**Amazing Azul Game Report**

This document details the deign choices in our implementation of Azul with respect to data structures, ADTs and overall efficiency.

**Data Structures**

Linked List

We chose to use our implementation of a linked list (TileList) to represent the bag and the lid. This is due chiefly to only ever needing to remove elements from the front of the TileList or add elements to the back.

A linked list has excellent O(1) efficiency for adding or removing elements to the back/front as there are is no need to interact with any elements other than the desired one.

We further improved efficiency by removing the need to iterate through the entire TileList when moving tiles from the lid to the bag. By implementing both the bag and the lid using a TileList, when the bag becomes depleted, we can simply swap the pointers for bag and lid. This results in a replenished bag in the correct order without iterating through any elements.

Vector

Vectors are used in a several areas:

1. A vector is used to store players in the game.
2. A vector is used to store the central pile of tiles.
3. A vector is used to shuffle the tiles before they are then placed in the bag.
4. Vectors are used in other minor areas where a variable length data structure is useful (e.g. storing indexes of elements to be deleted from the center pile)

In each of these situations a vector was chosen for one of two reasons, ease of implementation and future scalability.

In the case of player storage, a vector allows us to easily expand the game to include multiple players in future.

A vector is used to store the central pile of tiles as it is frequently changing size. Ideally a multimap would be used in this situation due to its ability to store collections of elements under a single key, allowing for much easier removal of a given tile type. However, multimaps were out of the scope of this assignment.

Lastly, a vector is used to facilitate shuffling of tiles as LinkedList structures are difficult to dynamically change due to the nature of having to iterate through the list one by one. Thus, storing the data in a vector to shuffle then passing to the linked list is an effective and efficient method.

Array

Both 1D and 2D Arrays are used frequently throughout the system, including factory tile storage, a player’s personal wall, the master pattern wall and as temporary storage in a number of loops.

The main ADT (“Mosaic”), which stores the majority of a player’s data, incorporates a 2D Boolean array to store whether a position on the players wall is filled. It also includes a second 2D TileType array which contains the master pattern. Through this implementation comparisons between the two arrays are used to determine tile location, eliminating the need for char comparisons or the use of upper- and lower-case chars to represent tiles.

Factories are stored in an array as there are a set number (defined in Types.h) which never need to be removed until the program is exited. This is also reflected in how tiles are stored in the factories, again the size of the structure never needs to change, and its contents are never null.

**ADTs**

Our group was influence both by object-oriented programming and the MVC model. As such, we designed and implemented ADTS in a very ‘game-like’ fashion which reflected the real-world structure of Azul. Each section of gameplay was spit up into an object, for example the GameEngine, Menu, Factory, Player, Mosaic, Lines, and Wall.

All game logic is contained in the GameEngine class which contains a vector of Players and an array of Factories. Each player has a name and score, in addition to their own Mosaic which is comprised of their ‘pattern lines’, ‘broken line’ and ‘wall’.

Each of the classes contain all methods needed to retrieve and modify their data. This helps ensure only correct modifications are performed and safe data is returned.

We chose not to implement a Tile class but rather use an Enum. This decision was influenced by two primary factors: a tile has/needs no methods and contains no data, and the game logic does not require distinct tile objects but rather just counts of colours (e.g. 1 Black, 3 Yellow etc.). This decision reduces overall memory usage and drastically reduces the number of pointer and complexity of memory management.

**Efficiency**

The structure of each ADT was carefully considered to maximise efficiency while minimising implementation complexity. For example, the inefficiency of iterating through a whole linked list makes it unsuitable for a factory, which is frequently iterated through at random indexes, but perfect for a TileList where data can be retrieved consecutively.

Another example is our implementation of Tiles, as detailed above.

**Testing**

Our testing was performed using premade save files designed to trigger bugs/crashes. Our general testing methodology is:

1. Create save file for a specific game state, e.g. switching the lid and bag
2. Load save file and perform turn
3. If the program crashes, return to step one but step through with a debugger
4. Otherwise check output save file and compare it to expected output.

We tested common actions and inputs as well as edge cases such as invalid save games, player names with whitespace etc.

In this way we were able to pinpoint and fix most issues. One confusing exception was a seemingly random segmentation fault which occurred when drawing from the center pile.

Our initial implementation used a reverse iterator to erase elements (that had been drawn) from the rear of the vector to the front, thus avoiding segmentation faults caused by the removal of elements changing the indexes of proceeding elements.

Despite handling the scenario where the checked index decreases past the end of the vector, we still encountered random seg faults when drawing. When the program was stepped through with the debugger, the seg faults disappeared. Even more strangely, it seemed that once a specific move had been stepped through with the debugger, that same move (which originally caused the seg fault) would function normally.

Ultimately, we changed our implementation switching from a reverse iterator to a vector which stores the indexes to be erased.

As well as general I/O tests where all possible edge cases are considered in order to ensure quality software. Such as some edge cases for save states considered; Invalid new game, Valid 2nd Round, before winning, etc.

**Group Management/Co-ordination**

Github was employed for code sharing and collaboration, Microsoft teams for general communication and file sharing and a combination of Microsoft teams and RMITs collaborate ultra for video and text conferencing.

In the initial project setup stage, VSCode’s live share plugin was utilised to allow the group to write code on the same machine and view each other’s contributions in real time. At this stage Lucid Chart was also used to collaborate on system design planning.

The group was always open to discussing and implementing changes and also elaborating on what had been changed.

Overall, the group communicated well with no issues and we were able to get the majority of our work done comfortably and on time, in an efficient manner.