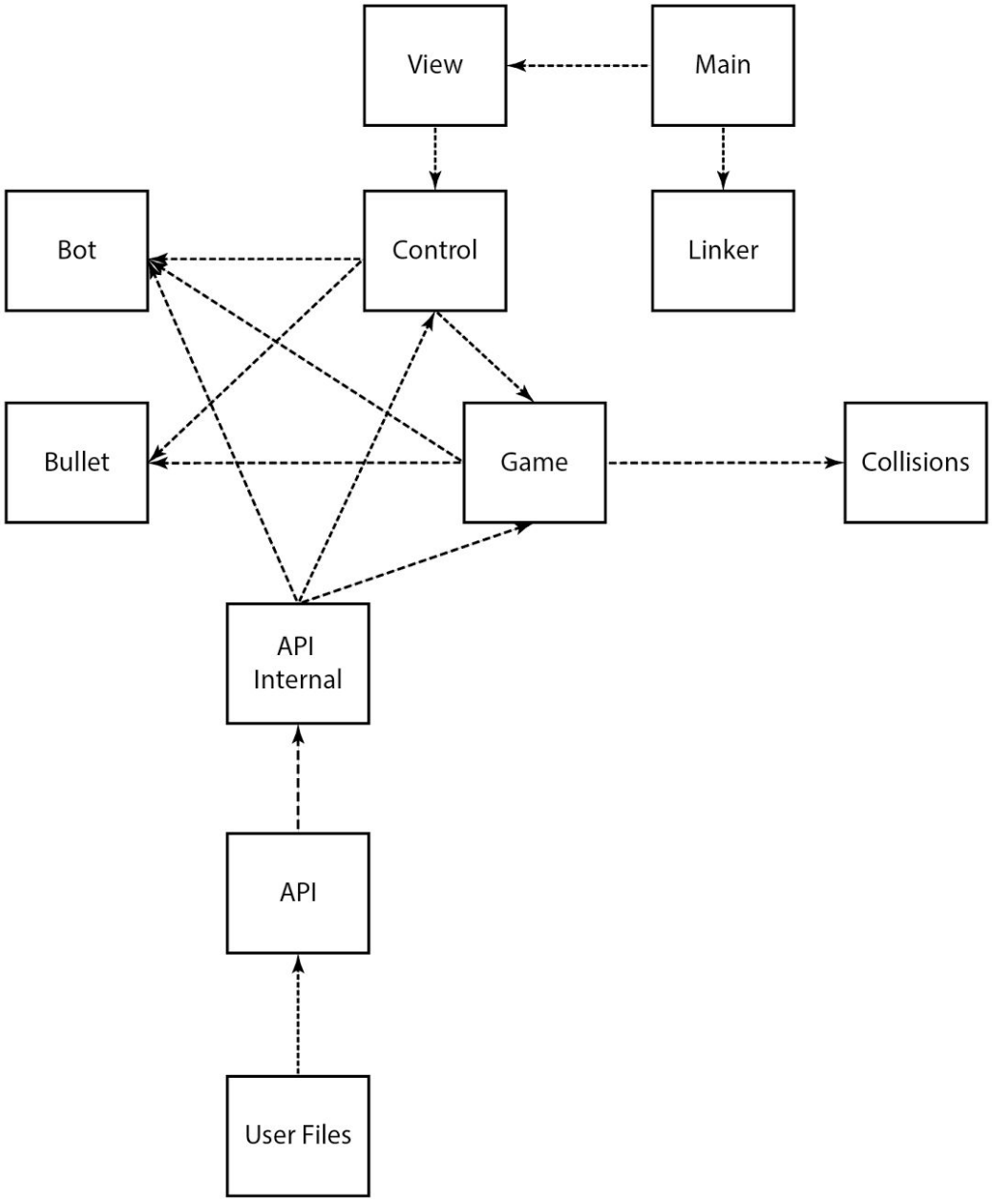
Architecture:

Our implementation will implement Model-View-Controller architecture, which is outlined by the C&C diagram below:



It is important to note that the Model and View components are not coupled with each other at all. This is to keep the game logic separate from the way in which it is presented to the user. Control serves as the bridge between the two. Changing Model or View will likely require some small changes in control as well, but will not affect each other. Additionally, control is responsible for reading user input, however because BatlCaml is designed to be a zero-player game, the only real input that will ever be read from the user is the AI code, and this is not exactly input in the traditional sense of a GUI application. Hence, control will effectively serve only as an abstraction between Model and View.

System design:



We aim to have several modules (necessarily OCaml modules) that will consolidate features together and decrease coupling while increasing abstraction, especially between the modules that make up the Model and View components.

The Model component will consist of the following .mli files:

- bot.mli: This interface will provide a way for the program see the state of an individual bot, and provides a step function to update the state of a bot.
- bullet.mli: This interface will contain a way to see the state of a bullet launched by a bot, as well as a step function to update its state.
- game.mli: This interface will bundle the state of the game together, and provide a step function to update that state.
- api.mli: This interface will provide users with a set of high-level functions that they can use to control the robot that they can call from their ai implementations. They will use these functions in their implementation of the step function.
- apiinternal.mli: This interface is the same as api.mli, but is hidden from users of the program
- collisions.mli: This interface is used by the game module to handle collisions in the course of the simulation.

The Control component will consist of the following .mli file:

- control.mli: This interface will handle all necessary abstractions to get the Model component to communicate with the View component, as well as handle any user input. It will also repeatedly call the step function in simulator as needed to run the game logic.

The View component will consist of the following .mli file:

- view.mli: This interface will handle all necessary functions required to interact with control so as to actually provide the user with some context to see the game, be that through a GUI or console application.

Module design:

See attached .zip file for pertinent information on the design of each module.

Data:

The controller module, which sends pertinent data to the view module that it may be displayed to the user, contains state data relevant to the simulation as a whole. This simulation data will contain state data of each bot, as well as information about the game as a whole. We will not need to rely on any data structures, but will instead use functions that return new state objects on each step of the simulation.

External dependencies:

We rely on Oasis to dynamically load and compile user ai code to run our application. We use ANSITerminal to display our application in the command line in an aesthetically pleasing way. Finally, we rely on Ounit for our unit tests.

Testing plan:

We tested our application using a combination of glass and black box unit tests and play tests. Because of our implementation approach, we were able to confirm that our model functions were correct before we moved on to implement control and then view, and further were able to confirm that we did not mess up any of these functions when implementing new functionality.

Our unit tests served to confirm the basic functions of our model. This meant testing to make sure that each branch of the execute function in Game worked correctly, and that the step function in game worked as expected. By writing tests to confirm the correctness of these functions, we were able to ensure that our program would run correctly when looped in a regular simulation, as these were the only functions that the correctness of that loop relies on.

After adding the view and control modules, we used vigorous play-testing to ensure that our application ran smoothly and correctly. This mean ensuring that the view displayed the simulation as we expected it to look, and ensuring that the loop in the control module called the correct module functions and correctly ran the simulation. By play-testing on a variety of ais, frame rates, seeds, and steps we were able to ensure that our implementation was both correct and that it was displayed to users in a readable and accurate format.

Division of labor:

Our group worked on the vast majority of this project together (the implementations of the model and control modules). Ryan focused on implementing the dynamic loading of user code, while the rest of the group focused on implementing the view. All group members worked together to test the application.