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 $400. 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 it for, but may also opt to sell any of the roads or vehicles.

For each vehicle purchased, the player determines the origin and destination that the vehicle cycles through using the available roads, as well as the good to collect and sell. 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 interest to purchase exclusive rights to road segments to disadvantage players for a set period of time.

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 game then ends and you can see how long you played the game for.

Key features:

- **The town of Camelot:**
    - Locations: Nodes that accept/produce a 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 the goods.
- **Money:** budget of each player grows by transporting goods, and is depleted as they 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.

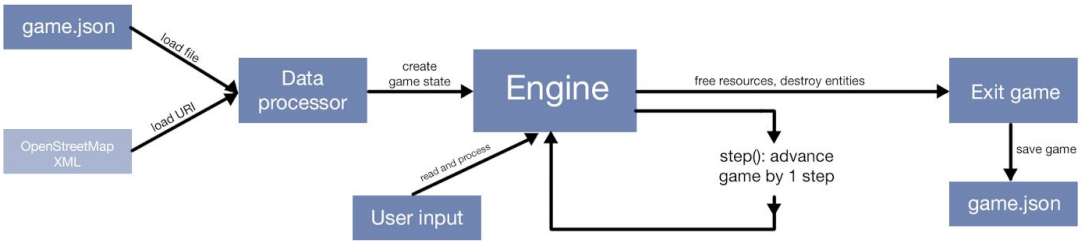
Architecture

The game will use 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,** or **XML data** retrieved from OpenStreetMap (OSM), 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.

Note that OSM data processing is a tentative feature subject to time and resources.



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 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 will call functions and use 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 will check for user input on the screen and through the keyboard and use it to generate specific processes, which will be used to perform an action that modifies the game state. It also contains functions that will be used to convert player inputs to processes as well as determine AI actions.

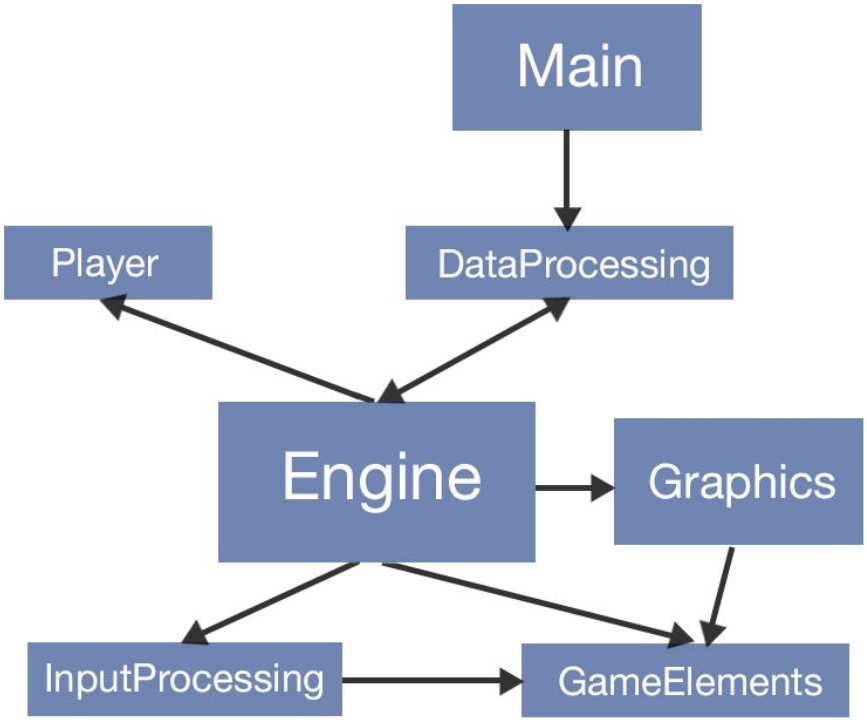
GameGraphics: This module will handle the updating of the display screen. The main display function in the module will take in a game_state and display its representation on the screen. It will also display the buttons needed to interact with the game, and then returns a process list if any of the buttons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

Player: This module will have a type to represent a player (both human or AI) and AI difficulty.

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

> **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 will maintain 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 will be done through the GUI and mouse clicks. **Mouse click locations** will be used to determine player commands, which will update information on each object as necessary. Our game also will store 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.

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.

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 will be used to draw the GUI and record mouse events. It will not be used to change text size, unfortunately, so we made our own font and print 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 will be used to implement the graph representing the vertices (locations) and edges (roads) of the town. OCamlgraph also implements various graph algorithms that we will use (such as Dijkstra's shortest path algorithm).

Piqilib:
Actively developed and versatile data serialization library with support for JSON and XML. This library will be 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

General notes:

Testing will be done function by function with modules. **Glass box testing** and **black box testing** will be done by the writer of the module. Each module should be testable individually, in accordance with the idea that modules should have low coupling.

Ultimately, it will be up to each member to be responsible for writing unit tests for their own code. **The deadline for completing a certain module will be equivalent to the deadline for completing tests.**

Specifics:

Individual module testing will not be too difficult. Those testing road/location updaters will be required to create a graph using the OCamlgraph functors as stated previously, while vehicle and player module writers will create their own vehicle and player lists.

The difficulty that may arise from this project comes from the fact that the engine will ultimately be responsible for ensuring that all of the updates are occurring when they should be occurring, since the game will be real-time rather than turn-based (ideally). Rigorous testing of the real-time loop will require most other module functions to be complete, so it will be done last.

Finally, **gameplay testing** will be used both to extensively check our code for correctness and also to ensure that we create a balanced game that is neither overly easy, nor impossibly difficult to play. We will also have friends with no knowledge of our design play our game to determine if the goals and instructions are clear.

Updates:

Testing was still done incrementally with regards to the actual gameplay. The main difficulty involved with the program was game saving and process handling. Process testing was done incrementally since each process does not depend on any others. The engine was fairly straightforward to implement so the issue described previously in the design document did not come to pass.

AI testing was done through gameplay. As the game is played in real time, it would be fairly difficult practically to determine that the AI was making the right decision at any given time, especially since there exists a certain degree of randomness in the game. It was only possible to test the AI after process handling was complete, but this was done early enough that AI testing could be comprehensive. 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.

Json handling was done by the person writing the Json conversions. Once again, testing was not difficult to implement.

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.

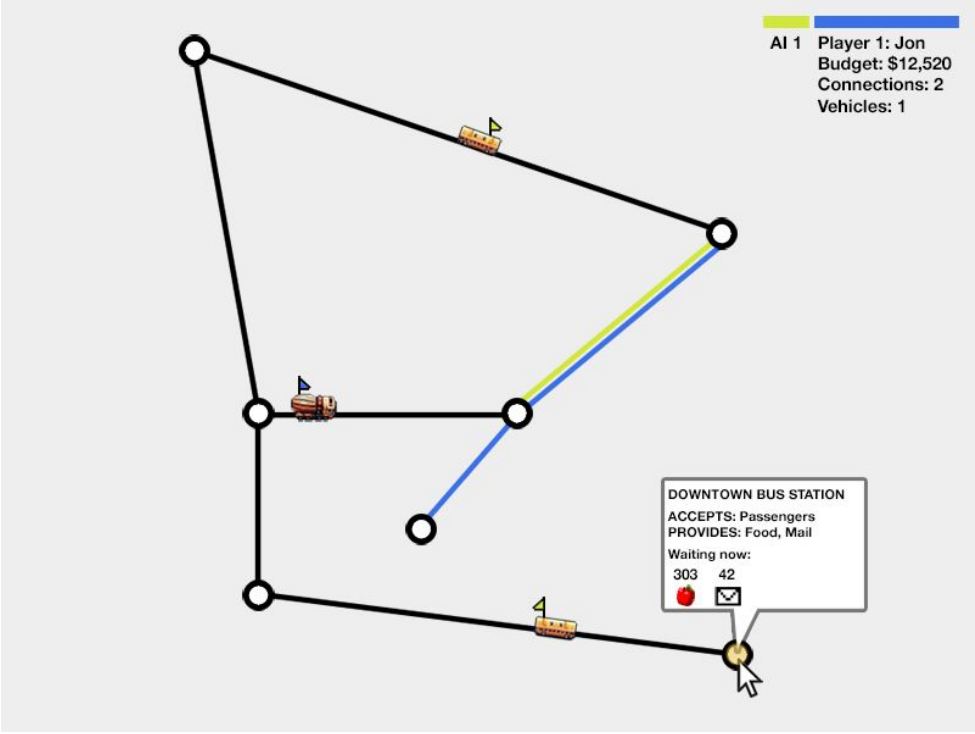
Patrick handled the main loop of the game and the engine that updates the game state to make new frames to be displayed each update. This involved both the handling of game processes and the updating of gameElements that change every frame, the actual frame updating system that aims to keep as consistent 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.

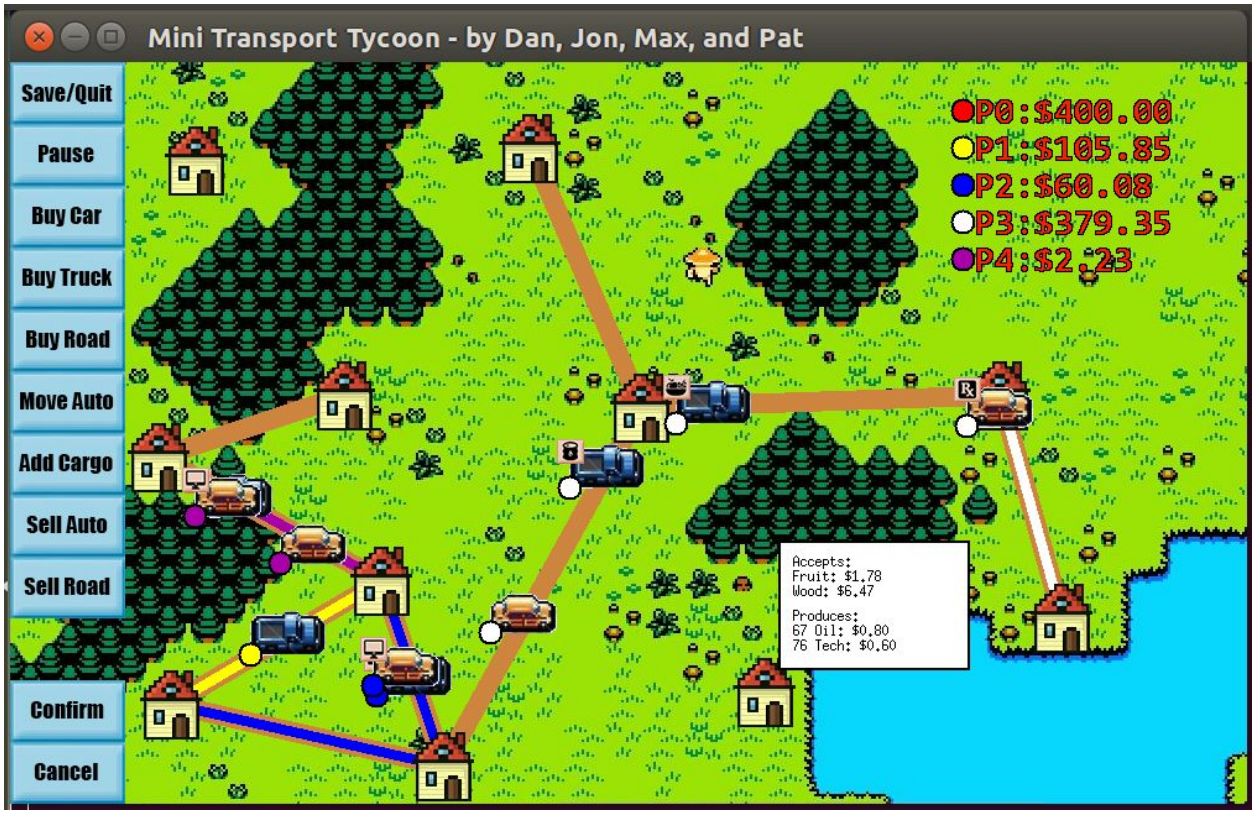
Hours spent on the project include hours spent in design meetings.

<u>Art for the GUI</u>

Original concept art:



Final Product:



General 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.

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.

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) and more randomness in terms of AI decision making.

There's some functionality that's included but not implemented (such as vehicles breaking down) that we decided not to implement in order to reduce the number of complications for the player to handle.