The Hong Kong University of Science and Technology
School of Engineering
The Department of Computer Science & Engineering

COMP 2012H - Honours Object-oriented Programming and Data Structures
2018 - 2019 Fall

Programming Assignment 3
Report

Pipes Game

Student: WONG, Tsz Nok
Joshua 20506440

Introduction

Pipes is a game which aims at delivering water from the source to the outlet without any leakage. The game contains a board with numerous individual rotatable blocks. Each block can be one of the following types: T-junction, turn, straight, cross and empty. Player will click on the blocks to rotate clockwise by 90 degrees until the object achieved.

This assignment is using `C++` and `Qt` as a development kit so as to create a GUI.

A new game mode is added to the game which combines the game rules with 2048. An animation modification is also made to enhance the playing experience. The details of the new features will be presented in this report.

Design

Structures & Enumerations

There are few structure and enumeration types added to assist the game development.

`enum BlockType`

The enumeration consists of following type: `TJUNCTION`, `TURN`, `STRAIGHT`, `CROSS` and `EMPTY`. This is designed for enhance readability of the code related to block types.

`struct BFSNode`

The structure consists of the direction of water from and the coordinates of the block. The structure is needed for implementing the queue in a concise way due to these are the only information required for the breadth first search algorithm.

`enum BFSStatus`

The enumeration consists of following type: `CONNECTED`, `LEAKAGE` and `STUCK`. This is designed for enhance code readability in breadth first search algorithm.

`struct BFSResult`

The structure consists of the status and cycles ran in breadth first search algorithm. This is needed to return multiple signals return in the searching.

`struct BlockData`

The structure consists of the type and orientation of the block. This is designed for computing the change in game map in the new feature added. It also only contains required information for the computation.

Data Members & Member Functions in `GameInstance`

In the `GameInstance` class, there are some variables and functions added to facilitate the end game checking algorithm.

`bool isChecking`

A boolean variable for flagging the status of is the end game checking process ongoing. It is designed to prevent calling check game more than once at the same time.

`static const int animateTime`

A integer variable for storing animation interval in milliseconds.

`bool** initTravelled()`

Initialise and return a boolean 2D array with size of `MAP_SIZE` × `MAP_SIZE` for indicating whether the block has travelled in during breadth first search algorithm for end game checking.

`void deleteTravelled(bool** &travelled)`

Delete a boolean 2D array with size of `MAP_SIZE` × `MAP_SIZE`. This is required to prevent a memory leak.

`inline int opposite(int direction)`

Return the opposite direction of the input direction.

`inline int deltaY(int direction)`

Return the change in y-coordinates by following the input direction.

`inline int deltaX(int direction)`

Return the change in x-coordinates by following the input direction.

`void updateBlockImage(int y, int x, bool highlighted)`

Update the block appearance by parameters and indicating the highlighted status of the block.

`BFSResult bfsBlocks(bool animate = false)`

Run breadth first search algorithm and animation and return the searching result.

Data Members & Member Functions in `Block`

In the `Block` class, there are some variables and functions added to facilitate manipulation of the block. Moreover, some variables and functions are changed to use the custom data structures.

`void updateImage()`

Updating the appearance of the block directly.

`void setProperties(BlockType type, int orientation)`

Allow setting type and orientation from GameInstance to facilitate the new feature implementation.

`int direction`

An integer variable of indicating the flow direction of the block. This is implemented in a bitmask structure.

`int get_direction()`

An accessor of the direction variable.

`BlockType type`

The type of block is changed using enumeration `BlockType` to accommodate the implementation.

`BlockType get_type()`

Return enumeration `BlockType` instead to accommodate the implementation.

Implementation

There are some special implementations to achieve the endgame checking condition and animation. The objectives are accomplished by implementing bit mask direction data with the breadth-first search algorithm.

Bitmask Direction Flow

The allowable direction flows of the block are represented by bitmask structure. From the bit at the rightmost position, representing left, up, right and down correspondingly. Such structure is used due to the ease of computation of add multiple directions (bitwise operator `|`), rotate (shift the bits to left, add back the overflow direction and crop overflow bits), check enabled direction (bitwise operator `&`) and remove direction (subtract operator `-`).

Breadth First Search Algorithm

The algorithm is implemented with the structure queue of `BFSNode` called frontier. A 2D boolean array representing travelled block is initialised. The array is used to prevent recomputing the same block and end up an infinite loop. The default status of searching is `STUCK` which is not leaking but also not successfully connect to outlet. The algorithm is initialised with the top-left block with direction flowing from `LEFT` and counting cycle `1` for animating purpose.

While frontier is not empty, the first `BFSNode` is popped and prepare for checking. If the node is in the outlet position, the result will be `CONNECTED`. The node will not spread and continue the algorithm to check any leakage. Then, it checks whether the current node out of the game map, which represents a leakage. Also, it will check whether the node accepts the flow from the direction. If not, it is leaking. For leakage happened, the algorithm will stop immediately and return the result.

Then, if the node is travelled before, it skips spreading water to other direction, otherwise mark the current node as travelled. As water is entered from one direction, the water can only spread to other allowable directions and thus, removing inlet direction. Next, iterate through all available direction of the current node and push a new BFSNode to the frontier. The next iterate node inflow direction will be opposite of the outflow direction of the current node, and its position is computed through the direction used.

The cycle ends and the cycle counter increment by one. The loop continues until the frontier is empty or leakage is found.

Animation

The animation is performed by highlighting blocks in scheduled time. The scheduled time is calculated from cycle counter and timed by `QTimer`. While iterating frontier node, the change of image delays product of animation interval and cycle counter.

New Feature

A new game mode is added to the game which combines the game rules with 2048. The new game mode is available by pressing the button below start named “Feature”. The game starts with an empty game map and adds some random pipes in it. The player can press arrow keys to move the blocks in the map with the same rule as 2048. The blocks will merge according to the following rules:

1. If the blocks are `CROSS`, they will merge into `TJUNCTION`;
2. If the blocks are `TJUNCTION`, they will merge into `STRAIGHT`;
3. If the blocks are `STRAIGHT`, they will merge into `TURN`;

Such rules are defined with different considerations. The blocks will merge into pipes with the less allowable direction which can easily reduce the probability of having leakage in the map. Meanwhile, keeping the game challenging as consecutive non-`TURN` pipes will be merged and destroy what player has constructed. The game has been tested which is possible to complete, with a very challenging play.

An animation modification is also made to enhance the playing experience. Whenever the done button is clicked the water flow animation will start until it reaches leakage or connected.

Design

Data Members & Member Functions in `LoginWindow`

Some data members and member functions are added to `LoginWindow` to start the new 2048 game mode.

`bool startedFeature`

A boolean indicating the current started game is in 2048 mode. It is designed for changing some behaviours in the close game.

`void start_feature_game()`

Launch a new game level of featureLevel.

`void on_feature_button_clicked()`

This function will be invoked when the “Feature” button is clicked

Data Members & Member Functions in `RecordManager`

In `RecordManager`, some extra variables are added for storing a record of new game mode.

`static const QString record_path_feature`

The absolute path where the `record_feature.txt` stored.

`int featureRecord`

The minimal passing step will be stored in the record.

Data Members & Member Functions in `GameWindow`

There is a need to add signal function and related to capturing key press in `GameWindow`.

`void keyPressEvent(QKeyEvent *keyEvent)`

This function will be invoked when key press detected in GameWindow.

`void keyPressed(QKeyEvent *keyEvent)`

Signal function connected to GameInstance.

Data Members & Member Functions in `GameInstance`

The major implementation will be done in `GameInstance`.

`void loadFeatureMap()`

Initialise the game map with empty blocks and randomly add pipes to the map.

`void randomAddPipe()`

Adding pipes to a random empty position. This is needed to follow the rule of 2048.

`BlockData** initBlockData()`

Initialise and return a BlockData 2D array with size of `MAP_SIZE` × `MAP_SIZE` for storing key information of the blocks.

`void deleteBlockData(BlockData** &blockData)`

Delete a BlockData 2D array with size of `MAP_SIZE` × `MAP_SIZE`. This is required to prevent a memory leak.

`void replace(BlockData** blockData)`

Replace blocks with new block information in blockData. This is needed as blockData is used only for computing changes.

`void combine(BlockData &destination, BlockData &part)`

Combine two blocks by merging part to the destination. This is used to follow the rule of 2048.

`void rotateClockwise(BlockData** blockData)`

Rotate the blockData clockwise by 90 degrees so as to allow use fundamental swipe function.

`void swipeLeft(BlockData** blockData)`

Swipe the blocks to the left according to the rules of 2048.

`void swipeRight(BlockData** blockData)`

Swipe the blocks to the right according to the rules of 2048.

`void swipeUp(BlockData** blockData)`

Swipe the blocks to the ceiling according to the rules of 2048.

`void swipeDown(BlockData** blockData)`

Swipe the blocks to the floor according to the rules of 2048.

`static const bool animationChangeEnabled`

A flag for enabling or disabling the animation enhancement.

Implementation

The fundamental swipe function is `swipeLeft(BlockData**)`. For each row, the algorithm detects consecutive non-`EMPTY` same type block and try to merge them. The merge row will then shift all non-`EMPTY BlockData` to the left and padding `EMPTY` block until the row is filled. For other directions, it will first rotate to the equivalent direction and then rotate back to original direction.