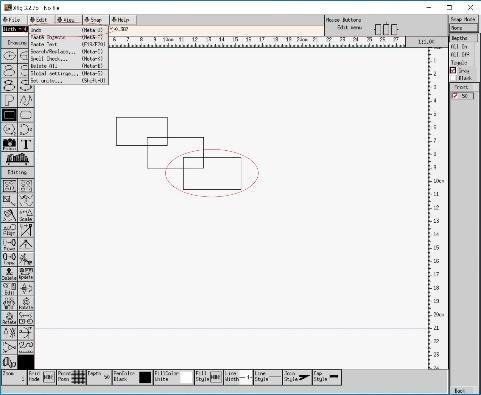
**Software Change Request**

| **Software** | **Baseline Version** | **Feature Name** | **Difficulty** |
| --- | --- | --- | --- |
| xfig | 3.2.8a | Enhanced Undo/Redo | High (Est. 17 files; 850 LOC) |

***Current Behaviors:***

Currently the Undo edit tool (Menu --> Edit --> Undo) allows a user to undo only the last operation such as object creation, deletion, or modification. This would deliver an unnatural behavior. For example, when Undo is clicked twice, the program will go back to the state before the first Undo rather than undoing the previous two actions. Multi-level undo is not supported. Figure 1 illustrates the current behavior through an example figure with three rectangles.

Graphical user interface, application

Description automatically generatedDiagram, engineering drawing

Description automatically generated

*Figure 1: The state before undo and the menu with undo (left), the state after the first undo clicked (middle) – deleting the last rectangle, and the state after the second undo (right) – Adding back the last rectangle*

***Expected Behavior:***

The current Undo feature needs to be enhanced to support multi-level undo. This new feature will allow a user to undo or redo an unlimited number of times. The maximum number of undo/redo operations is only limited by the amount of available memory determined by the computer system. Furthermore, the new feature must support the same set of drawing objects the current undo/redo operations support without any change.

***Solution Hints for Instructor:***

The implementation of the current undo/redo operation is located in source code files *u\_undo.c* and *u\_undo.h*.

One solution is to make use of two stacks to keep track of undo or redo actions. Every time an action is made to the figure, that operation shall be pushed onto the undo stack. When an undo is requested, an action shall be popped out of the top of the undo stack, undone, and pushed onto the top of the redo stack. A redo action shall pop the most recent operation from the redpo stack, “redo” the operation, and push it back onto the undo stack. Figure 2 illustrates the redo & undo stack and their relationship.

Also note that Undo is implemented by redrawing everything except the undone action.

Chart, box and whisker chart

Description automatically generated

*Figure 2: The redo & undo stacks and their relationship*

The solution implements these stacks using singly linked lists of “history” structs. Each one of these objects has a pointer to a compound named saved\_objects, which is used in the default undo feature. The other two important variables saved to the history struct are **last\_action** and **last\_object**. These variables dictate what undo function to execute, in regards to what object or objects. The rest of the values saved in the history struct can be seen in the code section in the object.h file.

Redo is implemented by reusing the undo functions. By default, undoing a second time redoes the last action. This means that internally after an undo, values are set in such a way that the next undo operation is the opposite of the last. For example, if the last operation was to create an ellipse, after undoing the creation of this ellipse (by deleting it), the last action is switched from deletion to creation. This is the reason why different stacks can be passed into each function; redoing and undoing are essentially the same thing.

The difficulty comes from modifications behind the scenes, in addition to the reuse of functions. As an example, the undo\_addpoint() function uses functions that add objects to the master object compound. Since these functions need to be modified to save any changes, the stacks must handle these extra changes accordingly. These severity of these problems range from requiring minor variable name changes to near-complete rewrites of some functions. An approximate difficulty rating of each undo function is included in the Code section.

An additional problem that must be solved is when to save an action to the undo stack. Actions are scattered throughout Xfig’s code. Many exist in u\_list.c, which is where objects are added to the master objects list. However, some actions such as scale or move need to be set in their respective functions.

It should also be mentioned that the current solution does not fully implement some features, such as joining splines, moving points, opening compounds, and rotation. These unfinished implementations will be tagged as such, and should still be able to serve as a general guide unless explicitly noted.

Possible Student Pitfalls

The following are some problems that were encountered while attempting to complete this project:

• While each drawable struct contains pointers to the next struct, these cannot be used to keep track of changes in isolation. Due to poor understanding of the inner workings of Xfig, the first attempt used one compound to keep track of all changes. Any new changes would be added to the end of each list of drawable objects by changing the last object’s pointer. For example, if line\_A was already drawn to the canvas, and line\_B was newly drawn, line\_A->next would be set to line\_B. This does not work. The master objects list uses, and changes, these pointers already.

• An over complication that was made in the process of writing the solution was in the way the history struct was written. In an attempt to save a small bit of memory, each history object contained pointers to each drawable object instead of a pointer to a compound. While this most likely could work, it has the potential to significantly complicate the code. Several helper functions would need to be written, with each necessitating a good understanding of how the function works, due to the fact that existing code could not be reused.

• Ideally, whenever a new action is executed (outside of undoing and redoing), the redo stack should be cleared and freed. This can be the source of problems for a couple reasons. The obvious problem is freeing every object in the redo stack. If an ellipse was created then moved, and the move was undone, a pointer to that ellipse will be in the redo stack (in the move operation history element). Since the ellipse is still created, and a part of the master object list, freeing it will naturally cause problems. The other issue comes from the aforementioned intermediate operations. If any non-undo/non-redo operation causes the redo stack to be cleared, and such operations occur within some undo operations, the redo stack can be cleared by undoing certain actions. In the solution, a workaround in the form of a boolean variable is used. If this variable is true, redo will not be cleared. This can be set at the beginning and end of any problematic undo/redo operations.

• Another freeing-related mistake may consist of not resetting the saved\_objects to Null after each change. If this is not done, lingering objects may be added to the redo stack, and accidentally freed.

• While it isn’t necessarily a pitfall, the key mapping configuration was particularly difficult to find. These bindings are located in the app-defaults/Fig and app-defaults/Fig.in files, at the root of the xfig source download.

• Depth is an internal counter which needs to remain consistent. It keeps track of the number of objects of each type on each layer. If this count is wrong, it can cause objects to appear invisible, or cause Xfig to hang in a loop if objects are linked strangely.

• After modifying an existing object, that object is sent to the back of the master objects list. This means that objects will not always be in the same order after undoing/redoing.

Code

Approx. difficulty of each undo function:

• Add / Delete (Easy)

• Move (Easy)

• Edit (Medium)

• Glue / Break (Medium)

• Load (Medium)

• Scale objects (Medium / Unfinished)

• Scale compounds (Medium / Unfinished)

• Add / Remove point (Medium)

• Add arrow head (Medium)

• Remove arrow head (Medium / Unfinished)

• Convert (Medium / Unfinished)

• Open / Close (Unfinished)

• Join / Split (Medium / Unfinished)

Note: Unfinished means that the code is either partially or not functional at all. Functions not tagged with this work with most objects, but have not been tested with compounds

Files modified:

e\_addpt.c e\_arrow.c e\_convert.c e\_deletept.c e\_edit.c e\_glue.c e\_joinsplit.c e\_scale.c main.c u\_create.c u\_drag.c mode.h u\_free.c u\_undo.h u\_undo.c u\_list.h w\_cmdpanel.c