DESIGN Document for SLOGO

Ken Kalin (kdk33), Jim Huang (zh120), David Liu (ql139), Emily Shao (eys9), Sophie Williams (smw92)

Team 8

Role(s)

- Ken Kalin
    - Wrote Controller
    - Setup Model, View
    - History Class
    - Tests
    - (Changes) UI Preset Buttons
    - Documentation (README.md)

- Sophie Williams
    - Model (Instruction Objects, Parsing, Variable Manager, Command Manager, Model Turtle, Model Canvas, Argument Objects)
    - Instruction Bundles and communication with frontend
    - SceneController
    - Turtle animation and drawing
    - Color selection
    - Instruction tests

- David Liu
- Emily Shao
- Jim Huang

Design Goals

The purpose of this project is to implement a version of "SLOGO" to animate the movement of a turtle via text commands inputted by a user. The commands control the movement of the turtle on the screen which can draw a line of varying colors (like an etchesketch). The project will showcase our understanding of encapsulation, abstraction, and higher level design concepts like Model View Separation and Controller classes.

Features implemented:

- Enter commands to the turtle interactively by entering text commands
    - Commands can be nested, include variables, and include user-defined commands
    - Enter commands to the turtle interactively by clicking on buttons
    - See the results of the turtle executing commands displayed visually
    - The turtle starts in the center of the display which is considered (0, 0)
    - See errors that result from entered commands in a user friendly way (i.e., not just printed to the console)
    - See history of successful commands run previously
    - See user-defined variables
    - See user-defined commands
    - Set an image to use for the turtle (instead of the CSS style's default)
    - Set a background color for the turtle's display area (instead of the CSS style's default)
    - Set a color to use for the pen (instead of the CSS style's default)
    - Set the speed to animate the turtles' movement and turning (where the max speed does no animation, just displaying the final result)
    - Can save the current history of the program as a .slogo file (through Settings window)
    - Can load a .slogo file into the program and run the commands in the file (through Settings window)

High-Level Design

Frontend

When you start main, the `start()` method is called which creates a sceneController and passes it a Stage. The Scene controller creates a new "splash" scene. On this splash scene, the user can choose to load a language file and then start the SLogo program by clicking Begin. This triggers a callback to the Scene controller to generate the `MainScene`. The MainScene is created and the controller for the Main class (`MVController`) is created.

The Main Scene initializes the following classes which make up the UI

- `CanvasView` which displays the turtle and the path that it draws.
- `ToolButtonPane` has the `ComonpentStyleButton` which opens the settings window (which configures light/dark modes). It can also set the background color and and has a button to open the help window.

- `ControlButtonPanel` contains the play and pause animation buttons and the update Animation rate slider and frame skipper It also contains dialogs to change the pen width and select a pen color.
- `UserInputPane` Contains the command bar for text input, the submit button, and the GUI controlls that triggger preset commands such as `Forward` and `Left` and the ability to draw premade shapes such as `Square` and `Triangle` and to turn on and off the pen.
- The `History Pane` contains the history of commands that have been executed.
- The `UserVariablePane` contains the variables that have been created by the user.

`HelpView` is a separate scene that is created when the user clicks the help button on the `ToolButtonPanel`. It contains a list of all the commands that can be used in the program. As defined by XML files that are loaded in the beginning of the program. It includes **definitions, examples, code segments, and expected return values for each command.**

Controller

The `MVController` is the controller for the entire program. It is created when the `MainScene` is created. It is passed a reference to the `MainScene` and creates a `Model` obejet. The controller sets up a listener within the view that will call `sendSignal()` to notify the backend when the user submits a new command. MVController also sets up a JavaFX timeline object which starts the animations.

At each step of the timeline, if the listener has been notified that a command has been submited, the Controller will call the models `handleInstructionInput()` method. This method will parse the command and then call the appropriate method in the model.

The `listner` has variables to store the instruction and has a boolean flag that shows when a new instruction is ready to be read and exectued. It has method `sendSignal()` that will store the instruction and set the flag to `true`. It has a `clearSignal()` method that will set the flag to `false` and clear the instruction.

If there is a bundle available from the model and the view is not currently playing an animation. The controller will call the `.getBundle()` method from the model to get the newest instruction bundle and it will call the view's `updateView()` and pass in that bundle for the view to be updated.

XML Reader

The `XMLReader` class is used to read in the XML files that contain the information for the commands. It stores the command information in an `XMLData` object which contains a list of `commandInfo` bundle objects. Each `commandInfo` bundle contains the following information: * `name` : the name of the command * `Name Variants` of the command * `Description` of command * `Example` of command *

`Num of Args` : number of arguments the command takes * `Class` : the class that executes the command

The XML Data object is passed into the `HelpView` to display the information to the user.

Backend

The `Model` class is the backend of the program. It is created by the `MVController` and given the `language` which is used to setup the parser to run in the correct language.

The Backend Contains:

`Parser` : The parser is created when the model is created and is passed the language. The parser is used to parse the user input and create the `InstructionBundle` objects.

- When an instruction is sent to the model through the `handleInstructionInput()` method, the parser parses the instruction into an `InstructionList` object. The parser uses the following algorithm to decomose each line of instructions into executable `Instruction` objects.

1. It will split the instruction by spaces into a linked list of ListNode objects.
2. For each ListNode it will determine if it is a:
   - `Command`
   - `Variable`
   - `Constant`

3. If it is a command, it will check if it is a valid command. If it is not a valid command, it will throw an error. It will create an `Instruction` object of the correct name using reflection. (It will convert alternate names of the command into the original form before doing this) By getting the number of expected arguments from the instruction object, it will scan ahead in the list the correct number of arguments.

   - If any of the next arguments are parsed as another command the parser will recursively populate that commamnd object and pass it as an argument to the original command.
   - This is handled by the `Arguments` interface which allows commands to accept any type of argument (constants, insturctions, varaibles). Each command exepts `Arguments` and doesn't try to access them until runtime (after everything has been parsed).

Once the entire line is parsed the `handleInstructionInput()` method will call the instuction list `execute()` method. This will call the `execute()` method on each `Instruction` object in the list. After executing, `InstructionBundle` objects are created and placed onto the `InstrucitonBundleQueue` . The `InstructionBundleQueue` is a queue of `InstructionBundle` * Executing the instruction updates the corresponding `ModelTurtle` object's state which is used in the backend for calculations about the turtle's movement. * `InstructionBundle` object contain information for the frontend to update the view.

Any errors generated by the parser are added to a `DataStorage` object called `myError` which keeps track of the error messages until prompted by the frontent when it generates an error popup.

`VariableManager` : The variable manager is used to store the variables that are created by the user. It contains two HashMaps. One for `UserVariables` and one for `UserCommands` . The `UserVariables` are stored as Strings mapped to Doubles. The `UserCommands` are stored as Strings mapped to `CommandTemplateObjects` which store a list of commands that need to be executed based on how the user defines the structure of the command Macro (using the `TO` command).

Assumptions or Simplifications

Assumptions or Simplifications:

- The Turtle was placed at the center of the screen by translating it by 250 in each direction. To compensate, the `getTurtleX()` and `getTurtleY()` methods in the `ModelTurtle` class return the turtle's x and y coordinates after subtracting 250 to correct.

- The Help Window is only setup to display predefined commands based on XML files supplied before runtimie and will not show user defined commmands that were created as the program was running.

- The user is also required to select a language at the start of the program and cannot change it during runtime.

Known Bugs:

- When the turtle travels past the edge of the canvas, it does not stop,
    - The pen will not draw outside the canvas, but the turtle will keep moving
    - There is no error checking for turtle out of bounds

- If a resource fails to load, the program will crash and only the Help Window will appear
- The turtle image is not centered on the turtle (this means that tests are off by +/- turtle width)
- When putting the pen back down, it will automatically still draw a line from where it picked the pen up

Changes from the Plan

Originally, the program was going to execute the commands as they were parsed. This would allow all arguments to be evaluated into constants before being inputted into a command. While this technique would work well for simpler instructions like "fd fd 100", this would not work as cleanly for control instructions such as repeats and if statements. Realizing this, we decided to rework the backend of the program so instructions would be able to take a number of different types of arguments such as constant, variables, and even instructions. This way parsing and execution of the instructions could be separated.

Features unimplemented:

- Change the language of the program during runtime
- Speech to Text input

How to Add New Features

Icons

The Icons folder in the resources folder contains all the icons used in the program. They are all in the `.png` format.

Stylesheets

The Stylesheets folder contains the `.css` files that are used to style the program. There are seperate file sheets for `Light` `Dark` `Duke` `UNC` and `Original` themes.

XML Files XML Files are the descriptions of each command (within their own langauge) used to popualte the helpview and the parser and instruction objects.

Language Files

The language files are used to store the names of buttons and labels in the front end of the program in their respective langauge.

Properties Files

The properties files are used to store the default values for the program. They are used to set the default values for the program when it is first started.

Text.properties

- Screen width
- Screen height
- Default shapeSize

Syntax File (Syntax.properties)

- Syntax for valid command, constant (regex)
- Syntax for valid variable (regex)
- Syntax for valid comment (regex)

Turtle Images There are three included turtle images that can be used in the program. They are stored in the `TurtleImages` folder in the resources folder. These can be selected by clicking on the turtle once the game has started.

- Key/Mouse inputs:
- Arrow keys: Move the turtle in the direction of the arrow key (up, down, left turn, right turn)
    - After a command is run, the UI focus will jump back to CanvasView so that the user can continue to use the arrow keys to move the turtle.
- The user can move the turtle by typing commands into the text input box. The commands are case insensitive and are documented in the `HelpView`
- The user can also move the turtle using the provided GUI buttons which can move the turtle in predefined directions and shapes.
- They can also click on the turtle to change the image on the turtle
- There are also dialogs to change the state of the pen including the color and thickness of the pen and whether it is currently set to draw.