An Investigation on Reaction-Diffusion-Based Mazes

Jonathan DL. Casano
Computer Science Department
Ateneo de Naga University
jonathancasano@gmail.com

Allan A. Sioson
Computer Science Department
Ateneo de Naga University
allan@adnu.edu.ph

ABSTRACT
In this work, we investigate the RD based maze generation algorithm of Wan et al. [3] This algorithm starts with any image with enough detail and produces a maze-like image which preserves the prominent features of the input image. As a proof of concept, we developed an application where users can specify two points that defines a path. The application is largely based on the algorithm of Wan et al. and is subjected to a set of test input images. The application is developed in C++ using the OpenCV library.

Categories and Subject Descriptors
[**Image Processing and Computer Vision**]: Image Transformation—*Neighborhood Convolution*

General Terms
Algorithms, Performance, Design, Experimentation

Keywords
Reaction-Diffusion, Connected Components, Neighborhood Convolution

1. INTRODUCTION
In the work of Alan Turing entitled, *T*he Chemical Basis of Morphogenesis , he defined Reaction-Diffusion as a phenomenon that happens when two chemical concentrations meet each other, and as the name suggests, begin a countinuos process of reaction (with each other) and diffusion through space (as caused by the reaction) until equilibrium is reached [1]. This is predominantly accepted and agreed upon by biologists today, as what is behind some of the spontaneous patterns that occur in nature such as, botanical structures, spots and stripes formations in animals like giraffes, cats and zebras [3]. Studies in this field found a specific reaction-diffusion variation that results to an organic apperance similar to that of mazes as we know them today. This paper is about controlling the number of iterations which affect the formation of these labyrinth-like patterns resultion from simulating a reaction-diffusion phenomenon in order to come up with a functional maze, with the aim of preserving whatever salient featues the source images have from which the mazes would be built from. This work aims to automatically generate mazes out of pictures and photographs.

2. RELATED STUDIES
This work is largely based on the work of Wan et al. [3] In particular, we implemented the algorithm they described in their *Evolving Mazes from Images* paper based on our interpretation of the mathematics involved.. In their work, they were able to provide a user interface that allows the user to specify two points to function as the beginning and end of the maze. a cloning template was specified to function as the filter to evolve the original image and turn it into an image that shows a maze-like pattern. In their research they were able to control how organic the maze would look and how wide the passages would appear to be and how thick the walls would be rendered. The same GUI offered paint tools with which users could specify areas in the original image that they would like to evolve in a certain manner, i.e. wall brush, passage brush, organic brush. The user will simply choose an image to be evolved, use the paint tools to affect the evolution if desired and click another button that produces the maze.

In the paper of Xu and Kaplan [4] , no user interface is offered. Their maze construction method begins with the formation of a grid from vertical and horizontal lines. From the grid, a rectangular maze is constructed. This rectangular maze is then subjected to the relaxation algorthm of Singh and Pedersen which evolves the rectangular maze to become organic. This relaxed image is now the maze.

3. TECHNICAL BACKGROUND
Turing, [1] defined Reaction-diffusion to be a process in which two or more chemicals diffuse over a surface and react with one another to produce stable patterns. Reaction-diffusion can produce a variety of spot and stripe patterns, much like those found on many animals such as zebras, cougars, and tigers. Developmental biologists think that some of the patterns found in nature may be the result of reaction-diffusion processes like perhaps the formation of petals in sunflowers [2]. Turing, in his paper The Chemical Basis of Morphogenesis [1], stated that a system of chemical substances, called morphogens, reacting together and diffusing through a tissue, is adequate to account for the

main phenomena of morphogenesis. Morphogenesis is the formation of the structure of an organism that involves the differentiation and growth of tissues and organs during development. It's the process responsible for differentiating the cells to become body parts. Wan et al. [3] used this concept of Turing and utilized Reaction-Diffusion as a basis for pattern generation.

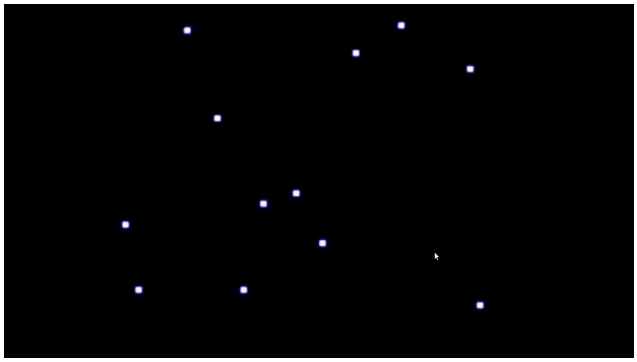


Figure 1: representation of unstable morphogens in a reaction diffusion system

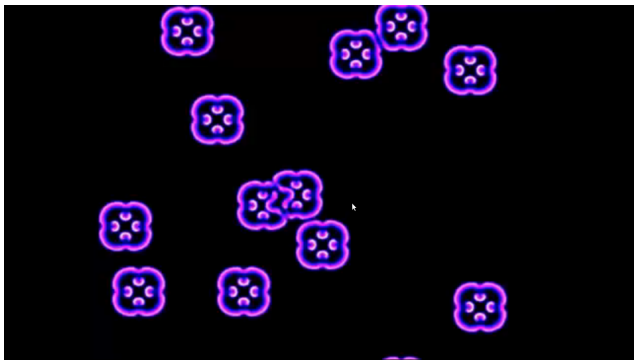


Figure 2: representation of morphogens in the same RD system reacting with themselves

Figures 1 to 5 are a visual representations that seek to explain the concept of Reaction- Diffusion. Each dot in figure 1 represents an unstable morphogen that innately would react with itself and upon reacting with itself, would cause diffusion to happen (figure 2) This constant process of reaction and diffusion, would make the morphogens expand through the available space (figure 3) seeking 'equilibrium', or a state in which it could no longer react and diffuse due to the fact that the simultaneous reactions and diffusions happening around the morphogen in concern does not anymore allow it to. It is at this point that the reaction-diffusion system is called stable (figure 5). Note that although the system is already at equilibrium, subtle evolutions still occur as caused by the continuous reaction-diffusion happening. This focus on subtle evolution could be observed at figures 3 and 4. It is because of this phenomenon that in generating patterns using a Reaction- Diffusion based technique, a maximum iteration needs to be specified because in theory, a reaction-diffusion system will keep on evolving unless stopped [1].

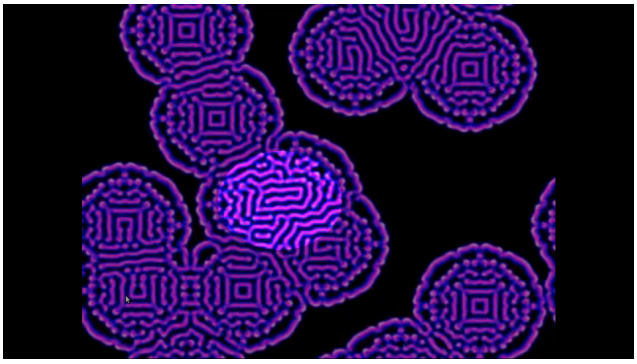


Figure 3: representation of partial equilibrium in an RD system (1)

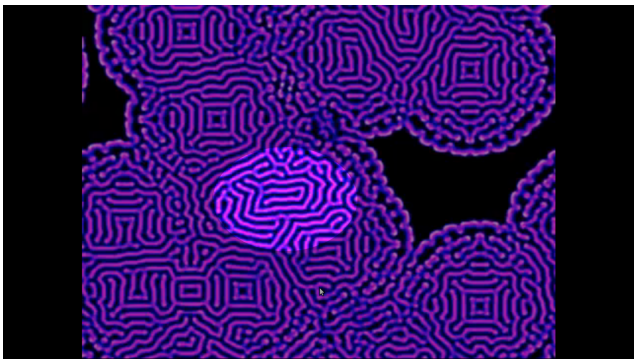


Figure 4: representation of partial equilibrium in an RD system (2) subtle evolutions even when already near equilibrium

4. METHODOLOGY

The maze generation process has five steps based on the work of Wan et al.

4.1 First Step : Addition of extra whitespaces around the edges of the image

white pixel padding `(pixel value 255, IPL depth 8U, 2 channels)` is added onto the sides of the image. This is to make sure that the image will have enough space when the evolution takes place since, as noticed during the testing phase, the image grew in area as the reaction-diffusion algorithm is applied.

In this paper the researchers used the wrapper `RgbImage-Float()` . This allows the pixels of the images to be interpreted as values ranging from 0 to 1, zero being optically interpreted by the human eye to be black and one interpreted to be white.

This program simply adds 90 pixels of whitespace padding (pixels that have the value 1) around the image, to make room for the evolution that would take place come the next step wherein the output image of this program will be passed as input of the next program, processImg.cpp.

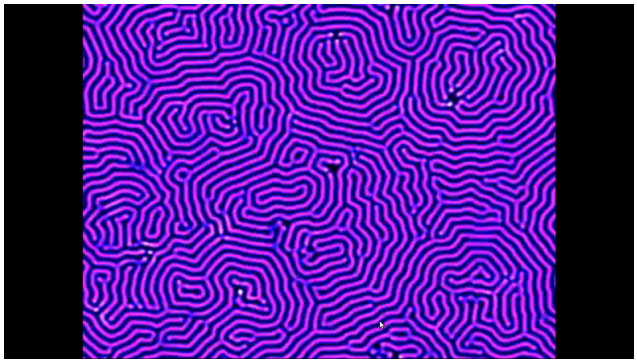


Figure 5: representation of a stable reaction-diffusion system

Algorithm 4.1: ADD PADDING AND REMAP ($originalimage$)

```
for i ← 0 to image−>height
  do for j ← 0 to image−>width
       do { newImage[i+padding][j+padding].r = originalImage[i][j].r/255.0;
            newImage[i+padding][j+padding].g = originalImage[i][j].g/255.0;
            newImage[i+padding][j+padding].b = originalImage[i][j].b/255.0;
```

4.2 Second Step : Applying the Reaction-Diffusion Formula to generate a maze-like image

This step implements the Reaction-Diffusion formula presented by Wan et al. [3] The output of this program is a maze-like pattern which preserves the salient structures of the input image. However, in this program's output no path is generated yet. This program shall be executed twice. The other occurrence being after the third step. This is to ensure that the final output is organic looking, which means that, the lines bridging the connected blobs do not look like plain lines but would evolve as well to blend with the output maze pattern.

Figure 6: Input image with white padding on all sides

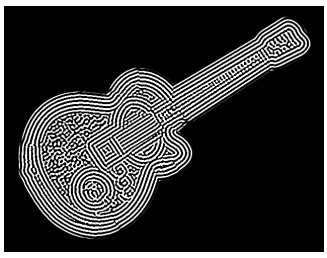


Figure 7: Padded image after being subjected to the RD formula

Algorithm 4.2: GENERATE MAZE-LIKE IMAGE ($paddedimage$)

```
{ load(inputImage)
  grayscaleAndRemap(inputImage)
  generatedImageI(inputImage)
  generateImageX0(inputImage)
  generateImageY0(generatedX0image)
  generateImageYFiltered0(generatedY0image)
  generateImageDeltaX(generatedX0image, generatedY0image)
  for i ← 0 to maxIteration
    do { generateImageY0(imgNextX, imgNextY)
         generateImageYFiltered0()
         generateImageNewDeltaX()
         generateImageNextX()
```

First, the padded image, that came from generating an output using addPadding.cpp, is grayscaled. To generate a grayscale image means to use the values of the red, green, and blue channels that comprise each individual pixel to get the average channel value for that pixel. This average value is then assigned as the new value for the channels of the pixel. Hence, given a pixel whose R, G, B channel values are

```
image[i][j].r = 90; // R channel
image[i][j].g = 100; // G channel
image[i][j].b = 110; // B channel
```

to generate a grayscale version of the pixels means to do the process below

```
image[i][j].r = (image[i][j].r +
image[i][j].g + image[i][j].b) / 3;
image[i][j].g = (image[i][j].r +
image[i][j].g + image[i][j].b) / 3;
image[i][j].b = (image[i][j].r +
image[i][j].g + image[i][j].b) / 3;
```

This process should be applied for all pixels to generate a grayscaled image.

The grayscaled image will then be remapped for the values to fall in the range `[R1 = 0.396, R2 = 0.888]` . This is to ensure that the evolution would happen. The reaction-diffusion theory claims that there is a so called 'search for equilibrium', a pixel being in a pure black (0), or a pure white (1) state is said to be 'already in equilibrium' or 'already stable'. This is the reason why the image needs to be remapped between the range R1 and R2, for them to be in an unstable state (gray) that could search for equilibrium. The search for equilibrium happens during each iteration, how exactly this happens is explained in the succeeding paragraphs.

To understand better the process, the researchers find the need to differentiate IMAGES from STATES. Omission of this explanation would make the paper a difficult read.

An IMAGE is a collection of pixels presented in a two dimentional matrix whose values are within the range that the RgbImageFloat wrapper is made to handle (0 - 1). A STATE, is just like an IMAGE, however, some if not all of the values that it holds, exceed that of the range that the RgbImageFloat wrapper is expected to handle. This differentiation is crucial to note since, STATE pixels, even though they have values that exceed the range are still visibly interpreted. For example,

IMAGE

```
pic[i][j].r = 1.0;
pic[i][j].g = 1.0;
pic[i][j].b = 1.0;
```

produces the color white since RgbImageFloat interprets pixels from 0 - 1.

STATE

```
pic[i][j].r = 35278.2376;
 pic[i][j].g = 23838.23;
pic[i][j].b = 987.232323;
```

This also produces the color white. As it tries to check whenever a pixel is given a value greater than 1 to interpret, it interprets it to be white. On the other hand, the 'significant' values remain unchanged. These out of bounds values, are needed in the succeeding processes and the algorithm will fail when certain STATES are remapped to become IMAGES when they do not need to be.

4.2.1 GENERATING IMAGE X0

To generate image X0 all pixel values are simply multiplied by two, and the pixel values that go beyond 1 are remapped back to 1. This makes the image lighter since doubling the values of the pixels make them closer to white.

Algorithm 4.3: GENERATE IMGX0 ($inputImage$)

$imgHolder \leftarrow cvCreateImage$
$C1 \leftarrow 2$
for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
do { **for** $i \leftarrow 0$ **to** $imgHolder \rightarrow width$
do { $image[i][j].r = pic[i][j].r * C1 - 1.0;$
$image[i][j].g = pic[i][j].g * C1 - 1.0;$
$image[i][j].b = pic[i][j].b * C1 - 1.0;$
return ($imgHolder$)

4.2.2 GENERATING IMAGE I

To generate image I, the graycaled image is simply cloned. This means each pixel that the grayscaled image has is the same set of pixels that image I will have.

Algorithm 4.4: GENERATE IMAGE I($inputImage$)

$imgHolder \leftarrow inputImage$
return ($imgHolder$)

Algorithm 4.5: REMAP IMAGE I ($imageI$)

$imgHolder \leftarrow cvCreateImage$
for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
do { **for** $i \leftarrow 0$ **to** $imgHolder \rightarrow width$
do { $image[i][j].r = pic[i][j].r/1.0 * 2.0 - 1.0;$
$image[i][j].g = pic[i][j].g/1.0 * 2.0 - 1.0;$
$image[i][j].b = pic[i][j].b/1.0 * 2.0 - 1.0;$
return ($imgHolder$)

4.2.3 GENERATING STATE Y

State Y is generated by using this formula

$$Y_{i,j}^{(t)} = \frac{1}{2}[\mid X_{i,j}^{(t)} + 1 \mid - \mid X_{i,j}^{(t)} - 1 \mid]$$

For each pixel, the formula evolves it to be between a value in the range -1 to 1. It takes as input the previous X state and outputs a state Y.

Algorithm 4.6: REMAP IMAGE I ($imageI$)

$imgHolder \leftarrow cvCreateImage$
for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
do { **for** $i \leftarrow 0$ **to** $imgHolder \rightarrow width$
do { $image[i][j].r = pic[i][j].r/1.0 * 2.0 - 1.0;$
$image[i][j].g = pic[i][j].g/1.0 * 2.0 - 1.0;$
$image[i][j].b = pic[i][j].b/1.0 * 2.0 - 1.0;$
return ($imgHolder$)

4.2.4 GENERATING STATE FILTERED Y

To generate the state filtered Y, state Y is masked using the given cloning template. Masking an image means computing a pixels value using its old value and the values of pixels in its vicinity. How wide the vicinity is will depend on the dimensions of the mask used. In this case, the given 5 x 5 cloning template is used.

A mask is a small matrix whose values are called weights. Each mask has an origin, which is usually one of its positions. The origins of symmetric masks (a mask whose dimensions are the same) are usually their center pixel position.

```
float cloningTemplate[5][5] =
{
{-0.025, -1.0, -1.5, -1.0, -0.25},
{-1.0, 2.5,7.0, 2.5,-1.0},
{-1.5, 7.0, -23.5, 7.0, -1.5},
{-1.0, 2.5,7.0, 2.5, -1.0},
{-0.025, -1.0, -1.5, -1.0, -0.25}
}
```

In this paper, the mask is a 5x5 cloning template from the study of Wan et al. The reference paper did not clearly state why such values have been used to comprise the 5x5 mask. The reference study simply stated that, convolution using

the given cloning template proves to be the most ideal to use. Disclosure as to why this is so was not discussed.

Presented below is the formula used by Wan et al. to generate the maze convolution.

$$\frac{dX_{i,j}^{(t)}}{dt} = -X_{i,j}^{(t)} + \sum_{k,l\epsilon\eta(i,j)} a_{k-i,l-j} Y_{k,l}^{(t)} + I_{l,j}$$

For each pixel in the image to be masked, the mask is conceptually placed on top of the image with its origin lying on that pixel. The values of each input image pixel under the mask are multiplied by the values of the corresponding mask weights. The results are summed together to yield a single output value that is placed in the output image at the location of the pixel being processed on the input.

Algorithm 4.7: IMAGE YFILTERED ($imageI$)

for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
  do for $j \leftarrow 0$ **to** $imgHolder \rightarrow width$
  do { **for** $k \leftarrow -2$ **to** 3
    do for $l \leftarrow -2$ **to** 3
    do { **if** $(i+k >= 0 \&\& i+k < img \rightarrow height$
    $\&\& j+1 >= 0 \&\& j+1 < img \rightarrow width)$
    do **then** $filterTemp[i][j] +=$
    $imgHolder[i+k][j+l].r *$
    $cloningTemplate[2+k][2+l]$

In this case, the one to be masked is state Y using the cloning template as the mask to generate state filtered Y.

4.2.5 GENERATING STATE DELTA X

The state delta X is created by adding the pixels of IMAGE I and STATE FILTERED Y to the negative pixels of the previous STATE X in the iteration. Image I, being part of what comprises STATE DELTA X, ensures that the maze evolution will take up the form of the input image. Another way to convey this idea this would be to say that, the salient structures of the input image is continuously fed to the maze making loop as it iterates, making sure that the evolution does not become arbitrary as to not evolve according to the features of the input image.

Algorithm 4.8: IMAGE YFILTERED ($imageI$)

for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
  do for $j \leftarrow 0$ **to** $imgHolder \rightarrow width$
  do { **for** $k \leftarrow -2$ **to** 3
    do for $l \leftarrow -2$ **to** 3
    do { **if** $(i+k >= 0 \&\& i+k < img \rightarrow height$
    $\&\& j+1 >= 0 \&\& j+1 < img \rightarrow width)$
    do **then** $filterTemp[i][j] +=$
    $imgHolder[i+k][j+l].r *$
    $cloningTemplate[2+k][2+l]$

4.2.6 GENERATING SUCCEEDING X STATES

From IMAGE X0, the succeeding images that are to be the basis of the next iterations, are remapped to become states. Specifically, these are remapped to have the values within the range -1 to 1, forming stripe patterns composed of black and white pigments. Suceeding X states are created by adding the corresponding pixels of the change in X (delta X state) to the previous X state. Observe that the pixels of image I which is the grayscaled version of the input image remapped to -1 - 1 is also added to generate the next X state. This is to ensure that the evolution will pattern itself according to the salient structure of the input image.

Algorithm 4.9: IMAGE DELTA X+ ($filteredY, nextX, imgI$)

$imgHolder \leftarrow imgNewDelta$
for $i \leftarrow 0$ **to** $imgHolder \rightarrow height$
  do for $j \leftarrow 0$ **to** $imgHolder \rightarrow width$
  do { $imgNewDelta[i][j].r =$
    $imgNewDelta[i][j].g =$
    $imgNewDelta[i][j].b =$
    $filteredY[i][j].r - nextX[i][j].r$
    $+imgI[i][j].r$
return ($imgHolder$)

4.2.7 GENERATING THE MAZE LIKE IMAGE

To generate the final image means to take the last generated X STATE and use openCV's convertScale() function to translate it into an IPL DEPTH 8U image. The conversion is necessary since openCV does not support the saving of Float images (0 - 1), but does support the saving of int images. Int images being images whose pixel values range from 0 - 255.

4.3 Third Step : Allowing the user to input the start point, end point and guiding curve

This step allows the user to enter two points, a start point and an end point. White lines are drawn between two blobs which belong to the user specified solution curve, whose start point and end point is at the pixels whose distance from each other is the least with respect the the two blobs being processed.

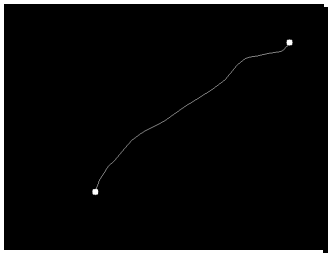


Figure 8: The user specified guiding curve with start and end points

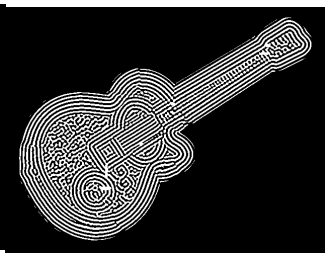


Figure 9: Maze image after specifying guiding curve

This third step overriddes openCv's `findContours()` function. First, upon running this program, a user would be presented a window and would be prompted to sketch a guiding

curve. A guiding curve is a line drawn to specify where the user most likely would want the path for the maze to be in. It is not the actual path, but a guide as to where the path should appear. This line is saved as an image and is used as the basis of the next process, locating the connected components (or blobs) that the guiding curve hit and providing them similar id's. OpenCv's `findContours` function interprets connected components as individual objects that have attributes. Giving the guiding curves id's means simply just checking if a line hits a blob. If it does and it's 'flag' attribute doesn't yet have a value, the program gives it a value. Connecting the blobs with the same id's does NOT mean doing a `connect()` function upon traversing the whole image and finding a blob with an id. This could be done, but the possibility of connecting every single blob in the image is very high. This would defeat the purpose of the guiding curve which is to make sure that the solution path would somehow follow the user specified curve, essentially connecting only those blobs that the guiding curve binds together. The solution would be to utilize pointers, as the findContours() function used pointer to reference blobs from each other. The `findContours()` function stores in a linked-list the blobs the it recognizes from BOTTOM LEFT to UPPER RIGHT. This exposes the problem that 'a blob with an id is not necessarily preceeded or followed by a blob with an id also'. The guiding curve could have stored blobs that are not immediately following each other in the linked-list. The solution used in this paper was to reconstruct where the pointers pointed to. This would mean, for each blob, the program finds the nearest blob and processes the two blobs to locate where their nearest pixels are. A line is then drawn to connect these two lines.

4.4 Fourth Step : Making the maze look organic

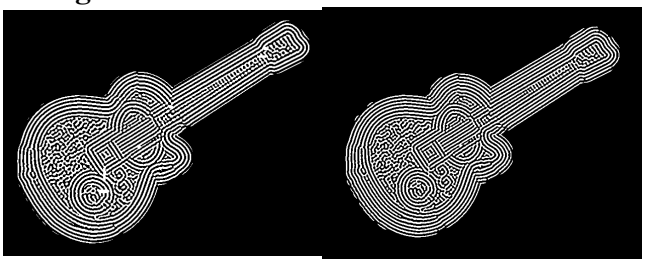


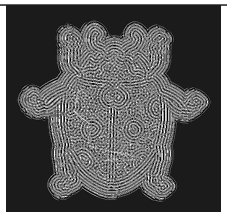
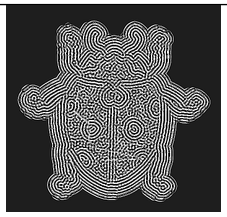
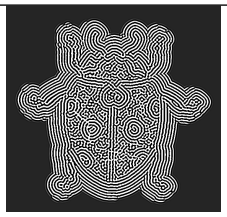
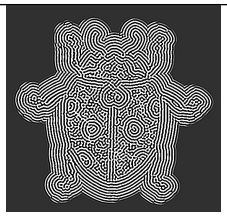
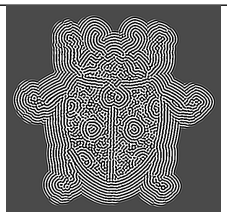
Figure 10: Maze image after specifying guiding curve

Figure 11: Resulting image after added Reaction-Diffusion Iterations

4.4.1 Additional reaction-diffusion Iterations for maintenance of organicity

After attempting to connect the blobs to bridge the start point and end point, it becomes obvious that lines were drawn and that these lines do not blend well with the maze. The fourth step subjects the resulting image to the RD formula again to ensure that the added lines that bridge the two points blend with the maze like pattern (organic). This time though, the iterations are controlled. Upon further evolution, the maze may not fully reach it's state of equilibrium. This may result to gray `(pixel value between 50 and 200, IPL DEPTH 8U, 1 channel)` areas around the image. This will cause errors when the connected blobs are colored to highlight the existence of a solution path as gray will be interpreted to be a connected component (since it is not fully black) and Hence, the process will not be able to produce a maze. Included in this step is the process to make sure that the pixel values `(IPL DEPTH 8U, 1 channel)` for the already organic bridge image would be close to binary, so as to produce an images composed of strong white and black pigmentations whose values the fifth step can already interpret.

Table 1: Added Iterations and Organicity

ADDED ITERATIONS	RESULTING IMAGE
5	
10	
20	
50	
150	

4.5 Fifth Step : Coloring the connected components

The premise is that, if the blob where the start and end points reside are colored with the same random color, then they are connected, hence, there is a functional maze in the generated image.

.This process simply makes use of openCV's `drawContour` function to color at random the connected components. The

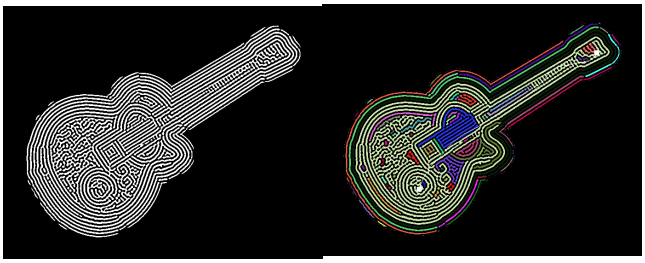


Figure 12: Maze image after added reaction diffusion Iterations

Figure 13: Resulting image after coloring the connected components

function of simply just drawing the contours was overriden to include in the output image the linePath generated by the user during the third step.

`drawContours.cpp` uses openCv's `drawContour()` function to color the connected components of the image.

5. RESULTS AND DISCUSSION

5.1 Program performance during Maze generation

Table 2: Relationship between Image Size, Iteration and Run Time

IMAGE SIZE	ITERATIONS	RUN TIME
440 x 420	20	12 secs
.	100	25 secs
.	400	1:57 secs
.	800	13:44 secs
.	1000	OutOfMemoryException
640 x 480	20	30 secs
.	100	1:12 secs
.	400	3:07 secs
.	800	OutOfMemoryException
.	1000	OutOfMemoryException
1024 x 768	20	3:19 secs
.	100	OutOfMemoryException
.	400	OutOfMemoryException
.	800	OutOfMemoryException
.	1000	OutOfMemoryException

During times when the iteration specified by the user exceeds that of what the processor can handle, it throws an `outOfMemoryException`. The same is true if the user inputs an image whose size the processor cannot handle. see table 2 It is due to this limitation in the program's efficiency that there sometimes would be a need to stop the maze generating process somewhere halfway when the pigments still have not yet reached equilibrium (there still are gray areas). `organicInit.cpp` tries to translate the partially evolved image into an image that seemingly already reached its equilibrium by having a specified range (t is greater than 50 and is less than 200) and making 'white' all the pixels that fall under this range. The specified range tries to capture the gray areas in the image; The areas that are partially evolved. The choice to turn the pixels that fall under the range to be white is to give importance to generating white passages instead of prioritizing the generation of black walls. The image is then passed back to `processImg.cpp` to evolve the image using 20 - 25 iterations to maintain the mazeâĂŹs organicity. During the testing it has been observed that the optimal number of added iterations should be at least twenty and not exceeding 50. This is to balance the need to make the added lines blend in with the maze with the need to not make the maze evolve further. In table 4.2, during the 50th added iteration, a maze patterned padding already developed on the borders of the region of interest. This is not desirable because the goal is to preserve the salient structure of the image, as much as possible the added iterations should limit evolving the image further as it will tend to lose its original salient features, see last image in table 2.

5.2 Final Examples

Figure 14: original image | seal.png

Figure 15: seal.png generated maze

Figure 16: original image | ladybug.png

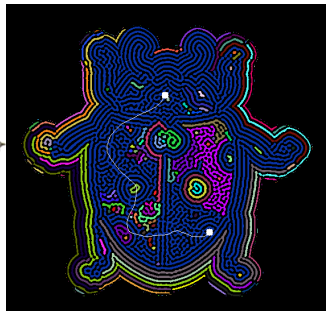


Figure 17: ladybug.png generated maze

Figure 18: original image | logo.png

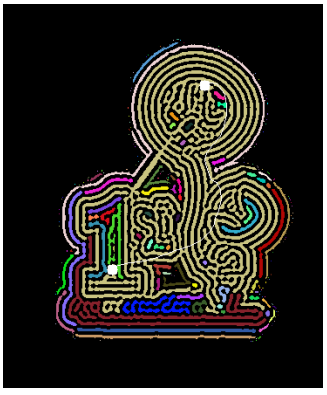


Figure 19: logo.png generated maze

6. FUTURE WORK

1. Apply Maze generation procedure on mobile platforms specifically for mobile games *educational games*.

2. Experiment on feeding the algorithm with parameters to control the wall thickness, passage spacing and pattern organicity. Alternatively, one may create flag images during the evolution as to apply a different filter or do different processing in regions of interest.

7. REFERENCES

[1] Alan M. Turing, *The chemical basis of morphogenesis.* Philosophical Transactions of the Royal Society of London. Series B, *Biological Sciences, 237 (1952), pp. 37-72*

[2] G. Turk, *Generating textures on arbitrary surfaces using reaction-diffusion.* Computer Graphics, *1991, pp. 289-298*

[3] L. Wan, X. Liu, T.-T. Wong, and C.-S. Leung, *Evolving Mazes from Images*, IEEE Transactions on Visualization and Computer Graphics, *16 (2010), pp. 287 - 297*

[4] J. Xu and C.S. Kaplan, *Image-guided maze construction*, ACM Trans Graph, *26 (2007), p. 29*

[5] R. C. Gonzales and R. E. Woods, *Digital Image Processing*, Pearson Prentice Hall, *3rd ed., 2008.*

[6] The OpenCV website. `http://opencv.willowgarage.com/wiki/`.