5-2018

# Procedural Generation and Rendering of Ink Bamboo Painting

Yangyang He

Procedural Generation and Rendering of Ink Bamboo Painting

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Computer Science from
The College of William and Mary

by

Yangyang He

Accepted for Honors

Timothy A. Davis, Director

James Deverick

Chun-yu Lu

Williamsburg, VA
May 4, 2018

# Procedural Generation and Rendering of Ink Bamboo Painting


Yangyang He

Computer Science Department

College of William and Mary,

Williamsburg, VA 23187-8795, USA

Email: yhe04@email.wm.edu

**Abstract**

This thesis describes an algorithm that generates various ink bamboo paintings. First, a completely procedural model is used to generate the geometric shape of bamboos. The model uses a grammar-like approach that recursively generates new parts of the bamboo in a randomized manner. The random parameters are bounded by rules that simulate the natural form of bamboo.

The structure of the bamboo is represented line segments with directions. Various ink stroke sprites of stalk, branch, or leaf shapes are mapped to line segments, using reverse mapping and bilinear sampling to eliminate aliasing effects. The sprites are mapped in different degrees of transparency to simulate the effect of various shades of ink produced by changes in forces when using an ink brush. Finally, a seal is applied to sign the work and enhance the visual effect.

The algorithm is implemented in Python 3 and can be run on any computer with the `imageio` library installed. The output of the program is saved in a PNG image file, which can be used for various types of illustrations. This model is able to produce unique images during every run, and would significantly reduce human labor in painting stylistically similar artworks of ink bamboo paintings.

## Acknowledgment

This thesis could not be done without the support from my advisor, my committee members, my family, and friends.

Foremost, I would like to express my sincere gratitude to my honors advisor, Dr. Timothy Davis, for his patience, immense knowledge, and continuous support during my project. His guidance helped me in all time of my undergraduate study and lead me into the field of computer graphics. I would like to thank the rest of my thesis committee: Professor James Deverick and Dr. Chun-yu Lu, for their encouragement, insightful comments, and instructive feedback provided during my thesis defense.

Besides my committee, I would like to thank Dr. Pieter Peers, for his informative lectures in Computer Graphics and Game Design. Both of the courses provided me substantial technical grounds to build up this model. I would also like to thank Dr. Xin Conan-Wu. Her Chinese Art History class inspired me to come up with this inter-disciplinary research topic.

Lastly, I am immeasurably grateful to my family and friends for their patience and encouragement that fueled me throughout my final year at College of William & Mary.

# Contents

# Chapter 1

# Introduction

## 1.1 Art Background

Ink landscape painting has been an important branch of East Asian art and art history for hundreds of years. It originated in the Fourth Century as the backgrounds of Buddhist murals. However, in paintings from Tang Dynasty, landscapes still only serve as structural elements. They did not became the subject matter of ink paintings until the Eighth Century, and reached its peak in Song and Yuan Dynasty. A large number of Chinese, Japanese, and Korean artists have been active in ink landscape painting since then, and different schools have developed the art over a long period of time.

There were many genres of ink landscape paintings, including religion, figure, architecture, foreign custom, dragon and fish, mountain and water, horse and buffalo, bird and flower, ink bamboo, and vegetable and fruit. Among these subgenres of East Asian painting, bamboo has been the most popular motif, as it represent the reclusive dream of the literati class. In a work of bamboo painting in ink, a skilled artist and calligrapher will paint a bamboo stalk or group of stalks with leaves. The contrast between the foreground and background, and between the varying textures represented

Figure 1.1: A Page from Wu Zhen's *Ink Bamboo Album* (Yuan Dynasty)

by the stalks and the leaves, gave scope to the painter to demonstrate his or her mastery with an inkpot and a brush. [14]

Each famous bamboo painter developed his or her own style. For example, the work of Wu Zhen (Yuan Dynasty), as seen in Figure 1.1, features lighter ink strokes, while Zhen Banqiao (Qing Dynasty) preferred stronger contrasts (Figure 1.2). Because of the volume of bamboo works painted over time, ink bamboo remains the most popular landscape genre even in modern days, and the production of a work of ink bamboo became one of the standard subjects to which an East Asian student could be set in a competitive examination.

## 1.2 Related Work

In the digital era, the modeling and rendering on ink paintings has attracted much attention from graphics researchers. Various research work has been done on the modeling of trees and 2-D rendering of ink paintings (Chu and Tai 2005[1]; Huang et al.

Figure 1.2: A ink bamboo work by Zheng Banqiao (Qing Dynasty)

2003[2]; Wang and Wang 2007[4]; Way et al. 2002[6]; Xie et al. 2011[7]; Xu et al. 2002; Yang et al. 2012[9]; Yu et al. 2002[11]; Yu et al. 2003[12]). Other researchers pushed their work into three dimensional space (Zhang et al. 1999[13]; Li and Yu 2006[3]).

In terms of procedural generation, each species of vegetation would require a different model to generate. Wang et al. [5] developed a model in 2014 that does 3-D renderings of oak and elm trees; however, despite bamboo being an important subject in ink landscape painting, less work has been done on the modeling and rendering of ink bamboo painting. This paper fills the vacancy in this field of research by introducing a new procedural method that generates 2-D bamboo models and renders the structure by mapping various styles of ink strokes.

# Chapter 2

# Procedural Generation

Through observation of ink bamboo paintings from both ancient and contemporary artists, I noticed that, although each bamboo looks different, the structure of the bamboo follows certain patterns. Thus, a procedural algorithm, as opposed to a manual one, could be used to generate the forms of the bamboo to produce structurally different, yet visually similar, models. I set up a group of random parameters within certain a bounding range, to determine the quantity, position, height, length, and orientation of the bamboo stalks, branches, and leaves.

## 2.1 Structural elements

Bamboo features tree-like structures. With three levels of nodes: `Stalk`, `Branch`, and `Leaf`, similar to the reported technique in [Wang et al. 2014]. These nodes are connected by `Joints` and are grouped into lists.

A `Joint` is an innovative structure introduced by this paper, that connects different parts of the bamboo. It has a position, which is a 2-D point, a parent, and two children. Joints are represented by ”X”s in the structure graph. They are not visible in the ren-

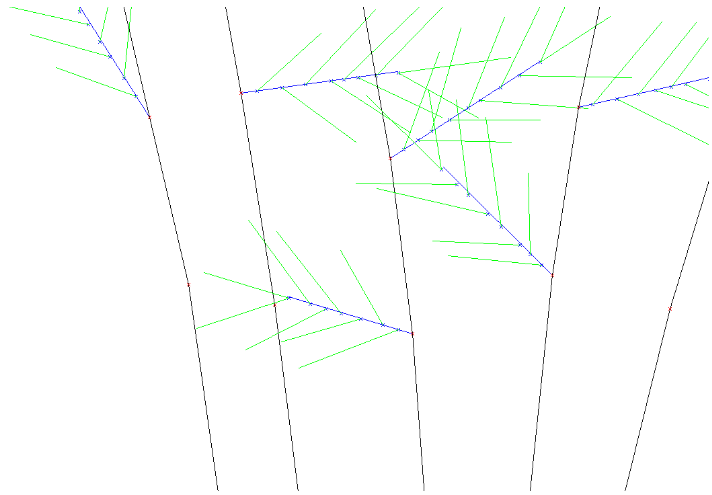


Figure 2.1: An example of generated bamboo structure

dered image.

A `Stalk` represents one independent bamboo. It has a `root` and a `top`, both of which are joints, that represent the bottom end and the top end of the current stalk, respectively. `Stalk` has a list of line segments (representing stalk segments), a list of joints, and a list of branches that grow out from the main stalk of the bamboo. `Stalk` also has a direction field that stores a 2-D vector representing the upward direction of the top segment of the current stalk. Stalks are represented by black line segments in the structure graph, as shown in Figure 2.1.

A `Branch` is structurally very similar to a stalk, except that instead of having a list of child branches, it has a list of leaves that grow from the current branch. Each `Leaf` is represented by a line segment. Branches are represented by blue lines while leaves are represented by green lines in the structure graph.

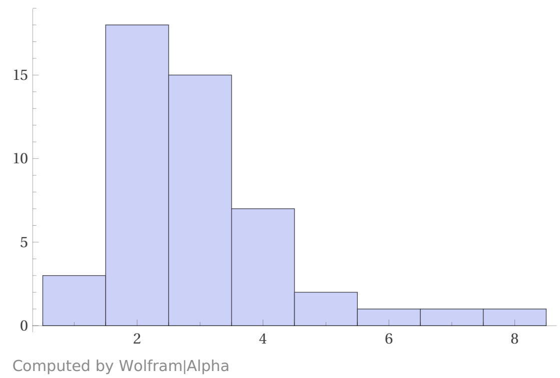



Figure 2.2: Distribution of bamboo quantities in 50 artworks

## 2.2 Quantity of Stalks

The number of bamboo stalks significantly influence the visual effect of an image. If there are too many bamboos, the image will be overcrowded, and the details would not be easily seen. If there are too few bamboos, the image may look empty.

Artists have varying opinions on the number of bamboo stalks in their artwork. For this paper, 50 ink bamboo images by both ancient and contemporary artists were selected and analyzed to discover the number of bamboo stalks in them. Most paintings have no more than four stalks. The distribution of the quantity is shown in Figure 2.2.

This data was converted into a probability distribution table and used as the probability distribution when generating the random number for bamboo quantity. The upper bound of the number was set to 5 to simplify the model and save time during the rendering of the image. As a result, the probability of the stalk quantity in the model is also shown in Table 2.1.

| *Stalk Quantity* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *Possibility* | 6% | 38% | 30% | 14% | 12% |

Table 2.1: Probability distribution of stalk quantities

## 2.3 Positions of Stalks

The positioning of the bamboo stalks is essential to the composition of the image as well. The stalks should not scatter at the marginal areas, resulting in an unbalanced composition. They should neither be too close, so that the ink strokes would overlap with each other.

In the experimented model, each stalk procedurally grows from its root. Thus, the horizontal position of the root determines the area that the stalk covers on the image. The x-coordinate of the first root was set at a random position between the trisect point and half point of the width of the image. Each of the following roots was placed to the right of the previous root with a random separation between 100 and 200 pixels, as shown in the code below. This way of positioning would ensure that the bamboo stalks appear near the golden ratio points on the width of the image. Every root was placed 10 pixels below the x-axis to hide the lower end of the stalk.

```
def get_root_position(width):
    return Vec2d(random.randint(int(0.33 * width), int(0.5 * width)), -10)

def get_new_root(previous_root):
    return Vec2d(previous_root + random.randint(100, 200), -10)
```

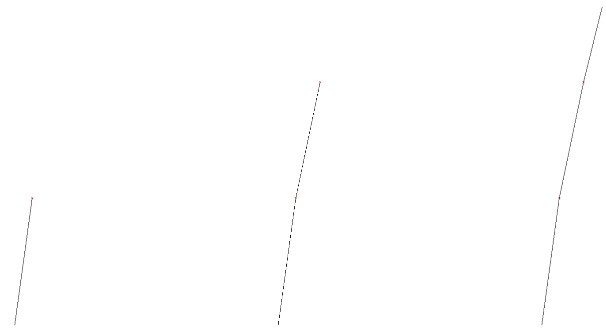


Figure 2.3: Growing process of a stalk

## 2.4 Growth of Stalks

For each root, a new stalk was constructed. A random integer between 3 and 5 was chosen as the number of segments of that stalk. The `grow()` method of the stalk was called the corresponding number of times.

During each call, a new segment of a random length grew upwards in a slightly rotated direction from the current top of the stalk. The first segment rotated $[0, 10]$ degrees, and the proceeding segments rotated $[0, 5]$ degrees in the same direction. The length fell between that of the shortest and longest stalk texture obtained (in the range $[242, 335]$). New segments of the same stalk rotated towards the same side to create the bending form of the bamboo. A new `Joint` was constructed as the new `top` of the stalk. This process is shown in the code below and illustrated in Figure 2.3.

```
def grow(self):
    rand_length = get_segment_length()
    rand_angle = get_random_angle()
    new_seg = Segment(self.top.position, \
        self.direction.rotate(rand_angle * self.bend_side), rand_length)
    self.top = Joint(new_seg.get_end().x, new_seg.get_end().y, new_seg)
```

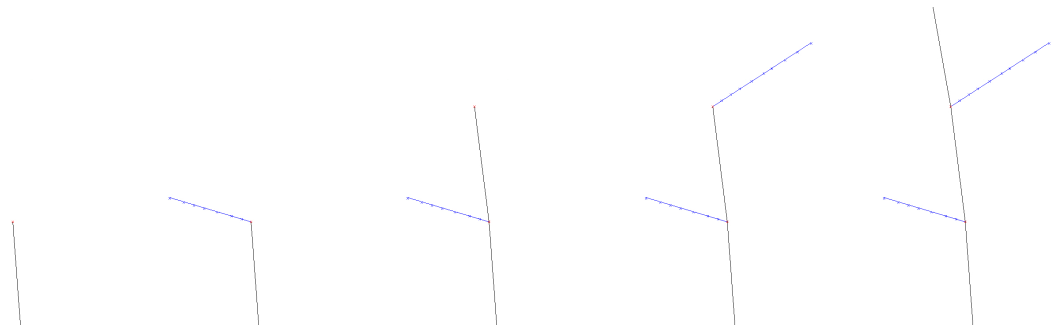


Figure 2.4: Growing process of two branches on one stalk

## 2.5 Branches

After each growth of new stalk segments, the model decided if a branch would grow from the new joint. If so, it would grow on the opposite side of the stalk of the previous branch. The direction of the branch was obtained by rotating the direction of the stalk a random angle from the vertical direction towards the determined side. The angle was between 30 and 85 degrees to ensure that a branch always grew upward. The growth of new branches from a stalk is shown in the code below.

```
# within Stalk.grow()
if self.decide_grow_branch():
    new_branch = Branch(self.top, self.branch_side)
    self.branch_side *= -1
    for j in range(get_segment_count()):
        new_branch.grow()
    new_branch.grow_tip_leaf()
    self.top.l_branch = new_branch
    self.branches.append(new_branch)
```

The growth of a branch was similar to that of a stalk. The `grow()` method will be called a random number (in the range $[6, 10]$) of times. The segment length, which determined the distance between leaves, fell between 20 and 40 pixels, so that the total length of a branch varied between 120 and 400 pixels. The new branch segments remained in the same direction as the first segment. During rendering, a branch was considered as a single line segment, whose origin was at the joint with the stalk and end was at the tip of the branch. The growth of a branch is shown in the code below. Figure 2.4 shows the process of growing two branches on a single stalk.

```
def grow(self):
    seg_length = random_seg_length()
    leaf_length = random_leaf_length()
    new_seg = Segment(self.top.position, self.direction, seg_length)
    self.top = Joint(new_seg.get_end().x, new_seg.get_end().y, new_seg)
```

## 2.6 Leaves

After each growth of new branch segments, one leaf of random length grew at alternate sides of the branch to simulate the alternate leaf pattern of bamboo, as shown in the left of Figure 2.5. The angle between the leaf direction and the branch direction was between 30 and 45 degrees. This process is shown in the code below.

```
# within Branch.grow()
# grow one leaf to each side at the joint
leaf_angle = random_leaf_angle()
leaf = Segment(self.top.position, \
  self.direction.static_rotate(leaf_angle * self.leaf_side), leaf_length)
self.leaf_side *= -1
self.top.l_branch = leaf
self.leaves.append(leaf)
self.joints.append(self.top)
self.segments.append(new_seg)
self.count = len(self.segments)
```

At the tip of each branch, one leaf facing toward the direction of the branch was added, as shown below. The generated leaf pattern is shown in the right of Figure 2.5.

```
def grow_tip_leaf(self):
    leaf_length = random_leaf_length()
    leaf = Segment(self.top.position, self.direction, leaf_length)
    self.top.l_branch = leaf
    self.leaves.append(leaf)
```

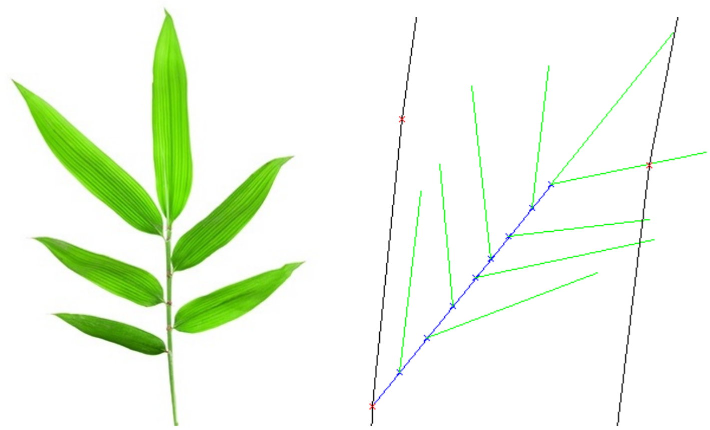


Figure 2.5: The alternating leaf pattern

## 2.7 Result of Generation

To summarize the generation process at a high level, the model first generated a list of roots. For each of these roots, the model grew a stalk consisting of several segments. For each joint between the segments, there was a chance that a new branch would derive. The branch consisted of several segments, and at each joint, a pair of leaves would derive. There was also a single leaf at the tip of the branch. The resulting structure is shown in the beginning of this chapter (Figure 2.1).

Once the generation process is complete, we have a collection of lists of line segments, which represent the positions of stalk segments, branches, and leaves. Examples and more details on the generation results are discussed in section 4.1.

# Chapter 3

# Rendering

## 3.1 Creation of Sprites

The ink stroke sprites were created by cropping out single stalks segments, branches, and leaves from Natali Myasnikova's "Japanese Bamboo Illustration". They were then rotated so that their local y-axes (the lines that connects the two ends of stalks and branches or the bottom and tip of leaves) align vertically.

In order to prevent unwanted overlap during rendering, different shades of black in the sprites needed to be changed to different degrees of transparency (or opacity), and the white backgrounds of the sprites needed to be changed to transparent, as done by the code below.

```
px = imageio.imread(filename)
for y in range(px.shape[0]):
    for x in range(px.shape[1]):
        shade = px[y][x][0]
        px[y][x] = [0, 0, 0, 255 - shade]
imageio.imwrite("out" + filename, px)
```

In total, this process resulted in 3 stalk styles, 2 branch styles, and 5 leaf styles. (Figure 3.1 - 3.3) When applied to various structures, they can produce hundreds of

Figure 3.1: Ink strokes for stalk segments

Figure 3.2: Ink strokes for branches

different bamboo forms.

## 3.2 Reverse Texture Mapping

The traditional forward mapping method maps pixels from the source image onto the destination image. For some cases, a pixel on the source image might be mapped to several pixels on the destination image, creating aliasing effects. In this model, the sprites are mapped to the generated structure using a reverse mapping method with bilinear sampling and alpha-compositing. First, every pixel on the canvas is transformed to the coordinate plane of the sprite. The pixels are first rotated around the origin of the line segment, so that the direction of that segment aligns with the y-axis. This was accomplished by translating the points to the origin, rotating the points

Figure 3.3: Ink strokes for leaves

by multiplying them with a rotation matrix (3.1), and then translating the coordinates back to their original location, as shown in the code below (3.1).

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.1}$$

```
def rotate_to_align(v1, v2, origin, point):
    """ rotate point around origin so that v1 aligns to v2 """
    sin = v1.sine_angle(v2)
    cos = v1.cosine_angle(v2)
    p = point - origin
    result = np.matmul([[cos, -sin], [sin, cos]], [p.x, p.y])
    return Vec2d(result[0], result[1]) + origin
new_coord = rotate_to_align(seg.direction, Vec2d(0, 1), seg.origin, Vec2d(x
    , y))
```

Note that we do not need to compute the degree of $\theta$. The sine and cosine of *theta* can be calculated from the normalized vectors $v1$ and $v2$.

```
def cosine_angle(self, v2):
    return self.dot(v2) / (self.length() * v2.length())

def sine_angle(self, v2):
    return self.x * v2.y - v2.x * self.y
```

The pixels are then stretched (or squashed) along the y-axis, so that the line segment to be textured has the same length as the sprite. Since a line segment does not have

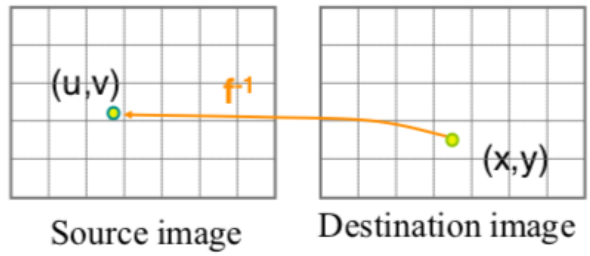



Figure 3.4: Reverse mapping

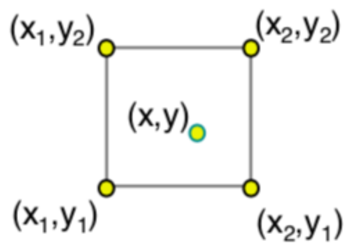



Figure 3.5: Bilinear sampling

a width, the mapped area on the canvas is not proportional to that of the sprite. The width of the sprite is preserved. This result is actually desirable, because segments of different length should have the same width.

```
def stretch_y(point, seg, scale):
    dy = (point.y - seg.origin.y) * scale
    y = seg.origin.y + dy
    return Vec2d(point.x, y)
new_coord = stretch_y(new_coord, seg, sprite.shape[0] / seg.length)
```

After this transformation, each pixel $(x, y)$ on the canvas is mapped to a point $(u, v)$ in the sprite's coordinate system (Figure 3.4). Because the referenced position $(u, v)$ does not usually have integer coordinates, the color from the four neighboring pixels are sampled using bilinear interpolation, as shown in the code below and in Figure 3.5.

```
def bilinear_sample(point, sprite):
    """ get the target color using bilinear interpolation """
    x1 = math.floor(point.x)
```

```
        x2 = x1 + 1
        y1 = math.floor(point.y)
        y2 = y1 + 1
        dx = point.x - x1
        dy = point.y - y1
        a = sprite[y1][x1] * (1 - dx) + sprite[y1][x2] * dx
        b = sprite[y2][x1] * (1 - dx) + sprite[y2][x2] * dx
        result = a * (1 - dy) + b * dy
        return result
```

## 3.3 Applying Different Shades

In traditional ink paintings, different weights of ink strokes result in different shades of black color on the image. Although this model could not simulate weights of ink strokes in a dynamic manner, it can simulate the effect by using alpha-composition when applying colors onto the canvas.

During the rendering process, each stalk is assigned a different degree of shading, ranging from 0.6 to 1.0. The number determines the opacity of the sprites applied to that bamboo.

```
def composite(c1, c2, shade):
    """ alpha-composite color c1 and c2 """
    a1 = c1[3] * shade / 255
    a2 = c2[3] / 255
    r = alpha_composite(c1[0], c2[0], a1, a2)
    g = alpha_composite(c1[1], c2[1], a1, a2)
    b = alpha_composite(c1[2], c2[2], a1, a2)
    a = a1 + a2 * (1 - a1)
    a = int(a * 255)
    return [r, g, b, a]

def alpha_composite(ch1, ch2, a1, a2):
    """ alpha-composite on chanel c1 and c2 """
    return int((ch1 * a1 + ch2 * a2 * (1 - a1)) / (a1 + a2 * (1 - a1)))
```

In the end, the color of pixel $(x, y)$ on the canvas is obtained by

Figure 3.6: Seal

```
    self.pixels[y][x] = composite(bilinear_sample(new_coord, sprite), \
        self.pixels[y][x], shade)
```

## 3.4 Signature Seal

Chinese painters usually carve their name onto a stamp and seal it at the corner of the painting once it is completed to represent their signatures. A seal with my Chinese name was also applied to the image after the rendering. The Chinese characters are in ancient fonts and read: "Signed by Yangyang He". It serves as a signature as well as enhances the overall visual effect. The seal is pre-produced in Adobe Photoshop, and is applied to a fixed position at $180 \times 180$ pixels from the bottom-left or bottom-right corner.

## 3.5 Output

During the generation process, the bamboo grows towards the positive direction of the y-axis, assuming that the origin of the coordinate system is in the lower-left corner. The origin of an image coordinate system, however, is at the upper-left corner. Thus, it is necessary to flip the canvas up-side-down before writing the color information into an image file.

The rendered image can be saved as a PNG file using the `imageio` library. Depending on the complexity of the bamboo model and the size of the canvas, it may take several minutes to render the image. Because the sizes of the models and the sprites are independent of the size of the canvas, the model could be rendered in various sizes and proportions. To obtain an appropriate field of view, the images were rendered as portrait (vertical) images in a resolution of $480 \times 720$ and landscape (horizontal) images in $1160 \times 720$.

# Chapter 4

# Results

Two PNG images are generated after each run of the model. The first shows the structure of the bamboo, and the second is the rendered image. The structure data of the bamboo models can also be saved in Python lists and dictionaries that can be written into a JSON (JavaScript Object Notation) file for later access or re-rendering.

## 4.1 Structure Graph

The structure graph shows the skeletons of the bamboos' structures. Figure 4.1(a), 4.2(a), 4.3(a) are some examples of structure graphs. In the picture, black lines represent stalk segments, blue lines represent branches, and green lines represent leaves. The crosses represent joints between different elements.

## 4.2 Rendered Image

The rendering result is written into PNG image files using the `imageio` library. Figure 4.1(b), 4.2(b), 4.3(b) are rendered images of the structures shown before them. Note that Figure 4.1(a) is a vertical image of resolution $480 \times 720$, while the other two images

(a) Structure graph of Figure 4.1(b) (b) Rendered image of Figure 4.1(a)

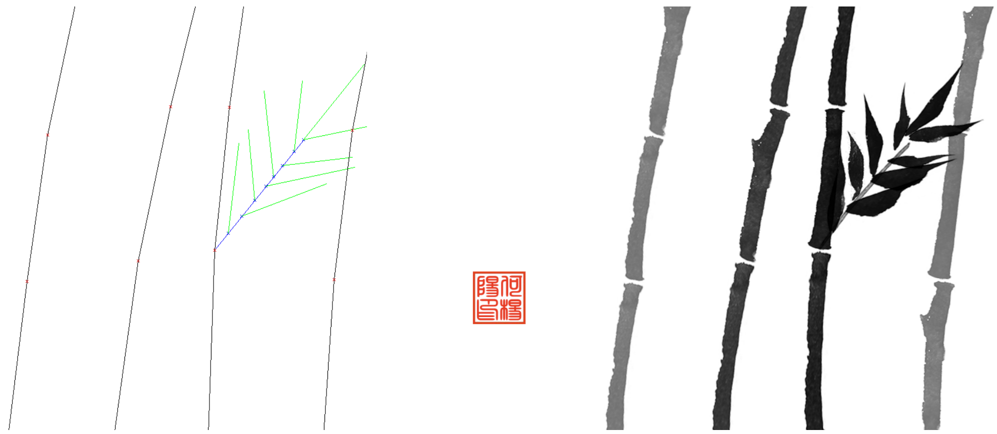
Figure 4.1: Sample rendered images

are of size 1160 × 720 pixels.

## 4.3 Structure Data

Each bamboo has three categories of elements: stalks, branches, and leaves. Each element is represented by a line segment, which has an origin, a direction, and a length. Thus, each category is a collection of line segments, which can be put into a Python list. The structural data of a single bamboo can be stored in a Python dictionary with three entries. The keys of the entries are the lists of line segments, and a constructed scene is just a list of such dictionaries.

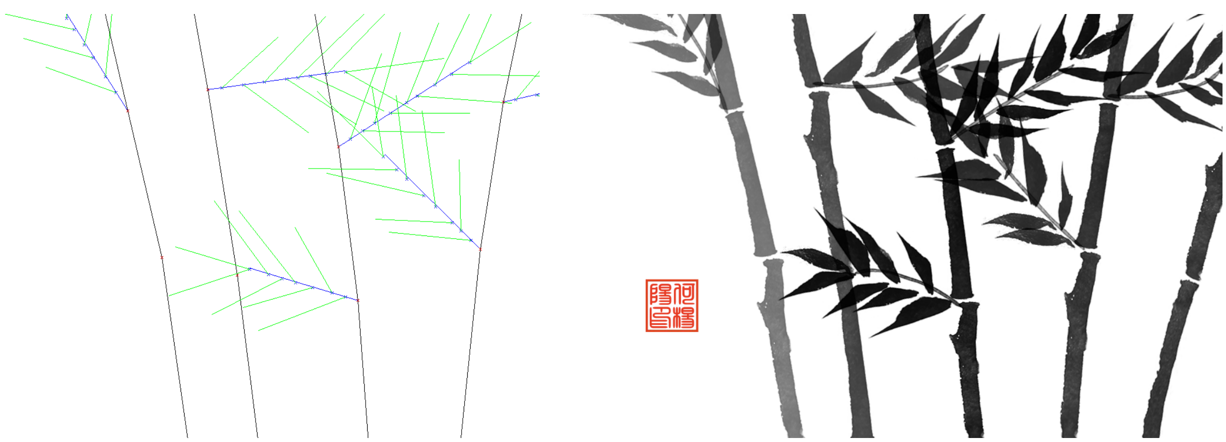
These structural data can be stored in a JSON file for future access. The user can later use this file to re-render the same scene (same structures of bamboos) with different shades of ink, different seals, or other different render parameters.



(a) Structure graph of Figure 4.2(b)

(b) Rendered image of Figure 4.2(a)

Figure 4.2: Sample rendered images



(a) Structure graph of Figure 4.3(b)

(b) Rendered image of Figure 4.3(a)

Figure 4.3: Sample rendered images

## 4.4 Performance Analysis

The time needed to produce a single image depends largely on the complexity of the structure (number of stalks, branches, and leaves) that was generated. Over 60 test runs, the time consumed for each image varied from a couple of minutes to over ten minutes. More testings show that the majority of the time was spent on the rendering process. For example, in a case with 2 stalks (6 segments in total), 2 branches, and 19 leaves, the generation process took less than 0.001 seconds, while the rendering took 445 seconds.

The reverse texture mapping is the most time consuming component in this model. The algorithm had to transform the coordinates of each pixels on the canvas to the coordinate system of the texture for each segment to be mapped. This means that for an $1160 \times 720$ image, more than 800K transformations were performed for each segment to be mapped. The limited performance of the Python programming language also reduced the speed of rendering. As section 4.6 would discuss, the optimization of this algorithm would significantly improve the performance of the model.

## 4.5 Applications

The model presented here could be used for application where basic ink artwork is required to reduce the cost of human labor. Because of the randomness during the construction and rendering process, each run of the model would produce different images. The advantage of the model is that it can produce a large number of unique works in a comparatively short amount of time to that of a human artist. For example, the output of Figure 4.1(a) can be used as the background design of letter paper. As the model is able to produce images of a variety of sizes and shapes, other usages may include: wall decorations, gifts, illustrations, etc.

## 4.6 Future Work

As discussed in section 4.4, currently, the model takes several minutes to render one image, depending on the quantity of segments in the scene. The reverse mapping algorithm could be further optimized to save rendering time, if methods like bounding boxes were used to reduce the number of unnecessary computations.

Altering the effect of different degrees of shades by changing the alpha channel of the image when compositing is not ideal. Since the structure of the bamboo consists of directed line segments, such line segments can be used as the trajectory guidance of an ink brush. A more realistic render could then be done by simulating a virtual ink brush model that paints in different weights and speed along the generated trajectory. Yang et al. [10] developed a model that could reproduce the writing process of calligraphic artwork; similar method could be used in this model to produce dynamic simulation of the brushwork. This would create animations of the painting process, which can be used for the training of entry level painters.

# Chapter 5

# Conclusions

This paper proposes a procedural model that generates various ink bamboo paintings. The model chooses random horizontal positions for the bamboo roots. Bamboo segments then grow out of the roots in a recursive manner: First, a stalk segment grows. If there is a branch at the top joint of the segment, then a branch segment grows. A leaf grows on top of the branch before another branch segment grows. When a branch is constructed, the growth of the stalk continues. After a whole bamboo is completed, another bamboo is generated following a similar procedure until the randomly determined quantity is reached. Each random parameter in the model is controlled by rules observed through real ink bamboo paintings. The growth rule of the leaves simulates the phyllotaxy (arrangement of leaves) of bamboo.

Stalk, branch, and leaf sprites are created from Natali Myasnikova's artwork. They are applied to the canvas using a reverse mapping method and bilinear interpolation. This approach reduces the aliasing effect that forward mapping might cause. Sprites of different bamboo are applied with different degrees of shades, which is controlled by the alpha channel when compositing the sprites and the images. Finally, a Chinese seal is stamped to enhance the visual effect and serve as a signature.

# References


[1] CHU, N., AND TAI, C. L. 2005. Moxi:real-time ink dispersion in absorbent paper. ACM Transactions on Graphics 24, 3 (Aug.), 504511.

[2] HUANG, S. W., WAY, D. L., AND SHIH, Z. C. 2003. Physical- based model of ink diffusion in Chinese ink paintings. Proceedings of the 11th International Conference in Central Europe on Computer Graphics,Visualization and Computer Vision, 520 527.

[3] LI, X. X., AND YU, L. 2006. Simulation of Chinese ink-wash painting based on landscapes and trees. In: proceedings of Second International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications, 328333.

[4] WANG, C. M., AND WANG, R. J. 2007. Image-based color ink dif- fusion rendering. IEEE Transactions on Visualization and Com- puter Graphic 13, 235246.

[5] WANG, Y., WANG, L., Hao W., Ning X., Shi, Z., Zhao M. 2014. A Stylization Method of Chinese Ink Painting for 3D Tree. VRCAI 2014.

[6] WAY, D. L., LAY, Y. R., AND SHIH, Z. C. 2002. The synthesis of trees in chinese landscape painting using silhouette and texture strokes. Journal of WSCG, 10, 3, 499507.

[7] XIE, N., HAMID, L., SUGURU, S., AND MASAYUKI, N. 2011. Contour-driven sumi-e rendering of real photos. Computers and Graphics, 35, 1, 122134.

[8] XU, S., TANG, M., AND LAU, F. 2002. A solid model based virtual hairy brush. Computer Graphics Forum, 21, 3, 299308.

[9] YANG, I. S., YU, Y. J., AND LEE, D. H. 2012. Ink-and-wash painting based on the image of pine tree using mean curvature flow. In: Proceedings of the 11th ACM SIGGRAPH Interna- tional Conference on Virtual-Reality Continuum and its Appli- cations in Industry, 189194.

[10] Yang, L. J., Xu, T. C., Li, X. S., Wu, E. H. Feature-oriented Writing Process Reproduction of Chinese Calligraphic Artwork. SIGGRAPH Asia 2014.

[11] YU, Y., LEE, Y. B., AND CHO, H. G. 2002. A model based technique for realistic oriental painting. In Proceedings of Pacific Graphics, Beijing,, 452453.

[12] YU, J. H., LUO, G. M., AND PENG, Q. S. 2003. Image-based synthesis of chinese landscape painting. Computer Science and Technology, 18, 1, 2228.

[13] ZHANG, Q., SATO, Y., TAKAHASHI, J., MURAOKA, K., AND CHIBA, N. 1999. Simple cellular automaton-based simulation of ink behavior and its application to suibokuga-like 3d rendering of trees. Journal of Visualization and Computer Animation, 10, 2737.

[14] Bamboo Painting. Wikipedia, Wikimedia Foundation, 6 Mar. 2018.

# Appendices

# Appendix A

# Code listings

main.py

```
import random
import time
import imageio

from Painter import Painter
from Canvas import Canvas
from Vec2d import Vec2d
from Stalk import Stalk

IMAGE_WIDTH = 1160
IMAGE_HEIGHT = 720


def main():
    # Initializations
    timestamp = str(int(time.time()))
    filename1 = "./sample_output/" + timestamp + "_sturcture.png"
    filename2 = "./sample_output/" + timestamp + "_render.png"
    canvas_struct = Canvas(IMAGE_WIDTH, IMAGE_HEIGHT)
    canvas_render = Canvas(IMAGE_WIDTH, IMAGE_HEIGHT)
    png_painter = Painter()

    # Read sprites
    stalk_sprite = [imageio.imread('./sprite/stalk-1.png'), imageio.imread('./
        sprite/stalk-2.png'), imageio.imread('./sprite/stalk-3.png')]
    branch_sprite = [imageio.imread('./sprite/branch-1.png'), imageio.imread('
```

```
    ./sprite/branch-2.png')]
leaf_sprite = [imageio.imread('./sprite/leaf-1.png'), imageio.imread('./sprite/leaf-2.png'), imageio.imread('./sprite/leaf-3.png'), imageio.imread('./sprite/leaf-4.png'), imageio.imread('./sprite/leaf-5.png')]
seal_sprite = imageio.imread('./sprite/seal-90.png')

print("Constructing...")
t0 = time.clock()

# start construction
quantity = get_bamboo_quantity()
newest_root = get_root_position(IMAGE_WIDTH)
roots = list()
roots.append(newest_root)
for i in range(quantity - 1):
    newest_root = get_new_root(newest_root.x)
    roots.append(newest_root)

stalks = list()
for root in roots:
    length = get_segment_count()
    new_stalk = Stalk(root, length)
    for j in range(length):
        new_stalk.grow()
    stalks.append(new_stalk)

t1 = time.clock() - t0
print("Construction time", t1, "seconds.")

# start rendering
stalk_count = 1
stalk_total = len(stalks)
for stalk in stalks:
    print("Rendering stalk", stalk_count, "/", stalk_total)
    shade = get_shade_degree()
    for seg in stalk.segments:
        canvas_struct.paint_seg_black(seg)
        canvas_render.paint_seg_sprite(seg, stalk_sprite[get_stalk_sprite_index()], shade)

    for branch in stalk.branches:
        seg = branch.get_segment()
        canvas_struct.paint_seg_blue(seg)
        canvas_render.paint_seg_sprite(seg, branch_sprite[
```

```
                get_branch_sprite_index()], shade)
            for joint in branch.joints:
                canvas_struct.paint_joint_blue(joint)
            for leaf in branch.leaves:
                canvas_struct.paint_seg_green(leaf)
                canvas_render.paint_seg_sprite(leaf, leaf_sprite[
                    get_leaf_sprite_index()], shade)
        for joint in stalk.joints:
            canvas_struct.paint_joint_red(joint)
        stalk_count += 1

    canvas_render.paint_seal(seal_sprite)

    print("Rendering time", time.clock() - t1, "seconds.")
    print("Saving images...")
    png_painter.paint_png(canvas_struct, filename1)
    png_painter.paint_png(canvas_render, filename2)


def get_bamboo_quantity():
    i = random.randint(1, 100)
    if i <= 6:
        return 1
    elif 6 < i <= 44:
        return 2
    elif 44 < i <= 74:
        return 3
    elif 74 < i <= 88:
        return 4
    elif 88 < i <= 100:
        return 5


def get_segment_count():
    return random.randint(3, 4)


def get_root_position(width):
    return Vec2d(random.randint(int(0.33 * width), int(0.5 * width)), -10)


def get_new_root(root):
    return Vec2d(root + random.randint(100, 200), -10)

```

```python

def get_stalk_sprite_index():
    return random.randint(0, 2)


def get_branch_sprite_index():
    return random.randint(0, 1)


def get_leaf_sprite_index():
    return random.randint(0, 4)


def get_shade_degree():
    return random.choice([1.0, 0.9, 0.8, 0.6])


main()
```

Branch.py

```python
import random
from Segment import Segment
from Vec2d import Vec2d
from Joint import Joint


class Branch:

    def __init__(self, root, side):
        self.root = root
        self.top = Joint(root.position.x, root.position.y, self)
        self.segments = list()
        self.joints = list()
        self.leaves = list()
        self.count = 0
        self.direction = Vec2d(0, 1)
        self.leaf_side = 1
        self.grow_first_seg(side)

    def grow(self):
        seg_length = random_seg_length()
        leaf_length = random_leaf_length()
        new_seg = Segment(self.top.position, self.direction, seg_length)
        self.top = Joint(new_seg.get_end().x, new_seg.get_end().y, new_seg)

        # grow one leaf to each side at the joint
        leaf_angle = random_leaf_angle()
        leaf = Segment(self.top.position, self.direction.static_rotate(leaf_angle * self.leaf_side), leaf_length)
        self.leaf_side *= -1
        # right_leaf = Segment(self.top.position, self.direction.static_rotate(-leaf_angle), leaf_length)
        self.top.l_branch = leaf
        # self.top.r_branch = right_leaf
        self.leaves.append(leaf)
        # self.leaves.append(right_leaf)

        self.joints.append(self.top)
        self.segments.append(new_seg)
        self.count = len(self.segments)

    def grow_first_seg(self, side):
```

```python
        seg_length = random_seg_length()
        leaf_length = random_leaf_length()
        seg_angle = random_branch_angle() * side
        new_seg = Segment(self.top.position, self.direction.rotate(seg_angle), seg_length)
        self.top = Joint(new_seg.get_end().x, new_seg.get_end().y, new_seg)

        # grow one leaf to each side at the joint
        leaf_angle = random_leaf_angle()
        leaf = Segment(self.top.position, self.direction.static_rotate(leaf_angle * self.leaf_side), leaf_length)
        self.leaf_side *= -1
        # right_leaf = Segment(self.top.position, self.direction.static_rotate(-leaf_angle), leaf_length)
        self.top.l_branch = leaf
        # self.top.r_branch = right_leaf
        self.leaves.append(leaf)
        # self.leaves.append(right_leaf)

        self.joints.append(self.top)
        self.segments.append(new_seg)
        self.count = len(self.segments)

    def grow_tip_leaf(self):
        leaf_length = random_leaf_length()
        leaf = Segment(self.top.position, self.direction, leaf_length)
        self.top.l_branch = leaf
        self.leaves.append(leaf)

    def get_segment(self):
        length = (self.segments[len(self.segments)-1].get_end() - self.root.position).length()
        return Segment(self.root.position, self.direction, length)


def random_branch_angle():
    return random.randint(30, 85)


def random_leaf_angle():
    return random.randint(30, 45)


def random_leaf_length():
```

```python
    return random.randint(120, 170)


def random_seg_length():
    return random.randint(20, 40)
```

Canvas.py

```
import numpy as np
import math
from Vec2d import Vec2d


SEAL_X = 180
SEAL_Y = 180


class Canvas:

    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.pixels = np.ndarray(shape=(height, width, 4), dtype=np.uint8)
        for i in range(height):
            for j in range(width):
                self.pixels[i][j].fill(255)
        # print(self.pixels[0][0])

    def paint_pixel(self, x, y, r, g, b, a):
        if x in range(0, self.width) and y in range(0, self.height):
            self.pixels[y][x] = [r, g, b, a]

    def paint_pixel_black(self, x, y):
        self.paint_pixel(x, y, 0, 0, 0, 255)

    def paint_pixel_red(self, x, y):
        self.paint_pixel(x, y, 255, 0, 0, 255)

    def paint_pixel_blue(self, x, y):
        self.paint_pixel(x, y, 0, 0, 255, 255)

    def paint_seg(self, seg, r, g, b, a):
        x0 = seg.origin.x
        y0 = seg.origin.y
        x1 = seg.get_end().x
        y1 = seg.get_end().y
        dx = abs(x1 - x0)
        dy = abs(y1 - y0)
        x, y = x0, y0
        sx = -1 if x0 > x1 else 1
```

```
        sy = -1 if y0 > y1 else 1
        if dx > dy:
            err = dx / 2.0
            while x != x1:
                self.paint_pixel(x, y, r, g, b, a)
                err -= dy
                if err < 0:
                    y += sy
                    err += dx
                x += sx
        else:
            err = dy / 2.0
            while y != y1:
                self.paint_pixel(x, y, r, g, b, a)
                err -= dx
                if err < 0:
                    x += sx
                    err += dy
                y += sy
        self.paint_pixel(x, y, r, g, b, a)

    def paint_seg_black(self, seg):
        self.paint_seg(seg, 0, 0, 0, 255)

    def paint_seg_green(self, seg):
        self.paint_seg(seg, 0, 255, 0, 255)

    def paint_seg_blue(self, seg):
        self.paint_seg(seg, 0, 0, 255, 255)

    def paint_joint(self, joint, r, g, b, a):
        x = joint.position.x
        y = joint.position.y
        self.paint_pixel(x, y, r, g, b, a)
        self.paint_pixel(x+1, y+1, r, g, b, a)
        self.paint_pixel(x+1, y-1, r, g, b, a)
        self.paint_pixel(x+2, y+2, r, g, b, a)
        self.paint_pixel(x+2, y-2, r, g, b, a)
        self.paint_pixel(x-1, y+1, r, g, b, a)
        self.paint_pixel(x-1, y-1, r, g, b, a)
        self.paint_pixel(x-2, y+2, r, g, b, a)
        self.paint_pixel(x-2, y-2, r, g, b, a)

    def paint_joint_red(self, joint):
```

```python
        self.paint_joint(joint, 255, 0, 0, 255)

    def paint_joint_green(self, joint):
        self.paint_joint(joint, 0, 255, 0, 255)

    def paint_joint_blue(self, joint):
        self.paint_joint(joint, 0, 0, 255, 255)

    def paint_seg_sprite(self, seg, sprite, shade):
        x_bound = sprite.shape[1] - 1
        y_bound = sprite.shape[0] - 1
        for y in range(self.height):
            for x in range(self.width):
                # transform every pixel to the coord of the sprite
                new_coord = rotate_to_align(seg.direction, Vec2d(0, 1), seg.origin, Vec2d(x, y))
                # new_coord = stretch(new_coord, seg, sprite.shape[0]/abs(seg.origin.y - seg.get_end().y))
                new_coord = stretch_y(new_coord, seg, sprite.shape[0] / seg.length)
                # new_coord = stretch_x(new_coord, seg, 0.4)
                new_coord.x -= seg.origin.x - sprite.shape[1] / 2
                new_coord.y -= seg.origin.y

                # bilinearly sample color and do alpha-composition
                if new_coord.x < x_bound and new_coord.x > 0 and new_coord.y < y_bound and new_coord.y > 0:
                    self.pixels[y][x] = composite(bilinear_sample(new_coord, sprite), self.pixels[y][x], shade)

    def paint_seal(self, sprite):
        x = SEAL_X
        y = SEAL_Y
        for q in range(90):
            for p in range(90):
                self.pixels[y + q][x + p][0] = sprite[89 - q][p][0]
                self.pixels[y + q][x + p][1] = sprite[89 - q][p][1]
                self.pixels[y + q][x + p][2] = sprite[89 - q][p][2]


def rotate_to_align(v1, v2, origin, point):
    """ rotate point around origin so that v1 aligns to v2 """
    sin = v1.sine_angle(v2)
    cos = v1.cosine_angle(v2)
```

```python
    p = point - origin
    result = np.matmul([[cos, -sin], [sin, cos]], [p.x, p.y])
    return Vec2d(result[0], result[1]) + origin


def stretch_y(point, seg, scale):
    dy = (point.y - seg.origin.y) * scale
    y = seg.origin.y + dy
    return Vec2d(point.x, y)


def stretch_x(point, seg, scale):
    dx = (point.x - seg.origin.x) * scale
    x = seg.origin.x + dx
    return Vec2d(x, point.y)


def composite(c1, c2, shade):
    """ alpha-composite color c1 and c2 """
    a1 = c1[3] * shade / 255
    a2 = c2[3] / 255
    r = alpha_composite(c1[0], c2[0], a1, a2)
    g = alpha_composite(c1[1], c2[1], a1, a2)
    b = alpha_composite(c1[2], c2[2], a1, a2)
    a = a1 + a2 * (1 - a1)
    a = int(a * 255)
    return [r, g, b, a]


def alpha_composite(ch1, ch2, a1, a2):
    """ alpha-composite on chanel c1 and c2 """
    return int((ch1 * a1 + ch2 * a2 * (1 - a1)) / (a1 + a2 * (1 - a1)))


def bilinear_sample(point, sprite):
    """ get the target color using bilinear interpolation """
    x1 = math.floor(point.x)
    x2 = x1 + 1
    y1 = math.floor(point.y)
    y2 = y1 + 1
    dx = point.x - x1
    dy = point.y - y1
    a = sprite[y1][x1] * (1 - dx) + sprite[y1][x2] * dx
    b = sprite[y2][x1] * (1 - dx) + sprite[y2][x2] * dx
```

```
    result = a * (1 - dy) + b * dy
    return result
```

Joint.py

```
from Vec2d import Vec2d


class Joint:

    def __init__(self, x, y, parent):
        self.position = Vec2d(x, y)
        self.parent = parent
        self.stalk = None
        self.l_branch = None
        self.r_branch = None
```

Painter.py

```python
import imageio

class Painter:

    def __init__(self):
        pass

    def paint_p3(self, canvas, filename):
        file = open(filename, 'w')
        file.write("P3\n")
        file.write(str(canvas.width) + " " + str(canvas.height) + "\n")
        file.write(str(255) + "\n")
        for i in range(canvas.height - 1, -1, -1):
            for j in range(canvas.width):
                file.write(str(canvas.pixels[i][j][0]) + " ")
                file.write(str(canvas.pixels[i][j][1]) + " ")
                file.write(str(canvas.pixels[i][j][2]) + " ")

        file.close()

    def paint_p6(self, canvas, filename):
        file = open(filename, 'w')
        file.write("P6\n")
        file.write(str(canvas.width) + " " + str(canvas.height) + "\n")
        file.write(str(255) + "\n")
        for i in range(canvas.height - 1, -1, -1):
            for j in range(canvas.width):
                file.write("%B" % canvas.pixels[i][j][0])
                file.write("%B" % canvas.pixels[i][j][1])
                file.write("%B" % canvas.pixels[i][j][2])
            file.write("\n")

        file.close()

    def paint_png(self, canvas, filename):
        # Reflect the image vertically
        for i in range(canvas.height // 2):
            canvas.pixels[[i, canvas.height - 1 - i]] = canvas.pixels[[canvas.height - 1 - i, i]]
        imageio.imwrite(filename, canvas.pixels)
```

Segment.py

```python
from Vec2d import Vec2d


class Segment:
    """Line segment class"""

    def __init__ (self, origin, direction, length):
        self.origin = Vec2d(origin.x, origin.y)
        self.direction = Vec2d(direction.x, direction.y).normalize()
        self.length = length

    def get_end(self):
        return Vec2d(int(self.origin.x + self.direction.x * self.length), int(
            self.origin.y + self.direction.y * self.length))
```

Stalk.py

```python
import random
from Segment import Segment
from Vec2d import Vec2d
from Joint import Joint
from Branch import Branch

class Stalk:

    def __init__(self, root, length):
        self.root = Joint(root.x, root.y, self)
        self.top = Joint(root.x, root.y, self)
        self.segments = list()
        self.joints = list()
        self.branches = list()
        self.count = 0
        self.bend_side = random.choice([-1, 1])
        self.direction = Vec2d(0, 1).rotate(get_initial_angle() * self.bend_side)
        self.length = length
        self.branch_side = random.choice([-1, 1])

    def grow(self):
        rand_length = get_segment_length()
        rand_angle = get_random_angle()
        new_seg = Segment(self.top.position, self.direction.rotate(rand_angle * self.bend_side), rand_length)
        self.top = Joint(new_seg.get_end().x, new_seg.get_end().y, new_seg)
        if self.decide_grow_branch():
            new_branch = Branch(self.top, self.branch_side)
            self.branch_side *= -1
            for j in range(get_segment_count()):
                new_branch.grow()
            new_branch.grow_tip_leaf()
            self.top.l_branch = new_branch
            self.branches.append(new_branch)
        self.joints.append(self.top)
        self.segments.append(new_seg)
        self.count = len(self.segments)

    def decide_grow_branch(self):
        if len(self.segments) + 1 == self.length:
            return False
```

```
        i = random.randint(1, 100)
        if i <= 45:
            return True
        else:
            return False


def get_segment_count():
    return random.randint(6, 10)


def get_segment_length():
    return random.randint(242, 335)


def get_random_angle():
    return random.randint(0, 5)


def get_initial_angle():
    return random.randint(0, 10)
```

Vec2d.py

```python
import math

class Vec2d:
    """2D Vector/Point class"""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def normalize(self):
        self.x = self.x / self.length()
        self.y = self.y / self.length()
        return self

    def rotate(self, angle):
        rad = angle * math.pi / 180
        x = self.x * math.cos(rad) - self.y * math.sin(rad)
        y = self.x * math.sin(rad) + self.y * math.cos(rad)
        self.x = x
        self.y = y
        return self

    def static_rotate(self, angle):
        rad = angle * math.pi / 180
        x = self.x * math.cos(rad) - self.y * math.sin(rad)
        y = self.x * math.sin(rad) + self.y * math.cos(rad)
        return Vec2d(x, y)

    def sqaured_length(self):
        return self.x * self.x + self.y * self.y

    def length(self):
        return math.sqrt(self.sqaured_length())

    def dot(self, v2):
        return self.x * v2.x + self.y * v2.y

    def cosine_angle(self, v2):
        return self.dot(v2) / (self.length() * v2.length())

    def sine_angle(self, v2):
        return self.x * v2.y - v2.x * self.y
```

```python

    # override methods
    def __add__(self, other):
        return Vec2d(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vec2d(self.x - other.x, self.y - other.y)
```

make_sprite.py

```python
import imageio
import sys


def main():
    filename = sys.argv[1]
    print(filename)
    px = imageio.imread(filename)
    for y in range(px.shape[0]):
        for x in range(px.shape[1]):
            shade = px[y][x][0]
            px[y][x] = [0, 0, 0, 255 - shade]
    imageio.imwrite("out" + filename, px)


main()
```