Computer Graphics Design Journal

Yuxiang “Tom” Lin

Session 1, Fall 2023

Contents

Chapter 1

Programming Contest Club Poster

1.1 Concept

1.1.1 Brief

This is a poster which my club (DKU Programming Contest Club) could use during a Club Expo. Our club's mission is to take part in programming contests. Therefore, the poster should convey a geeky impression, but also not without a sense of humor. The poster may be both stuck to the walls and handed out, so the aspect ratio for an A series paper in portrait orientation is chosen.

1.1.2 Text Element

The text element uses a "coded text" design, which is basically an if statement in C-style code saying that "if you are an elite hacker, you should join the Programming Contest Club." It also references the "leetspeak" trope with "1337 h@ck3r" standing for "elite hacker." The code highlighting and the extra large "Programming Contest Club" requires manual adjustments to its color and placement. The font I chose is Brass Mono, which is a "retro monospaced font inspired by 70's electrical and mechanical design," and a "solid choice for writing code." Therefore, the choice of this font could give a "programming impression."

1.1.3 Graphics Element

The most essential feature is that all the graphics are drawn on a hexagonal grid in an abstract fashion similar to a pixel art. The overall layout is inspired by the "two soyjaks pointing" meme, with two persons in the bottom corners of the screen having surprised faces and the person to the right pointing to the center of the poster. This directs the focus to the center of the poster, which is a person sitting in front of the computer screen. (Also note that the "Programming Contest Club" text is directly above it.) The three people featured in this poster represents a typical team in the ICPC programming contest, where three team members have to share one computer. The vibrant Tomorrow color palette (which is also a color palette for code text editors) is used in this poster. The careful placement of different colors is able to give a sense of balance. Gradients and noises are added to provide the finer details.

1.2 Iterations

1.2.1 Iteration 1

I implemented utilities for working with hexagonal coordinates and drawing basic shapes in a hexagonal grid. A random hexagonal grid and some shapes are drawn for testing. I also used a ”pencil” tool (which also shows the current hexagonal coordinate) to draw some sketches for text and graphics placements.

1.2.2 Iteration 2

I implemented a utility for drawing texts with different colors before using it to draw highlighted code texts.

1.3 Final result

The coder sitting in front of the computer and the two soyjaks pointing at them is added. I used callback functions passed to the shape functions to produce the gradient and noise details.

1.4 References

- Hexagonal grid: https://www.redblobgames.com/grids/hexagons
- Hexagonal coordinate tricks: https://observablehq.com/@jrus/hexround

- Triangle drawing algorithm: https://github.com/ssloy/tinyrenderer/wiki/Lesson-2:-Triangle-rasterization-and-back-face-culling#old-school-method-line-sweeping
- Brass Mono font: https://github.com/fonsecapeter/brass_mono
- Tomorrow theme: https://github.com/chriskempson/tomorrow-theme

Chapter 2

Wavy Illusion Field

2.1 Concept statement

This project aims at recreating and experimenting with an optical motion illusion work named "Primrose's field," which consists of a checkerboard with cross-like shapes of alternating colors placed at the corners of the cells. When looking at Primrose's field, the straight and perpendicular lines appear wavy, and that the illusory motion of the waves could be observed when looking at the different regions of the image.

I will first create a grid of large squares and place the small squares at the corners of the large squares. Then I will manipulate the size, color pattern, and the shape of the small squares to see how will they contribute to the optical illusion and whether I can do better than the original pattern.

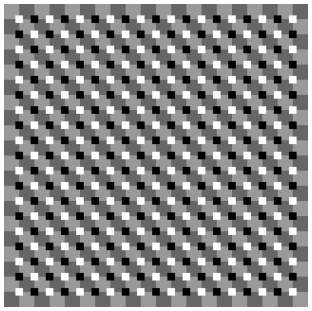
2.2 Generative rule

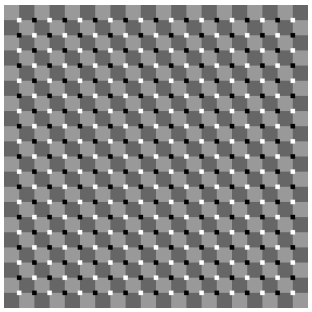
1. A simple checkerboard pattern is drawn in the background with alternating light gray and dark gray patterns.
2. At the corners of the cells in the checkerboard pattern, either a small white square or or a black square could be drawn according to some rules. The "squares" could also be stylized into shapes such as crosses.
3. The pattern could be colorized if needed.

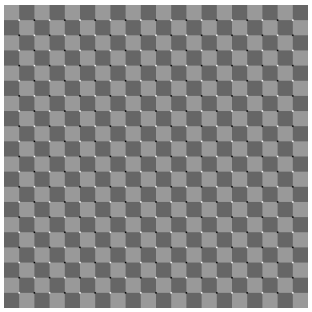
2.3 Design iterations

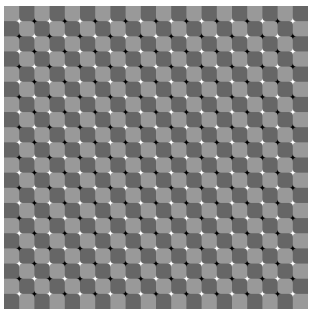
2.3.1 Iteration 1

I filled the large squares with light gray and dark gray checkerboard patterns. The small squares are of the same pattern but filled with white and black instead. Then I produced four images with different sizes/shapes for the small squares: large squares, medium squares, small squares, and crosses formed by quadratic curves. From my perspective, no optical illusion is observed with the large square patterns. For the medium and small squares, there is a sight illusion that the lines are slanted instead of being perfectly perpendicular to each other. However, this illusion is most strongly observed with the cross-shaped corners, which hint at the fact that the shape of the cross in the original design might be beneficial in inducing optical illusions.







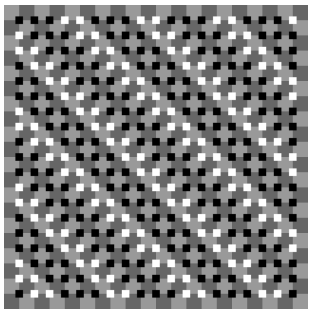


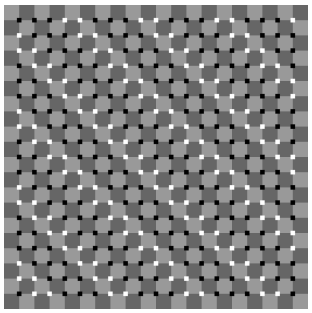
2.3.2 Iteration 2

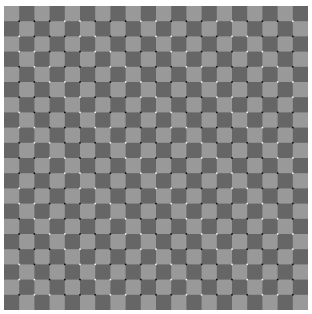
I decided to keep the large checkerboard pattern unchanged, but change the small squares to the following pattern shifted to the right every time you go one row down the board:

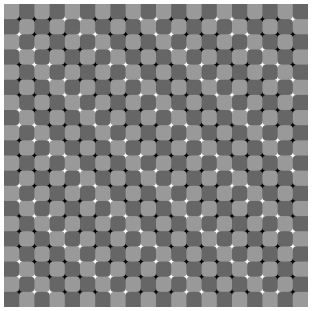
```
BBBWW BBBWW ...
 BBBWW BBBWW ...
  BBBWW BBBWW ...
...
```

This pattern aims at recreating the alternating diagonal lines in the original Primrose's field and analyze its contribution to the overall illusory effect. There appears to be a wavy pattern in the medium and the small squares version, and this wavy illusion is further amplified in the pattern with cross-shaped corners. Nevertheless, the illusory effect is still weaker than the original Primrose's field and little to no motion could be observed when focusing the eyes at different regions of the image.







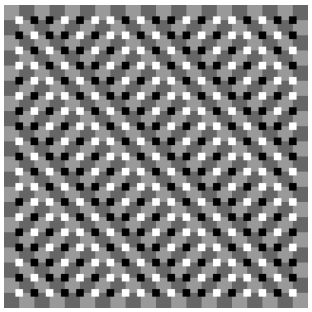


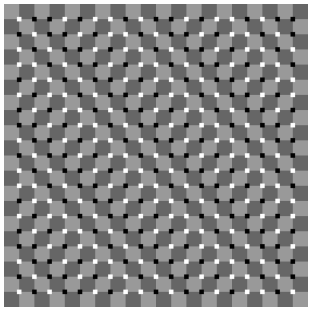
2.3.3 Iteration 3

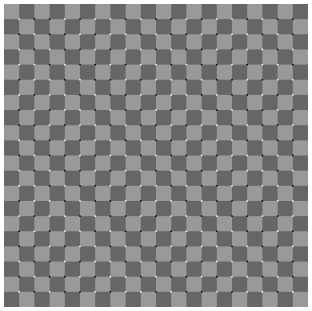
I will now use the original Primrose's field patterns:

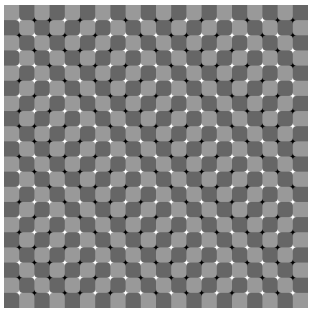
```
WBBW BWWB WBBW BWWB ...
 WBBW BWWB WBBW BWWB ...
  WBBW BWWB WBBW BWWB ...
...
```

The wavy lines illusion could be also observed in the medium and the small squares version, with the small squares version having the stronger illusion. The motion illusion can now be observed slightly in the small squares version. As is before, the crosses version have the strongest overall illusory effect, pretty much replicating the effect observed in the original Primrose's field.







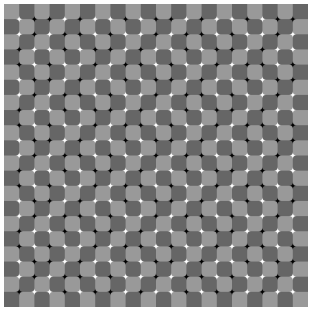


2.3.4 Iteration 4

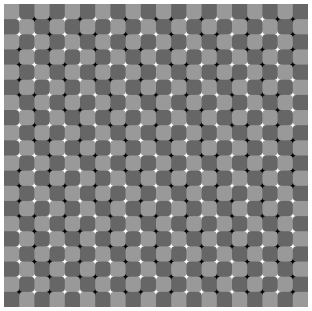
I now shift each row down to the right by two positions:

```
WBBW BWWB WBBW BWWB ...
12WBBW BWWB WBBW BWWB ...
```

This sort of reflects the diagonals vertically, but also looks a bit differently from an actual reflection.

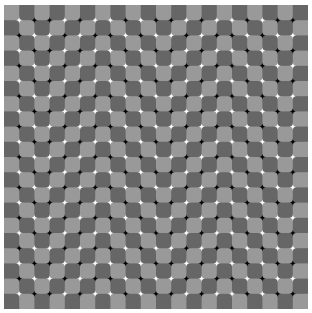


Shifting to the right by three positions doesn't look very interesting:

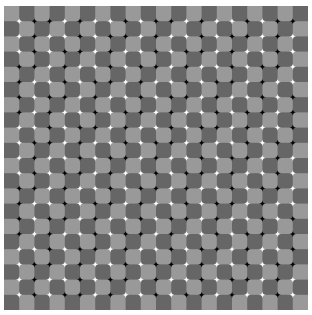


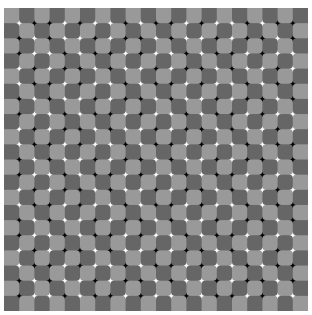
Shifting to the right by four positions creates an interesting "funnel" pattern that sort of looks like this:

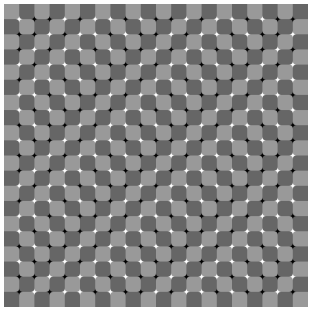
\/\/\/



Shifting to the right by five, six, and seven positions are just shifting to the left by three, two, and one positions.

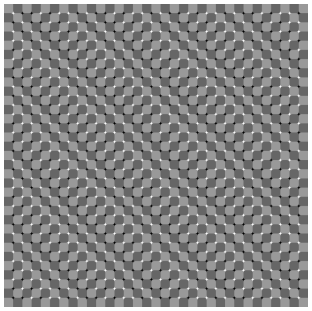






2.3.5 Iteration 5

This is the Primrose's field on a 33x33 grid:

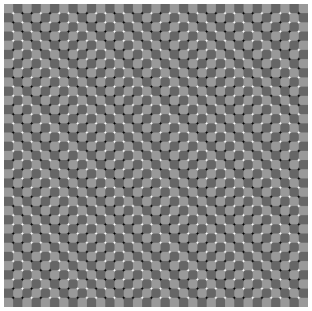


Now, instead of the original Primrose's field pattern, I thought that the Thue–Morse sequence is quite similar to it, which looks like this:

```
W
W B
WB BW
WBBW BWWB
WBBWBWWB BWWBWBBW
```

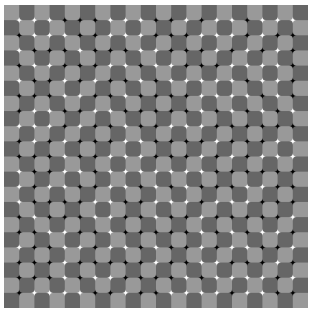
This sequence could be constructed recursively by appending the inverse of itself to itself repeatedly, or by calculating the parity of the binary representation of the indices. It could be observed that the Thue-Morse sequence of length 8 is the same as the original pattern.

I chose to fill the first row with the Thue-Morse sequence and repeatedly shift it to the right by one when going down the rows. This creates the pattern below, which doesn't look very different from the original design at a glance, and the strength of the illusory effect is approximately the same:



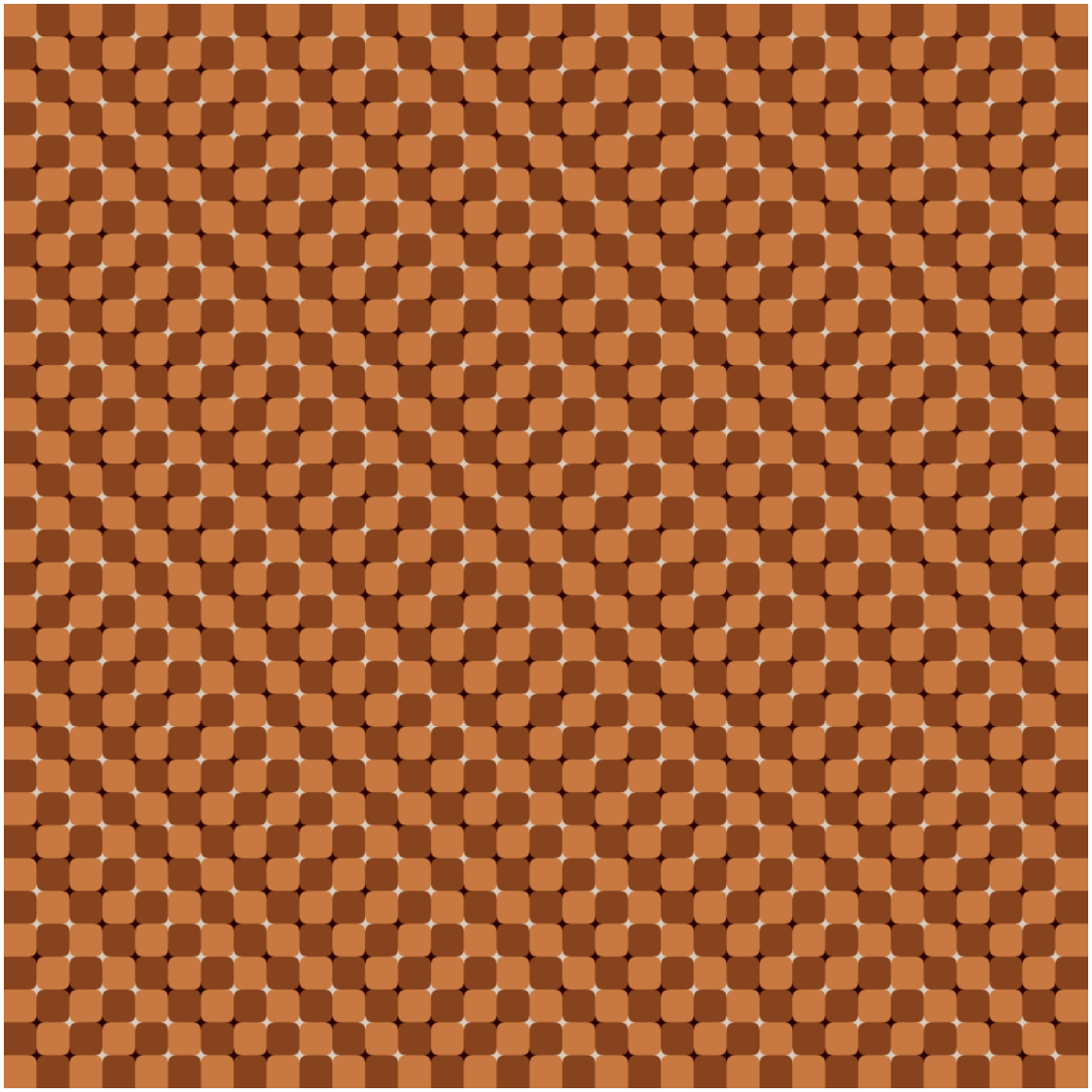
2.3.6 Iteration 6

I also tried to fill the small squares with white and black randomly, which creates a rather random distortion.



2.4 Final results

I took the Thue-Morse sequence pattern from iteration 5 and colorized it. I also animated this pattern by shifting it to right repeatedly every few frames, but this doesn't appear to have a significant impact on the optical illusions effect.



2.5 References

Akiyoshi's illusion pages: https://www.ritsumei.ac.jp/~akitaoka/index-e.htmlThue–Morse sequence: https://en.wikipedia.org/wiki/Thue%E2%80%93Morse_sequenceColor palette: https://colorhunt.co/palette/29000187431dc87941dbcbbd

Chapter 3

Recursive Reconstruction of Images

3.1 Concept

This project is based on the wave function collapse algorithm, which places tiles onto a grid according to certain rules such as whether tile A can be placed to the left of tile B. It optimizes this process by placing a tile on the position with the least entropy each time. The entropy of a position is decided by how many different types of tile could potentially be placed at that position (as decided by its neighboring tiles that are already placed). For example, if only one type of tile could be placed at (x_0, y_0), but there could be two types of tile at (x_1, y_1), then (x_0, y_0) has a lower entropy. This, for example, can be used to procedurally generate a image when given a small image. The small image in this case provides both the tiles (chopped-up pieces of the image) and describes the generative rule (what pieces can be placed together in the small image can be placed together in the generated image).

My idea is to tweak the wave function algorithm so that a) the placement of the tiles are not strict, but the way they neighbor each other is similar to the original image and b) I will sample and "reconstruct" the original image iteratively in increasing larger sizes that are powers of 2 (e.g. 1x1, 2x2, 4x4, ...). This is also inspired by the way how procedural textures are created, where layers of noises of various frequencies are added together so that the lower frequency noises form the overall shape, while the higher frequency noises form the details. Put it another way, I would like to create a image that is "similar" to the original in a sense that both the larger chunks and the small pieces in the output image would relate to their neighbors in a way similar to the original image.

Although not intentionally thought of, the way I tweaked the original algorithm could be comparable to glitch art, where technologies are used in ways that deviate from their original designs to introduce glitches. In a similar fashion, I modified the wave function collapse algorithm according an abstract concept of reconstructed a image with progressively larger tiles. The end result also bears similarity to glitch art aesthetically despite the fact that I actually prefer a more smooth looking. The "blocky" output could be explained by that I only consider the average color of a tile instead of how it would connect to its neighbors.

3.2 Generative rule

My project also implements the wave function collapse algorithm which generates a new image from the rules described by the original image, but it deviates from the usual implementation in four aspects:

1. The placement of the tiles do not adhere to strict rules, but the probability of one tile having color A in the generated image is decided only by the probability of a) the overall probability of a tile having color A in the original image and b) the probability of a tile having color A neighboring another tile of color B in the original image when a tile of color B is already placed in the generated image neighboring the given tile. Formally, let the probability to have tile of color A at (x, y) be p. Let the probability to randomly select a tile of color A in the original image be $P(A)$. Let the probability for a tile to have color B to the left of a tile of color A be $P(B|A)$, and the same goes for the tile to the right, up, and down of a tile of color A, which are denoted by $P(C|A), P(D|A), P(E|A)$ respectively. Assuming the independence of the joint probability of all the neighboring tiles, p is calculated as follows according to Bayes' theorem:

$$p = P(A)\frac{P(B|A)P(C|A)P(D|A)P(E|A)}{P(B)P(C)P(D)P(E)}$$

2. The process of wave function collapse does not always select the position with the least entropy, but the position is selected each time through a weighted random sampling. The weights are the maximum entropy minus the entropy at the current position. This change favors collapsing the positions with a smaller entropy, but avoids the problem of the output being too determined by the initial decision. This problem can be illustrated by an example: Suppose that we always select the position with the least entropy, and that a blue tile is almost always next to a blue tile and a red tile is almost always next to a red tile in the original image. When the initial tile is set to be blue, its neighbors will have the lowest entropies and will be almost always set to blue. This will continue until the image is filled almost entirely by blue.

3. The generated image is of the same dimensions as the original, but the tiles are of progressively larger sizes that are powers of 2 (e.g. 1x1, 2x2, 4x4, ...). For tiles that are larger than 1x1, its color means the average color of its pixels. When placing the tiles, I would first only consider their average colors before finally assign the actual tile randomly from a pool of tiles sharing the same color.

4. The generation process is iterative:

    1. The original image is sampled by 1x1 tiles to calculate the probabilities.
    2. A new image is generated by placing 1x1 tiles of different colors according to the probabilities.
    3. The new image is sliced into 4x4 tiles for the next iteration.
    4. The original image is sampled again by 4x4 tiles, where the average color of each tile will be considered.
    5. A new image is generated by placing the 4x4 tiles generated from the previous iteration.
    6. The process repeats until the desired tile size is reached.

3.3 Design iterations

3.3.1 Original image

3.3.2 1. RGB12

The original image is converted from 24-bit RGB to 12-bit RGB by using only the highest 4 bits for each channel.

3.3.3 2. Quantization

The most frequently used 16 colors (excluding those that are too similar to each other) are chosen to form a palette, and the pixels from the original image is replaced with the closet colors from the palette.

3.3.4 3. Tilelify

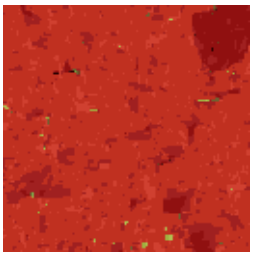
The utility function for grouping pixels into tiles is implemented. Below is a example of grouping pixels into 4x4 tiles like this:

1122
1122
3344
3344

The upper-leftmost pixel from each tile is extracted to form the image below:

3.3.5 4. Wave function collapse by picking the lowest entropy position

In this iteration, I implemented the wave function collapse algorithm by placing a tile on the position with the lowest entropy in each step (see the `peppers_min` folder for more details). This, however, has the problem of the result being too determined by the initial decision as mentioned above.



3.4 Final results

A weighted random approach is used to generate the final result to address the problem encountered in iteration 4. In the final implementation, results from the previous iteration is fed into the current iteration to iteratively "refine" the result. (Nevertheless, the outputs of later iterations tend to become more blocky.) I also added a log to record the order that the positions are collapsed, which is later played as a video.

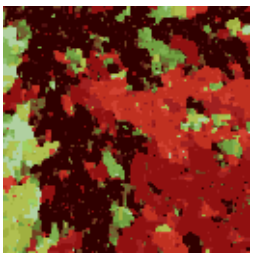
- Images of the outputs from different iterations: original*image*name/iter_#.png
- Video of the wave function collapse algorithm running: original*image*name/collapse.mkv

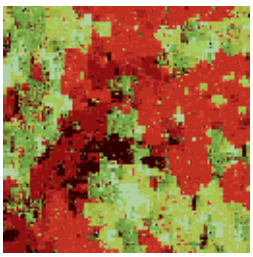
Note: If you zoom into the outputs for a closer look, they might look blurry. This is because interpolation is often used to upscale an image.

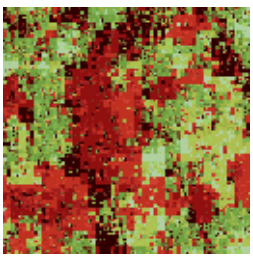
3.4.1 Peppers

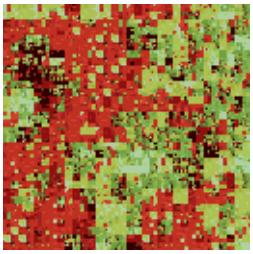
Original

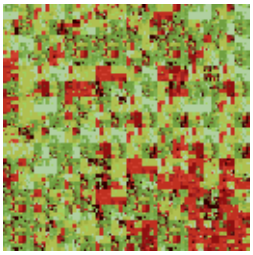
Outputs of iterations 0 to 5

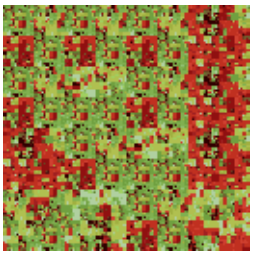








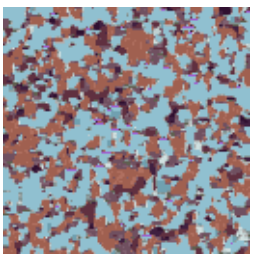


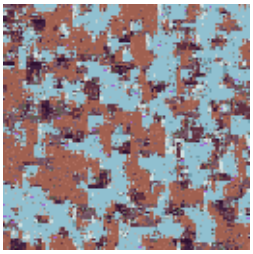


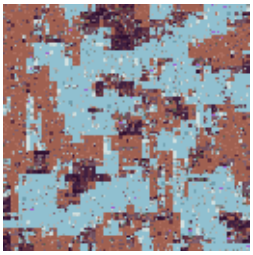
3.4.2 House

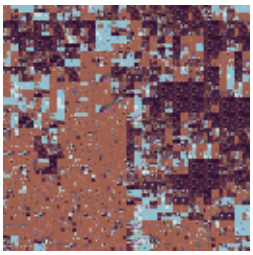
Original

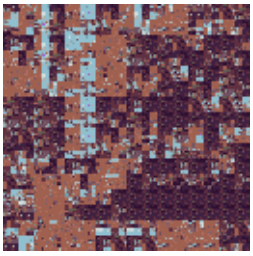
Outputs of iterations 0 to 5

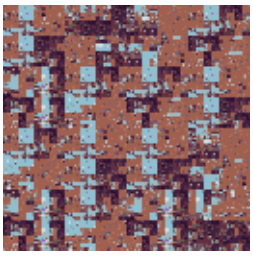








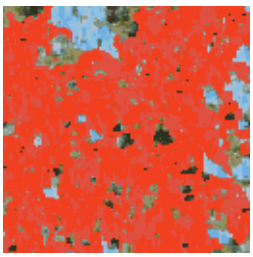


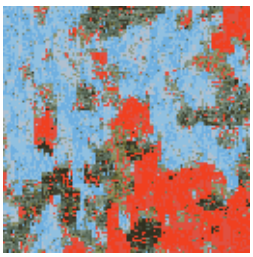


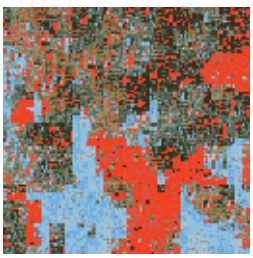
3.4.3 Mandrill

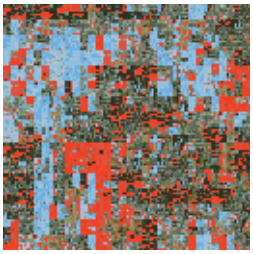
Original

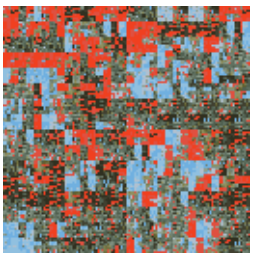
Outputs of iterations 0 to 5

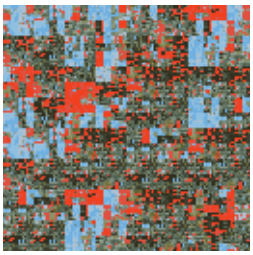












3.5 Dependencies

- This specific fork of Video Export Library: https://github.com/hamoid/video-*export*processing/tree/kotlinGradle
- ffmpeg

3.6 References

- Wave function collapse algorithm: https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/
- Color quantization: https://stackoverflow.com/questions/29244307/effective-gif-image-color-quantization
- SIPI Image Database - Misc: https://sipi.usc.edu/database/database.php?volume=misc

Chapter 4

Illegible Black Metal Band Logo Generator

4.1 Concept

This project aims at recreating the font for stereotypical unintelligible black metal band logos, which could be characterized by the two following features:

1. Tree-like branches emerging from the base text font.
2. Blood-dripping effect along these branches.

These two features can be observed in the logo of the black metal band Darkthrone given below:

It should be noted that not all black metal band logos exhibit this style as there are also many bands with clean-looking logos (for example, Emperor). Darkthrone is also not the band with the most unintelligible logo since there are much worse offenders. However, this project simply aims at parodying this "unreadable logo" trope in general.

While these two features could be implemented separately (recursive branching and drawing vertical lines attached to the branches), I took the approach of combining these two features by having the branches bending down gradually like a willow tree.

There are other features that are common in black metal logos. The most prominent is the black and white color, which will be used in this project. Another is the frequent use of Gothic font, which will not be used in this project as I would prefer to draw the base text font on my own using simple strokes and that Gothic fonts are usually used in logos that are clean and engraved-like as opposed to the sprawling tree-like logos that I would generate.

4.2 Generative rule

4.2.1 Visual object

The visual object is drawn with Agent, which are objects that draw a point (of given stroke) at their current locations and advance their positions every frame. The two types of Agent are:

1. Trunk: Draws the base text font, which contains all the uppercase Latin characters.
2. Branch: Emerges from the base text font to create the branching and blood-dripping effects.

4.2.2 Parameters

```
// Scale of the noise xy-coordinates
final float noiseScale = 0.01;

public class Trunk implements Agent
{
    // Maximum time (in frames)
    private static final int   tMax = 300;
    // Stroke weight at beginning and middle
    private static final float swBeg = 15, swMid = 25;
    // Stroke weight interpolation exponent
    private static final float swExp = 1.5;
    // Bezier curve amount at corners
    private static final float smoothAmt = 0.2;
    // Amount of displacement along noise gradient
    private static final float dispScale = 5;
    // Probability to create a new branch
    private static final float branchProb = 0.15;
    // Scale of the branch initial stroke weight
    private static final float branchScale = 1;
    // Minimum and maximum branch length
    private static final float branchLenMin = 150, branchLenMax = 300;
}

public class Branch implements Agent
{
    // Maximum time (in frames)
    private static final int   tMax = 200;
    // Stroke weight interpolation exponent
    private static final float swExp = 0.75;
    // Amount of random gradient added to branch movement
    private static final float randAmt = 0.55;
    // Maximum downward shift
    private static final float yShiftMax = 5;
    // Downward shift interpolation exponent
```

```
    private static final float yShiftExp = 3;
    // Maximum branch depth
    private static final int   depthMax = 2;
    // Probability of creating a new branch
    private static final float branchProb = 0.05;
    // Scale of the branch initial stroke weight
    private static final float branchScale = 0.8;
    // Minimum and maximum branch length
    private static final float branchLenMin = 50, branchLenMax = 100;
}
```

4.2.3 Algorithm

Trunk

Each character is defined by a series of strokes, which in turn are line segments connecting a series of points on a 5x5 integer coordinate grid. Below are the two strokes composing an "A" character numbered with 1 and 2:

```
..1..
.1.1.
.222.
1...1
1...1
```

The Trunk Agent linearly interpolates along the sum of all stroke lengths in each frame to render a point from a certain stroke. However, if the current point lies on the corners of a stroke, a Bézier curve would be used to connect the two neighboring line segments smoothly. The stroke weight is also interpolated so that the middle of the stroke has the highest weight while the beginning and the end of the stroke has the lowest. Both the curved corner and the changing stroke weight are for simulating drawing a stroke with a pen.

Displacement is then applied to the points so that they are shifted slightly along the noise gradient, which can be comparable to the jittering of the pen.

Finally there is a random chance every frame that a branch will be created from the current position with the initial stroke weight proportional to the current stroke weight of the Trunk.

Branch

The Branch Agent draws a point at its current position and moves forward by a gradient every frame. This gradient is a sum of three vectors:

1. Noise gradient: This can be thought of as extruding the screen along the z-axis to provide a terrain for the branches to "flow" like blood from high to low ground.
2. Random gradient: This allows the branches to grow randomly, providing a tree-like texture.
3. Y-shift: This bends the branches downward gradually. The more the branches extend, the more they will move downward along the Y-axis.

Combining these three vectors gives a mix between tree the tree-branching and blood-dripping effects.

There is also a chance for a branch to recursively create a subbranch every frame, limited by the maximum branch depth.

4.3 Design iterations

4.3.1 Iteration 1

Lines are drawn along the strokes to compose the characters.

4.3.2 Iteration 2

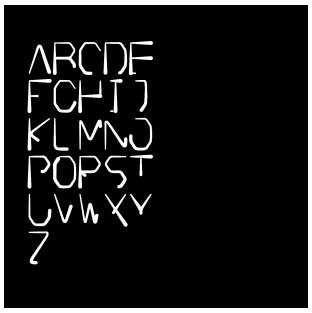
Stroke weight is linearly interpolated along each stroke.

4.3.3 Iteration 3

Bézier curve is used to smooth out the corners connecting the line segments.

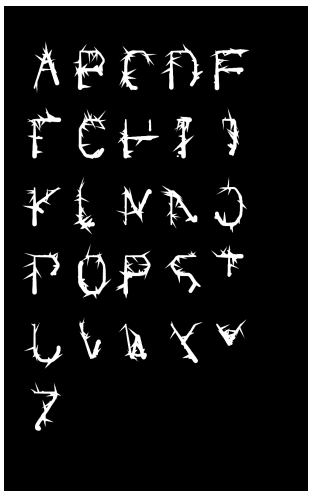
4.3.4 Iteration 4

Trunk object is added. Base text font drawing code is rewritten so that the process of drawing is animated.



4.3.5 Iteration 5

Branch object is added. Branches are created randomly from the base text font toward random directions.



4.3.6 Iteration 6

A 2D noise height map is added and that the gradient at each point is calculated with finite difference. The gradient field is then drawn on the background similar to a normal map with color red for the X component and green for Y component. The branches now move along the gradient field and that a ”flowing” texture can be observed.

Stroke weight interpolation is changed to have the thickest part in the middle.

4.3.7 Iteration 7

Random gradient and downward Y-shift are added to Branch so that the branches can have more of the ”willow tree” appearance.

Noise gradient displacement is added to the base text font so the text also appears jittered.

Noise scale is adjusted and that the noise is made symmetrical so that it may produce a more symmetrical band logo theoretically (which unfortunately does not appear to work for the alphabet).

4.3.8 Iteration 8

Recursive branching is added.

4.4 Final results

4.4.1 ”Default” parameters

This is drawn using the parameters that look the most ”correct” to me. (See ”Generative rule: Parameters” for the exact parameters.)


ABCDE
FGHIJ
KLMNO
PQRST
UVWXY
Z

4.4.2 Thin beginning and end, fat middle stroke

The following parameters are changed:

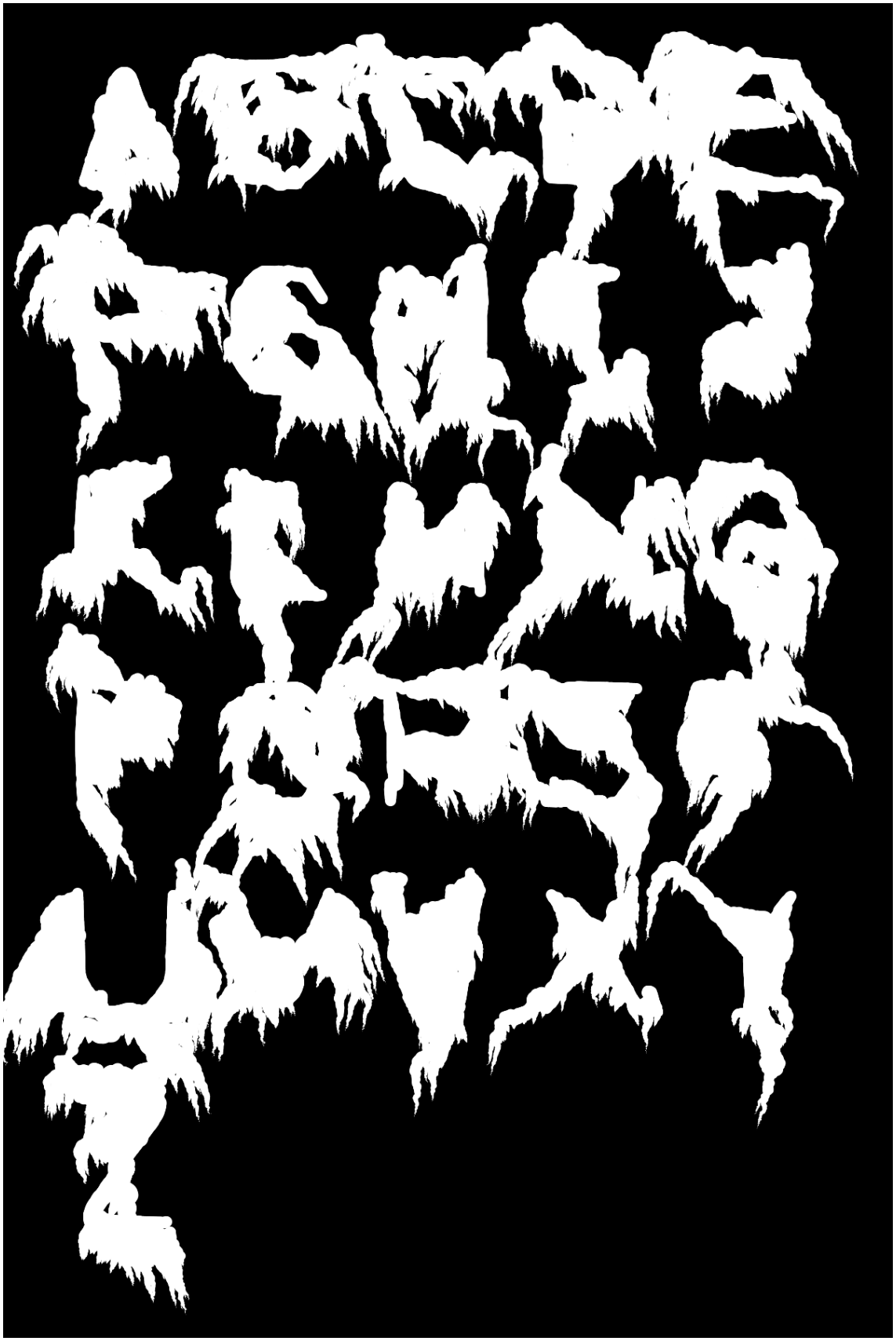
```
// Stroke weight at beginning and middle
private static final float swBeg = 5, swMid = 50;
```


ABCDE
FGHIJ
KLMNO
PQRST
UVWXY
Z

4.4.3 ”Convex” branch stroke weight function

The branch stroke weight shrinks increasingly faster:

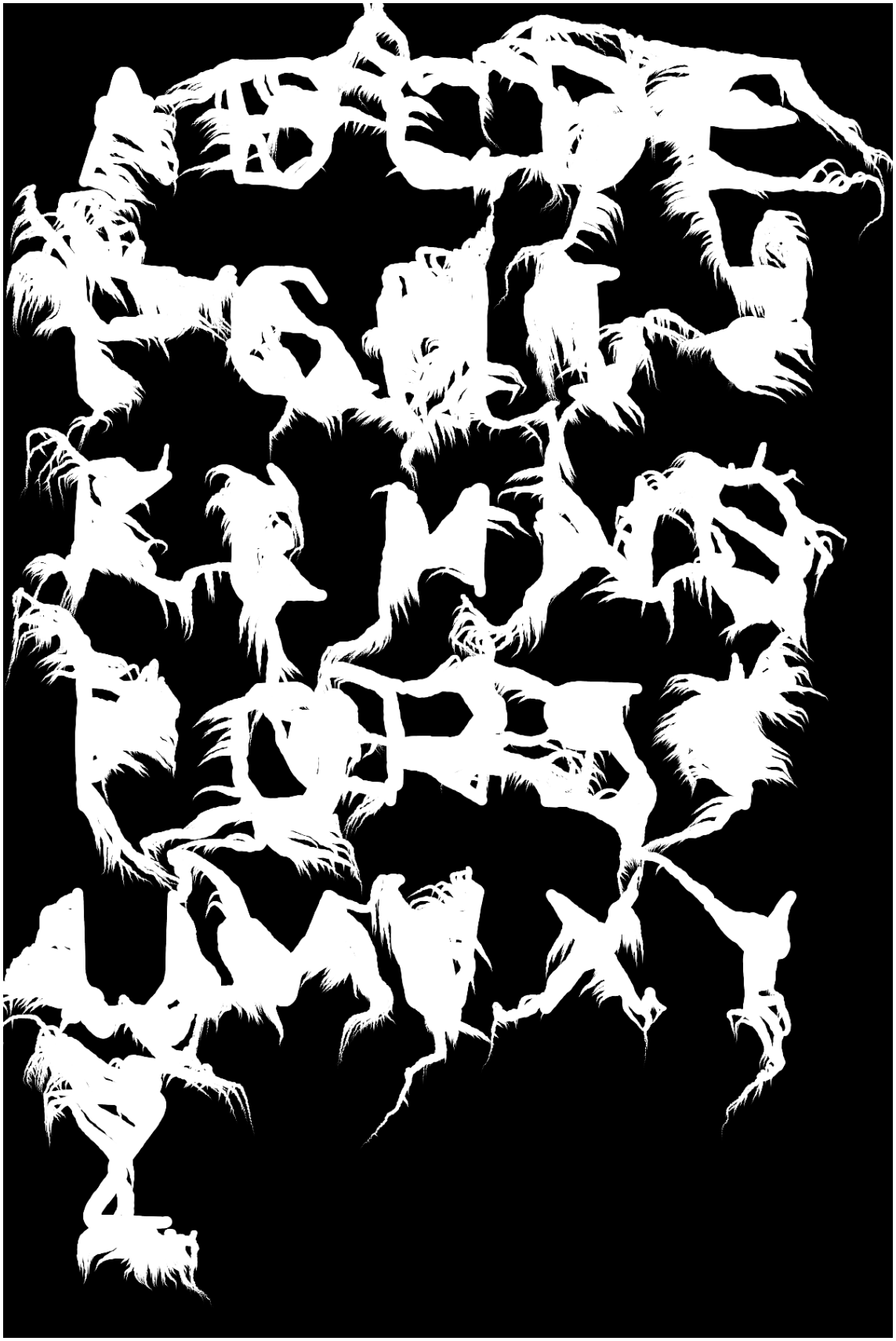
```
// Stroke weight interpolation exponent
private static final float swExp = 3;
```



4.4.4 More fluid-like branches

The amount of random gradient is reduced:

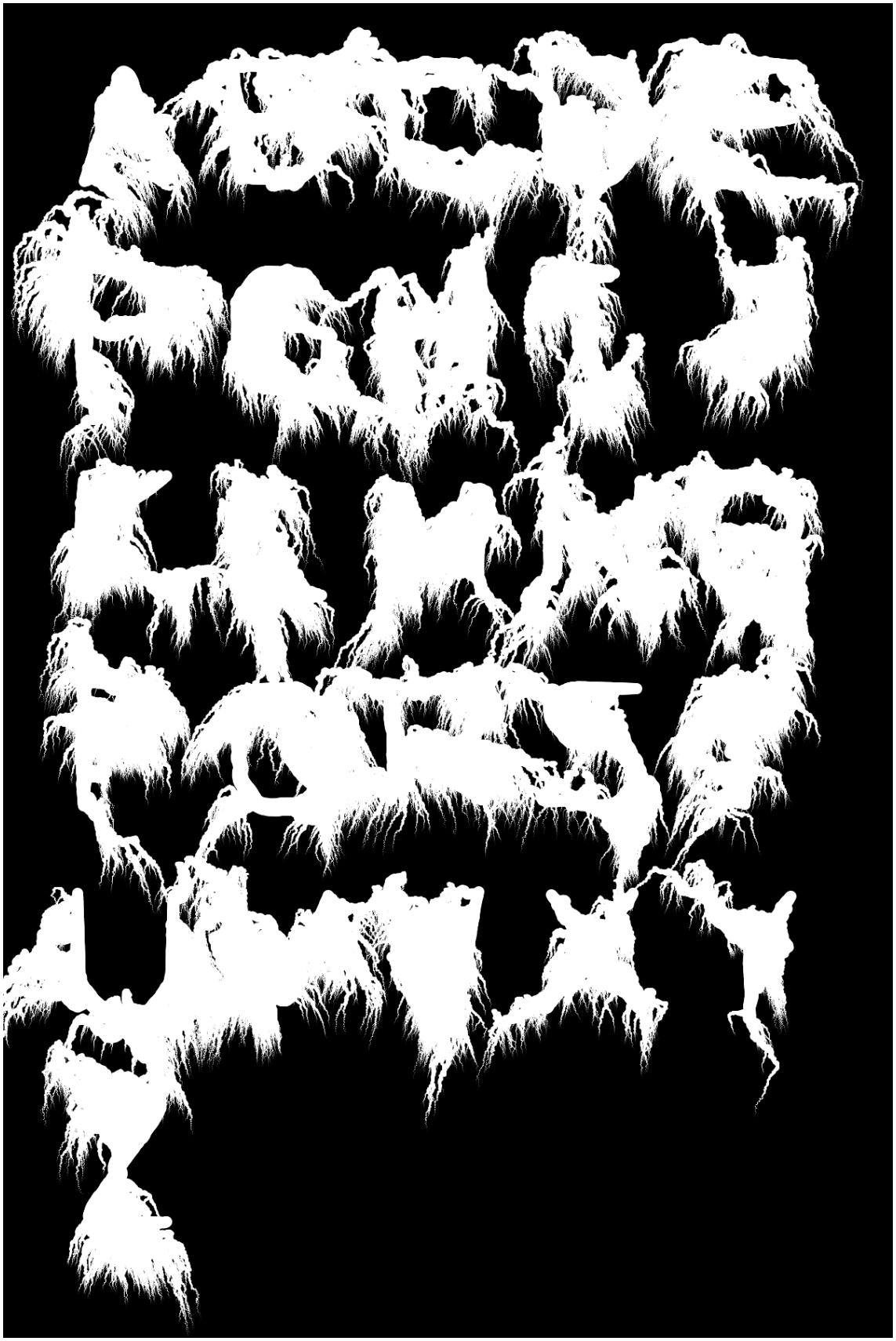
```
// Amount of random gradient added to branch movement
private static final float randAmt = 0.3;
```



4.4.5 "Furry" branches

Recursive branching depth is increased:

```
// Minimum and maximum branch length
private static final float branchLenMin = 200, branchLenMax = 400;
// Maximum branch depth
private static final int   depthMax = 3;
// Probability of creating a new branch
private static final float branchProb = 0.02;
// Minimum and maximum branch length
private static final float branchLenMin = 100, branchLenMax = 200;
```



4.5 References

- Darkthrone logo: https://sk.pinterest.com/pin/426434658442686417
- Extruding a font along a field: https://openprocessing.org/sketch/1989053
- Bézier curve: https://en.wikipedia.org/wiki/B%C3%A9zier_curve

Chapter 5

Chinese income visualization

5.1 Concept

I want to explore the relationship among income, industry, and working hours. As a multitude of relationships would be considered, and that there are many categories of industries (about twenty), I would produce three graphs about this topic:

1. The distribution of income in different industry categories with a histogram for each industry category.
2. The number of workers in different industries at different income levels and how some industry might have more workers than the others at different income levels. As there are numerous industry categories, the four broad divisions of industrial sectors would be used instead. The graph would be a hybrid between stacked bar chart and bump chart.
3. A scatter plot showing how working hours correlates with income level. This could demonstrate the current unfair situation of "more work, less pay."

The purpose of this project is to investigate how the income inequality of Chinese waged workers is related to the industries that they work in. On the left is a histogram representing the wage distribution within each industry. On the top is a stacked bar chart/bump chart hybrid representing the number of surveyed workers in each industrial sector at each income range. It can be observed that very few people work in the primary sector and that they usually have very low wages. People working in finance, technical service, and IT tend to have the highest wages. However, tertiary sector constitutes the majority of low income workers, while secondary sector makes up most of the medium to high income workers.

5.2 Generative rule

5.2.1 Dataset

Source: CFPS (China Family Panel Studies) 2018 survey

As the original data is very wide - surveying many aspects concerning the individuals and families in China, I picked out only the most relevant columns concerning the industry, occupation, weekly working hours, and yearly income of waged laborers.

5.2.2 Income distribution by industry histogram

Visualization strategy

- Density vs. **lightness**: To make the graph easier to read, I opt for a higher data-ink ratio and use spacing instead of explicit grid lines to line up the bars with data values.
- **Function** vs. decoration: I want this graph to be a straightforward visualization. Therefore, the elements shown are just the data and labels without decorations.
- **Abstraction** vs. figuration: The data is displayed as bars as opposed to fancier visual elements.
- **Familiarity** vs. originality: The simple histogram is a very traditional choice of data visualization.
- Redundancy vs. **novelty**: Each data point is only shown once. The income label is drawn twice though, as the chart is very tall.
- **Multi-dimensional** vs. uni-dimensional: Income is not only categorized by industries, but the distribution of the workers a certain income is also shown within each industry.

Mapping method

The data points would be grouped by the twenty industry categories. Then for each category, a histogram showing the distribution of workers of different income levels would be drawn. Note that these histograms would be proportional to the workers within that industry instead of all the waged workers surveyed, and there is a great disparity of worker numbers among different industries.

Objective

I would like to demonstrate how the income level of workers are distributed differently among different industries and the general tendency of income levels concentrating around a certain (and rather low) level. Still, the overall presentation would be informative and neutral.

5.2.3 Number of workers by income and sector bump chart

Visualization strategy

- Density vs. **lightness**: The graph has no grid and extra annotations as the focus is on the relative values of the data points instead of the exact values.
- **Function** vs. decoration: The graph displays data directly without much decoration.
- **Abstraction** vs. figuration: Data points are shown as abstract bars.
- Familiarity vs. **originality**: A combination of stacked bar chart and bump chart would be employed, which is a very original design.
- **Redundancy** vs. novelty: Parts of the graph is duplicated in a larger scale as some bars are too short to be shown clearly.
- **Multi-dimensional** vs. uni-dimensional: The sectors, income levels, the number of the workers, and their ranks are all shown in the same graph.

Mapping method

The data would be binned according to the income levels first. Within each bin, the number of workers in the four industrial sectors are counted respectively and that the four sectors are then ranked. The bins are then represented with bars that are further divided into four bars representing the four sectors. The sub-bars representing the same sector in the adjacent bars are then connected to form a hybrid between bar chart and bump chart.

Objective

The main objective is to show the distribution of workers and the inequality of wages among different industries. The overall presentation style would be informative and neutral.

5.2.4 Working hours vs. income

Visualization strategy

- Density vs. **lightness**: The graph focuses on the general trend between working hours and income instead of the specific values.
- **Function** vs. decoration: The graph displays data directly without much decoration.
- **Abstraction** vs. figuration: Data points are just points in the scatter plot as there are many data points.
- **Familiarity** vs. originality: The standard design of a scatter plot with a best fitting line is used.
- Redundancy vs. **novelty**: The information is present without duplicates.
- Multi-dimensional vs. **uni-dimensional**: Only the relationship between income and working hours is considered.

Mapping method

A scatter plot would be used to plot each worker as a circle according to their income and working hours. The circles plotted would be transparent to show their density. A best fitting line calculated through linear regression would be then drawn over it.

Objective

This graph aims to highlight the issue of people with higher income tending to working less and that a significant portion of people with low income working extremely long hours. The color scheme of black, white, and red is chosen to provide a sharp contrast and for their historical connection with labor rights and workers' revolution.

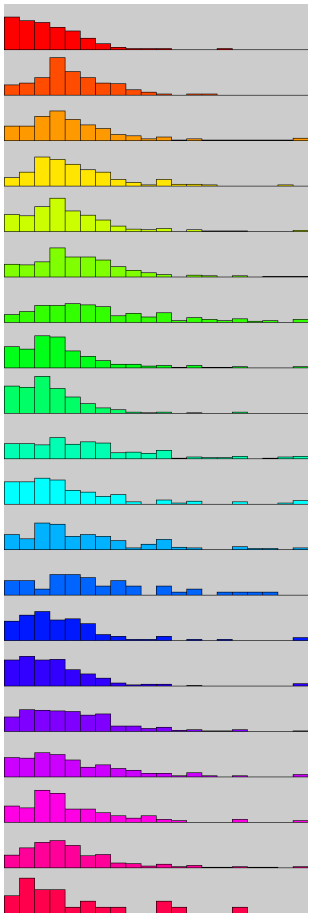
5.3 Design iterations

The most relevant columns are extracted from the original dataset using Python Pandas. Some basic analysis and visualization is than performed using Pandas and Matplotlib to determine the topics of interest.

5.3.1 Income distribution by industry histogram

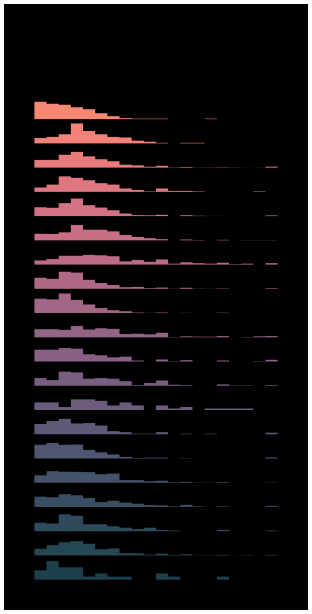
Iteration 1

Simple histograms for income distribution are plotted for each industry.



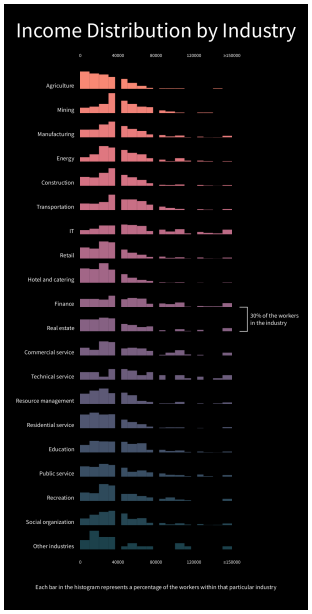
Iteration 2

A color palette is chosen for the histograms. I specified the spacing and layout for the graph to leave room for title and labels.



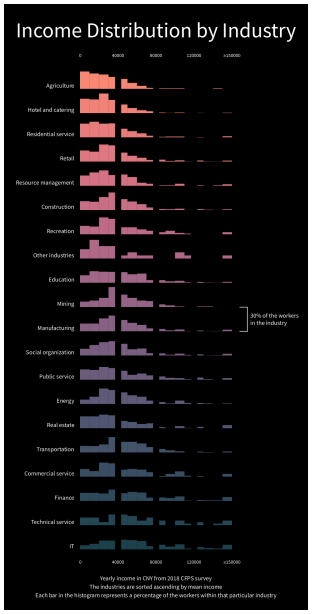
Iteration 3

Title and XY-axis labels are added. Vertical ticks that have the same color as the background are drawn to indicate income number. The annotation on the right shows that proportion of the workers represented by a full-length bar.



Iteration 4

The industries are now sorted ascending by mean income. More comments are added.



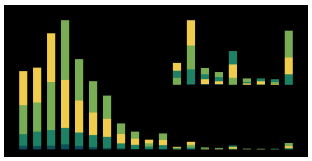
5.3.2 Number of workers by income and sector bump chart

Iteration 1

Spacing and layout for the graph is outlined.

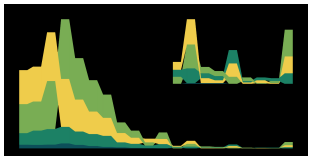
Iteration 2

A stacked bar graph is drawn. The sub-bars in every bar is sorted by its length. The right half of the graph is magnified.



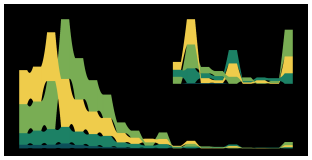
Iteration 3

The bars from the same sector is connected in a fashion similar to a bump chart.



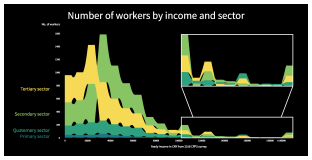
Iteration 4

The connections are smoothed with quadratic Bézier curves.



Iteration 5

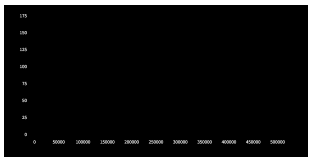
Title and labels are added.



5.3.3 Working hours vs. income

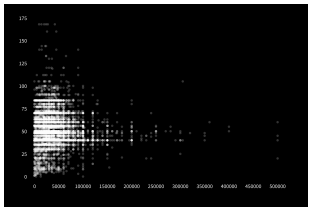
Iteration 1

X- and Y-axis ticks are drawn.



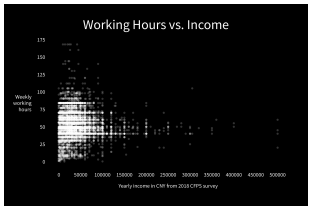
Iteration 2

A scatter plot is created by drawing transparent circles.



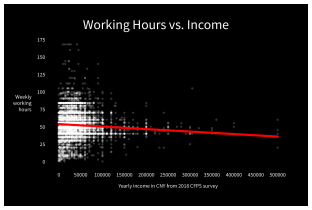
Iteration 3

Title and axis labels are added.

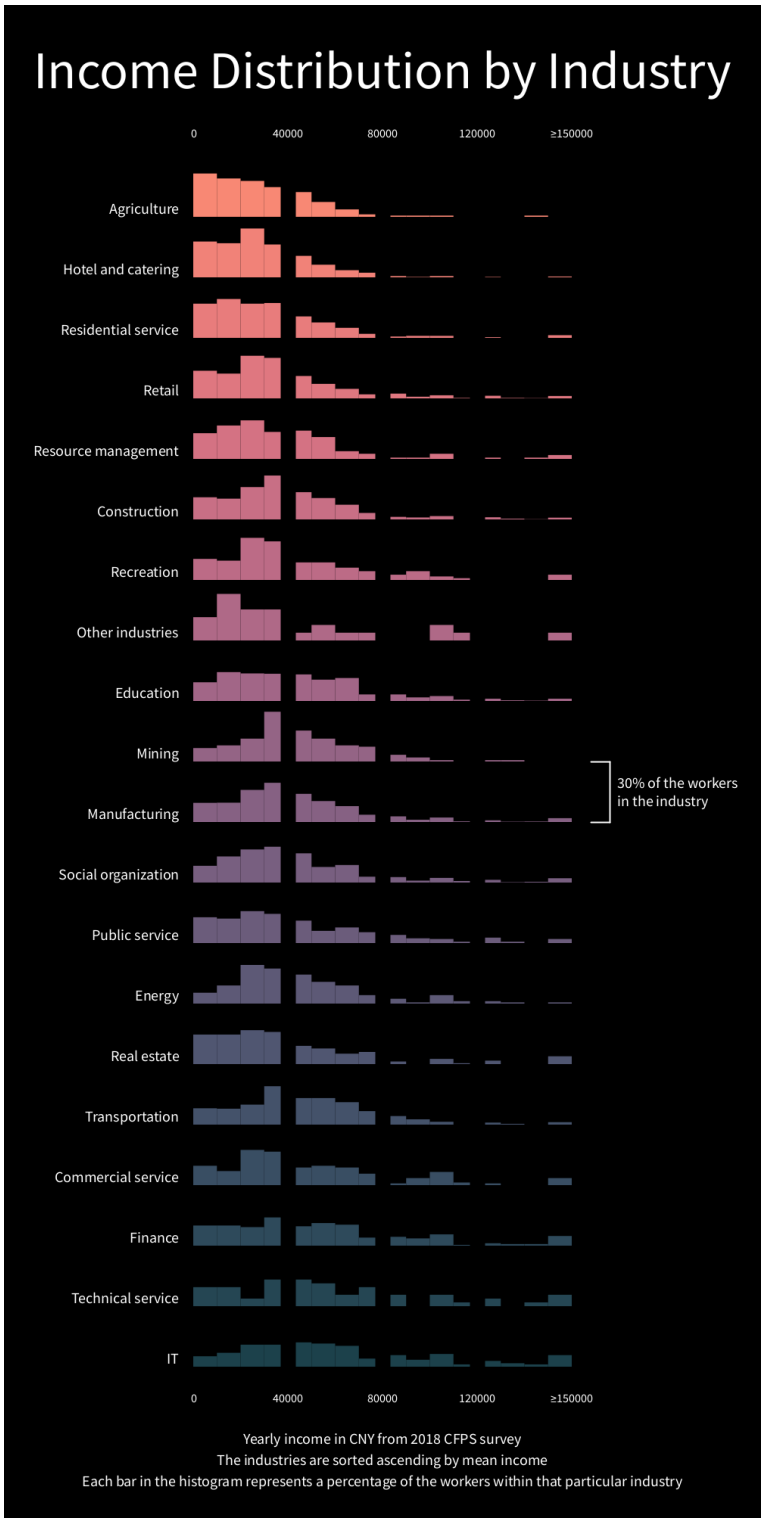


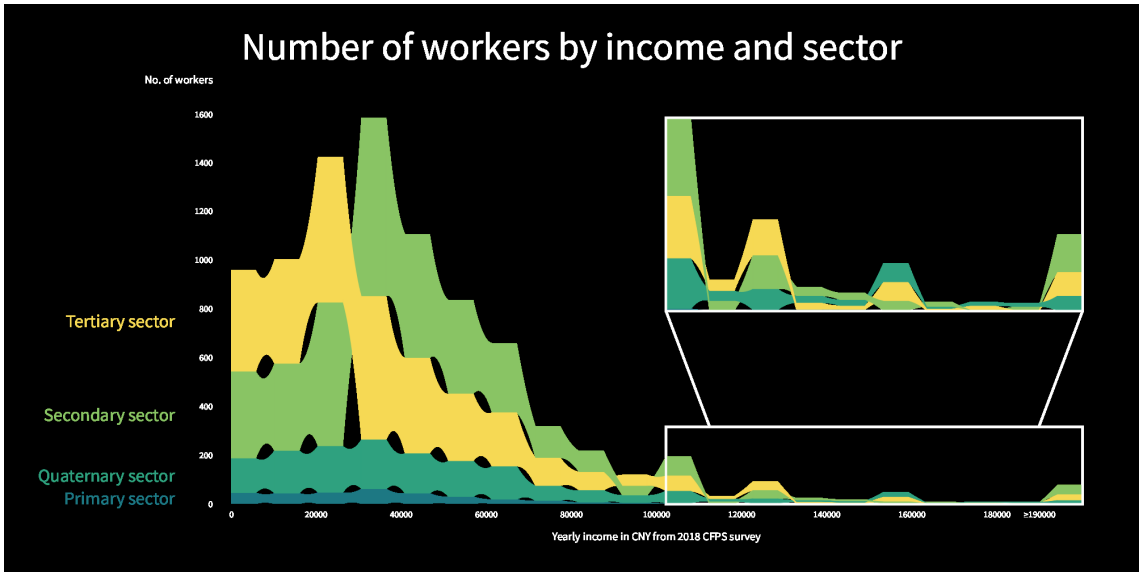
Iteration 4

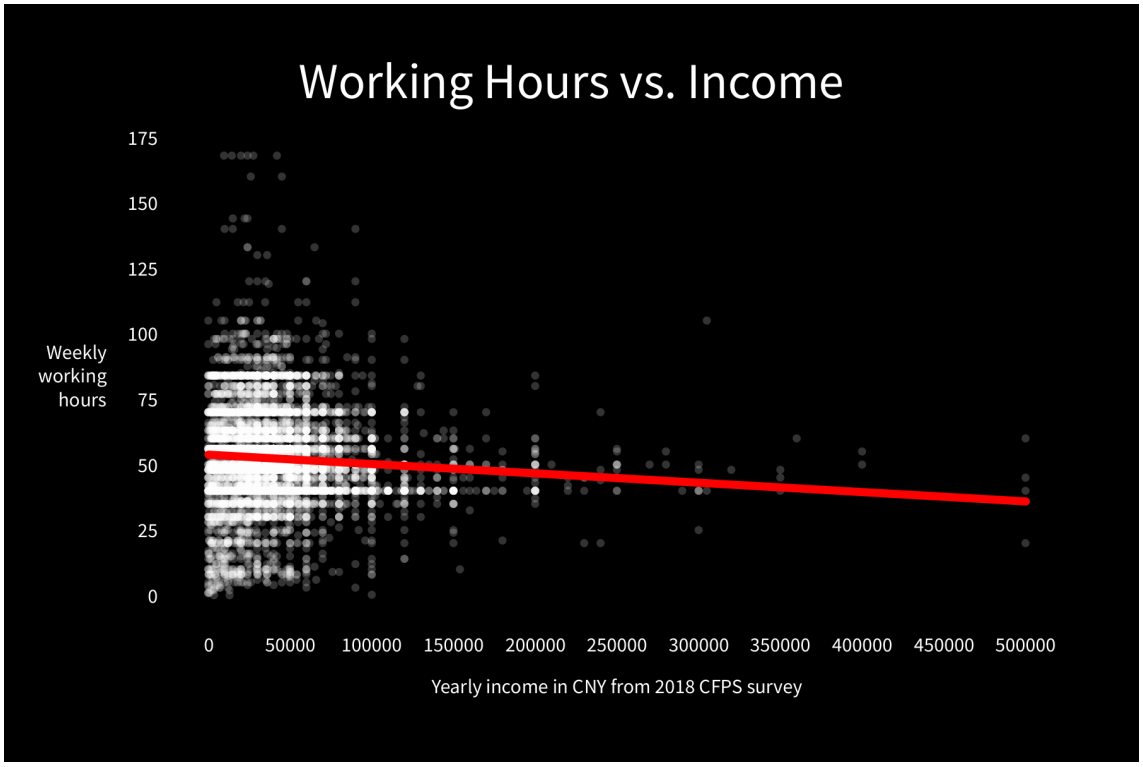
The best fitting line is found through linear regression.



5.4 Final results







5.5 References

CFPS: https://www.isss.pku.edu.cn/cfps/en/data/public/index.htmColor picker: https://tristen.ca/hcl-pickerChatGPT: Some code for data exploration and the code for linear regression

Chapter 6

Game: Infernoxene

6.1 Concept statement

This is a 2D pixel side-scrolling action puzzle game where you burn all the enemies to death by utilizing flammable oil and other objects on the map. It includes a mix of rigid body and falling sand physics to provide a physics-based immersive gameplay experience.

6.2 Generative rules

6.2.1 Inspiration

The real-life inspired aspect of this game is basically the amount of ways that you can tackle a problem (or, more specifically, kill someone) in real life compared to, for example, in an FPS game where you can only shoot the enemies to death. This game chose to specifically focus on the aspect of fire propagation and physics and how they could be used as weapons to deal with the enemies.

The video game inspirations is most directly Noita, a game where every pixel is simulated. There are also inspirations from games like Dark Messiah of Might and Magic and Half-life 2 where physics play an important role. The lethality of fire in Far Cry 2 is also an inspiration.

6.2.2 Gameplay

Starting point

You play as a character who can move up, down, left, and right in a given map. You can grab and throw objects and also cast sparks to light up flammable substances.

Objectives

Burn all the enemies to death while not getting caught up in the flames yourself.

Rules

Both the player and the enemy can move left and right, jump and ”fast fall”. Touching burning substance decreases both the player and the enemy’s health. Both the player

and the enemy die when they run out of the health.

The rigid bodies in the game world obey conservation of momentum except when external forces are applied. They could crash into each other and are under the influence of gravity and friction. The fluids are simulated through falling sand cellular automata rules (which doesn't fully obey rules in the real world). Fire is ignited by spark and are propagated through flammable substances.

6.2.3 Core mechanics and loop

Player move push and throw objects around and cast oil bottles to create a way to set enemies on fire. They can cast a spark to set oil on fire to damage the enemies while avoid receiving damage themselves. Simultaneously, objects can also be used as obstacles for the player to defend themself from fire damage or to obstruct the enemies from throwing objects.

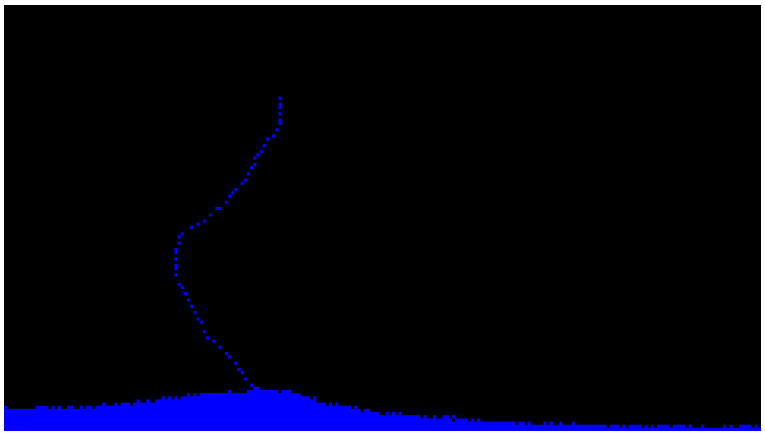
6.2.4 Anticipated player experience

Players would be thrilled in the brutal termination of the enemies while engaging in the intellectual problem-solving process of making the best of what they have on the map and to utilize the physics to conduct this process creatively.

6.3 Iterative process

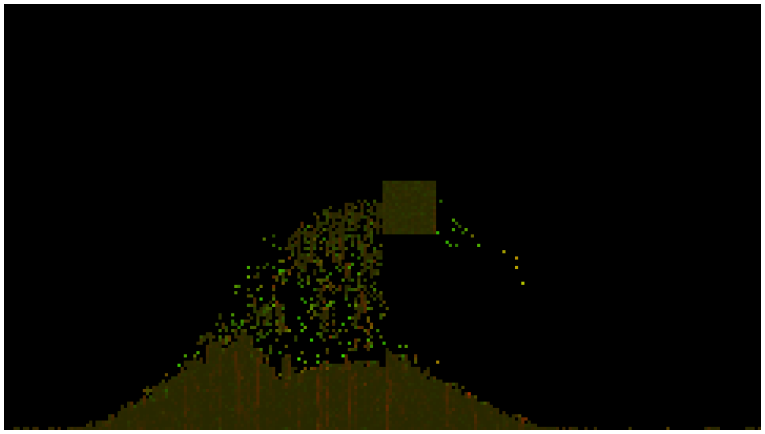
6.3.1 Iteration 1

I begin by implementing a basic falling sand simulation for water. This is done through simple cellular automata rules where each water pixel could move downward (including diagonally downward) and sideways during each simulation frame.



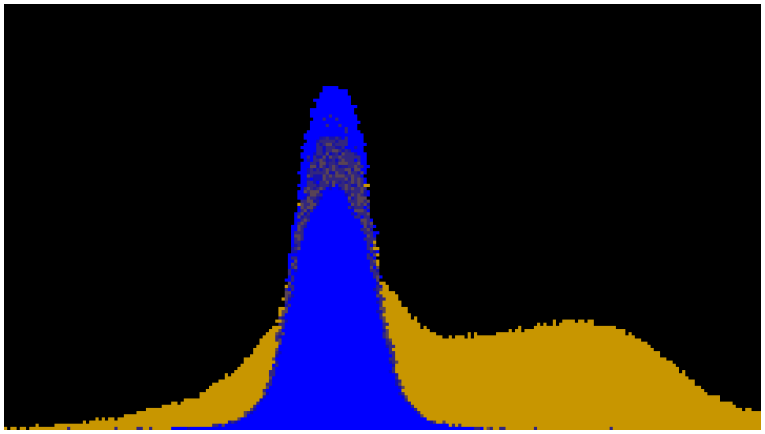
6.3.2 Iteration 2

A fail attempt at simulating every pixel with Brownian motion, surface tension, and pressure. I thought up some ungrounded ways to simulate these rules, which did not work well. Therefore, I went back to the basic approach afterwards.



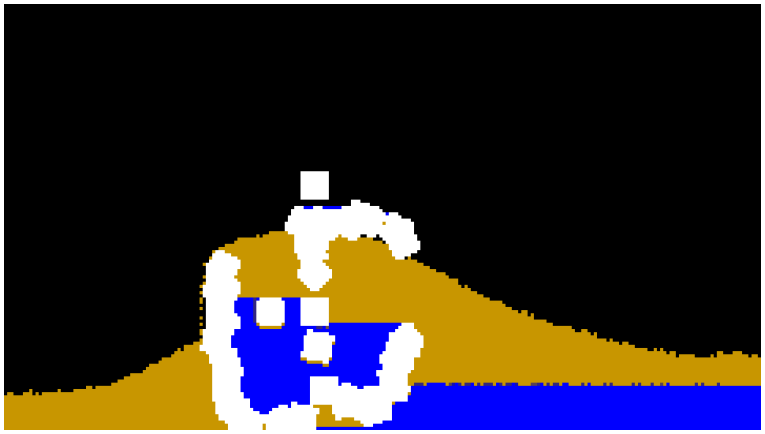
6.3.3 Iteration 3

Add oil into the fluid simulation. It can also interact with water by having each oil pixel moving up when contacting with water. Note that in the final game I did not include water at all as my time have ran out, but there are potential ways in which oil-water interaction could be implemented.



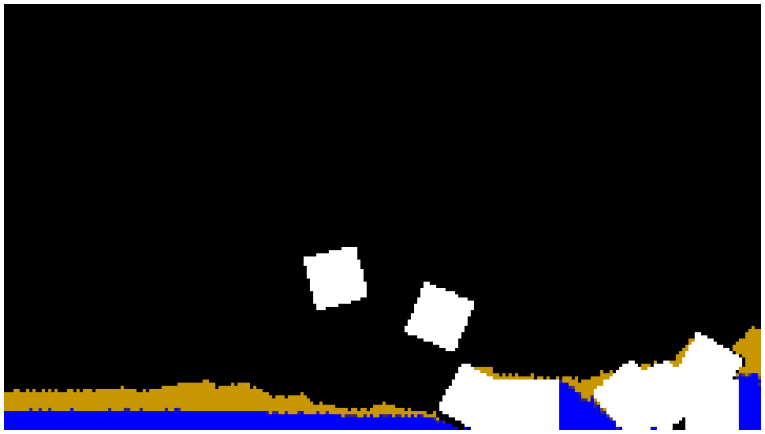
6.3.4 Iteration 4

Add rigid bodies that cannot move yet. They will however interact with liquids by displacing all the pixels that share the same space.



6.3.5 Iteration 5

Add collision detection and resolution concerning rigid bodies and the map border. More details are explained below.



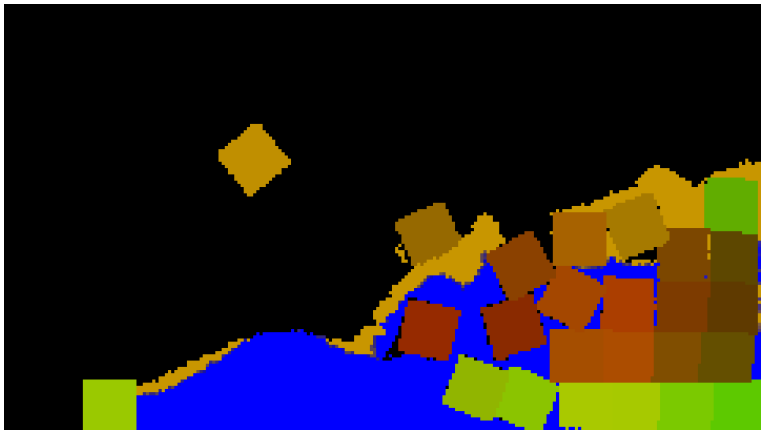
6.3.6 Iteration 6

Add collision between rigid bodies. Each rigid body is modeled as connected squares, but the edges of a rigid body have the shape a series of circles. This could be understood as this:

```
oooo
o##o
o##o
oooo
```

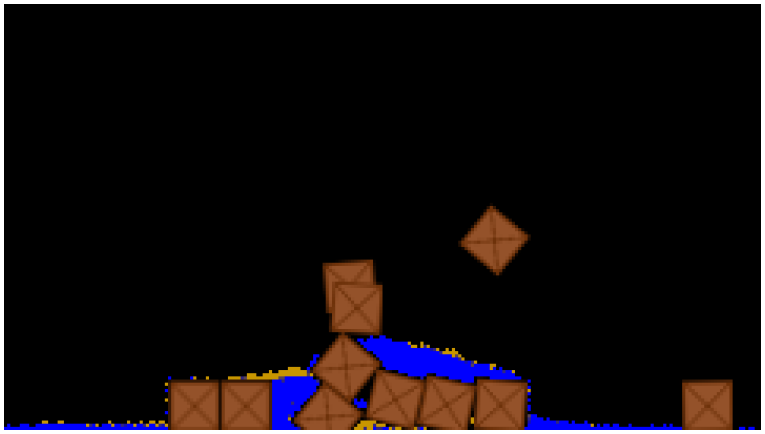
For a rigid body of square shape.

The collision between rigid bodies are then resolved using a hand-made sequential impulse solver.



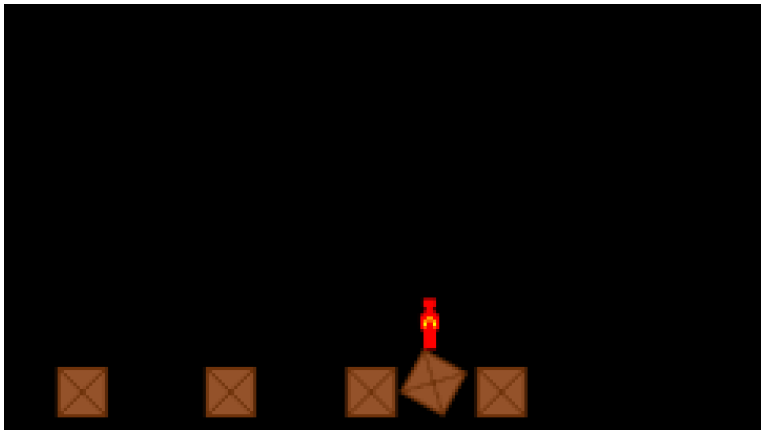
6.3.7 Iteration 7

Add textured sprites. I have used the components pattern from Game Programming Patterns to organize each game object, where each GameObject class would be consisted of a Sprite, a Input, a Logic, and a body.



6.3.8 Iteration 8

Add controllable player.

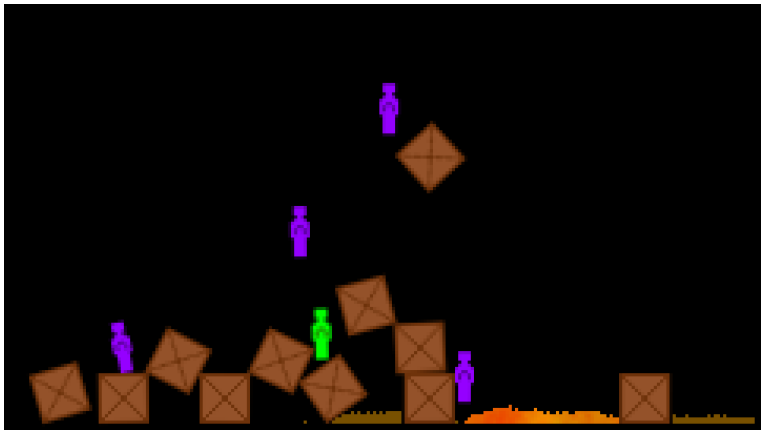


6.3.9 Iteration 9

Add oil bottles that break on hard impact and release oil.

6.3.10 Iteration 10

Add enemies that move around randomly. Oil can now be ignited with sparks and that fire will spread within oil. Player and enemy will die instantly upon contact with burning oil.

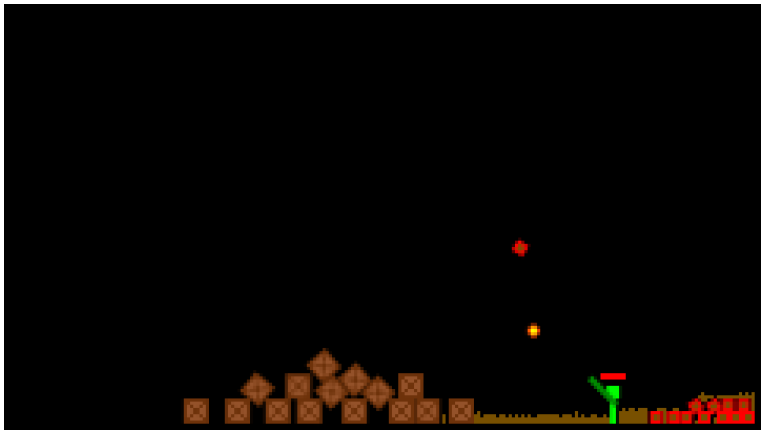


6.3.11 Iteration 11

Add health bar and that contact with burning oil is not a one-hit kill anymore, but will decrease the health of the agent that comes into contact with it proportionally.

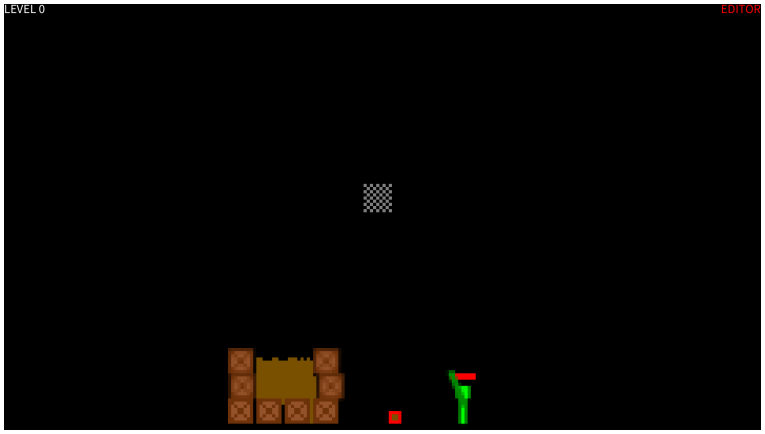
6.3.12 Iteration 12

The ability to throw objects and cast spark are implemented.



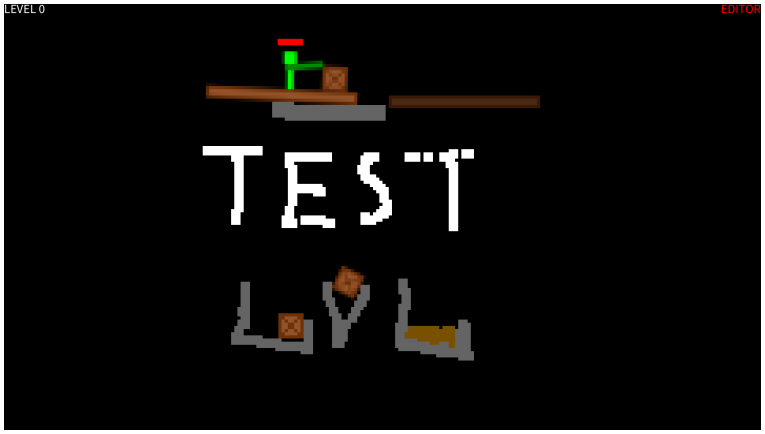
6.3.13 Iteration 13

A actual level editor is implemented. You may preview objects in the level editor. Level loading and saving are also implemented through manual serialization of GameObject.



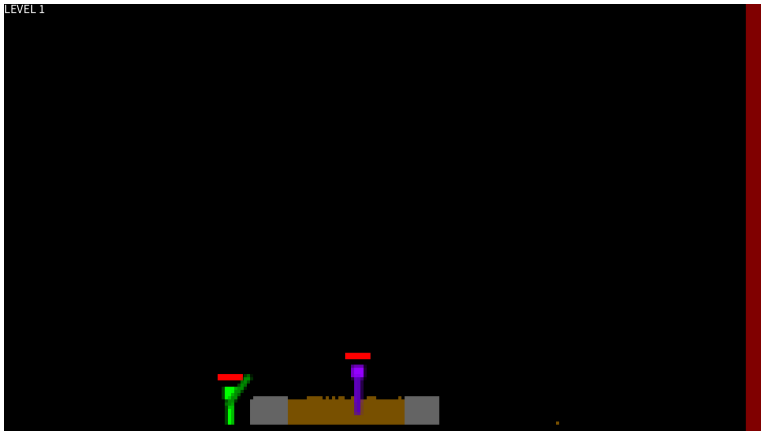
6.3.14 Iteration 14

Add decorative and solid pixels in falling sand simulation. Add plank object.



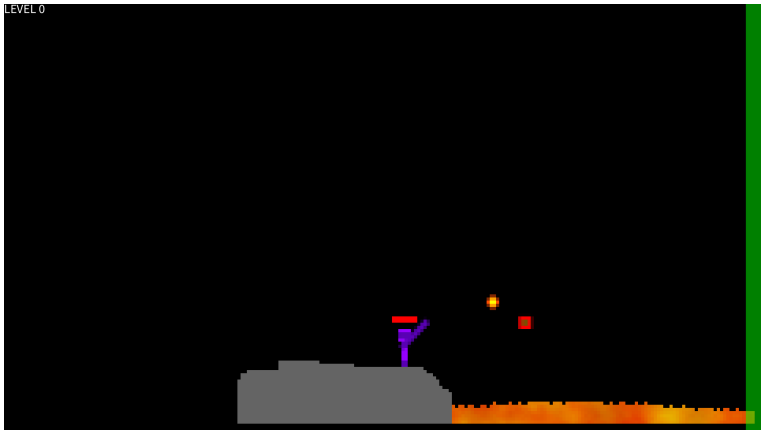
6.3.15 Iteration 15

Add level exit at the right side of the screen where you may exit the level by entering it while all enemies are dead.



6.3.16 Iteration 16

Add sentry, which is a type of enemy that stays still and cast oil bottle and spark alternatively continously.



6.4 Final result

6.4.1 Level 0

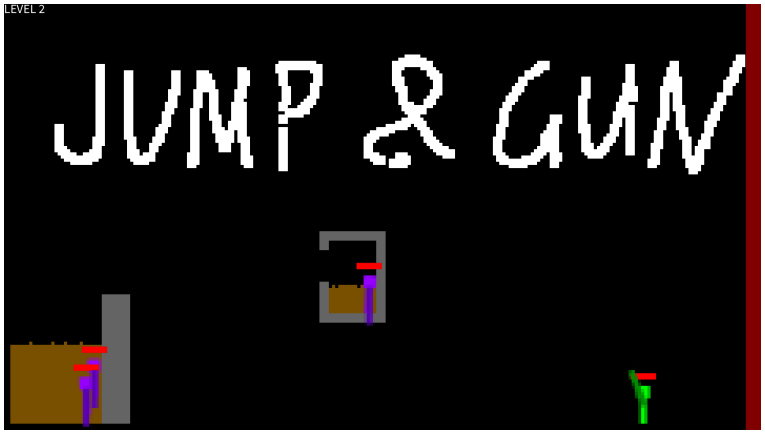
Title screen and introduction to basic movement.

6.4.2 Level 1

Introduction to the spark casting and fire propagation mechanics. The player must use spark to ignite the oil pool and burn all the enemies to death.

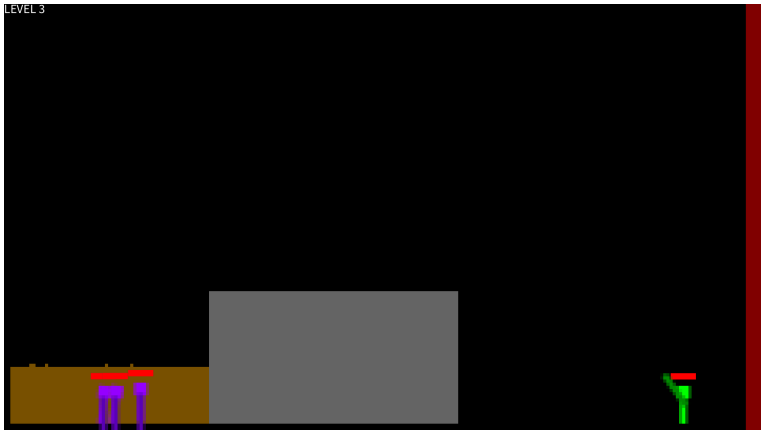
6.4.3 Level 2

Introduction to how the velocity of the spark projectile is relative to your movement velocity.



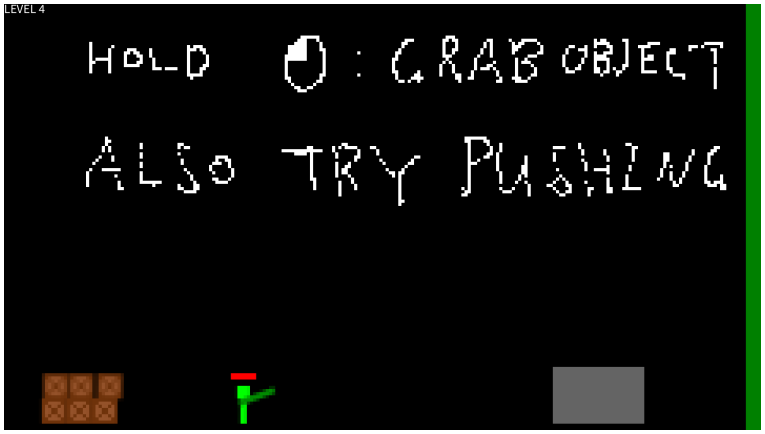
6.4.4 Level 3

This level requires the player to run, jump, and cast spark.



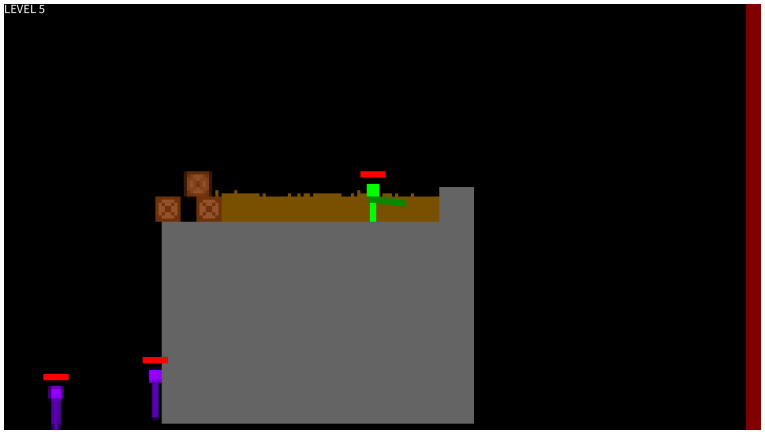
6.4.5 Level 4

This level introduces the grabbing mechanic and how you can push objects.



6.4.6 Level 5

This level introduces how a rigid body blocks the flow of liquid.

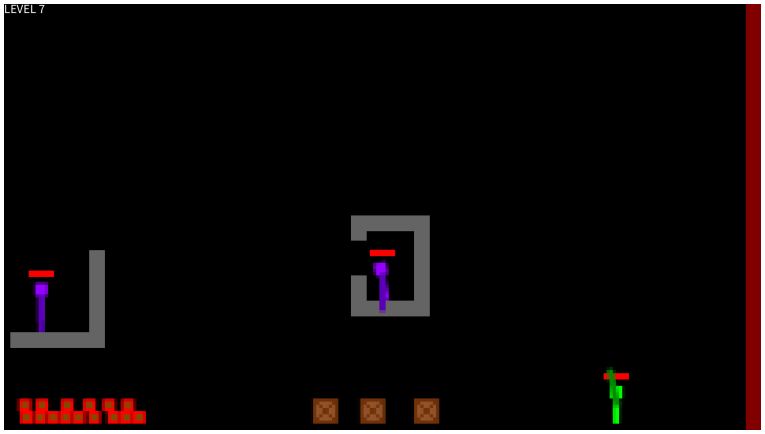


6.4.7 Level 6

It demonstrates how oil bottle works: How you can throw it and how it shatters on impact.

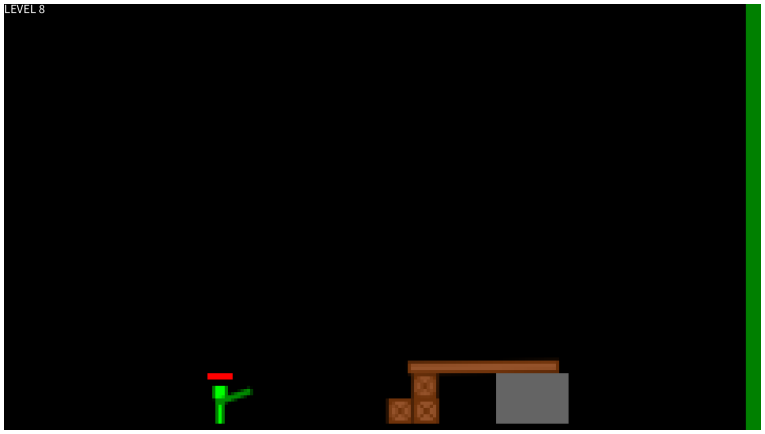
6.4.8 Level 7

This level further tests the players ability to utilize bottle throwing, spark casting, and other mechanics previously introduced.



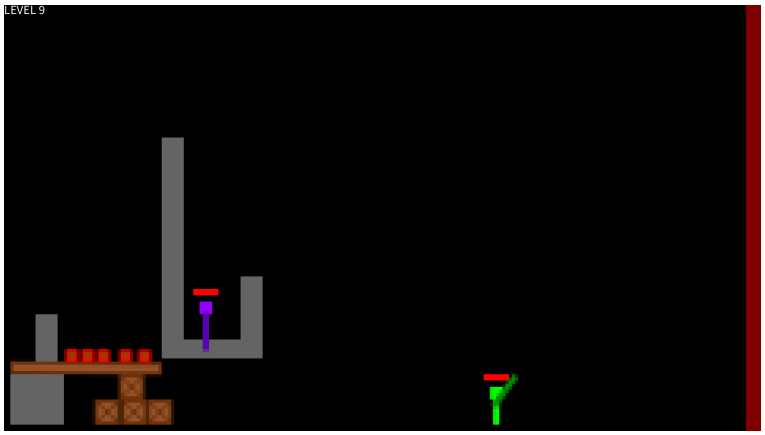
6.4.9 Level 8

This level shows how you can make a ramp using a tilted plank.



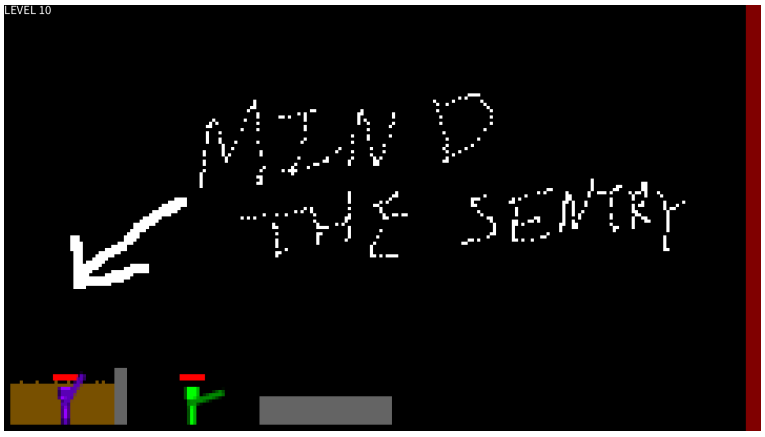
6.4.10 Level 9

This level demonstrates how you can remove the boxes to retrieve the bottles before using them as weapons.



6.4.11 Level 10

Introduction to the sentry enemy type.

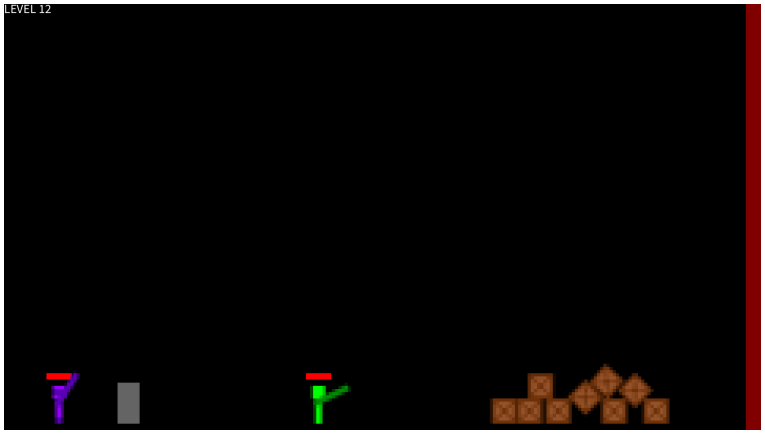


6.4.12 Level 11

The player is challenged to defeating the sentry using oil bottles.

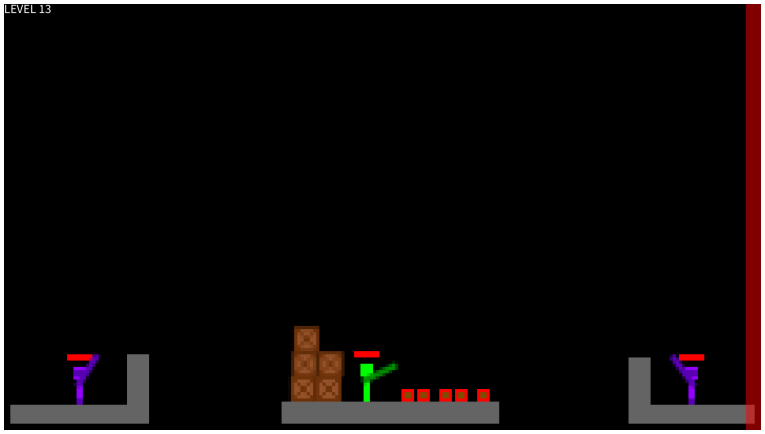
6.4.13 Level 12

The player could try blocking the flow of sentry's oil to have it burn itself to death.



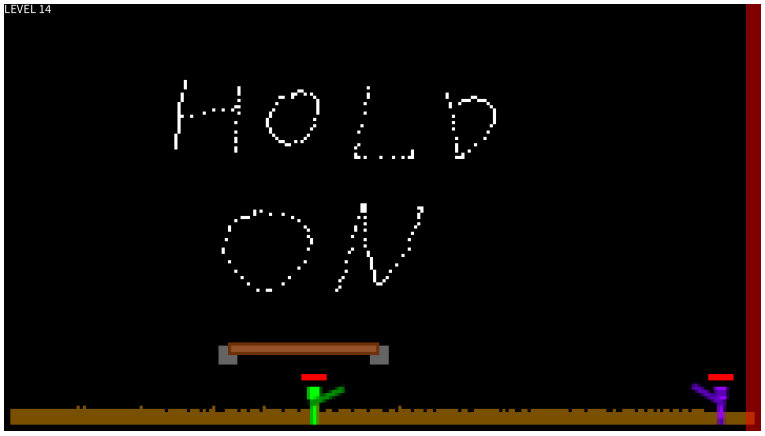
6.4.14 Level 13

A very difficult challenge where the player has to face two sentries simultaneously while the burning oil builds up.



6.4.15 Level 14

This level demonstrates that the player can hold on to the plank to escape the burning oil.



6.4.16 Level 15

A randomly arranged final level.

6.5 Controls

- P: Restart
- ‘ (The key above tab): Toggle playing/editor
- Enter: Screenshot

6.5.1 Playing

- WASD: Movement
- Left mouse button: Grab
- Right mouse button: Cast spark
- Drag mouse: Move arm

6.5.2 Editor

- Select the object to place:1: Solid2: Oil3: Box4: Plank5: Oil bottle6: Spark7: Player8: Enemy9: Sentry0: Decoration
- Backspace: Deletion tool for falling sand
- Tab: Save level
- Left/right arrow: Previous/next level

6.6 Notes

- Do try the editor mode to play it like a sandbox or to skip levels!
- The written from scratch physics engine is really clunky...

6.7 References

- https://www.youtube.com/watch?v=prXuyMCgbTc
- https://gameprogrammingpatterns.com
- https://www.cs.ubc.ca/~rhodin/2020*2021*CPSC*427/lectures/D*CollisionTutorial.pdf
- https://box2d.org/files/ErinCatto*SequentialImpulses*GDC2006.pdf

Chapter 7

Abstract 3D Plant Based on L-system (Extra Credit for Week 6)

7.1 Concept

The main goal of this project is to create a basic semi-realistic model of plants with L-system. L-system represents a plant with a string where each character is a component. This string is then drawn in a fashion similar to turtle graphics. The plant is "grown" by replacing the characters in the string according to certain rules that are modeled after the real-life cell division and elongation events that occur during plant growth.

7.2 Generative rule

L-system is a parallel rewriting system where a string representing the components are replaced simultaneously in each iteration according to certain rules. The string of components will be read from left to right and drawn. An imaginary "cursor" would represent the origin and the orientation for the current component to be drawn, which would also be updated after drawing the current component. Parentheses are used to store the cursor states on a stack to allow drawing branches. The components of this L-system would be the branches, leaves, flowers, and the apices where the new branches, leaves, and flowers are created (due to them being the places where cell divisions occur).

In this project, the process of planet growth will be animated. Therefore, rewriting will occur each frame where the new components are created from apices and that the existing components will also grow accordingly (cell elongation). The rules for rewriting will be stochastic, so that a apex might be replaced by different components probabilistically in each iteration.

The branches, leaves, and flowers are abstract shapes in this project. The configuration (for example, how many branches the apex might create each time) and the rewriting rules are static. Randomizing these rules and adding more details like branch textures are wind animations could be a project for the future.

7.2.1 Vocabulary and rules

- A: apex

- B: branch
- L: leaf
- F: flower

The following replacements will happen probabilistically in each iteration:

- A -> BA
- A -> BLA
- A -> BFA
- A -> B(BA)(BA)...(BA)A

The existing components will also grow in size if possible.

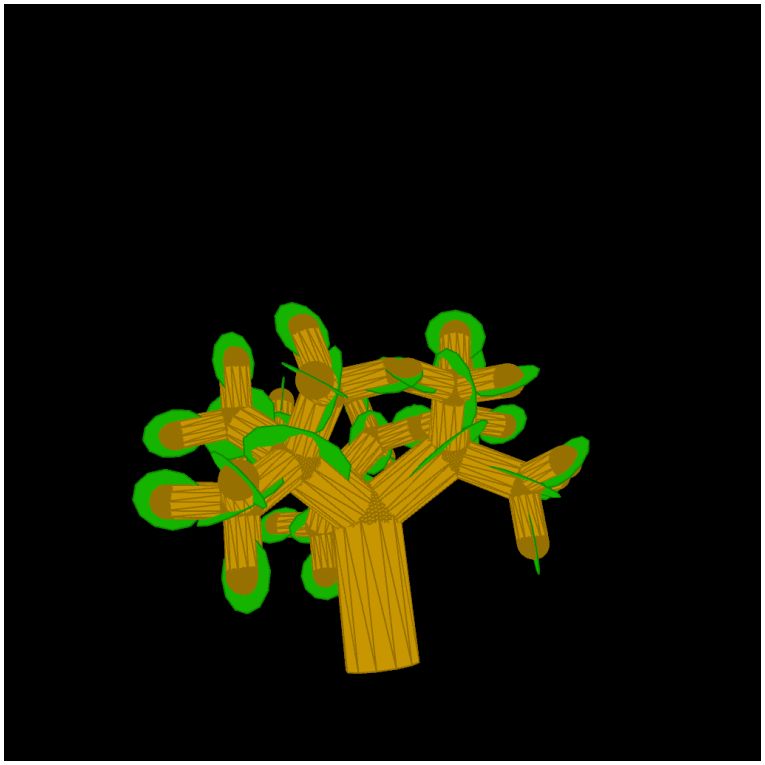
7.3 Iterative process

7.3.1 Iteration 1

Implementing the utilities for drawing a branch and rotating the coordinate space with mouse

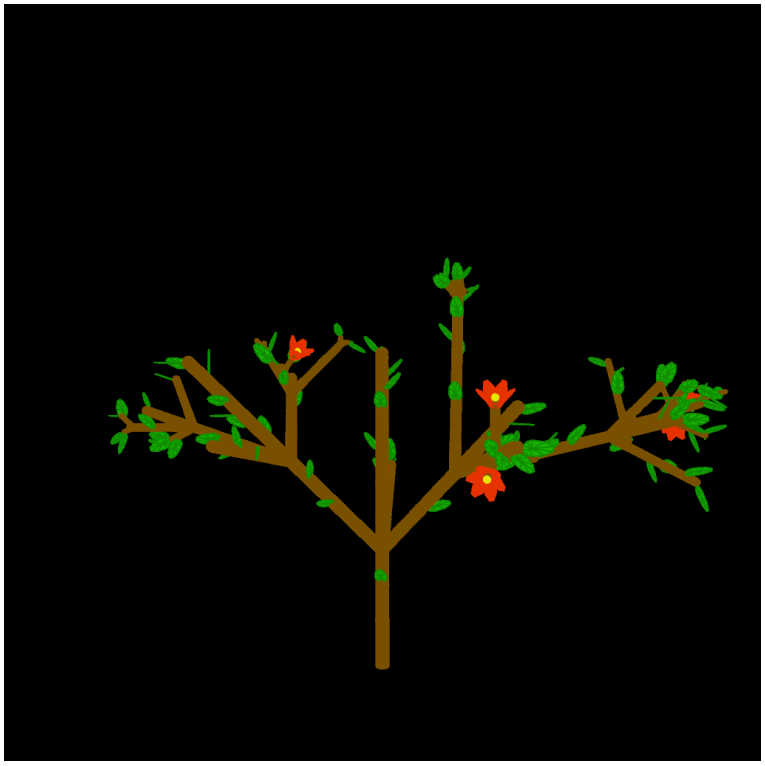
7.3.2 Iteration 2

A basic L-system with only rough prototypes of branches and leaves. It is unrealistic as branching does not happen only in apices.



7.3.3 Iteration 3

Relatively more detailed flowers and leaves are added. Leaves and petals are drawn as triangle fans fitted to unevenly squashed circles. Branching now only occurs at apices.



7.4 Final result

Configurations are tweaked slightly for the plant growth animation. Lateral branches now grow more slowly just like real plants (though this also reduces the computational workload).

![Final 1](/data/dku/mediart*206/week*6*extra/three branches){width=10cm} ![Final 2](/data/dku/mediart*206/week*6*extra/two branches){width=10cm} ![Final 3](/data/dku/mediart*206/week*6*_extra/four branches){width=10cm}

7.5 References

L-system: Prusinkiewicz, P., & Lindenmayer, A. (2012). *The algorithmic beauty of plants*. Springer Science & Business Media.Cylinder drawing: https://vormplus.be/full-articles/drawing-a-cylinder-with-processing