CS5617 Software Engineering

Design Specification

Shruti Umat, 111701027, Processing Module Team Member

Overview

We wish to support the following features for our shared white board application -

- Selection of objects
  A client should be able to select any object on the board - object created by them or another client's object.
- Deletion of the selected object
  Similar to selection, a client may delete any object present on the board.
- Show the tag of the selected object
  The tag of an object refers to the username of the client who modified the object most recently.

Objectives

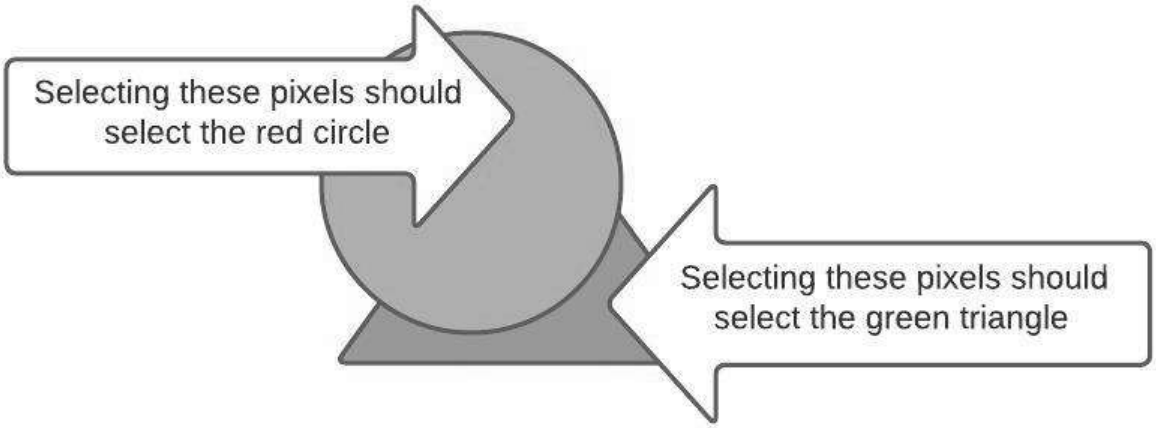
- Provide support for the above features
- Achieve multithreading at the client side operations
  For each processing object received from another client through the network module or the current client's local board objects, a new thread is spawned to execute functions (being run by threads) concurrently on a multi-core operating system.
- Review processing related to color change and rotate features

Design

Few fundamental challenges have shaped and made more precise our requirements and the design choices for the above targeted features.

Challenges

- Several objects can overlap on each other. The question arises: which object shall be chosen from multiple objects whose pixels intersect with the selected pixel region? The topmost object **at the selected pixels** is chosen. For any object to be selected by the user from the UI, there should exist some pixels on the board for which that object is the topmost object. Hence, on clicking at these pixels, the object will be selected.



- In the above case, if the triangle is selected and deleted and later the operation is undo-ed, the triangle should ideally occupy its previous position which is under the circle.
  This requirement indicates that there is an ordering in the board objects in a direction perpendicular to the plane of the board. In literature, this ordering is commonly referred to as **Z-ordering** (if the board is thought of as an X-Y plane, the ordering exists in layers i.e. in the Z-direction).
  Z-ordering is an interesting and complex concept. It can also allow us to implement features like layers, changing layers of objects on our board. Ex: Putting the triangle on top of the circle in the our previous example. However, our team is not targetting these features currently and decided to design the implementation of our targeted features without Z-ordering concepts.

Refined requirements

- Each board object is created in the top most layer, on previous objects that would have been present at those positions.
- Move, change layer of object features are not being implemented. Thus, an object's layer remains fixed (neither can it be brought above to the upper layers nor can it be moved to the lower layers) which is its layer at the time of creation. Further, undo-redo, colour change and rotate operations will not modify the object's layer.

Motivation

As noted above, the layer of an object is the topmost layer at the time of its creation and remains fixed throughout its presence on the board.
A crucial observation is: more recently created objects occupy upper layers (are at the top of the board). A relation exists between the time of creation of object and its layer i.e. higher the timestamp (more recent), higher the layer.

If we can *efficiently* maintain the objects at each pixel position according to decreasing value of their time of creation (`Timestamp` of a `BoardObject`), we can lookup the object with the highest/most recent timestamp at a particular pixel for select operation and conclude that this object is indeed the topmost object at this pixel position. After undo-redo operations, if the object reappears on the board, it must occupy its previous layer i.e. it should be ordered with the currently present objects according to its timestamp at the time of creation. This way, the object will be inserted into its previous layer.

A data structure to efficently maintain ordered elements is a heap. `PriorityQueue` class provided by java provides APIs for building a heap according to a custom comparator on the elements. Note that each pixel position requires a priority queue as different sets of objects are present at each position. However, objects occupy a large number of pixels. Thus, storing a priority queue of objects and their timestamps at each pixel would consume a lot of memory. Thus, at each pixel, we store a priority queue of pair of ObjectId and timestamp as ObjectId requires much lesser memory than an entire object which would have other data members also. We provide an ordered map for lookup from objectId to object to extract other fields of the board object.

The Design

We maintain three ordered maps - a map from `Position` to priority queue of pair of `ObjectId` and `Timestamp` , a map from `ObjectId` to `Object` and a map from `UserId` to username for all the users accessing the current board as part of the board's state, both at the client as well as at the server.

Selection

- The UI will get the positions on the board which are selected by the cursor on an `onClick` event. UI must highlight/display all the pixel positions of the object that got selected. Deletion, tagging, colour change and rotate operations proceed only after selection.
- For each pixel in the list of input pixels from UI, from the priority queue at that pixel (looked-up from the map), the topmost pair of `ObjectId, Timestamp` is looked up (the `top` operation of a priority queue). It is possible that different ObjectIds are present at the top for different positions from the the list of pixels received from UI, ex: selecting pixels at the boundary of two or more objects.

- The `ObjectId` present at the top of maximum number of pixels is chosen as the user most probably wanted to select that particular object.
- With the above most frequent `ObjectId` , the object is looked-up from the second map.
- From the object, the list of positions is returned to UI for display.

Deletion of selected object

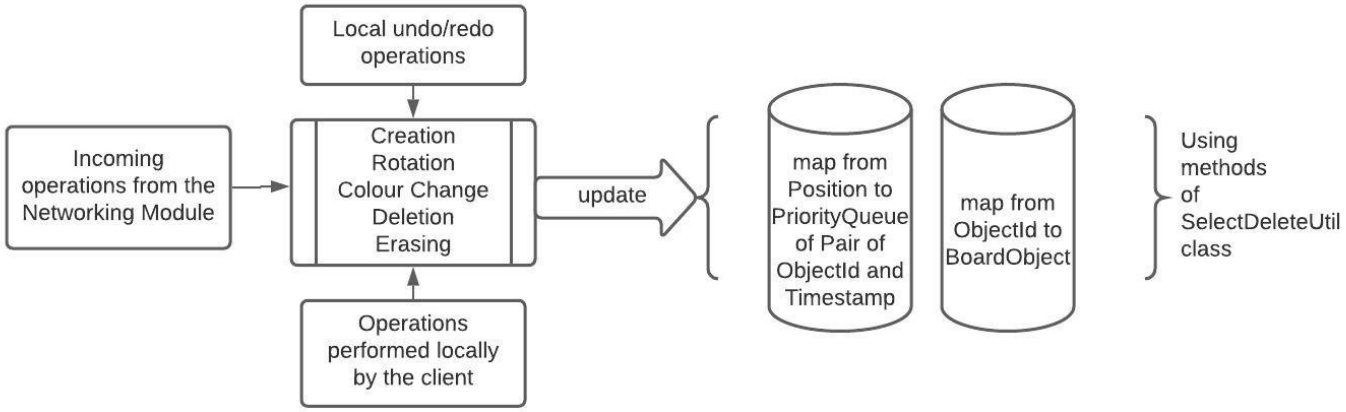
- The selected `ObjectId` 's object is looked-up from the second map and stored as a local variable.
- For each of the pixel positions which the above object covers, the `(ObjectId, Timestamp)` pair is deleted from the priority queue at the pixel.
- The (ObjectId, Object) pair is deleted from the second map.
- The delete operation is pushed into the undo stack so that this operation can be undo-ed/redo-ed.
- `provideChanges()` API in class `CommunicateChange` is called to compute intensity changes at the top of the pixels caused due to deletion. These changes will be published to UI (a subscriber to pixel intensity changes).
- The locally stored object is returned.

Tagging the selected object

- The selected ObjectId's object is looked-up from the second map and stored locally.
- The `UserId` field is read from the object above.
- The username corresponding to this userId is looked-up and returned.
- Unique userIds are generated by appending the user's username obtained from UI and the user's IP address.

For all the above operations to work, it is extremely important that after creation, rotation, colour change, erase (using an eraser), deletion (selecting an object and removing all of its pixels from the board) and undo-redo operations (which boil down to executing above operations), the maps are updated. A section on internal classes contains simple APIs that should be invoked by other classes in the processing module implementing above listed operations to update the maps with the changes caused by their operations.

put



Flow demonstrating how maps are updated when operations are carried out locally or received from the network

Interfaces

The following interface is provided to the UI for performing operations select, delete, colour change, rotate. The last three operations happen on the selected object, thus they are preceeded by the select operation.

```
public interface IOperation {

    /**
     * Returns all the pixel positions for the selected object if an object exists at the
     * @param positions the list of Positions selected by the cursor from UI
     */
    ArrayList<Position> select (ArrayList <Position> positions);

    /**
     * Deletes the selected object
     */
    void delete ();

    /**
     * Changes color of selected objected to specified intensity
     */
    void colorChange (Intensity intensity);

    /**
     * Rotates selected object by specified angle in counter clockwise direction
     */
    void rotate (double angleCCW);

    /**
     * Resets the board to an empty board
     */
    void reset ();
}
```

These classes are defined to abstract commonly used objects like intensity, coordinates of a pixel etc.
** There are more utility classes (ex: for handling `UserId, ObjectId`) but only the relevant ones are mentioned here for explanation.

```
// This class corresponds to RGB values
public class Intensity {
    public int r, g, b;
}

// This class corresponds to row and column of a position
public class Position {
    public int r, c;
}

// This class corresponds to position and intensity at that position
public class Pixel {
    public Position position;
    public Intensity intensity;
}
```

The following `BoardState` class maintains the data structures needed to represent the state of the client's board.
** Only the data members relevant to selection, deletion and tagging are mentioned here.

```
public class BoardState implements Serializable {

        /* maps - IN BOARD STATE */
        // map from pixel to priority queue (max-heap) of (ObjectId, Timestamp)
        // based on timestamp (max timestamp has more priority)
        public static Map <Position, PriorityQueue <Pair <ObjectId, Timestamp> >(0, n


        // map from ObjectId to Object
        public static Map <ObjectId, BoardObject> objIdToBoardObject;

        // map from UserId to Username (String) in our case
        public static Map <UserId, String> idToName;

        // CAUTION : All three maps above are unique for each client
        // Therefore need to synchronize while accessing/updating them

        // Need to synchronize functions using 'selectedPair'
        public static Pair <ObjectId, Timestamp> selectedPair;

        // more data members and member functions ....
}
```

Internal Classes

The `SelectDeleteUtil` class defined below provides methods to update the map from `ObjectId` to `BoardObject` and the map from `Position` to `Priority Queue` of pair of `ObjectId` and `Timestamp` (higher Timestamps are more recent and have more priority). It also has other helper methods to be called by `Operation` class defined later in this section to achieve select, delete and tagging operations.

The `BoardState` class maintains a static data member `selectedPair` that stores the ObjectId and Timestamp of the client's currently selected object. Since it is a shared variable between multiple threads that can write to it (ex: the user selected two different objects in succession and two threads are spawned to handle selection), its get and set methods are synchronized.

The private methods `getMostProbableObjectId`, `insertIntoPQ`, `removeFromPQ` are synchronized and access/update the Position to Priority Queue of (ObjectId, Timestamp) pair map defined in `BoardState` class.
On the other hand, `getBoardObject`, `insertObjectIntoMap`, `removeObjectFromMap` private methods access/update the ObjectId to BoardObject map and hence are synchronized.
These helper functions are provided so that the critical section of the code (accessing/modifying the maps) can be separated out from other non-critical section code like reading fields from BoardObject etc.

The public functions `insertObjectIntoMaps`, `removeObjectFromMaps` call the helper functions to handle the accesses/updates on both the maps.
The `getUserNameOfSelectedObject` returns the username of the user who most recently modified the selected object. These functions need not be synchronized as they perform non-critical operations and call the helper methods above, which are synchronized, to handle the critical section.

```
public static class SelectDeleteUtil {

        /**
         * Synchronized handles to get and set the 'selectedPair' data member
         */
        public synchronized static void setSelectedPair(ObjectId, Timestamp);
        public synchronized static Pair <ObjectId, Timestamp> getSelectedPair();


        /**
         * Gets ObjectIds at the top of Priority Queues at each position (x, y) in gi
         * The ObjectId that is present at the maximum number of positions i.e. which
         * the maximum number of times is returned
         * This is done as the selected pixel positions can be such that
         * some of the pixels belong to an object and some to others.
         */
        private synchronized static ObjectId getMostProbableObjectId(ArrayList <Posit

        /** Gets DrawingObject with the given ObjectId
         */
        private synchronized static BoardObject getBoardObject (ObjectId);

        /**
         * For each Position (x, y) in the input ArrayList, insert the ObjectId, Time
         * Priority Queue at the position (x, y) present in the map
         */
        private synchronized static void insertIntoPQ (ArrayList <Position>, ObjectId

        /**
         * For each Position (x, y) in the input ArrayList, delete the ObjectId, Time
         * Priority Queue at the position (x, y) present in the map, if it exists in
         */
        private synchronized static void removeFromPQ (ArrayList <Position>, ObjectId

        /**
         * Read the ObjectId field from the BoardObject and
         * insert the (ObjectId, BoardObject) key, value pair in the map
         */
        private synchronized static void insertObjectIntoMap(BoardObject);

        /**
         * Remove the input ObjectId's BoardObject from the map
         * Thus, the key, value pair (ObjectId, BoardObject) gets deleted from the ma
         */
        private synchronized static BoardObject removeObjectFromMap(ObjectId);


        /* APIs to Update maps to be used internally by other Processing Module class

        /**
```

```
         * Reads the ObjectId, Timestamp, ArrayList<Positions> fields of the input Bo
         * calls insertIntoPQ and insertObjectIntoMap to update both the maps
         */
        public static void insertObjectIntoMaps (BoardObject);

        /**
         * Performs a lookup in the ObjectId to BoardObject map to obtain the BoardOb
         * corresponding to this ObjectId
         * Reads the ArrayList <Position> and Timestamp fields from the BoardObject a
         * Lastly, it calls removeObjectFromMap with the given ObjectId
         * The deleted BoardObject is passed for serialization if applicable
         * i.e. only if this was a local delete operation and needs to be sent to oth
         */
        public static BoardObject removeObjectFromMaps (ObjectId);

        /**
         * Looks up the BoardObject of the selectedPair's ObjectId using the map
         * From the BoardObject, UserId field is read
         * Calls getUserNameFromId(UserId) function provided by the BoardState class
         */
        public static String getUserNameOfSelectedObject();

}
```

The `Operation` class below imports the above class and invokes its functions to perform selection and deletion.

```
public static class Operation {

    // selects the object present at the maximum number of positions
    // out of the many positions selected by the UI cursor
    public static ArrayList <Position> select (ArrayList <Position> positions);

    // deletes the selected object
    // calls the pushIntoStack function in the UndoRedo class to push the deletion op
    public static void delete ();

    // other functions like rotate, colour change, reset defined in IOperation interf
}
```

Analysis

In this section, we evaluate the performance, safety/reliability and utility of the implementation of the features.

Performance

Let `P` be the total number of pixel positions on the board, `N` be the total number of objects on the board currently and `U` be the total number of users accessing the board.

- Each lookup in the Position to Priority Queue map for a given position takes `O(log(P))` time.
  Note that most of the board remains empty with no object on top of it. Only when a pixel position is actually covered by an object, its priority queue is created and the (Position, PriorityQueue) key, value pair is inserted in the map. Thus, the actual number of key, value pairs in the map
  will typically be less than `P`.

- Insertion, deletion and accessing the top of the Priority queue require `O(log(number of elements in the queue))` time which is upper-bounded by `O(log(N))` in the case when a pixel position is covered by all objects.

- Each lookup in the ObjectId to Object map takes `O(log(N))` time.

- Each lookup in the UserId to Username map requires `O(log(U))` time.

- For `m` selected pixel positions as input from UI, computing the ObjectId selected works in `O(m * log(P) * log(N))` time and returning the list of all Positions of the selected object requires `O(log(N))` time for lookup in the ObjectId to Object map, making the overall selection time complexity as `O(m * log(P) * log(N) + log(N)) = O(m * log(P) * log(N))`.

- For deletion of an object after it has been selected, at each of its positions, the (ObjectId, Timestamp) pair must be deleted from the corresponding priority queue requiring `O(totalNumberOfPixelsInObject * log(P) * log(N))` and `O(log(N))` time for deleting the object from the ObjectId to Object map, making the overall time complexity as `O(totalNumberOfPixelsInObject * log(P) * log(N))`. Pushing into the undo stack requires constant time.

- For tagging an object after it has been selected, `O(log(N))` time is needed to lookup the BoardObject from the ObjectId using the map, `O(log(U))` time is needed to lookup the username from the UserId present in the BoardObject using the map present in `BoardState` class. Thus,
  `O(log(N) + log(U))` time is required for tagging.

- Space complexity for storing the ObjectId to BoardObject map is `O(N)` , for map from Position to Priority Queue is `O(P + (totalNumberOfPixels)_i)` for each object `i` from `{1, 2,... N}` and `O(U * maxLengthOfUserName)` for map from UserId to username.

Security

Since maps are a part of the `BoardState` class and are shared data members, access and update operations are provided using synchronized helper functions. Thus, the API is thread-safe ensuring consistency. Efficiency is also ensured by only synchronizing the methods executing critical section code like update/access of maps. Other methods' non-critical section code can be executed concurrently.

For erasing objects using an eraser or for resetting the board, we create a white object at the pixels where the eraser was used (or on the entire board in case of reset). This `BoardObject` has a boolean `isReset` which is true if and only if the object is a reset object.
In the case that the object is a reset object, `select` will be disabled on it during implementation. Thus, select, delete, colour change and rotate operations would not work on an erased object. This is indeed the desired behaviour as the white object is used to save computation time required to remove elements from the maps and the stacks. The white object can be deleted by an undo operation only.

Usability

Other classes in the Processing module call the public methods in `SelectDeleteUtil` and `Operation` class.
The internal details of synchronizing access and updates on maps and maintaining two maps consistently is abstracted from other classes. The `IOperation` interface is also implemented using functions from the `Operation` class.

Summary and Conclusions

- Time complexities for all operations are logarithmic.
  Ex: For `P = 1e6, N = 1e6` which represents a typical `1000*1000` resolution board with 10,00,000 objects only requires `O(m*20*20)` time for selection of `m` pixels. The select

operation will work within 1 sec if `m <= 2500` (assuming `1e6` number of operations/sec on a typical commodity processor, even though processors are able to execute `1e7` operations/sec today). Typical upper bound for `m` will around `1000` in case of a high resolution board with high PPI (pixels per inch). In practice, the number of objects on the board will be less than 10,00,000 most of the times. Thus, the operations are efficient even for a high resolution board and large number of objects.

- Ordered maps have been used instead of Hash Maps in java to strike a tradeoff between space and time complexity. Ordered maps require lesser memory than hash maps as they only store a value for a key if the key, value pair is inserted in the map. However, lookup time is logarithmic instead of expected constant time lookup provided by a hash map.