Mini Transport Tycoon: Design Document

Patrick Walsh (prw55), Jonathon Gibson (jtg98), Maksis Knutins (mk2228), Dan Liu (dl556)

System description

Core vision:

A real-time strategy game that involves building a transportation empire in the town of Camelot that consists of multiple markets and a variety of resources.

This game is heavily inspired by Mini Metro by Codepoint Limited (http://dinopoloclub.com/minimetro/) and Transport Tycoon Deluxe by Chris Sawyer (http://www.openttd.org/en/).

Description:

The player starts off in the town with a limited public road network, locations that produce and accept certain goods, and a limited budget of $250. Their goal is to build roads and vehicles that transfer goods between locations in the most lucrative way. The first player who reaches $2,500 wins.

At the core, the player can do one of two things:

- purchase new vehicles; and
- purchase new roads (either through privatizing public ones, or building private ones).

They are then to transport goods using their vehicles between the locations. To generate profit, they must sell the goods at a higher price than they bought them for, but they may also opt to sell any of the roads or vehicles for a reduced price.

For each vehicle purchased, the player determines a starting location for it, and they can then route that vehicle and purchase cargo for it at marketplaces. As the initial network is sparse, the player is incentivized to build new routes that are more efficient, thus generating more income more quickly.

As the player competes with other AI players, it might be in their best interest to purchase exclusive rights to road segments to disadvantage rival players.

Graphics-wise, the game has a 2D top-down interface, where the town is represented as a graph with nodes as locations and edges as the roads between them. Players use their mouse to interact with the GUI buttons, vehicles, and locations to carry out the actions outlined above. Once the goal is reached, the game ends, and the winner is announced. The duration of the game is also displayed and they may return to the title screen to play again or change the AI's difficulty.

Key features:

- **The town of Camelot:**

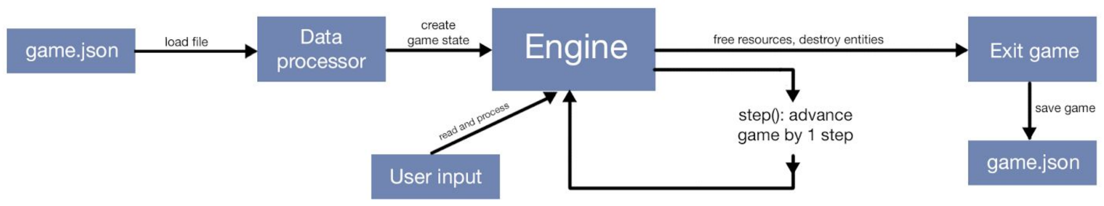
    - Locations: Nodes that accept/produce certain goods.
    - Roads: An interactive, dynamic network that connects the locations, which may be built/destroyed/monopolized, etc.
- **A Fleet:** set of vehicles that each player builds for transporting their goods.
- **Money:** Each player grows their available funds by transporting goods, and they can spend their money to build infrastructure, buy cargo, or expand their fleet.
- **Rivals:** AI players who compete with the player to expand their networks and claim access to the best routes. Beware, they are quite savvy businessmen.

Architecture

The game uses a modified **pipe and filter architecture**.

A game begins when a user selects a map to play - one of provided local options, a previously saved game, or tentatively, a city of choice anywhere in the world. The selected map, a local **JSON file,** is fed into a data processing component, which converts it into a valid game state, which is then passed into the game engine.

Engine, the game's central component, processes the initial game state and listens for user input. Based on the input, it processes subsequent game states until an exit instruction is received. Upon exiting, the engine frees resources and saves the last game state into a local JSON file. While the game is fairly short, this is useful to create a savepoint that you can return to when trying to beat the Hard or Brutal AI.



Component and Connector diagram of game architecture

System design

Modules that describe each separate system in the game are outlined below:

Main: This module serves as the starting point for the game. It prints some welcoming text and then launches the title screen.

DataProcessing: This module is responsible for saving and loading states of the game. This is used so that the player can return to a game at a later time without losing their progress.

Engine: This module executes each step of the game. In each step it calls functions and uses types in many of the other modules (see MDD) to update and display the game state, check for an input, etc. It helps break the task of updating every frame of the entire game into a single series of steps which can be executed using the other modules.

InputProcessing: This module checks for user input on the screen and through the keyboard and uses it to generate specific processes which are used to perform actions that modify the game state. It also contains functions that are used to convert player inputs to processes as well as determine AI actions.

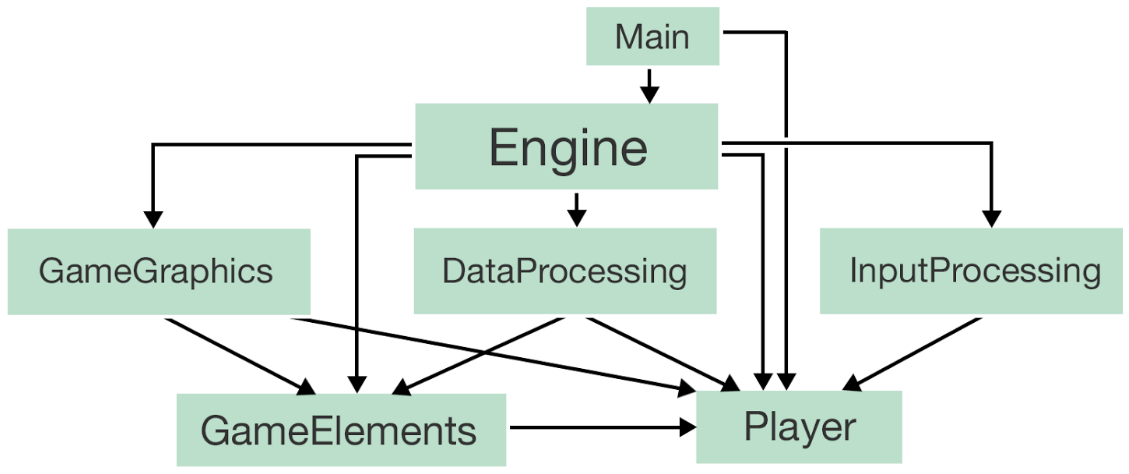
GameGraphics: This module handles the updating of the display screen. The main display function in the module takes in a game_state and displays its representation on the screen. It also displays the buttons needed to interact with the game, and returns a process list if any of the buttons are clicked.

Player: This module has a type to represent a player (either human or AI) and AI difficulty.

GameElements: This module contains the following types and functions necessary to build the graph and play the game:

> **Gamestate:** A type to represent an entire state of the game (including the map, a list of vehicles, a list of players, a game_age, and a paused flag)
>
> **Location**: A type to represent each location in the game world (including its coordinates, resources, etc.) and functions related to the modification and updating of locations.
>
> **Vehicle**: A type to represent each vehicle in the game world (including its type, cargo, destination, etc.) and functions related to the modification and updating of vehicles.
>
> **Good**: A type to represent each good in the game world (including its type, quantity, origin, etc.) and functions related to the modification and updating of goods (such as price updates).
>
> **Connection**: A type to represent each connection (i.e. road) in the game world (including its connected locations, position, owner, etc.) and functions related to the modification and updating of connections.

Module design



See .mli files for a more in depth look at what each module does.

Data

A *game_state* type maintains information on the set of **locations, roads, players, vehicles** and **goods** in the current game. It is possible to serialize the game state to a JSON file to support loading and saving.

User interaction is done through the GUI and mouse clicks. **Mouse click locations** are used to determine player commands, which will update information on each object as necessary. Our game also stores mouse clicks so that a user can pre-click the locations or vehicles before clicking a button; this is extremely useful when trying to beat fast AI but can be tricky to master for beginners.

Data on the roads and locations are updated with respect to a module that contains the graph, as created by **OCamlgraph**. OCamlgraph hides its implementation of **graphs**, so we cannot identify how the vertices and edges are stored (but we can access them through ocamlgraph functions).

Information contained in *game_state* will generally be stored as **lists**. Because vehicle updates will necessarily involve iterating through the entire set of vehicles (in terms of updating their current location), the use of lists should be equally as efficient as any other data structure.

External dependencies

We use **Graphics, OCamlgraph**, **Piqilib, Camlimages, Gtk+.**

Graphics:
This is used to draw the GUI and record mouse events. Due to an unimplemented function in the Graphics module, we could not use it to change text size (see Piazza Post @2538), unfortunately, so we made our own font and printed bigger text as images.

Camlimages:
We use only two functions from here, but they are vital for our GUI. We load an image from a file, and then convert this image to a color array array so that it can be used in Graphics.

Gtk+:
We didn't use Gtk+ ourselves, but Camlimages is dependent on it, so it must be installed.

Ocamlgraph:
This is a library that can be used to implement graph structures in OCaml. It is used to implement the graph representing the vertices (locations) and edges (roads) of the town. OCamlgraph also implements various graph algorithms that are used (such as Dijkstra's shortest path algorithm).

Piqilib:
Actively developed and versatile data serialization library with support for JSON and XML. This library is used by our DataProcessing module to load and save game states. It was selected over Yojson to support OpenStreetMap-based XML maps had they been implemented.

Testing plan

Testing was done incrementally. The main unit testable components of the game were process handling and data processing. Therefore for these components we made a test suite verifying that each process updated the game state properly and that the json files could be loaded and saved successfully. Because this could only be used to test a very small piece of the system in isolation, we mainly used gameplay testing to test that larger features were working together. By setting up our main_loop and functions so that we could still run the game and see the visual window before implementing all components, we were able to test as we went each feature of the game, without needing all other systems to be working.

AI testing was done mainly through running the game and watching the behavior of the AI. Since there are several elements of the AI's decision algorithm that are randomized and not immediately predictable by the human coding it, this was more about looking at what the AI did that seemed unintelligent and less about getting it to buy an exact road, vehicle, or cargo route. Therefore while unit testing could have been used, it would have told us very little and only information about a very small subset of cases that could actually arise. It was only possible to test the AI after process handling was complete, but luckily we planned our division of labor well so that by the time AI was ready to begin testing, process handling had been completed. Due to the visual nature of the game it was fairly straightforward to determine whether the AI was acting in the manner it was expected to act in.

While most of our testing was incremental and performed as soon as we had implemented a feature such as buying roads or selling vehicles (either using unit tests, playtesting, or both), we wanted to be sure to catch all bugs in our game very thoroughly at the end. We ended up catching a few rare cases, such as incorrect terminal helpful messages printed in the game when you performed a very specific action or crashing vehicle routing when they could not reach their destination, but after several hours we are fairly confident that no bugs exist. Finally, we spent several man-hours gameplay testing to improve the balance of the game and attain a nice level of difficulty while still keeping it fun to play. Of course this balance can be somewhat subjective, but after also enlisting a friend of ours to play it and give us some feedback we think it has a fair enough balance while still being a difficult game that is hard to beat on one's first try.

Division of Labor

Jon did the user interface for the project. This included designing the graphics, drawing the pictures to the screen, detecting clicking on the screen, and printing instructions for the user. He was in charge of basically anything that you can see or click. He spent around 50 hours total on all stages of the project, including finding, editing, and creating the images and font, and play testing to make sure everything looks nice and works as it should.

Max did the loading and saving for the game, which converts a game state to a json file and then can convert any valid json file back into a game state that can be played. He also did the formal test suite, which in addition to play testing, made sure that our functions worked as intended. He spent approximately 45 hours on the project in total.

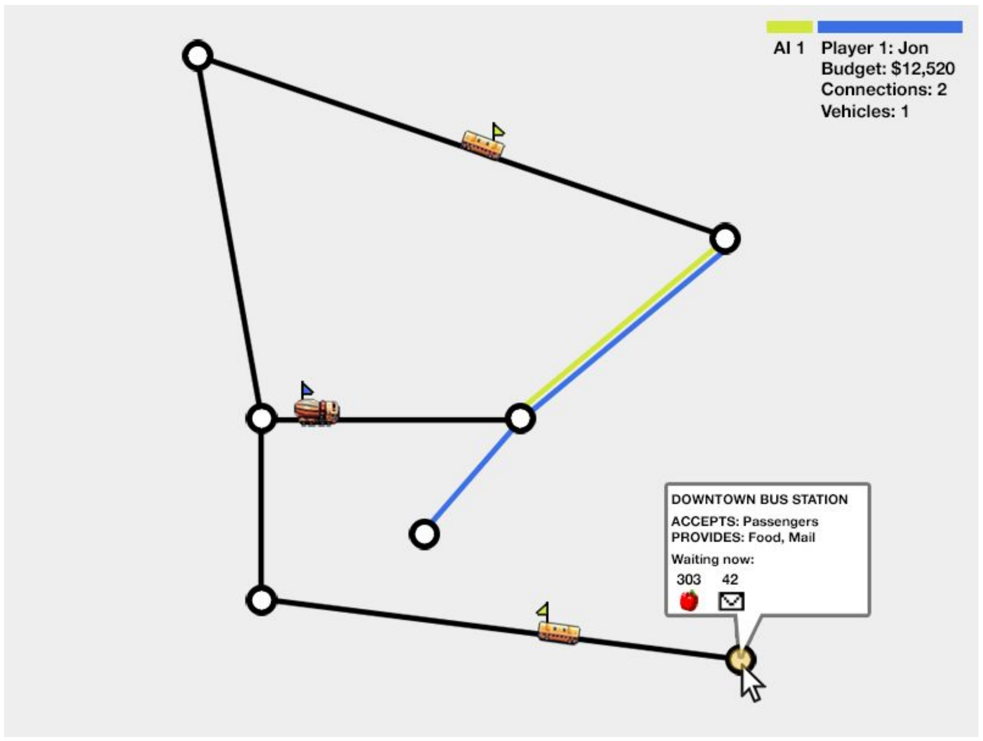
Patrick handled the main loop of the game and the engine that updates the game state to make new frames to be displayed each update. This involved the updating of the gamestate's fields that change every frame, the actual frame updating system that aims to keep as consistent a frame rate as possible, the creation of human processes (i.e. given a set of locations from mouse clicks create a road between two locations) and the handling of all player processes, both human and AI. He spent approximately 55 hours on the project in total.

Dan implemented the AI for the game. In order to make the AI act somewhat intelligently, he had to determine the best way to keep track of information regarding the most profitable roads and good purchases and good sales, which was somewhat difficult. He spent 50 hours on this project.

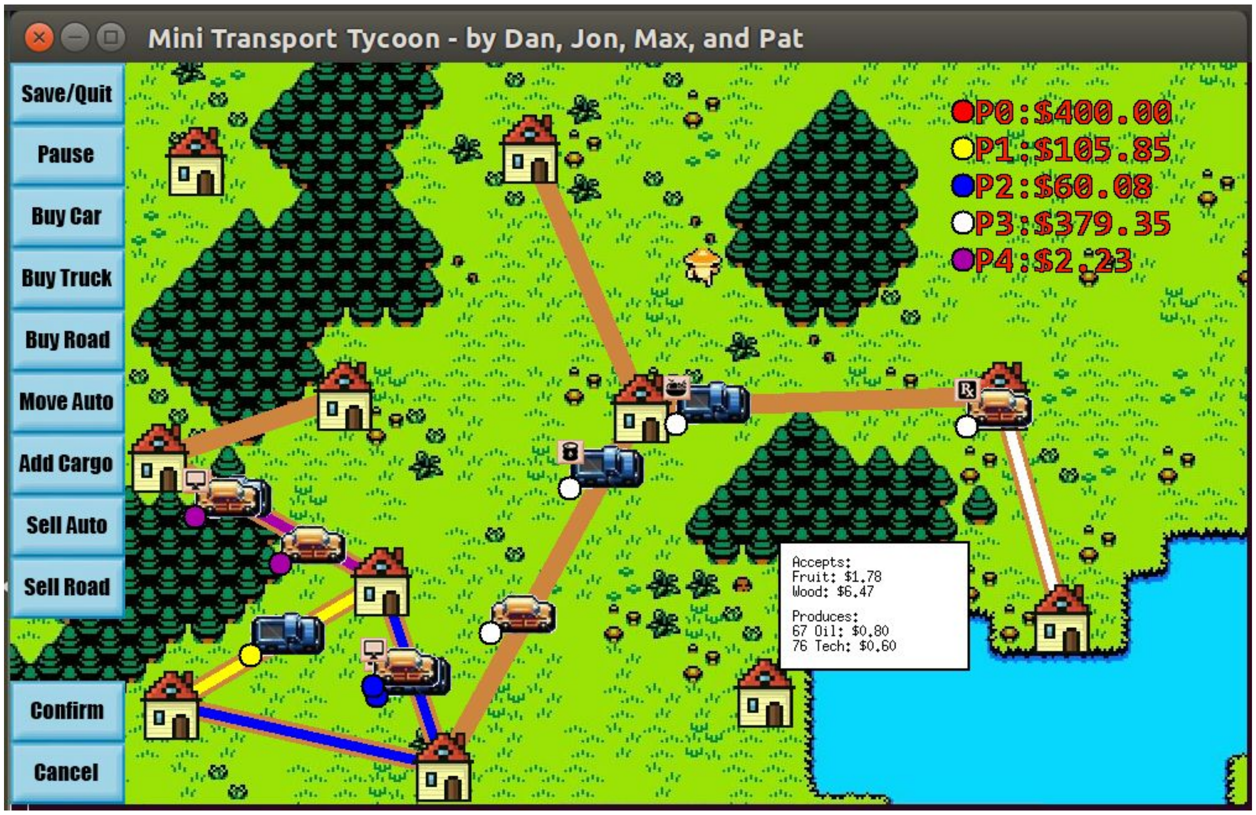
Hours spent on the project include hours spent in design meetings.

<u>Art for the GUI</u>

Original concept art:



Final Product:



```
Accepts:
Fruit: $1.78
Wood: $6.47

Produces:
67 Oil: $0.80
76 Tech: $0.60
```

Updates:

We decided not to implement OpenStreetMap with OCurl, as we felt that they would make the town not look as neat as our predefined map, and we felt our time was better spent ensuring are other features were exactly as we wanted them.

Module changes include a reduction in the functions needed in the player module, which now only includes information on the player type and AI difficulty. Because the AI's moves are determined solely by processes, it made more sense to put the function determining the AI's moves in the InputProcessing module. Similarly, functions that converted player clicks to processes were also implemented in InputProcessing.

The only known "issue" is that sometimes the graphics window flickers while you are playing the game. This does not seem to be caused by inefficiencies or processing problems in our game's logic because the game has about the same amount of flickering even if we triple or quadruple the frame rate by changing an in-game constant. We believe it is a limitation of our VMs, the Graphics module, and the relatively large amount of things we need to draw on screen each turn. There may be a way to improve this somehow, but we do not feel it draws away from the gameplay nor are our algorithms unnecessarily inefficient. We do regulate the frame rate by sleeping for the difference of a set frame duration and the actual amount of time the engine took to process the frame and draw it to the screen by calling a GameGraphics function. So that way, so long as we could do the processing within the maximum time duration of a frame, all frames will have nearly identical time delays between them. This creates nice smooth gameplay as opposed to choppy gameplay where each frame takes longer to update if more processing power is used.

Future goals include further improvements on AI, such as better road construction (dependent on the number of accessible locations from the two locations being connected), more randomness in terms of AI decision making, and randomized map making.

There’s some functionality that’s implemented but turned off via a code-flags (such as vehicles breaking down) that we decided to leave out to reduce the number of complications for the player to handle.

We hope you enjoy playing Mini Transport Tycoon!