(16) Image / Graphics Shapes to (x, y) converter

Preface

Task title:

```
Program pentru comanda
unui utilaj cu comanda
numerica
```

Task description:

```
- se va scrie un program pentru o masina
de taiat cu flama
- programul citeste un fisier care contine
traiectoria de taiere (secventa de segmente
si arce de cerc) si genereaza comezni
pentru deplasarea pe doua directii (x si y)
a capului de taiere; deplasarea capului de
taiere se va simula pe ecranul
calculatorului
(Java, C, C++, C# etc.)
```


Editor details:	
Author	Virghileanu Teodor
University	UTCN CTI EN
Year of study	3

- Git
- Repo
- QR image URL

> I suggest reading the PDF version of this README here https://github.com/GaussianWonder/scs-project/blob/main/README.pdf

Contents

Introduction

Context

The goal of this project is to **design and implement an algorithm** that **converts** *an image* or *a set of graphical shapes* **into a set of commands** that can be used by machines which work on 2D planar workspaces such as:

- Milling and Engraving machines
    - PCB prototype makers
- Plasma cutting machines
- 3D printers
- Drawing machines 😃

This algorithm should be written and packed such that **it runs on relevant platforms** without the need of language specific *adaptations*

The **output** of the algorithm should be **encoded simple** enough such that it can be *repurposed / transpiled / decoded* easily in order to match the language the user's machine is using. (**ie:** via regex transformations or content interpretation)

Specification

The algorithm will be **simulated** in a *configurable* **simulation** that accepts similar (or identical) commands to the output of the algorithm.

Objectives

The objective of this project is to **design and implement an algorithm** that **converts** *high level graphical content* into *simple commands* that draw outlines of the given graphical abstractions.

Because of the given specification, an implementation of a configurable simulator is required.

Bonus objective

Given enough time, implement **a configurable slicing algorithm** for *3D objects* and sequentially pipe them into the main algorithm described above to further demonstrate its usability.

Bibliographic Study

Since the topic of this project is `converting images or 2d primitives` into a set of 4 directional move commands, the first thing I research is `image processing` and `2d primitives` such that project requirements can be established.

Image Processing

From previous experience i know that OpenCV is a helpful framework in this regard.

Documentation can be found here and the github page here

OpenCV relevant API:

- Detecting outlines via cv::findContours
- Detecting shapes via cv::convexHull

For this purpose alone, OpenCV is overkill, so any other solutions that solve these problems in particular are well fitted, however implementation time must also be considered.

Edge detection

The Canny Edge Detector is an algorithm that solves this particular problem.

High Level steps:

- Grayscale Conversion
- Noise reduction / Blurring
  - Gaussian Blur
  - OpenCV example
- Determining Intensity Gradients
  - Detect edge intensity and direction by using edge detection operators
  - Sobel operator
- Non-Maximum Suppresion
  - This can be thought out as the thinning of the currently calculated outlines
- Double Thresholding
  - Filter out pixels by intensity
    - Strong
    - Weak
    - Non relevant 😃
- Edge Tracking by Hysteris
  - Transform weak pixels into strong ones
- Cleanup
  - Iterate throguh remaining weak edges and remove them

Articles that target this approach can be found here and here

There are other algorithms that can acomplish this as well, such as the `Scharr filter` and `Sobel filter`

Convex hulling

Although not necessary, it can be of use when debugging or for future features.

> This is currently marked as not a requirement

2D Primitives

The list of 2D Graphics Primitives is pretty narrow:

- Point
- Line
    - > 1st degree curve
- Polygon
- Ellipse
- Curve
    - Handy list of algebraic curves can be found here
    - Bézier curve
        - Examples of this can be found on p5.js source guided by it's documentation
        - Calculating the bounding box
    - Centripetal Catmull–Rom spline

> I did not include particular shapes of other listed items (such as triangles, rectangles,... which derive from polygons, circles which can be generated from ellipses,...)

Programming language

The second most important thing is the `programming language` that powers the whole system.

The algorithm that performs the conversion must be a standalone that works with commandline arguments in order to fit some criterias specified during the **Introduction**. Because of this, it does not matter what language I use for this part of the system, as long as algorithmic requirements established during the Bibliographic Study are also met.

Eventhough the simulation is not tied in any way to the algorithm mentioned above, I want both of them to be constructed using the same language and resources.

As far as the `Simulator` is concerned, I want it to live in a powerful, fast environment capable of creative graphics via any means available (ie: OpenGL)

`Memory safety` **and** `Thread safety` are important for this specific tasks. Given the generic nature this has to be implemented in, it follows that it should support batch processing in order to be included in other systems.

RUST

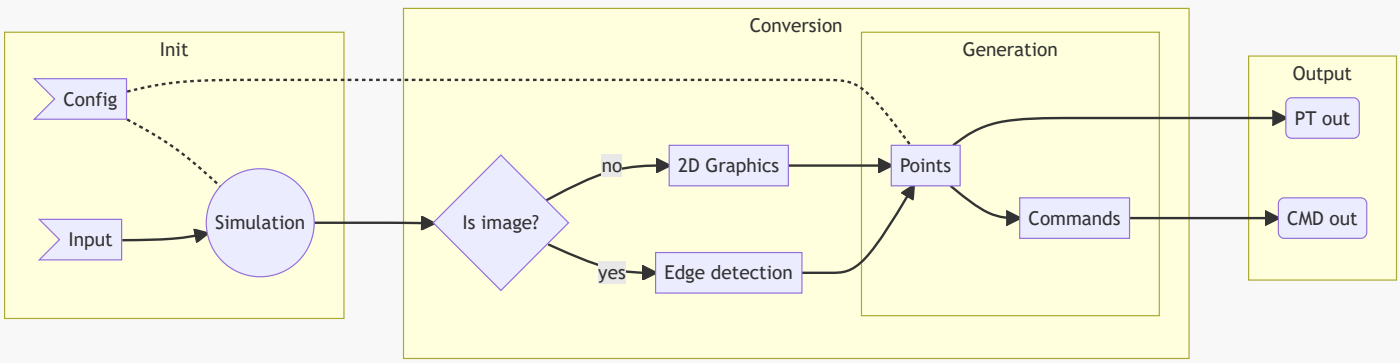
> A language empowering everyone to build reliable and efficient software.

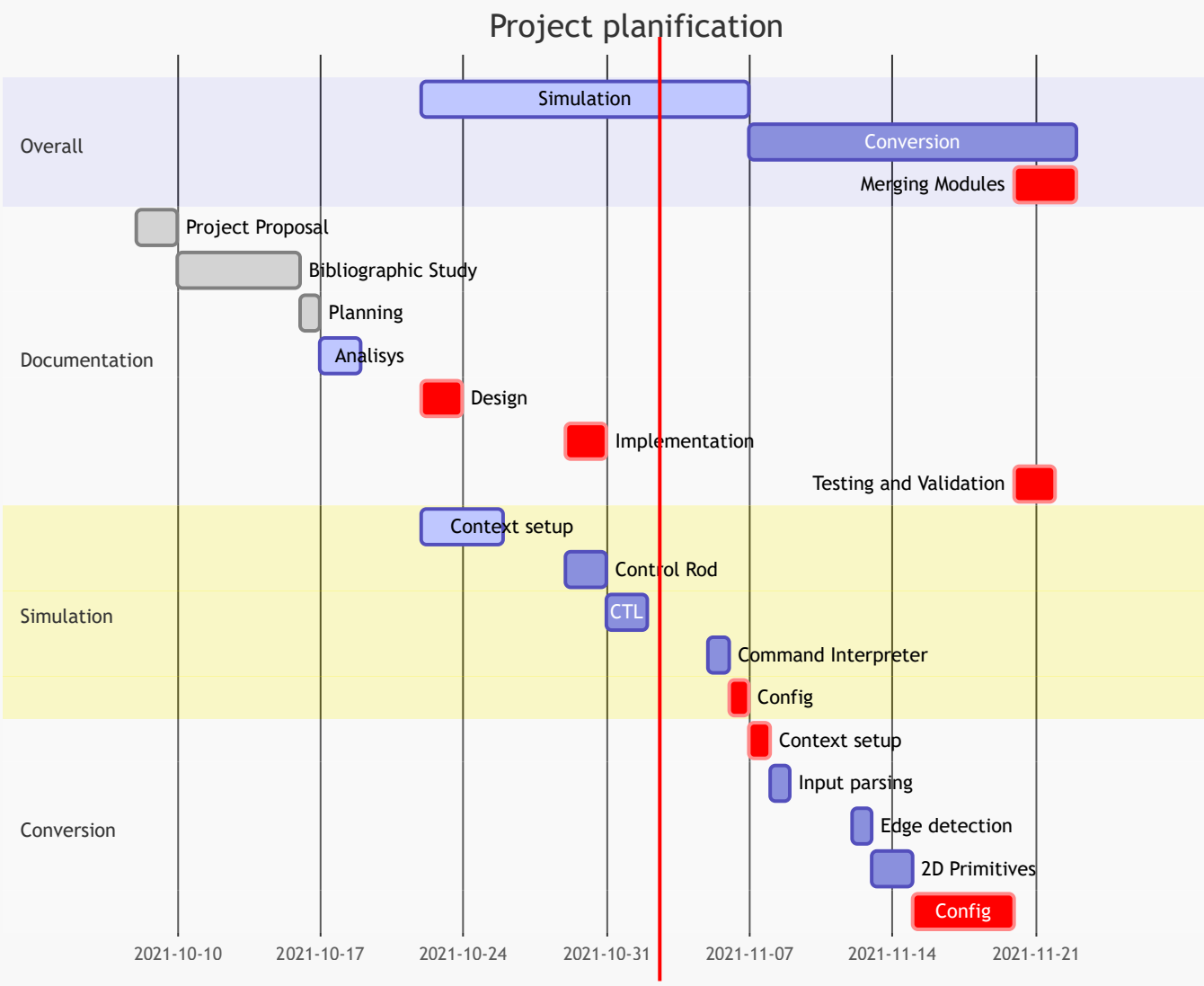
Rust is matching the given criterias so far. Docs are here

Requirement	Resource	Info	Targeted feature
Graphics	nannou	Safe, Reliable, Sufficient	2D Primitives
Image Processing	opencv-rust	Unstable, Untested	**OpenCV relevant API:**
Image Processing	rust-cv	New, Has Edge detection	**Edge detection**
Statically typed	types	✓	Programming language
Memory Safe	ownership	✓	Programming language
~~Garbage Collector~~	bamboozle	✓, Not a runtime GC	Programming language

Analisys

High level overview and planification:





Command output

What hasn't been discussed yet is the command output language.

The normal approach to this problem would be to compile the point collection into GCode since it is the most widely used CNC programming language. However i decided to go with **a custom approach** since the simulator that accepts these commands will be simple, and the commands themselves are simple enough too.

The program must accomodate a drawing mechanism. Such mechanisms need to be able to move on a plane (drawing or not) and optionally stay idle

Thus the command list compilation is

- **Pen** `down`
  - Puts the pen down
  - Any move statements after this will be *drawing* on the canvas
- **Pen** `up`
  - Raises the pen up
  - Any move statements after this will just *transport* the tip to another location
- **Move** `{X|Y} {DIST}`
  - Move the tip DIST on X or Y axis
- **GO** `{X} {Y}`
  - Move the tip to coordinate {X} {Y}

Config options

Both the simulation and the conversion algorithm benefit from the use of the same config options.

The simulation needs to be configured in order to precisely display the effects of the algorithm.

The conversion algorithm needs the configuration to match the capabilities of the machine that's going to be used.

Judging by the commands above, **machine precision** is a must configuration option. This will be a number that describes how far apart or *how close together* the generated points can be, before converting them into commands.

A flag that tells the algorithm whether to use **relative** or **absolute** tip movement is also required.

One last requirement is the possibility to map the input to some other **resolution**

The configuration will be described in a JSON file, or via arguments:

- **--precision** `2`
- **--absolute**
- **--map** `100 100`

```
{
  "precision": 2,
  "absolute": true,
  "map": {
    "x": 100,
    "y": 100
  }
}
```

Design

The simulation will display a simple control to choose an input file and a configuration file.

Once both are selected, **2/3** of the screen will be used to display the preview with several controls available to the user:

- **Toggle input**
  - Displays the input image below the preview
  - If no image was provided, it will draw the 2D Graphical shapes under the preview instead
- **Stop**
  - Pauses the simulation
- **Next**
  - If stopped, this will execute the next command
  - While normal execution, this button will not be available
- **Resume**
  - Resumes the normal execution of the preview
- **Reset**
  - Delete everything and restart the process

Changing the input will result in a **Reset**.

Implementation

Testing and Validation

Conclusions

```
[...]
```

Bibliography

- OpenCV
  - https://docs.opencv.org/
    - https://docs.opencv.org/4.5.3/df/d0d/tutorial_find_contours.html
    - https://docs.opencv.org/4.5.3/d7/d1d/tutorial_hull.html
    - https://docs.opencv.org/4.5.3/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html
  - https://github.com/opencv/opencv
- Canny Edge Detector
  - https://datacarpentry.org/image-processing/06-blurring/
  - https://en.wikipedia.org/wiki/Sobel_operator
- Canny Edge Detection Articles
  - https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123
  - https://justin-liang.com/tutorials/canny/
- Curves
  - https://www.2dcurves.com/
  - https://en.wikipedia.org/wiki/Bézier_curve
  - https://en.wikipedia.org/wiki/Centripetal_Catmull–Rom_spline
- Code examples
  - https://github.com/processing/p5.js/
  - http://nishiohirokazu.blogspot.com/2009/06/how-to-calculate-bezier-curves-bounding.html
  - https://p5js.org/reference/
- Rust
  - https://github.com/nannou-org/nannou
  - https://github.com/twistedfall/opencv-rust)
  - https://github.com/rust-cv/cv
- GCode
  - https://en.wikipedia.org/wiki/G-code