

# Illustrating Data Structure and Algorithms using Penrose

## PROJECT REPORT

*CISC499 Winter 2024*

Queen's University

Supervisor: Burton Ma

Student: Marcus Chiang – 20184995

### Introduction

Hand-made diagrams that communicate abstract concepts in computing can be difficult and tedious to create. Algorithms often require multiple diagrams to communicate changes in data structures across multiple iterations. The goal of this project is to create a Python module that allows the user to easily generate a diagram of a given array.

Penrose is a design tool that turns a trio of plaintext into a diagram. Each diagram requires a domain, style, and substance file. Penrose creates the diagrams, but not as easily as we would like and not directly from a Python data structure. Python can easily create text files though, all it needs is a filename, and data to write to the new/existing file. So, Python will be used to analyze an array and print out a substance file that accurately describes that array. Style and domain files will be created to be compatible with the substance that Python prints.

### Materials and Methods

#### *Software Sources*

The list of required software that is necessary to produce a substance file is nice and short. You will need Python 3, I specifically used Python 3.9.10.

The roger command line interface was used to generate the .svg files. You can use “roger trio example.domain example.style example.substance > your\_image.svg” to produce an image. The usage is described in the README.md of the repository. There are many ways to produce an image; like the Penrose editor, or methods that will be explained later in the discussion section.

Git and Github were used for version control. The repository can be found here:  
[https://github.com/marcuschiang1/Penrose\\_Array](https://github.com/marcuschiang1/Penrose_Array).

### ***Domain***

The domain file defines the objects and their relationships for use in the style and substance files. It just contains the type declarations and predicate declarations. The object types are familiar and exactly what we expect from an array. It declares an “Element” type, “Array” type, and “Index” type. The predicate “In” declares that an Element  $e$  is contained in an Array  $a$ . The “IndexOf” predicate declares that the index of an Element  $e$  is Index  $i$ . The “Middle” predicate is used when an element or two elements are at the center of the array. The top predicate is used to highlight the top element of a stack.

There is only one domain file, since not every type and predicate have to be used, it only defines the possible types. Any further expansion to define algorithm-specific relationships or types can be done freely in the domain file.

Functions may also be declared in the domain file. Here is an example of a vector addition function from the Penrose website function `addV(Vector, Vector) -> Vector`<sup>4</sup>. The functions take any number of arguments and produce an object of a defined type. This was not found to be very useful for creating arrays (so far at least). Though, it could be useful for future

developments that require more work in Penrose than Python. For example, array appending or creating subarrays could be done with a function within Penrose.

### *Style*

The style file is where all the objects are given shape and locations. The different style files for the stack and binary search are mostly the same in structure. Loops may be added to modify certain array elements, or the orientation of the array can change. The initial “array.style” is a good template to modify for future development.

The setup starts by defining a canvas size, which can be changed at will; although, the offset of the array may need to be changed as well. Then there is a namespace that contains various colours to be used during the drawing process. Namespaces in general are helpful for variable storage, you can also store values by attaching them to objects initialized in the substance file. For example:

```
collect Element e into array
where In(e, a)
foreach Array a {
    widths = listof labelWidth from array
    max_width = maxList(widths)
    a.element_width = max_width
}
```

This isn’t used anymore since feedback said to have a fixed element box size; but it shows the use case of attaching variables to objects. This was a section of code that allowed storage of the largest possible text width in the array. The “a.element\_width” can also be used in later sections that concern the Array a. The collect feature is very useful, even if it ended up not being used this time. That concludes the setup, the next step is to loop and create shapes.

A for-each loop is used alongside a condition to iterate through each element of an array.

```
forall Element e; Index i; Array a  
where IndexOf(e, i) {
```

The loops look at each element and index, then it executes the code block for each element and index that fulfills the condition “IndexOf”. It may also look at each Array a if you wanted to re-include the max element width or print multiple arrays in one image. The next step is to declare the scalars that will be used in the current iteration.

```
scalar index_width = 25  
scalar e.x = match_id*index_width
```

The index\_width is set to 25, which fits a two-digit integer. This was the value that changed from being the width of the largest element. The scalar “e.x” assigns an x coordinate to each element. The scalar match\_id is a value ranging from 0 to n, which corresponds to the current loop iteration. This scalar is essential to aligning each elements text and box. This value is multiplied by the index width to give each element their own consistent x coordinate. Element 0 would be 0\*25, Element 1 would be 1\*25, Element 2 would be 2\*25, and so on.

Shapes are created in each iteration that correspond to the current index object and element object. The first shape is a Text object that displays the value of the current element. This is represented by the “string” property in the shape, it is set to the label created in the substance file.

```
shape e.elementLabel = Text {  
  string: e.label  
  center: (e.x, 0)  
  fontSize: "16px"  
  fillColor: colours.black  
}
```

Labels will be discussed in the substance section, but just know that each object can be given a corresponding “label” that is of type string. The center property uses the e.x scalar as the x value and a constant 0 as the y value. The fontSize can be changed easily by the user but is set to a 16-point black font by default.

The next shape is the array box that contains the element, it’s just a simple rectangle that conforms to the element text.

```
shape e.icon = Rectangle {  
  center: e.elementLabel.center  
  height: 20  
  fillColor: colours.white  
  strokeColor: rgba(0, 0, 0, 1)  
  strokeWidth: 1  
  width: index_width  
  cornerRadius: 2.0  
}
```

Properties of other shapes can be accessed within the program to match new shapes. The width and center are set to match those of the previous elementLabel, while the remaining properties are just visual stylings that can be changed without needing much fixing.

The final shape is straightforward after learning about the previous two. It’s just the indices of each element; it uses the corresponding elements x coordinate, then goes 20 units lower than the element on the y axis.

```
shape e.indexLabel = Text {  
  string: i.label  
  center: (e.x, -20)  
  fontSize: "16px"  
  fillColor: colours.black  
}
```

The style files are slightly altered for stack and binary search. Binary search requires highlighting on the middle element(s), so an extra loop is added to change the fill colour of the array element's box. Where the condition of "Middle" holds, each elements fill colour is changed to a transparent red. The highlighting overrides the original fill colour that was set by the previous loop, but this is fine. The transparent highlighting colour can be changed by the user, the red just didn't obscure the text as much as other colours did. The stack retains this same loop at the end to highlight the top element; but the orientation of the array is changed to make more sense. The x coordinate of each element is a constant 0, and the height is the match\_id multiplied by the fixed height of each box.

### *Substance*

The substance file declares the instances of objects and relationships that can be used in the illustration. This is the file type that is generated by Python, so it has a consistent structure across each type of illustration. It begins by listing the instances of each type, for loops are used when multiple instances of a certain type are needed.

```
Array a_1  
Element e_i for i in [0,4]
```

Predicates are generated next; you can index the arguments and use a for loop to generate many at once.

```
IndexOf(e_j, i_j) for j in [0,4]
```

Labels are generated after this, they are the strings that represent their respective instance in the illustration. For example, the first index is labelled by the string “0” and the first element in the array is labelled by “10”.

```
Label i_0 $0$  
Label e_0 $10$
```

Predicates after the labels are appended by the stack or binary search functions. If any further development is made, the beginning substance file structure would probably remain consistent.

```
Middle(e_2,i_2)  
Top(e_3, i_3)
```

### ***Python Implementation***

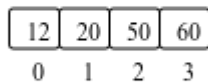
The Python implementation just gets info about an array, then turns that into a substance file that complies with the array style and domain. The PenroseArray object stores an array to be used by the functions within the class; the only parameter needed for those functions is a filename. The main function “generate\_substance”, creates a basic array illustration. It follows the same structure as the previously described substance files. It outputs the Array initialization, Elements, Index, IndexOf predicates, and the In predicates. There is a helper function named “create\_labels” that just generates the labels based on the length and content of the array. Both the types, predicates, and labels are then output to the given file; the substance file is ready to be used at this point.

Binary search starts by running the “generate\_substance” function on its own to ensure that a working substance file exists. It then appends predicates based on if the array is of even or odd length. If it’s odd, then just the middle element needs to be highlighted. If it’s even, then the middle two elements need to be highlighted. The substance file is usable once the predicates are appended.

The “penrose\_stack” function does mostly the same thing. It just appends an element that highlights the top of the stack.

Creating a substance file is not too expensive at  $O(n)$  time. The binary search function and stack function run at constant time, but you will always need to generate a basic substance file before they are run.

## Results



12	20	50	60
0	1	2	3

*Figure 1: Default array from array.style*



10	20	30	50	90
0	1	2	3	4

*Figure 2: First iteration of binary search, target is 50*

50	90
0	1

*Figure 3: Second iteration of binary search*

50
0

*Figure 4: Third and final iteration of binary search*

4	90
3	50
2	30
1	20
0	10

Figure 5: Stack

3	50
2	30
1	20
0	10

Figure 6: Stack after a pop operation.

## Discussion and Conclusion

### *Limitations*

The main limitation is that the images cannot illustrate multiple arrays at once. This would be helpful to illustrate each iteration of an array in the same diagram. To solve this, the user may have to save each iteration of the array in another array, then the PenroseArray object could hold a 2D array as well. This was working to some extent; multiple arrays could be present in the same diagram; but there were spacing issues that didn't end up getting solved in time.

The whole point is to make array diagram generation more convenient, but it still could be more convenient. Generating multiple arrays would help, but so would a better way to generate the images. Currently, you must use either the roger command line interface or the Penrose editor to generate the images from the Penrose trio. The next step would be to make the Python script generate the images themselves. There is a commented-out section that runs the roger command line from within the script; but it's not finished. It is also a weird solution to run a terminal command from within the script. A better solution will be in the future work sections.

Limited options are also a problem. The plan was to do as much of the work as possible in Penrose, then Python won't have to do much except generate substance. It looked like problems would arise if Python started to change the style files. Giving the user the ability to change the highlight colour would require that either Python changes the style file, or many style files be made for each colour. The same goes for other options like font colour, rectangle size, canvas size, etc. Both options seemed like it would have problems, so the user now needs to know a bit about Penrose to change those properties.

### ***Future Work***

The limitation section already listed a few points for future work. Generating multiple arrays would solve the problem of users having to individually generate images of each array iteration. More convenient ways to generate images from within the Python script would enhance usability. Finding a more elegant solution to handle all the options that the Penrose style file affords from within Python would allow the user to not even know about Penrose.

The Language API for Penrose allows for React and SolidJS integration<sup>5</sup>. A surprising amount of time was spent on trying to make this work; but just polishing the style files and main script was more a more meaningful use of time. The website provides JavaScript programs to use the Penrose JS module to attach the generated image to an HTML element. These could be used to create a website that is hosted on the user's computer and contains whatever image they generate. Using the Penrose API and finding a useful way to host the images on a website would be a meaningful expansion if CISC499 was a full-year course.

The original array style file could branch into more style files, like it does with binary search and stack. Array-based linked lists could be represented with each node as a vertical array that looks like a stack. A predicate "PointsTo(Element e1, Element e2)" could be used to draw an arrow from one array element to the next. This would require multiple arrays to be present in the drawing and an arrow to be illustrated wherever "PointsTo" holds. Sorting algorithms could be demonstrated as well with the current foundation. If multiple arrays are drawn, bubble sort could be easily drawn by highlighting the swapped elements of each iteration. Two-way arrows could be used to show a swap as well. There are many algorithms and data-structures that can expand upon this project since arrays are common.

## ***Summary***

The first step to replicate this project is to create a style, domain, and substance file for a basic array. Substance files should be created for edge cases like the empty array and large element sizes. A Python script should then be made to generate substance files for the simple array. It just needs to use a new/existing text file and output object initialization and labels. Then alternative style files can be made to demonstrate different algorithms and data structures. Corresponding substance files should be generated to demonstrate these style files. A demonstration Python script should finally be created to show the style files being used by their respective algorithms or data structures. The body of work overall contains Python scripts, domain files, style files, substance files, and generated images from the roger command line interface.

## **Works Cited:**

1. Ye, K., & Chen, W. (2020). Penrose: From Mathematical Notation to Beautiful Diagrams. Carnegie Mellon University.  
[https://penrose.cs.cmu.edu/media/Penrose\\_SIGGRAPH2020a.pdf](https://penrose.cs.cmu.edu/media/Penrose_SIGGRAPH2020a.pdf)
2. “Overview .” *Penrose*, penrose.cs.cmu.edu/docs/ref. Accessed 1 Apr. 2024.
3. “Using Penrose .” *Penrose*, penrose.cs.cmu.edu/docs/ref/using. Accessed 1 Apr. 2024.
4. *Functions* . Penrose. (n.d.-b). <https://penrose.cs.cmu.edu/docs/tutorial/functions>
5. *The language api* . Penrose. (n.d.-g). <https://penrose.cs.cmu.edu/docs/ref/api>