

Graphics

Now that you have the text-based version of the Traffic Jam game up and running, your job is to use what you have learned so far to create the graphics version of the game.



Figure 1: A screenshot of the Traffic Jame game.

The Graphics-Based Version

Your version of the game will ultimately look something like the version in figure 1, which has the car (represented as the red car), some trucks and autos, and the exit (as a red label).

Luckily for us, we already designed the `Vehicle`, `Board`, `Level`, and `Location` classes and will reuse all of them. This means that if we designed our classes well, we should only have to implement a class called `GraphicsGame.java` and not change anything that we have done before. The starter `GraphicsGame` is already in your Traffic Jam project.

Here is how the `GraphicsGame` interfaces with the rest of the classes. Notice that, just like `ConsoleGame`, it links up to `Level`, which is the intermediary to the actual game and its rules.

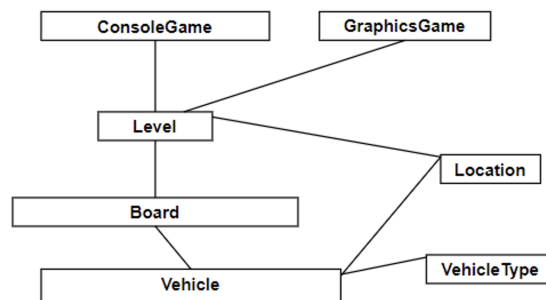


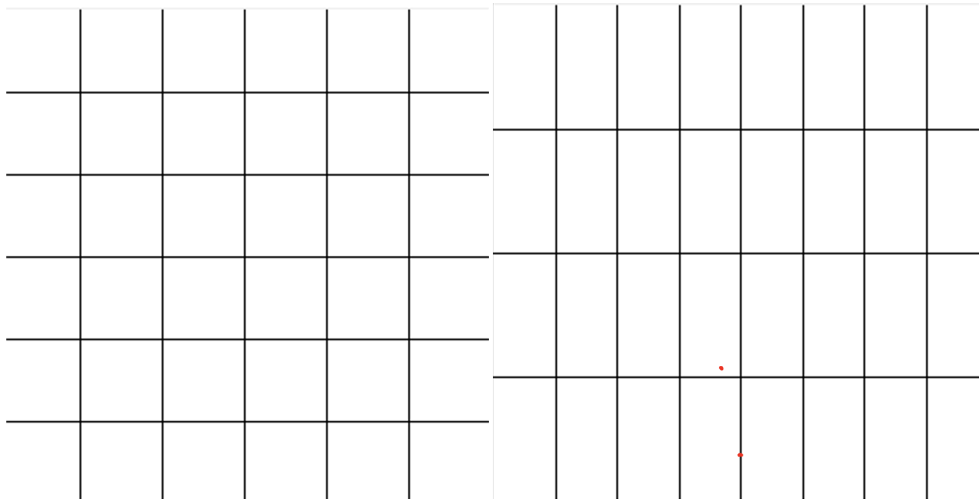
Figure 2: Relationship between classes in this assignment.

The Plan of Attack

1. `cellWidth()` / `cellHeight()` Based on the number of rows and columns for a particular level, return the width / height of a single element in the grid, which we will call a cell. This method could be written in one line and is meant to help you translate between row and columns and what is on the screen.



2. `drawGridLines()` Draw the lines to make a grid that represents the locations on the level. Your code should be general enough so that it works for levels with any number of rows and columns. For example, the figures below show two different grids (a 6x6 and a 4x8 one) that both take up the same space. Make sure to test this out before you continue!



3. `convertXYToLocation(double x, double y)` Convert the x, y coordinates given to you into a `Location` object that can be used to access a row and column on the board.

For instance, assuming a cell width and cell height of 100, calling:

```
convertXYToLocation(175, 225)
```

Would return the following location:

```
row 2 col 1
```

Make sure to test this by overriding `mouseClicked` and then having a print statement (or `GLabel`) for the location that corresponds to the cell where

you pressed the mouse button. Do this multiple times. For example, if you click inside the cell square that should be row 1 column 1, verify that this is what gets printed out. *As part of your submission, include this code but leave it commented out.*

4. `drawWinningTile()` Create a label named “exit” on the level’s goal cell and draw it appropriately. You will notice that there is a constant for the font that you will use here (named `LABEL_FONT`). The label should appear in the goal space but does not have to be centered.
5. `getVehiclesOnBoard()` This method is one that you will want to introduce to both `Level` and `Board`, but what you will simply want to return a list of all the `Vehicles` that have been added to the game. This will make it easier to work with the mouse events in step 8. Rather than trying to construct a list of vehicles on the fly, it will be easier if you declare and set up a new `ArrayList` as an instance variable, which turns `getVehiclesOnBoard` into a simple getter function. Then incrementally add the new vehicle into the list each time `addVehicle` is called.
6. `drawCar(Vehicle v)` Given a `Vehicle` object (named `v`), draw that vehicle on the screen based on its information. In order to draw a vehicle, you will have to create a `GImage` object. `GImages` take in a filename, `x`, and `y`. In this case, your starter project already has PNG images for you in the `images` folder. To use them, you just have to just the path “`images/`” followed by the name of the file that you want (e.g. “`car.png`”).

Notice that rather than having to rotate images, you are provided with separate (vertical) image files that have the same name as their counterparts – except that they have a “`_vert`” added to the end of the filename. So, for instance, “`car.png`” becomes “`car_vert.png`”. You can use these files in your program and there are additional constants at the top of the `GraphicsProgram` that will help you with these labels.

A few hints: The `VehicleType`’s `toString` method returns the same name as the filename that you want to use. Make sure to set the size of the images so that they take up the appropriate number of spaces on the grid. (The images should resize based on the number of rows and columns they take up on the grid.) You will want to test this to confirm that it works.

7. `drawLevel()` Use all of your helper methods to get a screen that looks like figure 1. Notice the 0 at the top left is the number of moves performed so far, so you will have to add that label, too. You can either create a separate helper method for this or just add it in this method.

8. `calculateNumSpacesMoved()` Create one additional method which will determine how many spaces a user wanted to move from their original location based on how far they dragged the mouse since they first started clicking on the mouse. (Consider whether you might need to store that information.) This is great to have when you ask the level to move a particular vehicle. For this assignment, you should not make the vehicle stop at a particular location if it cannot move any further. Wait until the user releases the mouse and then use `calculateNumSpacesMoved` to determine out how many spaces the vehicle is supposed to move.

You will need to figure out the number of spaces using the distance between where the mouse was first pressed and then released. Be sure to use the x and y coordinates in both cases. **You should store the original location when a user clicks on a space and – once the mouse button is released – print out the result from `calculateNumSpacesMoved`.** Use this small interaction to test that your `calculateNumSpacesMoved` is indeed working. **You must also print out the car that was clicked on and verify that it is in fact the right car.**

9. **Mouse Events** Use the methods above to finish implementing all of the mouse events and event handlers needed for the game. The five mouse event handlers were `mousePressed`, `mouseDragged`, `mouseReleased`, `mouseClicked`, and `mouseMoved`, with each passing in a `MouseEvent` `e`. You do not have to implement all of them, but you will have to decide how you want the code to behave.

After the user has attempted to click, drag, and **then let go of an object** is *the only time* you should ask the board if it is possible to move to that location. You can do so by sending the request to move, just like you did in the `ConsoleGame`, using `calculateNumSpacesMoved` to help you to know how far the vehicle should move.

Then, once the move happens, just update the car's location. Since the car will not move if it cannot, you can just ask the `Vehicle` where it currently is on the board and set its image location to that spot. If it moved, the `Vehicle` will give you a new row and column. If it did not, then it will give you the original location. Regardless, you can set the location using that space and the calculations you have done with cell width and height.

Note: If you get a `ClassCastException` when trying to drag a `Vehicle`, it may be because you actually clicked on a `GLabel` instead of a `GImage` (i.e. vehicle). You do not need to worry about this. However, if you do want to fix it, it is most likely caused by using `getElementAt` and then directly casting that as a `GImage` when it

is not, which raises the exception. To get around it, you can create an if-statement that checks the type of `getElementAt`, making sure it is a `GImage` before proceeding. Introducing something like this early in `mousePressed` should help to make sure you clicked a `GImage`.

```
obj = getElementAt(e.getX(), e.getY());
if(!(obj instanceof GImage)) return;
```

- 10. Implement How to Win** After each movement that the user makes, you should add the winning condition to check to see if the board is solved. If it is, then you can call `removeAll()` from the screen and add a new label that shows a congratulatory message.

UML Diagram

In order to help you keep track of everything that you need to write, we have revised the UML model below, which shows in more detail the methods for `GraphicsGame`. Other than `getVehiclesOnBoard`, all of your work for this assignment should go into `GraphicsGame`.

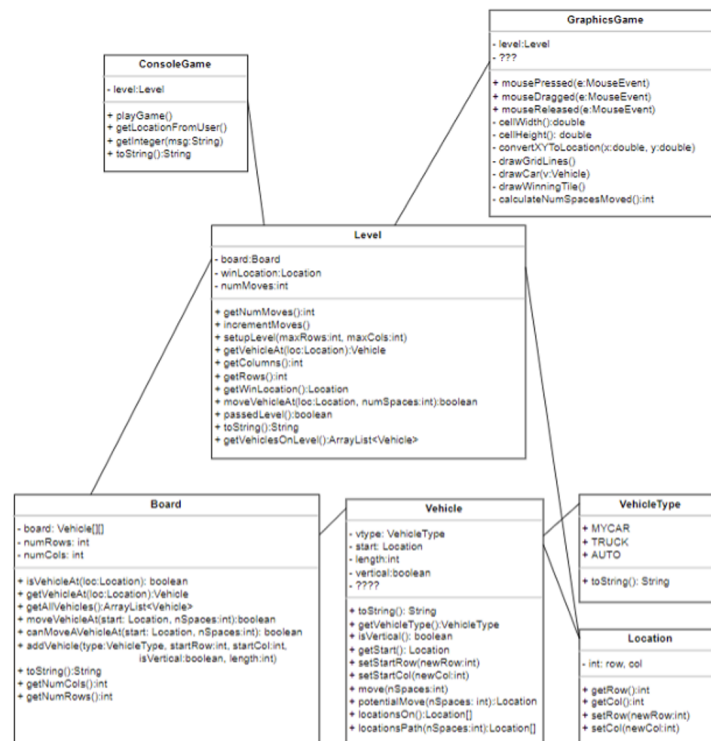


Figure 3: A UML diagram of the classes for this assignment.

Deliverables & Advice

Once you are done, submit the entire project as a .zip file like you did with the previous assignment. Please start early and if something does not make sense as you read the code or this handout, do not hesitate to ask me.

The challenge in this project is not the amount of code you need to write – if you write it correctly, the amount of code can be quite limited, making it easier to debug. The biggest challenge lies in figuring out the math to translate the x and y coordinates into movements on the row and column for the board and leveraging everything you have already written so far.