University of Bergen
Department of informatics

An Effective Generic Lasso Selection Tool for Multiselection

Author: Ole Magnus Lie
Supervisor: Jaakko Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2020

Abstract

Multiselection is widely available in the graphical user interfaces of common applications, often through rectangular or row-wise selection tools. Lasso selection, though often provided in image manipulation applications, is uncommon in applications of everyday selection tasks. Lasso selection would be a useful addition for many applications.

This thesis presents an effective and generic implementation of lasso selection. Its effectiveness is achieved by making the computation incremental: only the elements affected by the extension of the selection path are inspected. The solution is generic, easily reused in new selection contexts.

Acknowledgements

First, I would like to thank my knowledgeable supervisor, Jaakko Järvi, for your insight and helpful ideas when developing this thesis.

Second, a thanks to the friends I have aquired during my five years at the Department of Informatics, University of Bergen. It has been a truly motivational environment for studying, not to mention procrastinating.

Last, a special thanks to my family for your continuous support and help.

`Ole Magnus Lie`
2 June, 2020

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

Lasso selection is a well known selection tool. It is widely used in image editing, but less common in applications for everyday selection tasks, such as selecting files or image thumbnails. Lasso selection is perceived as non-trivial to implement [14]. Performance is a concern: the polygon representing the lasso can become humongous, consisting of a large number of line segments, which adds to the fact that there could be a large number of selectable elements.

Implementations of lasso selection in different applications work differently. Some provide a free hand selection, others a selection made up of line segments defined by mouse clicks. Some applications show which elements become selected while a lasso selection is in progress, but it is also common that the selected state is not shown until the lasso is closed.

We can sum up the motivation for this thesis as follows:

- Lasso selection is often not available where it would be useful.
- There is unnecessary variation in different realizations of lasso selection.
- Implementing a correct and efficient lasso selection is difficult.

1.2 Goal of work

The goal of this thesis is to make it trivial to implement an efficient lasso selection, so that any application programmer could offer lasso selection as one of the standard selection tools in practically any graphical user interface (GUI). We achieve this goal with a generic implementation that is easy to adopt to varying selection tasks.

An application programmer can take advantage of this generic implementation and deliver an efficient lasso selection for pixels in an image, file icons, image thumbnails, points in a scatter plot or even characters or words in a document. The application programmer must of course deal with application-specific details, but our approach gives clear guidance on how.

1.3 Key challenges and solution

This thesis presents solutions to the lasso selection problem based on the generic multiselection model *MultiselectJS* [17] by J. Järvi and S. Parent. In this model, the structure for handling input, the selection state and changes to the state are already in place; the contribution of this thesis is the design and implementation of an effective generic lasso selection algorithm for this context.

The goal of this work was to develop a lasso selection algorithm that is both generic and efficient. There are two main performance challenges: both the number of selectable elements and line segments in the lasso can become arbitrarily large. This work overcomes those challenges through an incremental lasso selection algorithm. The algorithm is incremental in that for each update to the lasso, we quickly identify the elements that *could* have changed their selection state, and inspect those elements only. All other elements are excluded from inspection.

We implemented the framework presented in this thesis in JavaScript for multiselection of HTML elements. Obviously the JavaScript implementation is not suitable for all purposes, for example if the application programmer needs the functionality to be implemented natively, or if the JavaScript engine does not have the required efficiency. The proposed approach is in no way tied to JavaScript; the same generic lasso selection algorithms can be implemented in any language.

Chapter 2

Background

This chapter introduces the concepts that are necessary for describing the contributions of this work. We first define the concept of multiselection. Second, we describe the library on which this thesis builds on. We explain the concepts of *MultiselectJS* [17] and its structures that are needed for creating new multiselection tools. Last, we discuss lasso selection, compare it to a more common multiselection tool, the rectangular selection. We discuss different lasso selection algorithms and explore existing lasso selection tools in common applications.

2.1 Multiselection

Multiselection is a concept where a user has the ability to select or deselect multiple elements rapidly, using a mouse or a keyboard. There are several different tools that deliver such functionality. Most common are rectangular selection and row-wise selection tools. Section 2.2.2 describes both of these tools, and others.

2.2 MultiselectJS

This thesis adds to the *MultiselectJS* library [17] constructed by J. Järvi and S. Parent. The library is described in the paper *One Way To Select Many* [14]. It provides an application programmer different selection tools, and structures for creating new multiselection tools. This section discusses the different concepts the library is based on, its API, and how a new tool can be implemented in this context.

2.2.1 Concepts of multiselection

The library is built with the following concepts in mind. The concepts are commonly mentioned throughout this thesis, so understanding them is essential.

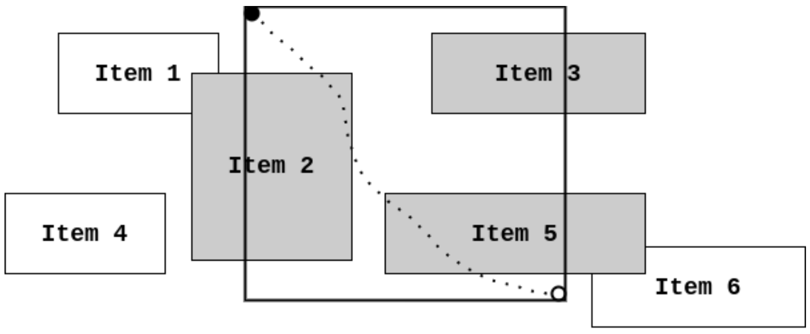


Figure 2.1: A snapshot of rectangular selection demonstrating concepts of multiselection.

Selection state

MultiselectJS assumes that each selectable element has a unique index. The *selection state* is thus a mapping from indices to a boolean value. The value `true` indicates that the element is selected, and the value `false` indicates that the element is not selected. The grey elements in Figure 2.1 all map to `true`, while the white ones map to `false`.

Selection space

The coordinate space where the selectable elements reside in. Points in this space can be represented as X- and Y-coordinate points, indices, or any other data representation that makes it convenient to identify the position of an element in the selection space.

Selection path

The sequence of points in a selection space, indicated by the user with a keyboard or a mouse, representing the selected area. The selection path is showed with the dotted line in Figure 2.1.

Selection operation

One action of selecting or deselecting a group of elements. The selection state is formed as a composition of selection operations. In Figure 2.1 the items 2, 3 and 5 are affected by the selection operation.

Selection domain

The set of elements that are affected by a selection operation.

Selection storage

Selection storage is an object that maintains the selection state. It stores it as a composition of selection operations, and provides methods for manipulating that composition. It also has methods for querying the selection state.

Selection geometry

A selection geometry is a set of functions that defines a selection tool. Different selection tools are defined through their implementation of the selection geometry. The central concept of a selection geometry is the `sdom` function, which defines the mapping from a selection path to selection domain, i.e., how the points a user indicates with a mouse or a finger determine which elements should be selected or deselected. Figure 2.1 shows the common rectangular selection geometry, where the `sdom` function interprets the first and last point of the selection path as a rectangle and determines which elements intersects with that rectangle.

Anchor and active end

The first point in the selection path is called the anchor, visualized as the black dot in Figure 2.1. The last point is called the active end, showed as a white dot in Figure 2.1. Some selection geometries, e.g. the rectangular selection geometry, determine the selection domain based on just these two points.

Mouse clicks

There are three types of mouse clicks. *Click* deselects all elements and discards the existing selection path. A new selection path is created with the clicked point as the anchor, and a new selection domain is computed. *Shift-click* adds a point to the existing selection path. *Command-click* (*Ctrl-click* on Windows) creates a new selection domain, while retaining the current selection state.

Rubber band selection

Rubber band selection refers to dragging the mouse to construct a selection path. Every mouse location recorded during a drag is added to the selection path, effectively working as consecutive *shift-clicks*.

Keyboard cursor

Keyboard commands can be used as an alternative to mouse clicks, e.g. finding the next element in an ordered grid based on an arrow key press. The keyboard cursor is the computed point in the selection space that is associated with a keyboard command.

Active domain

The selection domain computed based on the current selection path. Scenarios with multiple selection domains appear when using *command-click*.

Undo and redo stacks

Selection operations are pushed on a stack, so that they can be undone. Undone operations are pushed to a redo stack, so that they can be redone.

2.2.2 Selection geometries

A selection geometry captures what is specific to a particular selection tool. *MultiselectJS* already provides multiple selection geometries, which can be explored in the *MultiselectJS Demo Application* [18]. We discuss the provided selection geometries below. For each we describe the `sdom` function, which defines how the selection path determines the selection domain.

Rectangular selection

Rectangular selection is one of the most common tools for multiselection. It appears in file managers and desktops in most common operating systems. Image manipulation programs and other creative applications offer rectangular selection for quickly selecting groups of elements, be it figures or pixels in an image.

In rectangular selection, a rectangle is constructed with the anchor and active end as opposite corners, and every element whose bounding box intersects with this rectangle is selected or deselected. In this selection geometry the elements' positions can be arbitrary;

mouse coordinates are used as the selection space.

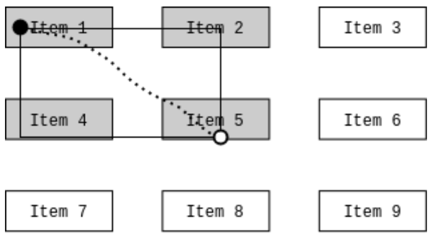


Figure 2.2: Rectangular selection geometry.

Row-wise selection

In row-wise selection, elements are sorted row-wise. Every element that lies between the element at the anchor and the one at the active end are the selection domain.

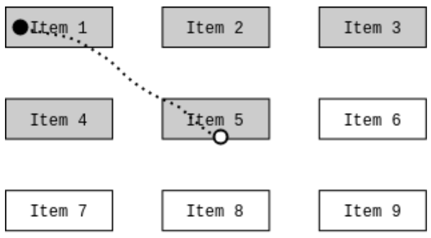


Figure 2.3: Row-wise selection geometry.

Snake selection

In snake selection, the user draws a selection path using a mouse, and every element whose bounding box intersects with the so constructed selection path is included in the selection domain.

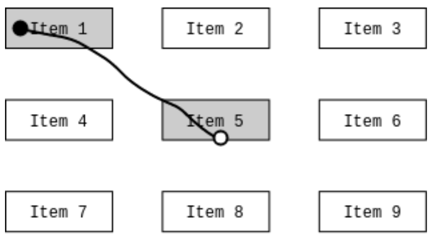


Figure 2.4: Snake selection geometry.

Point-wise selection

The user adds points to the selection path with a mouse drag or a *shift-click*, and every element that touches any of the points in the selection path are included in the selection domain.

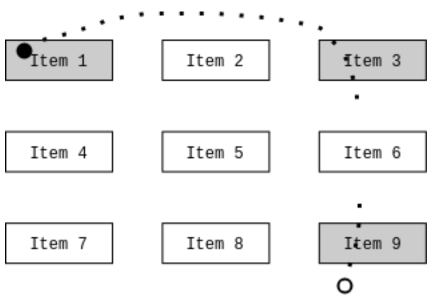


Figure 2.5: Point-wise selection geometry.

Mixed selection

This is a combination of the rectangular and row-wise selection. If the anchor is placed on an element, row-wise selection is applied. If the anchor is outside any element, rectangular selection is applied.

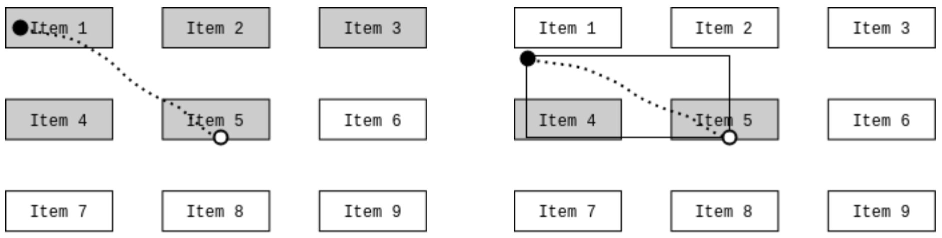


Figure 2.6: Mixed selection geometry.

2.2.3 API

MultiselectJS provides the following API, which we define based on the previously defined concepts. This API allows a programmer to implement multiselection in an application.

Selection state

The `SelectionState` class controls the selection state, selection path, undo and redo stacks, selection storage and which selection geometry is used. We explain all these concepts below, each in its turn.

The selection state is controlled by storing the composition of selection operations. The selection path is controlled by storing a list of points in the selection space. The `SelectionState` class also stores undo and redo stacks. The *undo* function removes the outermost selection operation, and stores it in a redo stack. The selection path is also cleared to make a subsequent *shift-click* predictable, i.e. working as a *click* rather than extending the undone selection operation. When a selection operation is placed in the redo stack, the *redo* function recomposes said selection operation.

When constructing a `SelectionState` object, five parameters can be passed: the selection geometry, a refresh callback function, a flag turning change tracking on or off, the maximum number of undo states and the selection storage. The selection geometry is the only thing that must be specified. Every other parameter has a default value.

The `refresh` callback is executed after every selection operation, passing the selected elements as an argument. This allows a client programmer to define what should happen when an element's selection state changes. If *change tracking* is turned on, a list of the changed elements is also passed to the *refresh* callback, allowing the function to just iterate over the changed elements. The default values of *refresh* and *change tracking* is an empty function and `false`, respectively.

The default implementation of selection storage is a JavaScript `Map`. It stores the selection state as a mapping from the unique indices of the elements to the corresponding selection state. The user can supply another implementation of storage if the default one is not sufficient. If a user decides to do this, the new storage has to implement the same functions as the default one. The API for implementing storage is described later.

The `SelectionState` class provides functions for accessing and modifying the selection of elements, selection geometry, storage and selection path. Functions defining the default behaviour of different mouse clicks are also available.

Selection geometry

The selection geometry classes are what define a selection tool. Therefore, to implement lasso selection means implementing a new selection geometry class. Every selection geometry has to define methods for handling mouse clicks, adding points to the selection path, computing the selection domain from a selection path, moving around with the keyboard and filtering elements based on a predicate function. The methods that handle mouse clicks are called from event handlers in the client, while the other functions are callbacks, only called from the library.

MultiselectJS provides an extendable class `DefaultGeometry`. Every selection geometry, such as the ones discussed in Section 2.2.2, implements its own extension of `DefaultGeometry`. The rest of this section describes the functions that allow an application programmer to create a new multiselection tool using the library.

The points in the selection path are not necessarily in the same coordinate system as mouse coordinates are. The `m2v(mpoint)` method converts the coordinates of the mouse click events' positions into coordinates in the selection space. This function has a default implementation that just returns `mpoint`. The `m2v` function is useful with selection geometries that do not rely upon the exact mouse position but instead, for example, the index of the element the mouse pointer lands on.

The handling of extending the selection path is done by implementing the `extendPath(path, vpoint, cache, cursor)` function. It adds `vpoint`, the new point in the selection space, to the selection `path` array. The `cache` parameter is an object used for storing information between calls to `extendPath`. The `cache` parameter can store anything, giving the application programmer the freedom to store application-specific information that helps in calculating the selection domain efficiently. The keyboard `cursor` can also be modified.

Computing the selection domain is done with `selectionDomain(path, J, cache)`. The function is called immediately after the selection path has been extended by `extendPath`. It creates the new selection domain based on the arguments. The `path` argument is the updated selection path. `J` is the previous selection domain, supplied to allow faster computation of the new selection domain. The `cache` argument is again for storing arbitrary information that should be preserved between subsequent calls to `selectionDomain`. Both `extendPath` and `selectionDomain` get passed the same `cache` object. The `cache` and the previous selection domain `J` are only preserved after a *shift-click* or a mouse drag — when the selection path is extended. When a new selection path is started, through a *click* or a *command-click*, both `cache` and `J` are `undefined`.

The `step(dir, vpoint)` method computes a new point that is "one step" from `vpoint`, in the direction that `dir`, one of the four directions corresponding to the arrow keys, defines. This function is used in keyboard selection to specify how the arrow keys modify the keyboard cursor. It returns `undefined` if the proposed step is not allowed.

The `defaultCursor(dir)` method returns the default location of the cursor when selecting with the keyboard. Again, `dir` is one of the four directions corresponding to the arrow keys.

The `filter(pred)` method returns a new selection domain consisting of the elements that satisfy the predicate function `pred`. This method allows for selecting elements based on other information than their position — selecting all files of type *.pdf* is a typical example.

Selection storage

As previously mentioned, the `SelectionState` class contains a default implementation of storage, using a JavaScript `Map` to store the selection state. However, an application programmer can supply another selection storage. For a selection storage object to be valid, it must implement the same functions as the default one:

- `storage.at(i)` — a function returning the selection state of the element with index i.
- `storage.selected()` — a function returning the indices of all selected elements.
- `storage.push(op, changed)` — a function adding a new selection operation *op* to the composition of selection operations. The `op` argument is the selection operation that specifies the elements to be selected or deselected. The `changed` argument can be undefined, but if it is not, `changed.value` needs to represent the changed indices from the preceding change to the composition of selection operations.
- `storage.pop(changed)` — a function removing and returning the previous selection operation. The composition of selection operations cannot be empty when running this function. If the `changed` argument is defined, it should contain the changed indices from the previous selection operation.
- `storage.top()` — a function returning a reference to the latest selection operation. The composition of selection operations cannot be empty when running this function.
- `storage.top2()` — a function returning a reference to the penultimate selection operation added to the composition. The composition cannot consist of less than two selection operations when running this function.
- `storage.size()` — a function returning the number of selection operations in the composition.
- `storage.bake()` — a function applying the latest selection operation to the base selection mapping, resulting in a new base. The applied selection operation is removed from the composition. The function has no effect if there are no selection operations stored in the composition.

- `storage.onSelected(J)` — a function determining if a selection domain J indicates exactly one selected element.
- `storage.modifyStorage(cmd)` — a function used to modify the storage, e.g. selecting, deselecting or toggling elements. The `cmd` parameter is the command that indicates how storage should be modified. The commands and their effects are defined by the client.
- `storage.equalDomains(J1, J2)` — a function determining if two selection domains $J1$ and $J2$ are equal.
- `storage.isEmpty(J)` — a function determining if a selection domain J is empty.

2.3 Lasso selection

This section first compares lasso selection to a comparable multiselection tool: rectangular selection. Second, we explore different lasso selection algorithms. Last, implementations of lasso selection in common applications are discussed.

2.3.1 Lasso selection compared to rectangular selection

Lasso selection is a free hand selection tool where a user constructs a selection path using rubber band selection. It can be seen as a generalization of the more common rectangular selection, where the user specifies a rectangle using a mouse and every element that is inside, wholly or partially, becomes selected.

In rectangular selection, the inner points of the selection path do not matter: the selected area is the rectangle constructed based on the anchor and the active end. In lasso selection, however, the selected area is the polygon determined by the entire selection path. Figure 2.7 shows the selection of the same elements using the two different tools. In both graphics, the solid black dot is the anchor, the white dot is the active end, while the dotted line is the selection path. In the rectangular selection graphic the solid black line represents the selected area. In the lasso selection graphic, the solid line is the last line segment of the polygon, the line from the active end back to the anchor.

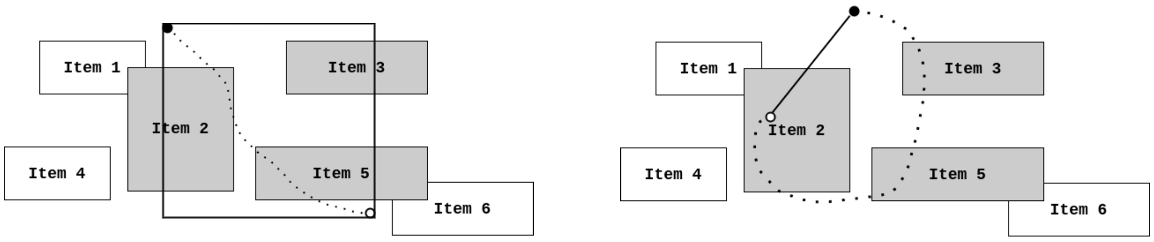


Figure 2.7: Rectangular selection and lasso selection of the same elements.

Lasso selection gives the user more freedom to specify which elements should be selected. While rectangular selection is effective in selecting large numbers of elements when they are organized in a row, column or a grid, a lasso is more effective when selecting smaller subsets of elements whose placement is irregular. Lasso selection supports the selectable elements being of varying shapes without any difficulty.

Lasso selection is sometimes more convenient even when elements are organized in a grid. Figure 2.8 shows an example. Rectangular selection limits the shape of the selected area, while lasso selection lets the user include exactly the desired elements.

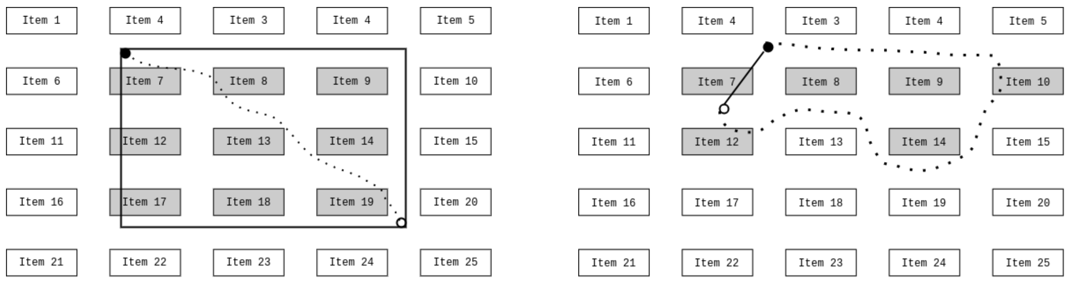


Figure 2.8: Lasso selection supports the selection of a set of arbitrarily placed elements.

2.3.2 Lasso selection algorithms

The workhorse of lasso selection is an algorithm that solves the following problem: for a point p, check if it lies inside the polygon defined by the line segments $[(p_1, p_2), (p_2, p_3)...(p_n, p_1)]$. Such a point-in-polygon algorithm is performed on a point that lies within the selectable element in question. There are multiple algorithms that solve this kind of a problem. J. Hao et al. [8] discusses and compares the different algorithms in their article. We present some known point-in-polygon algorithms that can handle both concave and self-intersecting polygons.

Ray-intersection

Ray-intersection [8], also known as ray-crossing [8, 10], crossings-count [7, 8], odd parity [8, 9], odd-even [8, 9] or even-odd [6, 8, 10] is a thorougly studied point-in-polygon algorithm. It is based on a simple idea: cast an infinite, horizontal ray from the point p, and count how many of the polygon's line segments the ray intersects with. If the number of intersections is even, the point is outside of the polygon. If the number is odd, the point is inside of it.

Figure 2.9 shows the idea the ray-intersection point-in-polygon algorithm is based on. The line from the green point has one intersection with the polygon, an odd number, and is deemed inside the polygon. Both red points, however, have two intersections with the polygon, an even number, and are not inside the polygon.

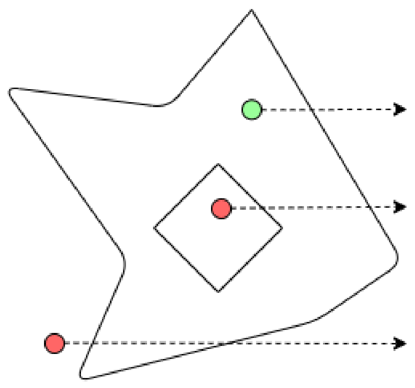


Figure 2.9: Ray-intersection point-in-polygon idea.

The algorithm is stable for all kinds of polygons, both concave and self-intersecting, but it is well documented that simplistic implementations of the algorithm can have problems finding the correct number of intersections if the point lies directly on or close to the polygon's edge [8].

Sum of angles

The sum of angles algorithm [8, 15, 11], or angle summation algorithm [8, 10, 7], solves the point-in-polygon problem by computing the angles where lines from each point in all the line segments $[(p_1, p_2), (p_2, p_3)...(p_n, p_1)]$ intersect through p, adding the signed angles together. If the total angle sum is zero, or close to zero, the point is outside the polygon. If it is anything other than that, the point is inside [7].

This algorithm has some obvious drawbacks: computing the angles is not efficient, and the algorithm is susceptible to rounding errors.

Winding number

The winding number algorithm [8, 10], or non-zero winding number algorithm [8, 9], counts how many times the polygon revolves counterclockwise around a point p. Thus, one needs to compute the direction of each line segment — either through angle calculations that are expensive, or through cross and dot products. If the number of counterclockwise revolutions is zero, p is outside of the polygon.

The winding number algorithm classifies points in self-intersecting areas as inside.

2.3.3 Lasso selection in common applications

Many variants of lasso selection appear in commonly used software. In this Section we describe some of these variants, and highlight some of their drawbacks and peculiar behaviours, if any.

jQuery

The JavaScript library jQuery supplies a plugin called *Selectable Widget* [16]. The library's documentation states that this widget provides lasso selection. While they may classify it as such, in our terminology jQuery's *Selectable Widget* provides a regular rectangular selection tool; free hand lasso selection is not available.

Adobe Photoshop

Adobe Photoshop has three different lasso selection tools [1]. All three support four operations to manipulate selection paths. *New* creates a new selection domain. *Add to* expands an existing selection domain. *Subtract from* removes the selected area from an existing selection domain. *Intersect with* creates a new selection domain consisting of the intersection of multiple selected areas.

Adobe's *Lasso tool* is a free hand selection tool. An anchor is defined with a mouse click, and the selection path is extended with a mouse drag. When the mouse is released, a line segment from the active end to the anchor is added, resulting in a closed selection path.

The *Polygonal lasso tool* is used for creating straight-lined polygons representing the selected area. The user constructs a polygon with mouse clicks, where each click defines the end of one line segment and the start of a new one. The polygon is closed when a line segment returns to the anchor, or with a double-click.

The *Magnetic lasso tool* is used for selecting complex objects against high-contrast backgrounds. The user defines the selected area using a free hand lasso tool with a mouse, and the tool snaps the added points to the desired edge. If it does not position correctly, anchors are available for modifications of the selection path. There is no public documentation on the implementation of this tool, but if we were to hazard a guess, edge detection is probably used.

GNU Image Manipulation Program

GNU Image Manipulation Program, GIMP, provides a lasso selection tool [23]. The user creates an anchor by clicking the starting position with a mouse. From there, the user can either use a mouse drag to construct a free hand polygon, or continue by clicking, defining straight line segments with each click. The polygon is closed when it is connected to the anchor, or with a double-click. There is no visual representation of the selection domain while selecting, only the selection path that is being constructed is visible.

Social Explorer

Social Explorer is a tool for creating reports and selections based on geography. In this software, a lasso selection tool is available for selecting countries or states.

There are three different variations of the lasso tool [5]. *Touching* selects every country that touches, or is enclosed by, the selection path. *Enclosed* selects every country that is completely enclosed by the lasso polygon defined by the selection path. That means that countries that have even one pixel outside of the lasso are not selected. *Centroid* selects every country where the center point is enclosed by the lasso. This seems like an arbitrary criterion, chosen for ease of implementation. We are left to wonder if there are countries whose center point is not within its own boundaries.

Social Explorer's lasso tool has some drawbacks [4]. First, if a region is selected and one wishes to create a new one, every selected element has to be manually removed — just creating a new selection domain will not deselect the old one. Second, there are no dynamic updates to the selection domain while the user performs the selection — only the selection path is shown while selecting, leaving the user without confirmation of whether the selection path specifies the desired elements. The user has to close the polygon, by double-clicking the mouse, before the selected elements are highlighted.

Chapter 3

Implementation

The lasso selection tool is essentially an adaptation of the snake selection tool defined in *MultiselectJS*. While the snake selection selects all elements that touch the selection path, lasso selection selects all elements that lie inside or on the polygon defined by the selection path. The lasso selection algorithm essentially determines whether two polygons, the element and the lasso, intersect. This can become a very expensive computation with arbitrarily large numbers of both line segments and elements. Throughout this chapter we present heuristics that mitigate the cost of the computation.

This chapter first presents the workhorse of the lasso selection: the point-in-polygon algorithm. We explain how we can use the functionality of snake selection to keep the point-in-polygon algorithm as simple as possible, while making the drawbacks it presents irrelevant. The element-in-polygon algorithm was implemented for use on rectangular HTML elements. We describe how the algorithm might need modification if the shapes of the elements are irregular.

We then present a non-incremental lasso selection implementation using the API presented in Chapter 2. We explain how the methods of the `SelectionGeometry` class are implemented, and the thinking behind our design choices. We describe this implementation of lasso selection as *non-incremental* because the selection domain is computed by running the point-in-polygon algorithm on all elements whose bounding box intersects with the bounding box of the selection path. Similar to the snake selection geometry, the implemented lasso selection tool supports the removal of points at the end of the selection path, in essence undoing a part of the lasso, by following the path backwards.

At last, we describe how the implementation can be made more efficient by making it *incremental*. Simple geometrical observations help us further eliminate lots of elements when computing the selection domain.

The implemented functionality presented in this chapter can be accessed at `https://github.com/omlie/multiselectjs`. The code exists in the *./multiselectjs/js/* folder. The selection geometries are implemented and exported from *../js/html_geometries.js*. The selection geometries imports the utilities module, containing multiple helper functions, from *../js/utilities.js*. The *MultiselectJS Demo Application* [18] has been modified to include the lasso selection tools. The demo application can be run locally by cloning the repository and accessing *../examples/demo/multiselect-demo.html* in a browser.

3.1 Selectable element contained by selection path

Determining if an element is contained by a selection path, and should be selected, is accomplished with a simple ray-intersection algorithm. The algorithm presented is a modified version of the one presented by P. Bourke in 1987 [2]. It has similarities to the ray-intersection algorithm presented by J. Hao et al. [8], which they found to be the optimal point-in-polygon algorithm. P. Bourke's point-in-polygon algorithm might struggle to correctly classify the point if it lies on the selection path. This worry is solved by using the function the snake selection geometry uses in *MultiselectJS* [17] to determine if an element intersects with a line segment of the selection path. If the element does not touch the selection path, every coordinate of the bounding box is either completely contained by the polygon, or completely outside of it. This allows us to apply the point-in-polygon algorithm to just one coordinate of the element.

The point-in-polygon algorithm requires a closed polygon to determine if an element is contained by a lasso. This is achived by adding to the processed polygon the line segment from the active end to the anchor, i.e. from the last point of the selection path to the first point.

The element-in-polygon algorithm presented in Listing 3.1 takes a path and the bounding box of an element as arguments. The `pathRectIntersect` function determines if the element intersects with the path, and allows us to run the point-in-polygon algorithm on just one coordinate of the element. The inspected point can be any point that lies within the element — we choose the top left corner of the bounding box as a representative point of it. This is the point p. The following steps are applied to p, to determine if p is contained by the polygon, or if the element touches the path.

Listing 3.1: Element-in-polygon algorithm.

```
const rectangleInPolygon = (path, rectangle) => {
  let counter = 0;
  let p1 = path[0];
  const p = topLeftCorner(rectangle);
  for (let i = 1; i <= path.length; i++) {
   let p2 = path[i % path.length];

   if (p.y > Math.min(p1.y, p2.y)) {
     if (p.y <= Math.max(p1.y, p2.y)) {
      if (p.x <= Math.max(p1.x, p2.x)) {
        if (p1.y !== p2.y) {
         const xIntersect =
           ((p.y - p1.y) * (p2.x - p1.x)) / (p2.y - p1.y) + p1.x;
         if (p1.x === p2.x || p.x <= xIntersect) counter++;
        }
      }
     }
   }
   p1 = p2;
  }

  if (counter % 2 !== 0) return true;
  return pathRectIntersect(path, rectangle);
};

const pathRectIntersect = (path, rectangle) => {
  let p1 = path[0];
  for (let i = 1; i <= path.length; i++) {
    let p2 = path[i % path.length];
    if (lineRectIntersect(p1, p2, rectangle)) return true;
    p1 = p2;
  }
  return false;
};
```

1. Loop through the path, starting and ending at the anchor, checking the following to determine if a ray through p intersects with the line segment defined by two consequtive points p_1 and p_2. When finished, continue to step 2.

   1.1. Lines 8 and 9: if the Y-coordinate of p is between the smallest and largest Y-coordinates of p_1 and p_2, continue to step 1.2. If not, the line segment is not relevant — the horizontal line segment starting from p cannot intersect with line segment defined by p_1 and p_2.

   1.2. Line 10: if the X-coordinate of p is smaller than the largest X-coordinate of p_1 and p_2, continue to step 1.3. If not, there is no possible intersection to the right of p and the line segment is not relevant.

   1.3. Line 11: if the line segment defined by p_1 and p_2 is not horizontal, continue to step 1.4. If it is horizontal, p lies on the line segment and an intersection is not possible.

   1.4. Line 13: calculate the point of intersection of a line horizontally from p, and the line segment defined by p_1 and p_2. Continue to step 1.5.

   1.5. Line 14: if p_1 and p_2 have the same X-coordinate, intersection is guaranteed based on line 10; increase the counter. If the X-coordinate of p is smaller than the point of intersection, the two lines intersect; increase the counter.

2. Line 22: if the counter is odd, the element is inside the polygon. If not, continue to step 3.

3. Line 23: the inspected point is outside of the path, but other parts of the elements might touch it. Check if the element intersects with the path, and return the corresponding boolean value.

The element-in-polygon algorithm is available from the *utilities* module in the repository. Note that the algorithm is implemented for use on rectangular elements, where elements are guaranteed to fill their entire bounding box. If the shapes of the elements are irregular, there is no guarantee that the top left corner of the bounding box lies within the inspected element. There is also no guarantee that an element intersects with the path even though its bounding box does. In such cases, this algorithm can be used as inspiration, but the application programmer has to ensure that the inspected point lies within the element, and that detecting intersections with the path is handled correctly.

3.2 Non-incremental lasso selection

This section describes a non-incremental implementation of lasso selection. We construct a new selection geometry, `LassoGeometry`, and implement the methods of the API that need overriding. The cache is used to store information that helps us compute the selection domain faster. How the implementation handles the extension of the selection path is presented before describing how the selection domain is computed.

Methods with default implementations

There is no ordering of elements in lasso selection — they can be positioned arbitrarily. We therefore use mouse coordinates as our selection space, making use of the default implementation of the `m2v` method.

Using the lasso selection with a keyboard is hardly useful. Lasso selection is meant to be used with a mouse, and therefore the `step` and `defaultCursor` methods need not be implemented.

Cache

Minimizing the amount of potentially selected elements is essential in making the computation fast. The cache is used to store information that helps eliminate elements that need not be inspected. It is initialized when the first point is added to the selection path, be it through a *click* or a *command-click*. Listing 3.2 shows the initialization of the `cache` object.

Upon initialization of the cache, a number of fields in the `cache` object are created:

- `removing` — a flag indicating if points should be removed, defaulting to `false`.
- `prevp` — a field for storing the previous point in the selection path, used when determining if points should be added or removed.
- `subAreas` — an object that essentially works as a mapping from a position to the indices of nearby elements.
- `rectangles` — a list of bounding boxes that surround the selection path.
- `pqueue` — a list of objects representing a queue of update commands.

One way to reduce the number of elements that must be inspected is to quickly identify elements that are positioned far away from the selection path, and ignore those elements.

Listing 3.2: Initialization of cache.

```
LassoGeometry.prototype._initCacheIfEmpty = function(cache) {
  if (Object.keys(cache).length == 0) {
   cache.removing = false;
   cache.prevp = undefined;
   cache.subAreas = util.splitSelectableArea(this.parent,
   ↪ this.elements);
   cache.rectangles = [
     {
      left: Number.MAX_SAFE_INTEGER,
      right: Number.MIN_SAFE_INTEGER,
      top: Number.MAX_SAFE_INTEGER,
      bottom: Number.MIN_SAFE_INTEGER,
     }
   ];
   cache.pqueue = [];
  }
  this._cache = cache;
};
```

For this purpose, the `subAreas` field in the `cache` object are created. It gets initialized using the `splitSelectableArea` function. This function splits the selectable area into smaller areas with a height of 100 pixels. Each subarea can be accessed by looking up the upper limit of said area, i.e. for the area ranging from 0px to 100px, the key is `100`. This enables us to look up the subarea for any point in the selection space, be it mouse event coordinates or the position of an element. Each subarea stores a list containing the indices of the elements that lie within said area. In the case of an element overlapping multiple subareas it is added to all of them. Note that a subarea is only created if an element lies within it. Listing 3.3 shows an example of how the subareas are represented. We use the `subAreas` object to quickly access the elements that lie near the selection path.

Listing 3.3: Example representation of subareas.

```
{
  100: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ],
  200: [ 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ],
  300: [ 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 ],
}
```

The splitting of the selectable area is not a necessary optimization when there are only a few elements. However, when the number of elements is large, this optimization significantly speeds up performance.

Another heuristic that reduces the number of elements that need inspection is to check if candidate elements intersect with the bounding box of the selection path. Candidate elements are efficiently eliminated — every element that does not intersect with the bounding box does not intersect with the polygon defined by the selection path. For every update to the selection path, the corresponding bounding box is added to the `rectangles` list. Figure 3.1 shows how the bounding box expands when new points are added.

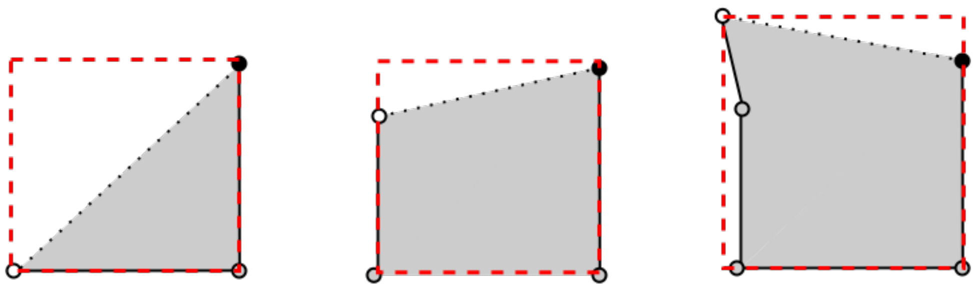


Figure 3.1: Bounding box expands for each new point.

Passing information about the extension of the selection path to the computation of the selection domain is done by storing a queue of objects in the cache, the `pqueue` field. The objects represent commands that the *sdom* function processes to determine the selection domain. Each command contains information that helps computing the selection domain efficiently. The updated selection path and its bounding box is stored in the object, ensuring immutability of the information between the calls to the `extendPath` and `selectionDomain` functions. The `subAreas` object is used to get elements that potentially lie inside the bounding box. The indices of the elements to be inspected are found by concatenating all subareas that the bounding box lies within, removing all duplicate elements, and storing the list in the command.

Extend path

For every new point p added to the selection path during a lasso selection, `extendPath(path,` ↪ `p, cache)` is called. The function determines if p implies removal of points in the *path*, or if p should be added to the selection *path*.

This function is also where point removal is realized. The point p indicates removal of the selection path if p makes the path turn more than 135 degrees. This will set the `cache.removing` flag to `true`. It will remove all points within 20 pixels of the point. Removal continues until a new point moves further away from the current last point. The `cache.prevp` field is updated for each new point, to determine if a new point moves further away than the one that indicated removal. This is the same functionality as found in the snake selection tool in *MultiselectJS* [17].

During removal of the selection path, the previously computed bounding box and the previous point of the selection path is popped from their respective lists. The previous rectangle, the updated selection path and the list of elements to be inspected are added to the queue of commands in the cache that the *sdom* function executes.

If p is added to the selection path, the previous bounding box of the selection path is updated to surround the updated selection path. It is added to the list of bounding boxes, and queued as a command in the cache. Adding them to the queued command as well as the list of bounding boxes is done to ensure immutability of the bounding boxes used when the selection domain is computed.

Selection domain

The `selectionDomain` function executes each of the commands queued in the cache. It does so by calling `_forEachAffectedByLine` with the following arguments:

- The bounding box of the selection path, defined in the command.
- The list containing the indices of the elements up for inspection, defined in the command.
- The selection path defined in the command.
- A function for selecting elements: setting the selection state of an element to `true`.
- A function for deselecting elements: setting the selection state of an element to `false`.

This method, defined in Listing 3.4, applies multiple functions to determine if the elements up for inspection intersect with the polygon defined by the selection path.

1. Line 14: if the element intersects with the bounding box of the selection path, continue to step 2. If not, the element is not relevant and the iteration is over.

2. Line 15: if the element intersects with the polygon defined by the selection path, select the element. If not, deselect the element.

Listing 3.4: Computation of the selection domain.

```
LassoGeometry.prototype._forEachAffectedByLine = function(
  rectangle,
  inspectableElements,
  path,
  select,
  remove
) {
  for (const index of inspectableElements) {
   const offsetRectangle = util.getOffsetRectangle(
     this.parent,
     this.elements[index]
   );

   if (util.rectangleIntersect(rectangle, offsetRectangle))
     util.rectangleInPolygon(path, offsetRectangle)
      ? select(index)
      : remove(index);
  }
};
```

3.3 Incremental lasso selection

This section describes an incremental implementation of lasso selection. We discuss the incremental concepts and explain how they propose changes to the non-incremental implementation.

3.3.1 Speeding up selection

This section explains approaches for speeding up selection and making the solution incremental.

Making the bounding box smaller

Minimizing the number of elements that need to be inspected is done by creating a bounding box of the triangle defined by the anchor and the two last points of the selection path (called affected triangle from here). The elements that intersect with the affected triangle are the only ones that can have changed during this update.

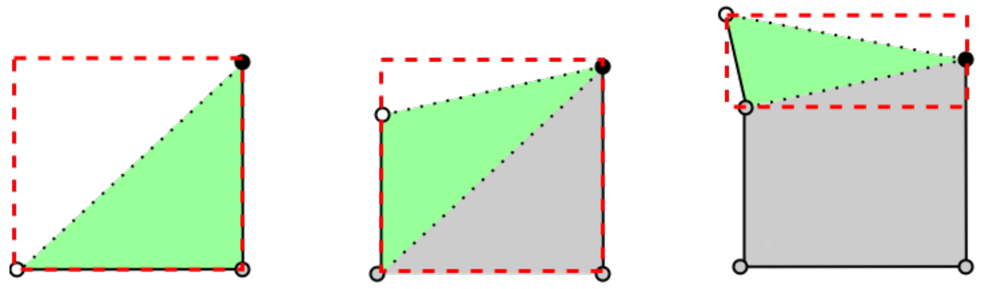


Figure 3.2: Bounding box surrounding the affected triangle.

Instead of updating the previous bounding box, a new one is computed. This allows us to get the indices of the elements that are close to the new bounding box, using the `subAreas` object. Instead of expanding the number of elements looked at, the only elements inspected are the ones that potentially are affected by the change to the selection path.

Defining the changed area

The idea in the previous section can be extended to narrow down the area even more. The only elements that can have a change in selection state are the ones that lie in the affected triangle defined by the anchor and the two last points of the selection path. This area is often smaller than the corresponding bounding box. Running the point-in-polygon algorithm on every element inside the bounding box of the triangle, with affected triangle as the path, allows us to eliminate even more elements. If the element is inside the affected triangle, the point-in-polygon algorithm is applied with the full selection path supplied.

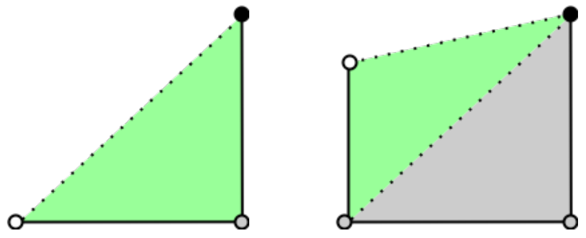


Figure 3.3: Affected triangle with addition of new points.

Figure 3.3 shows the affected triangle when new points are added. The solid black point is the anchor, the grey ones are the inner points of the selection path, and the white one the active end. The green area is the affected triangle.

The leftmost graphic in Figure 3.3 shows the affected triangle after three added points, which is the whole polygon. The rightmost graphic shows the situation after a fourth point is added. Only elements that intersect with the affected triangle need to be inspected.

3.3.2 Implementation

The implementation of an incremental lasso selection differs very little from the non-incremental implementation proposed in the previous chapter. The difference is the information supplied to the queued commands in the cache.

Cache

To achieve the proposed changes, a modification of the cache is required. The initialization of the cache is the same, but the information stored is different.

The stored rectangles are no longer the updated bounding boxes containing the whole selection path, but the bounding boxes of the affected triangle.

The information supplied to the queued commands in the cache have also changed. A bounding box is still passed, this time the one surrounding the affected triangle. The affected triangle is also passed. The `subAreas` object is used to get all subareas that the affected triangle lies within. The list containing all elements in said subareas are added to the command.

Extend path

The method of deciding if a new point represents removal of or addition to the selection path remains the same as defined in Section 3.2.

The only difference is that the computed bounding boxes stored in the cache are not updated versions of the previous one. For each addition to, or removal of the selection path, a new affected triangle appears. Every computed bounding box surrounds its respective affected triangle, and not the updated selection path.

Selection domain

Computing the selection domain is very similar to the non-incremental version, with a small change in `_forEachAffectedByLine` presented in Listing 3.5. The affected triangle is supplied as an additional argument, and it is used to quickly deem elements relevant or irrelevant.

Listing 3.5: Incremental computation of the selection domain.

```
 1 LassoGeometry.prototype._forEachAffectedByLine = function(
 2   triangle,
 3   rectangle,
 4   inspectableElements,
 5   path,
 6   select,
 7   remove
 8 ) {
 9   for (const index of inspectableElements) {
10    const offsetRectangle = util.getOffsetRectangle(
11      this.parent,
12      this.elements[index]
13    );
14
15    if (util.rectangleIntersect(rectangle, offsetRectangle))
16      if (util.rectangleInPolygon(triangle, offsetRectangle))
17       util.rectangleInPolygon(path, offsetRectangle)
18         ? select(index)
19         : remove(index);
20   }
21 };
```

1. Line 15: if the element intersects with the bounding box of the affected triangle, continue to step 2. If not, the element is not relevant and the iteration is over.

2. Line 16: if the element intersects with the affected triangle, continue to step 3. If not, the element is not relevant and the iteration is over.

3. Line 17: if the element intersects with the polygon defined by the selection path, select the element. If not, remove the element.

The addition of Step 2 drastically improves performance, further elaborated on in Chapter 5.

3.4 Visualizing lasso selection

Both the non-incremental and incremental implementations of lasso selection implements a `drawIndicators` function, allowing an application programmer easy access to visualization on a canvas. The function takes multiple parameters that determine how the visual representation should be.

- `selection` — the `SelectionState` class. Required to access information about the selection, e.g. the anchor, active end and the selection path.
- `canvas` — the canvas to draw on.
- `drawAnchor` — a flag to determine if the anchor should be visualized as a red circle.
- `drawCursor` — a flag to determine if the cursor should be visualized as a blue circle.
- `drawRubber` — a flag to determine if the selection path should be visualized as a green rubber band.
- `drawPathToAnchor` — a flag to determine if the auto-closing state of the lasso polygon should be shown as a grey line from the active end to the anchor. Defaults to `true`.
- `drawBoundingBox` — a flag to determine if the current bounding box, being used to exclude elements from inspection, should be visualized as a red rectangle. Defaults to `false`.

By calling the `drawIndicators` method with the desired arguments after every mouse event, a user can achieve simple visualization of the lasso selection being performed.

Chapter 4

Programmers guide to adopting lasso selection

This chapter explains how an application programmer can use the presented lasso selection tool in an application.

4.1 Accessing the implemented functionality

Both the library, utilities and the selection geometries presented in this thesis are available as CommonJS modules. A programmer can use them by importing them. Listing 4.1 shows how they are aquired. Note that `html_geometries` gives access to all selection geometries presented in this thesis, not just the lasso selection tools.

Listing 4.1: Import CommonJS modules.

```javascript
let multiselect = require("multiselect");
let multiselect_html_geometries = require("html_geometries");
let multiselect_utilities = require("utilities");
```

Alternatively, the programmer can access the modules by adding them as scripts in a HTML file. Listing 4.2 shows how to add the modules as scripts.

Listing 4.2: Import scripts.

```html
<script type="text/javascript" src="./dist/multiselect.js"></script>
<script type="text/javascript" src="./dist/multiselect_html_geometries.js"></script>
<script type="text/javascript" src="./dist/multiselect_utilities.js"></script>
```

4.2 Initializing the implemented functionality

Once the functionality is aquired, the user has to initialize the selection geometry to be used. Both the `LassoGeometry` and `IncrementalLassoGeometry` selection geometries have a constructor that takes the selectable area and the selectable elements as arguments. Both the selectable area and the selectable elements are left for the user to define. Listing 4.3 shows how the selectable area and the selectable elements can be defined programatically, before initializing the `IncrementalLassoGeometry`.

Listing 4.3: Initializing the selection geometry.

```
const selectableArea = document.createElement("div");
selectableArea.className = "selectable_area";
document.getElementsByTagName("body")[0].appendChild(selectableArea);

for (let i = 0; i < 500; ++i) {
  let e = document.createElement("span");
  e.setAttribute("class", "selectable");
  e.textContent = i;
  selectableArea.appendChild(e);
}

let selectables = selectableArea.getElementsByClassName("selectable");

let incrementalLassoGeometry = new
   ↪ multiselect_html_geometries.IncrementalLassoGeometry(
  selectableArea,
  selectables
);
```

When the programmer has defined the selection geometry, the `SelectionState` class can be initialized. As defined in Chapter 2, the constructor of the `SelectionState` class only requires the selection geometry to be passed. Listing 4.4, however, shows how to use a refresh callback function along with *change tracking* to quickly change the class of the selectable elements once their selection state changes. This way their appearance can change by applying CSS styles to the class `selected`.

Listing 4.4: Initializing the `SelectionState` class.

```
const refresh = (selected, changed) =>
  changed.forEach((value, index) =>
    selectables[index].classList.toggle("selected", value)
  );

let selectionState = new
   ↪ multiselect.SelectionState(incrementalLassoGeometry, refresh,
   ↪ true);
```

Once the `SelectionState` class is defined, the application programmer needs to handle mouse clicks and keyboard events, and call the corresponding functions in `SelectionState` accordingly. Visual representation of the anchor, the active end, the selection path and other visualizations that help the user determine the selection domain also need to be defined. However, all selection geometries in `html_geometries` implement a default visual representation. They can be used through the `utilities` module. The module contains example functions to initialize a canvas in the selectable area, as well as setup of mouse and keyboard events that trigger changes to the selection state and uses the default implementation of visualization in the selection geometry. Listing 4.5 shows how to use the example functions to handle mouse and keyboard events, as well as initialize the canvas used to provide a visual representation of the selection.

Listing 4.5: Setup canvas, mouse and keyboard events.

```
let canvas = multiselect_utilities.createCanvas(selectableArea);

multiselect_utilities.setupMouseEvents(selectableArea, canvas,
   ↪ selection);
multiselect_utilities.setupKeyboardEvents(selectableArea, canvas,
   ↪ selection);
```

As can be seen, all code is more or less boilerplate. The programmer does not have to think about geometry, interaction protocols or other difficult aspects.

Chapter 5

Experiments

5.1 Performance tests

To test the performance of the proposed lasso selection algorithms, a simple JavaScript test bed was constructed. We recorded the performance of multiple use cases of the lasso tools that we implemented. For each selection case, the results were stored in a JSON file. Every file contains performance data for 500 selections. For each selection, we recorded the total number of selectable elements, total number of performed clicks, total time elapsed in milliseconds, mean time per click and the percentage of selected elements of all selectable elements.

We wanted to test the performance of the implementations relative to the number of both elements and points in the selection path. Measuring how the number of elements impacts the performance, a randomly chosen number of selectable elements were created, between 0 and 10,000. Keeping the number of clicks constant allowed us to see how the number of elements impacts performance. For testing how the number of points in the selection path affect the performance, we used a constant number of elements and a random number of points in the selection path, between 3 and 10,000.

Every performance test first generates the selectable area and the selectable elements. The clicks are generated and stored in an array. This is done before measuring the time elapsed. Once the timing starts, every click in the array is processed and the time to compute the selection domain, determined by the constructed selection path, is measured. The only difference between the different performance tests are the number of elements and how the clicks are generated.

The results are visualized using *matplotlib* [12] in *Python*. The variable number — the number of elements or the number of points in the selection path — is plotted against the X-axis, while the mean time per click is plotted against the Y-axis. A red regression line shows the general slope of the results.

The test bed, results and visualization script can be found in our repository at GitHub [20].

5.1.1 Selecting all elements with four clicks

This section discusses the performance of lasso selection tools in a scenario where all elements are selected. The lasso polygon is formed by simulating a *shift-click* in each corner of the selectable area.

Motivation

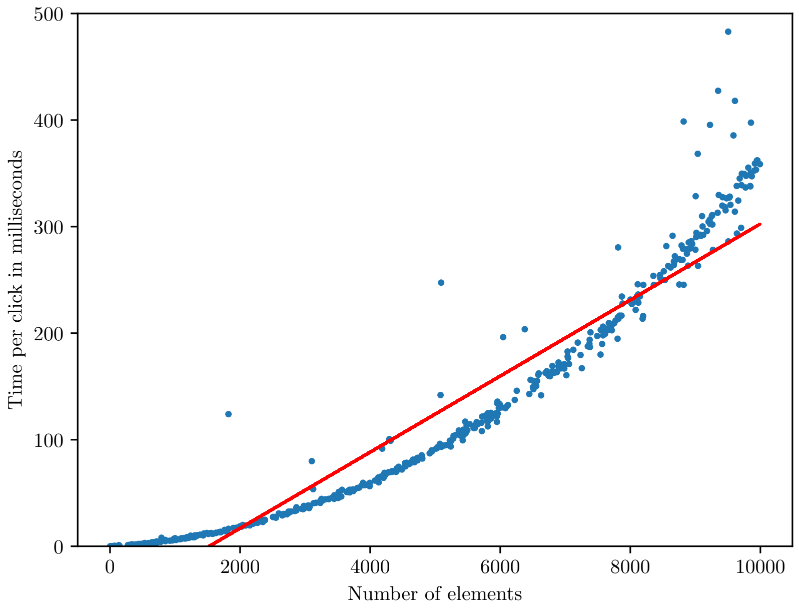
This performance test was done to explore if there is any change in performance between the non-incremental and the incremental lasso selection, and how the number of elements impacts performance.

Discussion

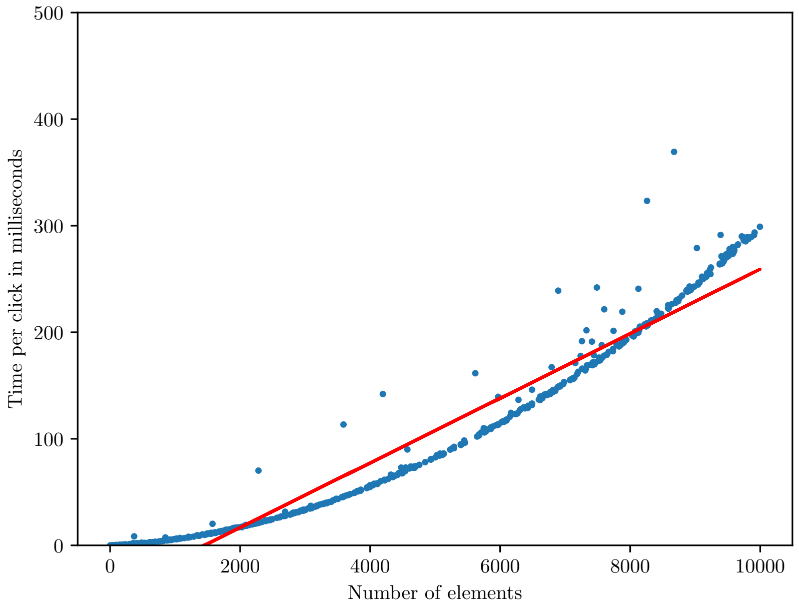
Comparing the graphics in Figure 5.1, one can see that on average the performance of both implementations are close to equal. However, the incremental implementation is both a bit faster and more stable, and has fewer outliers than the non-incremental implementation.

The reason for the similar performance of the two implementations is obvious: after the second click the bounding box is just a straight line, but after the third and fourth click the bounding box contains all elements. The only difference between the implementations is the incremental implementation's exclusion of elements that do not intersect with the affected triangle. This means that the incremental version runs the element-in-polygon algorithm on about half of the elements on the third click, and the other half on the fourth click. The non-incremental implementation, however, runs the element-in-polygon algorithm on all elements twice. This explains the tiny performance boost of the incremental implementation.

An interesting observation is that neither tests show any noticable increase in running time based on the number of elements before it reaches 2000.



(a) Non-incremental lasso selection.



(b) Incremental lasso selection.

Figure 5.1: Performance of the lasso selection implementations while selecting all elements with four *shift-clicks*.

5.1.2 Selecting all elements with 400 clicks

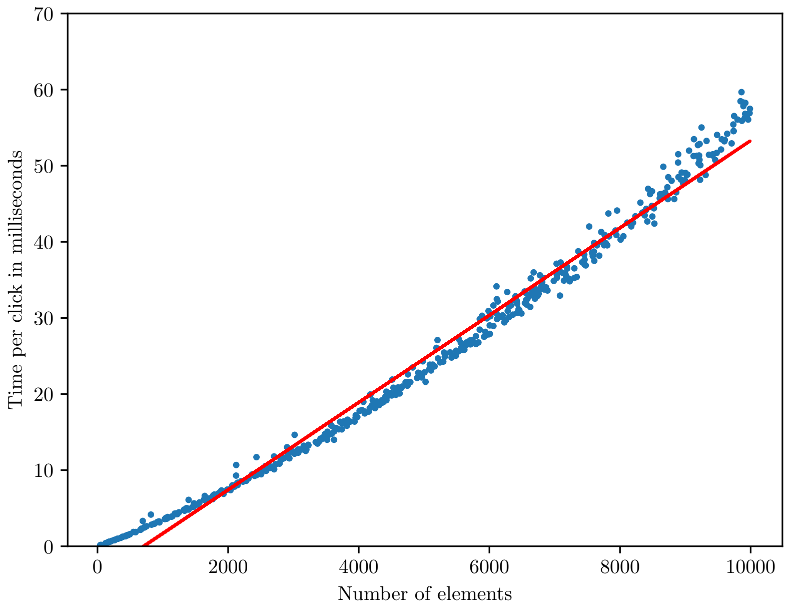
This section explores if the incremental lasso selection tool is faster than the non-incremental one with smaller and more realistic increments to the selection path. For this purpose, we constructed a test where all elements were selected, now using 100 *shift-clicks* along each edge of the selectable area.

Motivation

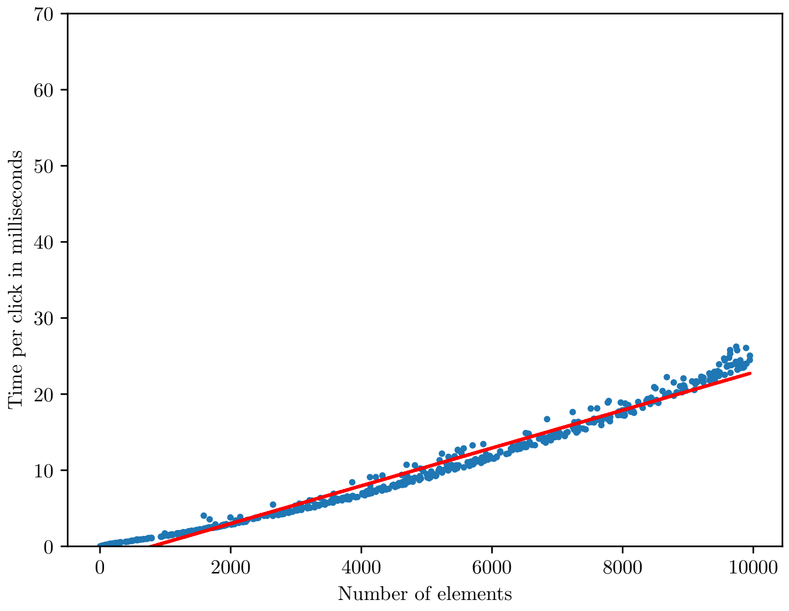
Since the incremental lasso selection tool inspects only a small number of elements for each addition to the selection path, the test should favor the incremental implementation over the non-incremental. We record the performance of the two implementations with smaller increments to the selection path to explore if the incremental implementation has the desired effect.

Discussion

Figure 5.2 shows where the incremental lasso tool thrives — smaller increments to the selection path reduces the number of inspected elements. The performance is both faster and more stable compared to the non-incremental implementation. The reason for this is that the non-incremental lasso tool inspects an ever-increasing number of elements, while the incremental one only inspects elements in the affected triangle.



(a) Non-incremental lasso selection.



(b) Incremental lasso selection.

Figure 5.2: Performance of the lasso selection implementations while selecting all elements in 400 *shift-clicks*, 100 clicks near each edge.

5.1.3 Selecting a random lasso with 500 clicks

While recording the performance of selecting all elements is useful in exploring the efficiency of the algorithm, it is not a realistic use of lasso selection. Other selection tools, rectangular rubber band selection, in particular, are more useful in performing such a task. We want to record how the lasso selection performs while processing selection tasks that are analogous to those that would occur in practice. For this purpose we generate lasso polygons by choosing points at random, but with restrictions that ensure realistic polygons.

Motivation

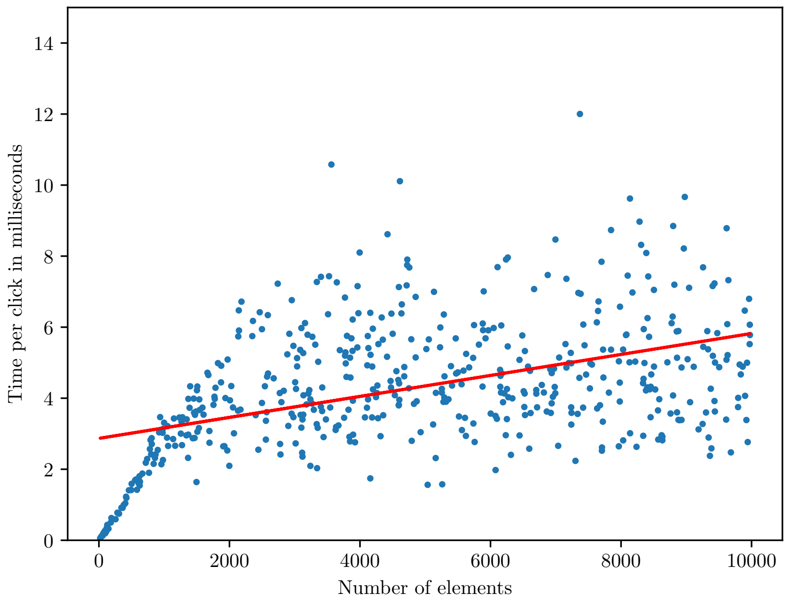
To mimic a realistic use of a lasso selection, a user drawing a large polygon during a mouse drag, we constructed a test that used lassos generated by 500 random *shift-clicks*. We compared the performance of the non-incremental and incremental implementations on these selection tasks.

To ensure that the randomly generated points form a realistic lasso, we picked each new point within 100 pixels of the previous point. The clicks might trigger both addition and removal of points in the selection path. They might also lead to self-intersecting lasso polygons.

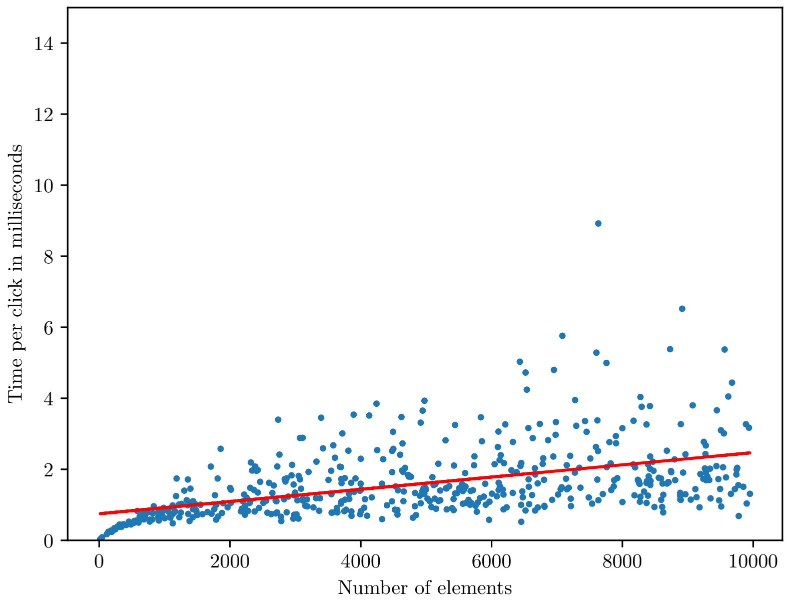
Discussion

Studying Figure 5.3 we can see that the performance is less stable than in the previous sections. This is to be expected — random selection paths lead to more spread in the running times. What is important to note is how the number of elements affect the running time. The non-incremental implementation has a steeper increase in running time, a higher minimum, and more outliers than the incremental solution.

Having a considerable amount of spread no matter how many elements there are in the selectable area is reassuring: it means that the performance is not halted by the number of elements, but rather the complexity of the selection path.



(a) Non-incremental lasso selection.



(b) Incremental lasso selection.

Figure 5.3: Performance of the lasso selection implementations while selecting a random polygon constructed by 500 *shift-clicks*.

5.1.4 Selecting a large lasso with random number of clicks

This section explores how both implementations of the lasso selection tools perform with a variable number of line segments in the selection path.

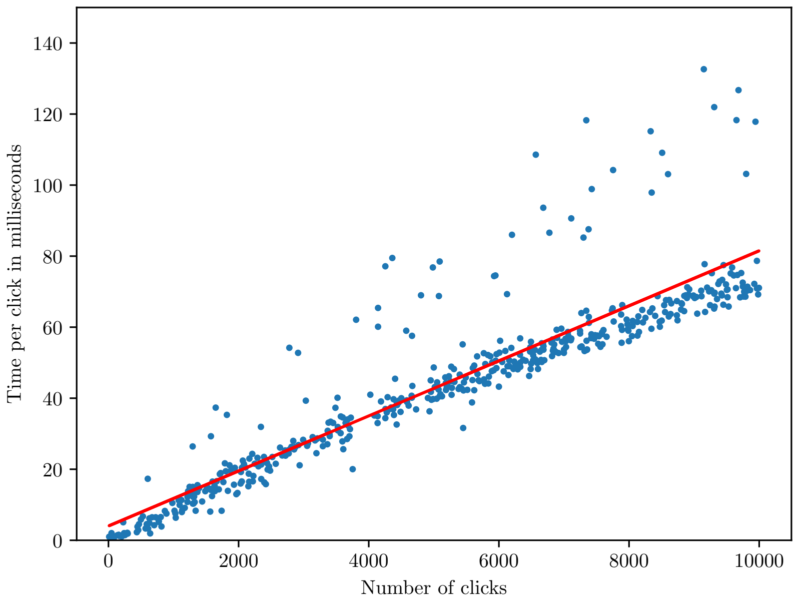
Motivation

We want the lasso selection tools to be efficient no matter how large the selection path is. We perform selection with random selection paths constructed with a variable number of points, between 3 and 10,000. A random starting point in the selectable area is generated, and each new point lies in a radius of 100 pixels from the previous point. Every selection task is performed on a selectable area with 2000 selectable elements.

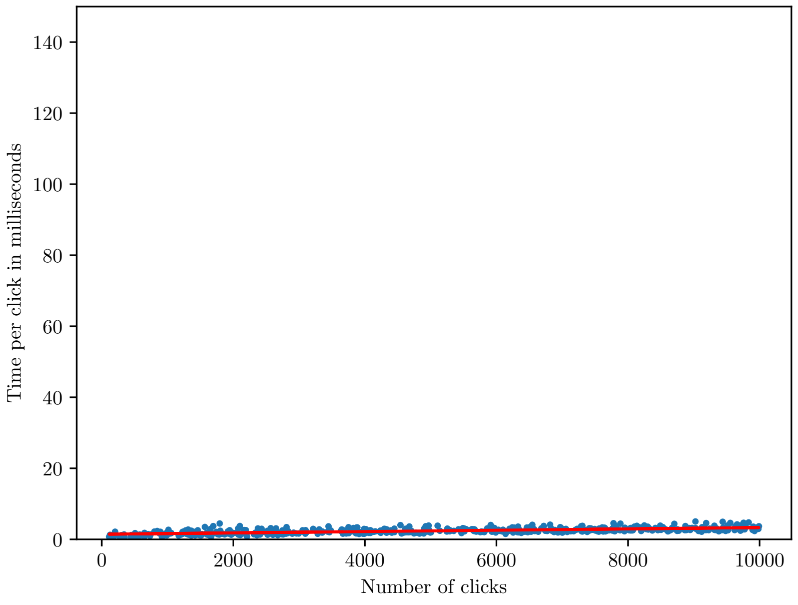
Discussion

Incremental lasso selection comes to its right with large selection paths, as shown in Figure 5.4. It has a slight decrease in performance when the path grows, but compared to the non-incremental version the performance degradation is miniscule.

The reason for the significant difference in performance between the two implementations is obvious: when the lasso covers a large area, the non-incremental version has to run the element-in-polygon algorithm on every element, while the incremental version only does so on elements in the affected area. This leads to fewer iterations over the selection path.



(a) Non-incremental lasso selection.



(b) Incremental lasso selection.

Figure 5.4: Performance of the lasso selection implementations while selecting a random polygon constructed by a random number of clicks.

5.2 Discussion and future work

This section discusses the results of the different performance tests, before defining aspects of the implementation that can be explored in the future.

5.2.1 Discussion

By constructing the test bed, we aimed to explore if the incremental implementation of lasso selection achieved the desired boost in efficiency compared to the non-incremental implementation.

It is apparent that the heuristics we have chosen have had the desired effect on performance — the incremental implementation is both more stable and efficient than the non-incremental implementation. It handles a large number of both selectable elements and points in the selection path.

It is important to note that these performance tests do not necessarily mirror the experienced efficiency when actually using the lasso selection tool, as all these performance tests have been performed programatically, ignoring a number of factors that may impact performance. For example, there is no guarantee that the browser can handle such a large number of elements.

The running times of the performance tests are also not completely comparable to actual usage of the lasso tool: they are performed with a sequential implementation of extending the path, while the one provided in *MultiselectJS* adds every call to `extendPath` to a queue. The queue is processed at predefined intervals. This lead to the performance test finishing immediately, so the functionality had to be removed to get precise timing results. This does not mean that the recorded performance tests are wrong, it means that recorded performance might be slower than the actual experienced efficiency, when one does not always wait for every call to `extendPath` to finish before computing the next one.

5.2.2 Future work

There are additional heuristics that we have not explored. This thesis explains how we split the selectable area into subareas to minimize the number of elements inspected. Future studies can include experiments based on this idea applied also to the selection path. This way irrelevant line segments could be excluded even before running the element-in-polygon algorithm.

The implementation presented above might perform poorly if the selectable elements are small and in very large numbers, such as is the case with pixels in a picture. If the selectable area is 500 pixels wide and the subareas are hundred pixels high, each subarea contains 50,000 selectable elements. Making the subareas vary in size based on the size of the elements might improve performance. Future studies could explore the option to give the programmer freedom to define how large each subarea should be. This way the number of elements in each subarea could be kept lower.

Another option to be explored is to only compute the selection domain when the selection path is finished, removing the dynamic aspect of the solution. This way the selection domain is computed only once, making it very efficient. This would lead to the same drawback as in other solutions to the lasso selection problem: no visual representation of how the selection path determines the selection domain. For large amounts of elements it might be a hit worth taking.

Performance is not the only concern, functionality is another. Future studies could extend the proposed lasso tool with similar functionality to the one found in Social Explorer [4, 5]: allowing the user to define if the lasso should select all elements that intersect with the area defined by it, all elements that are contained by it, or all elements whose center point is contained by it.

Chapter 6

Related work

There are not many works that study how to implement an effective lasso selection generically. In this thesis we focus mostly on efficiency and ease of implementation, though most existing works study the usability of different multiselection tools, lasso selection included. User studies are performed to compare these tools. The goal, however, is the same: effective ways to select many elements.

Some works look into other multiselection tools that can serve the same purpose as the lasso selection tool, but may be better suited for pen-based selection, such as selecting graphics on interactive whiteboards. Lasso selection is often compared to these variations of multiselection, and are found to be more time-consuming and fatigue-inducing.

6.1 Usability of lasso selection

S. Mizobuchi and M. Yasumura [22] performed a study comparing the effectiveness of circling selection[1] to tapping selection[2]. Tapping requires that each element to be selected is clicked. S. Mizobuchi and M. Yasumura hypothesized that circling was faster and more accurate than tapping overall, but that hypothesis was rejected. What they did find, however, was that the difference in selection time between the performed selection tasks varied less while circling, compared to tapping. Circling selection was not very efficient when performing selection tasks on elements with a low level of cohesiveness, i.e. elements were not adjacent to each other.

[1]Some papers classify lasso selection as *circling selection*.

[2]Tapping and clicking mean the same thing: interacting with an element by clicking it with a mouse or tapping on touch screens.

A similar study was performed by J. C. Jackson and R. J. Roske-Hofstrand [13]. Circling[1] was compared to clicking selection[2], but with simpler tests than S. Mizobuchi and M. Yasumura [22] performed. In the study of J. C. Jackson and R. J. Roske-Hofstrand, a participant had to select either a single element or pairs of elements, using either clicking or circling. The results showed that lasso selection were significantly slower at selecting a single element, but close to equal in selecting a pair of elements. J. C. Jackson and R. J. Roske-Hofstrand go on to conclude with the following statement:

> Circling seems to present a reasonable alternative to traditional clicking for selection of objects in mouse-based environments, and is particularly well-suited for selecting several objects in proximity to one another. It could certainly be considered as a supplemental selection technique in almost any interactive graphics display application.

This thesis can be considered a step towards making J. C. Jackson's and R. J. Roske-Hofstrand's suggestion a reality.

6.2 Alternatives to lasso selection

Harpoon selection tool, presented by J. Leitner and M. Haller [19], is a pen-based selection tool designed for interactive whiteboards. It is similar to snake selection in *MultiselectJS* [17], in that every element that touches the constructed selection path is selected. It differs from snake selection in that each point represents a selected area, differentiating coarse and fine-grained selection: the size of each area is defined dynamically based on the speed of selection. This is useful when selecting hand writing, for example grouping the dot and the line together in an "i". A user study was conducted where participants performed different selection tasks using multiple selection tools, *Harpoon* and lasso included. The study showed that the *Harpoon* selection tool significantly out-performed lasso selection in the speed of which the participants conducted the different selection tasks. However, when the participants were asked to name their overall favorite tool, the lasso selection tool came out on top.

M. Haller et al. [21] also present another tool to be used in conjunction with the *Harpoon* selection tool: *Suggero*. It is designed for interactive whiteboards. Based on the selection of one or more elements, using the *Harpoon* or another tool, it presents the user with a list of pre-computed perceptual groups containing, or being similar to, the selected content. This allows the user to quickly select similar elements or groups of elements. M. Haller et al. conducted a study which showed that *Suggero* decreased selection effort, interactions and stylus movement. All factors decrease fatigue when performing selections on interactive whiteboards.

H. Dehmeshki and W. Stuerzlinger [3] presents a multiselection approach designed with Gestalt grouping, the way human perception naturally groups objects together, in mind. The *Gestalt* multiselection approach groups elements together through a nearest neighbour graph, and detects groups based on proximity and good continuity. A user selects the desired elements by performing *clicks*, *double-clicks*, *shift-clicks* and *alt-clicks.* A conducted user study, with users performing different selection tasks with different selection tools, revealed that the *Gestalt* approach to multiselection surpassed both lasso selection and rectangular selection in selection time.

Chapter 7

Conclusions

Our motivation in this thesis were described through three simple bullet points:

- Lasso selection is often not available where it would be useful.
- There is unnecessary variation in different realizations of lasso selection.
- Implementing a correct and efficient lasso selection is difficult.

We have achieved what we set out to do: a generic efficient implementation of a lasso selection tool. It tolerates large numbers of both elements and points in the selection path. Two implementations were created, both very similar. Changing how the selection domain is computed resulted in a more stable, incremental lasso selection algorithm.

Any programmer can take the implemented lasso selection tool to use through the library *MultiselectJS*, allowing easy access to lasso selection in applications where it would be useful. Any programmer wishing to create a lasso selection tool can also take inspiration from this work — the heuristics presented should give a clear idea of how to mitigate the cost of computation.

While we have found multiple heuristics that mitigate the cost of lasso computations, there are others that have not been explored. Being able to limit the portion of the selection path that needs to be inspected seems particularly promising for increasing the efficiency of the algorithm.

Bibliography

[1] Adobe. Adobe Photoshop Lasso Tools, accessed: 5 April, 2020.
URL: `https://helpx.adobe.com/photoshop/using/selecting-lasso-tools.html`.

[2] Paul Bourke. Determining if a point lies on the interior of a polygon, 1987, accessed: 8 April, 2020.
URL: `http://paulbourke.net/geometry/polygonmesh/#insidepoly`.

[3] Hoda Dehmeshki and Wolfgang Stuerzlinger. Intelligent Mouse-Based Object Group Selection. In *Proceedings of the 9th International Symposium on Smart Graphics*, SG '08, page 33–44, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540854104. doi: 10.1007/978-3-540-85412-8_4.
URL: `https://doi.org/10.1007/978-3-540-85412-8_4`.

[4] Social Explorer. Explore Maps, accessed: 5 April, 2020.
URL: `https://www.socialexplorer.com/explore-maps`.

[5] Social Explorer. Polygon Selection Tool, accessed: 5 April, 2020.
URL: `https://www.socialexplorer.com/help/guides-videos/using-the-selection-toolbox`.

[6] Michael Galetzka and Patrick O. Glauner. A Simple and Correct Even-Odd Algorithm for the Point-in-Polygon Problem for Complex Polygons. In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, volume 1: GRAPP, pages 175–178, 2017.

[7] Eric Haines. *Point in Polygon Strategies*, page 24–46. Academic Press Professional, Inc., USA, 1994. ISBN 0123361559.

[8] Jian-Qiang Hao and Jianzhi Sun. Optimal Reliable Point-in-Polygon Test and Differential Coding Boolean Operations on Polygons. *Symmetry*, 10:1–16, 10 2018.

[9] Donald Hearn and M. Pauline Baker. *Computer Graphics, C Version*, pages 24–41. Prentice-Hall, Inc., USA, 1997. ISBN 0135309247.

[10] Kai Hormann and Alexander Agathos. The Point in Polygon Problem for Arbitrary Polygons. *Comput. Geom. Theory Appl.*, 20(3):131–144, November 2001. ISSN 0925-7721. doi: 10.1016/S0925-7721(01)00012-8.
URL: `https://doi.org/10.1016/S0925-7721(01)00012-8`.

[11] Chong-Wei Huang and Tian-Yuan Shih. On the Complexity of Point-in-Polygon Algorithms. *Comput. Geosci.*, 23(1):109–118, February 1997. ISSN 0098-3004. doi: 10.1016/S0098-3004(96)00071-4.
URL: `https://doi.org/10.1016/S0098-3004(96)00071-4`.

[12] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[13] J. C. Jackson and R. J. Roske-Hofstrand. Circling: A Method of Mouse-Based Selection without Button Presses. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, page 161–166, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913019. doi: 10.1145/67449.67483.
URL: `https://doi.org/10.1145/67449.67483`.

[14] Jaakko Järvi and Sean Parent. One Way to Select Many. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.14.
URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6108`.

[15] Juan José Jiménez-Delgado, Francisco R. Feito-Higueruela, and Rafael Jesús Segura. Robust and Optimized Algorithms for the Point-in-Polygon Inclusion Test without Pre-processing. *Comput. Graph. Forum*, 28:2264–2274, 2009.

[16] jQuery. Selectable Widget, accessed: 5 April, 2020.
URL: `https://api.jqueryui.com/selectable/`.

[17] Jaakko Järvi and Sean Parent. MultiselectJS, 2016, accessed: 5 April, 2020.
URL: `https://github.com/HotDrink/MultiselectJS`.

[18] Jaakko Järvi and Sean Parent. MultiselectJS Demo Application, 2016, accessed: 6 May, 2020.
URL: `http://hotdrink.github.io/multiselectjs/examples/demo/multiselect-demo.html`.

[19] Jakob Leitner and Michael Haller. Harpoon Selection: Efficient Selections for Ungrouped Content on Large Pen-Based Surfaces. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 593–602, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307161. doi: 10.1145/2047196.2047275.
URL: `https://doi.org/10.1145/2047196.2047275`.

[20] Ole Magnus Lie. omlie/lasso-testing: Lasso testing, results and script for plotting data, May 2020.
URL: `https://doi.org/10.5281/zenodo.3842702`.

[21] David Lindlbauer, Michael Haller, Mark Hancock, Stacey D. Scott, and Wolfgang Stuerzlinger. Perceptual Grouping: Selection Assistance for Digital Sketching. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '13, page 51–60, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322713. doi: 10.1145/2512349.2512801.
URL: `https://doi.org/10.1145/2512349.2512801`.

[22] Sachi Mizobuchi and Michiaki Yasumura. Tapping vs. Circling Selections on Pen-Based Devices: Evidence for Different Performance-Shaping Factors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 607–614, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137028. doi: 10.1145/985692.985769.
URL: `https://doi.org/10.1145/985692.985769`.

[23] GNU Image Manipulation Program. GIMP Free Selection (Lasso), accessed: 5 April, 2020.
URL: `https://docs.gimp.org/2.10/en/gimp-tool-free-select.html`.