Assignment 3, Image Processor

ITP - 132. Written by Bryce Summers. Last updated on October 17th, 2020

In this assignment, you will make extensive use of character vectors while implementing 2-dimensional capabilities. The assignment is grouped into three parts (A, B, and C), each with several independent tasks that you can complete. Some are longer / harder than others.

Instructions:

- Make sure you have the starter code file **`IP.cpp`**, that also includes an autotester.
- Pass as many of the auto-testers tests as possible or as close as possible.
- Some tests may give you a "wrong%" score. That means that between you and me, together we are at least that percentage wrong on the correct samples for an image.
- While it would be nice to get 0% wrong on every test, it can be hard to get perfect results due to subtle differences in the way we perform arithmetic operations on floating point numbers. Please just to the best you can and let me know if you think I have a logic bug in my test data.

Definitions

I have #defined the word "Image" to mean "vector<char>", since most of these functions will be presented in the context of image processing.

I have also #defined the word "Bounds" to mean "vector<float>" and you will be using Bounds arrays to store rectangular regions defined by 4 coordinates: [x1, y2, x2, y2] in part B. If it doesn't contain 4 values it is invalid should not be used!

Part A: Image encodings

Encodings are used to convert between 2D data 1D data that can be read and written to a file.

For each Image format (called BIFs[1]), you will write a function that encodes raw image data as a string and a decoder function that converts a valid encoded String back into a raw Image.

Here is the (raw) Image format and the 4 encoded string formats used in this assignment. Each encoded string format includes its encoding of each of the two (decoded) example images.

Name	Specification	Decoded Representation as an Image (vector<char>)
Image (raw)	An image is a vector<char> that stores the character data for an ASCII image in row-major order, with an added newline ('\n') character at the end of each row. <#columns> = index of first '\n'. <#rows> = size() / (<#columns> + 1)	**columns:** 0123456(7*) / 01234(5*) **rows:** 0 **0123456[\n]** / 0 **+++++[\n]** 1 **789ABCD[\n]** / 1 **+---+[\n]** 2 **EFGHIJK[\n]** / 2 **+---+[\n]** 3 **+++++[\n]**

Name	Specification	Encoded as a 1D String**	Encoded as a 1D String**
BIF_01	`BIF_01 <#rows> <#columns> <data>` *data is a sequence of rows*columns characters. Row major order.*	**BIF_01 3 7 0\ 123456789ABC\ DEFGHIJK**	**BIF_01 4 5 +\ +++++---++--\ -++++++**
BIF_02	`BIF_02 <#columns> <#rows> <data>` *data is a sequence of rows*columns characters. Row major order.*	**BIF_02 7 3 0\ 7E18F29G3AH4\ BI5CJ6DK**	**BIF_02 5 4 +\ ++++--++--++\ --+++++**
BIF_03	`BIF_03 <#columns> <data> '\n'` *rows = size of data / columns* *row major order. data shall **not** contain a '\n' char.*	**BIF_03 7 012\ 3456789ABCDE\ FGHIJK[\n]**	**BIF_03 5 +++\ +++---++---+\ +++++[\n]**
BIF_04	BIF_04 <#rows> <#columns> <data> *Data alternates between characters and a number indicating how many times it should be added to the image.* *This compression format is best for images with lots of repeated chars.* *No characters with 0 multipliers allowed.*	**BIF_04 3 7 0\ 1 11 21 31 4\ 1 51 61 71 8\ 1 91 A1 B1 C\ 1 D1 E1 F1 G\ 1 H1 I1 J1 K1**	**BIF_04 4 5 +\ 6 -3 +2 -3 +6**

*An raw Image contains an extra column in its decoding, which stores '\n' characters.

**If a line ends with \, then the string wraps around and doesn't include the \ or [\n] char.

[1] BIF stands for Bryce's image format. They are simple formats made up for this assignment.

Tasks for Part A

Task 1.1: Implement **`validateImage(Image & image)`** It should return true if and only if each row is terminated by a '\n' character, its size is exactly (#rows*(#cols + 1)) characters and there are no '\n' characters in columns 0 thru (#cols - 1).

Examples:

Valid:

`XXXXX[\n]`
`XXX[\n]` `XXX[\n]`

Not Valid:

`XXXXX`	`XXXXX[\n]` `XXXXX`
`XX[\n]` `XXX[\n]`	`XXXXX[\n]` `XXXX[\n]`

Task 1.2: Implement **`checkBounds(Image & image, int row, int col)`**It should return true iff row and col describe an actual location within the image, in other words if row is in the range 0 thru (#rows - 1) and col is in the range 0 thru (#columns - 1).

Note: Once you have your validation functions working, you should use them like assertions when implementing your encoding / decoding functions. Beginners often have trouble staying in bounds and using arrays perfectly and the C++ compiler might not tell you when you are doing something very silly.

Task 2.1: Implement **`decode_image_bif_01(istream & in, Image & out)`** and **`encode_image_bif_01(Image & in, ostream & out)`**

Both of them should return **true if** nothing goes wrong and immediately return **false if** the input does not follow the expected image or encoded character stream format.

Task 2.2: Implement **`decode_image_bif_02(istream & in, Image & out)`** and **`encode_image_bif_02(Image & in, ostream & out)`**

These functions follow the same return rules as BIF_01 as stated in task 2.1 above.

Task 2.3: Implement **`decode_image_bif_03(istream & in, Image & out)`** and **`encode_image_bif_03(Image & in, ostream & out)`**

These functions follow the same return rules as BIF_01 as stated in task 2.1 above.

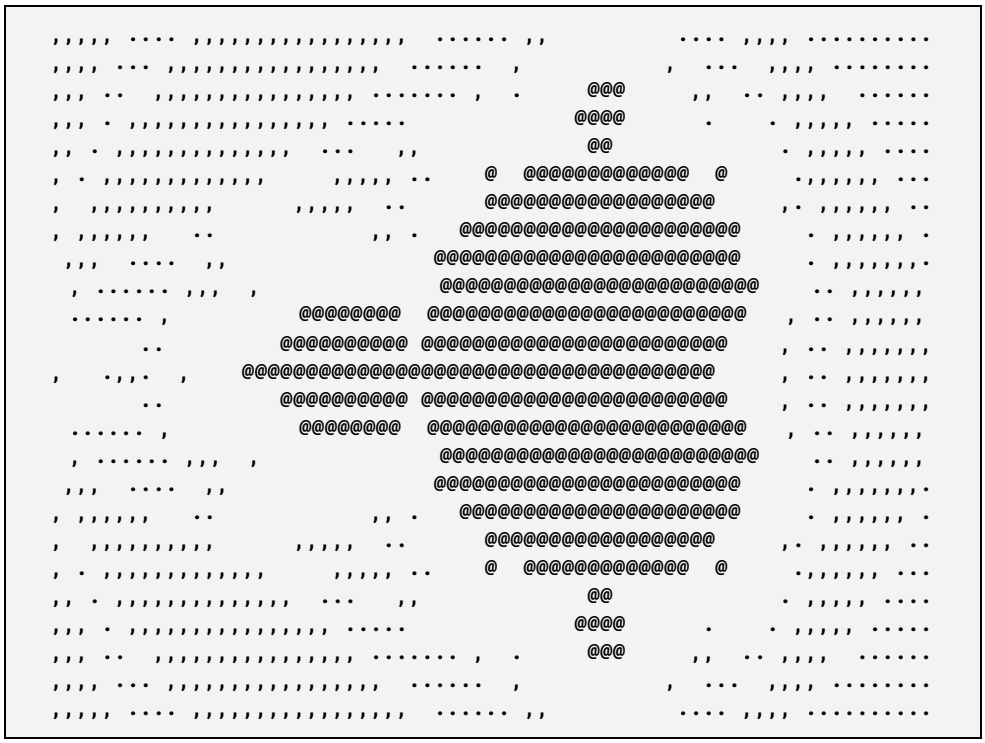
Task 2.4: Implement **`decode_image_bif_04(istream & in, Image & out)`** and **`encode_image_bif_04(Image & in, ostream & out)`**

These functions follow the same return rules as BIF_01 as stated in task 2.1 above.

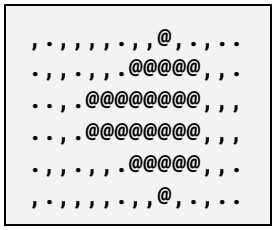
Part B: Image Rendering and Sampling.

Every two dimensional dataset needs to originate somewhere, sometimes you'll need to create data from scratch for personal edification, to produce test data, or to implement a dynamic modern user feature, such as responsive design that scales with the user's viewscreen.

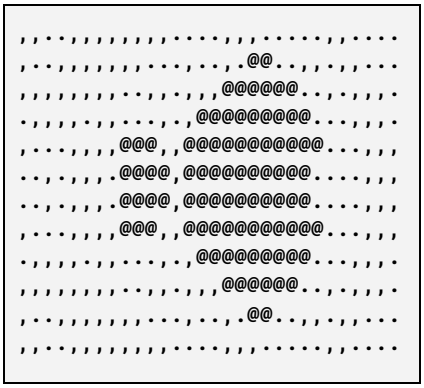
In part B, you will be implementing a mathematical process that produces an ASCII image of the mandelbrot set (https://en.wikipedia.org/wiki/Mandelbrot_set). Doing this should give you practice writing for and while loops, using helper_functions, as well implementing formulas using arithmetic operators. Your rendered images will be similar to these for the given dimensions:



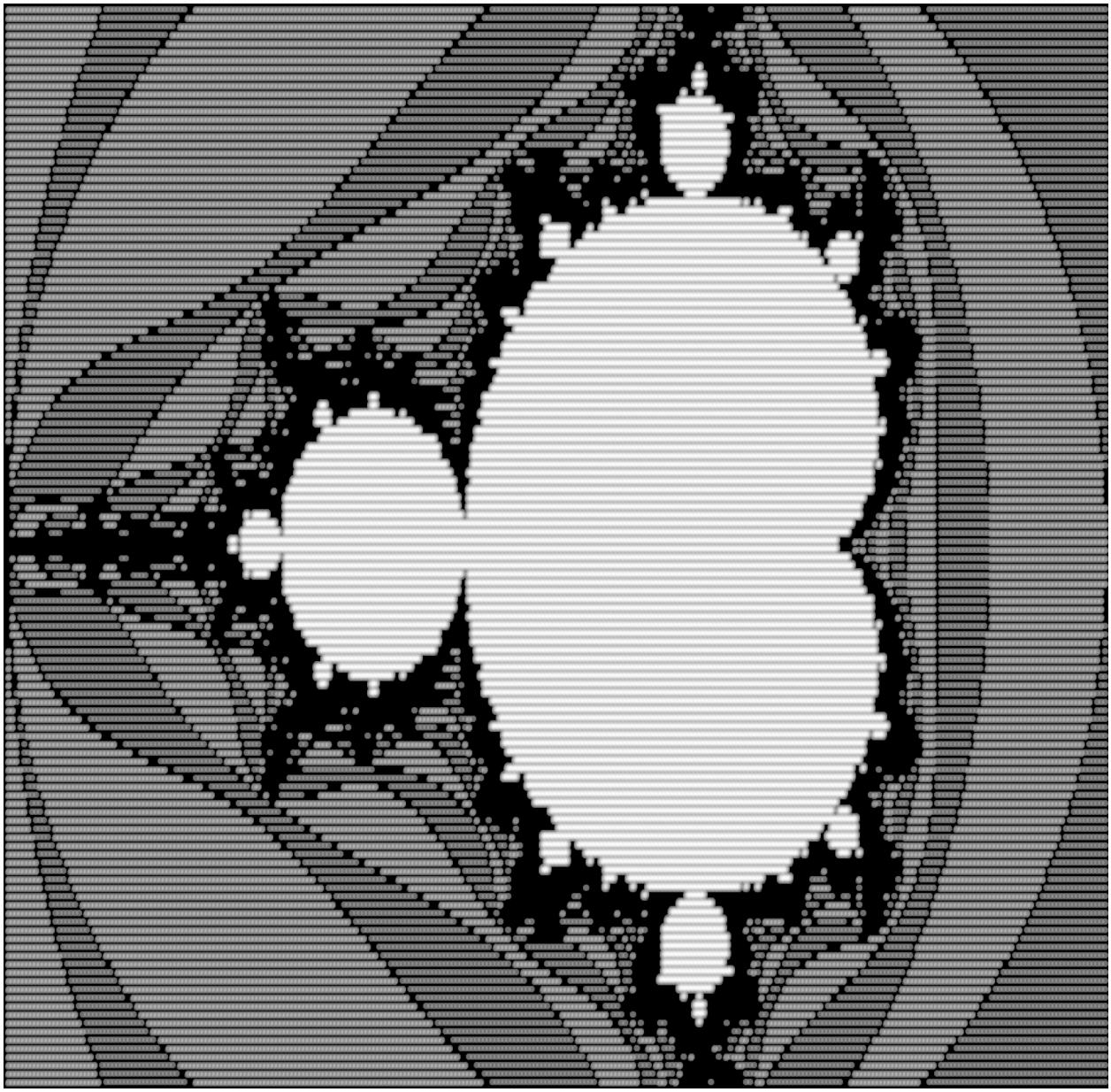
25 rows, 69 columns, 256 iterations, region sampled



6 rows, 15 columns,
100 iterations,
point sampled.



12 rows, 30 columns,
100 iterations, point sampled.



Mandelbrot Set, rows: 128, columns: 282, 100 iterations, region sampled.

```
    ,,o@@@@@@;,,
  .o@@@@@@@@@@@@@@;.
 o@@@@@@@@@@@@@@@@@@;
 o@@@@@@@@@@@@@@@@@@@;
;@@@@@@@@@@@@@@@@@@@@;
@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@
;@@@@@@@@@@@@@@@@@@@@;
 o@@@@@@@@@@@@@@@@@@@?
  o@@@@@@@@@@@@@@@@@?
   `o@@@@@@@@@@@@@?`
     ^^o@@@@@@?^^
```

Circle,
12 rows,
24 columns,
4 samples per region.

O'
12 rows,
24 columns
4 samples per region

```
    ,,o@@@@@@;,,
  .o@@@@@@@@@@@@@@;.
 o@@@@@@@@^^@@@@@@@@;
 o@@@@@^`      `^@@@@@;
;@@@@@`          `@@@@@;
@@@@@;            ;@@@@@
@@@@@;            ;@@@@@
;@@@@@.          .@@@@@;
 o@@@@@,.      .,@@@@@?
  o@@@@@@@@,,@@@@@@@@?
   `o@@@@@@@@@@@@@@?`
     ^^o@@@@@@?^^
```

Tasks for Part B

A point C: (c_x, c_y) is in the mandelbrot set iff the point Z(z_x, z_y) starts at the origin (0, 0) and then stays within a distance of 2 away from the origin (0, 0) after applying the following assignment statement an infinite number of times or at least until we give up.

$$Z' = Z^2 + C$$

Iteration (Assignment) Statement

Through these tasks, you shall build a mandelbrot set renderer from the bottom up, which will mark in set locations with '@' chars, out of set locations with '`' (and 'c' or '.' for partials)

Task 3.1: Implement **`escaped(float z_x, float z_y).`**
It should return true if the point Z is a distance of 2 or more from (0, 0).
In other words, it returns the truth value of this relational expression:

$$Z_x^2 + Z_y^2 \geq 2^2$$

Task 3.2: Implement **`iterate( float z_x, float z_y, float c_x, float c_y, float & x_out, float & y_out).`**
Given the point Z and the point C, execute the iteration assignment statement. Store the new z_prime_x value in x_out and the new Z_prime_y in y_out.

Assuming you might not be up on your complex number arithmetic, here is the translation of the iteration statement into how to compute z_prime_x and z_prime_out:

$$Z'_x = Z_x^2 - Z_y^2 + C_x$$
$$Z'_y = 2 \cdot Z_x Z_y + C_y$$

Be careful that you don't use Z_x, not Z_prime_x to compute Z_prime_y.

Task 3.3: Implement **`mandelbrotEscapeTime(float c_x, float c_y, int give_up)`**
Given a point C, create a point Z at the origin (0, 0) and count how many times the iteration assignment can be applied until Z has **`escaped`**. Stop iterating if Z escapes and return the iteration count. Otherwise, stop iterating if Z has not escaped after a number of iterations equal to the value of **give_up** and return **give_up + 1**.

Task 3.4: Implement **`sampleFromMandelbrot(float c_x, float c_y)`**.
Try to compute the **escapeTime** of the point C, but give up after 100 iterations. If the time exceeds 100 (it gave up) assume the point is in the set and return the V_IN character. Otherwise return `V_ODD` if the escape time is odd or `V_EVEN` if it is even.

Task 3.5: Implement **`lerp(float val1, float val2, float time).`**

Lerp linearly interpolates between val1 at time 0 and val2 at time 1.
To accomplish the interpolation, return the evaluation of this expression:

$$val1 \cdot (1 - time) + val2 \cdot time$$

Task 3.6: Implement **`void render(Image & image, Bounds bounds, char (*sampleAt)(float, float)).`**

1. Calculate the total number of rows and columns in the image's data. You will not need to push characters or otherwise modify the dimensions of the image.
2. **bounds** is a float vector with 4 float values in this order: [x1, y1, x2, y2].
3. Every location (row, col) in the image corresponds to a point (c_x, c_y) in the Euclidean plane as follows:

$$C_x = lerp\left(x1, x2, \frac{col + .5}{\#columns}\right)$$

$$C_y = lerp\left(y1, y2, \frac{row + .5}{\#rows}\right)$$

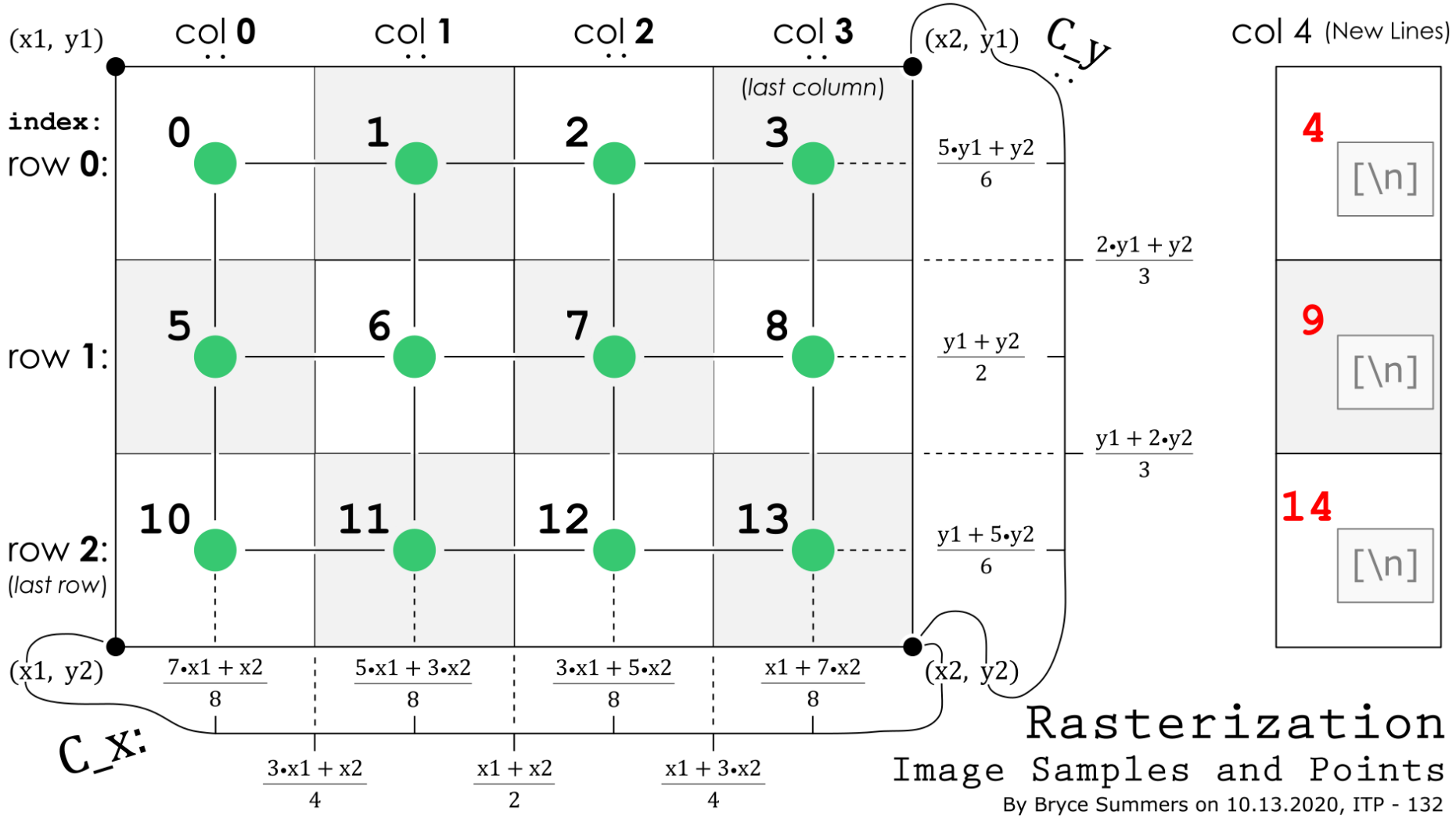
4. For each image location (row, col), assign it the value of **sampleAt(c_x, c_y)**. Please **look at the illustration on the next page** depicting the image indices, row-col locations and points for a 3 row by 4 column image!
5. Return true iff rendering completed successfully. Otherwise return false if the image or coordinates are malformed.

Task 3.7: Implement **`render(Image & image, int rows, int cols, char (*pointSampler)(float, float));`**

This is what is known as a wrapper function, one that protects a more fundamental worker function from bad inputs and translates between what is easy and convenient for the user to specify as arguments and what is easy for the original function to receive as parameters.

1. Clear the image of all character data. image.size() should be 0 afterwards.
2. Mutate image into a valid image that has dimensions(**rows**, **cols**). You can populate the data portions of the image however you want as they will be overwritten during the mandelbrot set rendering process.
3. Return false if **rows** < 1 or **cols** < 1. If this function ever returns false, then the state of the image is undefined.
4. render the default set view: top-left (-2, -1) to bottom-right (1, 1) onto the image by calling the worker render function you implemented in the previous task.

   `render(image, Bounds({-2,-1,1,1}), pointSampler);`
5. Return the success value received from the call to the worker render function.


(x1, y1)
col 0
col 1
col 2
col 3
(last column)
(x2, y1)
C_y
col 4 (New Lines)
index:
row 0:
0
1
2
3
4
[\n]
row 1:
5
6
7
8
9
[\n]
row 2:
(last row)
10
11
12
13
14
[\n]
(x1, y2)
(x2, y2)
C_x:
Rasterization
Image Samples and Points
By Bryce Summers on 10.13.2020, ITP - 132

Task 3.8: Implement **`superSampleMandelbrot(float c_x, float c_y, float region_w, float region_h)`**.

This averages the escape times of four points to determine the region's sample value.

1. If **x1** is `c_x - sample_w/4`, **x2** is `c_x + sample_w/4`, **y1** is `c_y - sample_h/4`, and **y2** is `c_y + sample_h/4`, then...
2. The 4 points re located at (**x1**, **y1**), (**x2**, **y1**), (**x1**, **y2**), and (**x2**, **y2**)
3. Calculate the **escapeTime** for each of the 4 points.
   (*call the function from task 3.3 four times using `MAX_ITERS` for **give_up***);
4. If the times are not all the same, return the `V_PARTIAL` character.
5. Otherwise, return `V_IN` if the **escapeTime** exceeds `MAX_ITERS`,
   return `V_ODD` if the **escapeTime** is odd, and
   return `V_EVEN` if the **escapeTime** is even.

Task 3.9: Implement **`render(Image & image, Bounds bounds, char (*sampleFromRegion)(float, float, float, float))`**.

This function is almost identical to the one you wrote in task 3.6. The only difference is that it uses a region sampler, like the one you implemented in Task 3.8, rather than a point sampler like you implemented way back in Task 3.4.

1. Assign to the image locations the values sampled from regions centered at the same points used in task 3.6, with widths equal to difference in x coordinates between each of the points and heights equal to the difference in y coordinates:

$$region\_w = \frac{x2 - x1}{\#columns}$$
$$region\_h = \frac{y2 - y1}{\#rows}$$

   More concretely, for each image location (row, col), assign it the value of **`sampleFromRegion(c_x, c_y, region_w, region_h)`**.

2. Just like in task 3.6, return true iff rendering completed successfully. Otherwise return false if the image or coordinates are malformed.

Task 3.10: Implement **`render(Image & image, int rows, int cols, char (* regionSampler)(float, float, float, float)).`**

Use your code from task 3.7, but pass the regionSampler function to render instead:

call: `render(image, Bounds({-2,-1,1,1}), regionSampler);`

Return true if successful. It fails and returns false in the same ways as in Task 3.7.

Part C: Binary Image CSG[2]

You will adapt some logic operations to Images. (*See examples on next page!*)

Each of them is equivalent to one of the 4 standard binary operators: NOT(!), AND(&), OR(|), and XOR(^), but applied to every location across a set[3] of parallel[4] images.
Each operation returns a new stack allocated Image with characters corresponding to the boolean[5] results for each location.

Task 4.1: Implement **`operator ! (const Image right)`**.

1. Create a new stack allocated Image called **out**.
2. For every character (**c**) in **right,** push a character to the back of **out**.
    a. If **c** is a new line char, then push a new line character: `'\n'`.
    b. If **c** is anything other than `V_FALSE`[6], push the char value `V_FALSE`.
    c. Otherwise push the char value `V_TRUE`.
3. Return **out**.

Task 4.2: Implement **`operator & (const Image left, const Image right)`**.

1. Create a new stack allocated Image called **out**. Return it at the end.
2. For every image **index** and the corresponding characters **`c1 = left[index]`** and **`c2 = right[index]`**, push a character to the back of **out**.
    a. If either char is a new line char, push a new line character: `'\n'`.
    b. If c1 and c2 are both said to be true[7], push the char value `V_TRUE`.
    c. Otherwise push the char value `V_FALSE`.
3. Return **out**.

Task 4.3: Implement **`operator | (const Image left, const Image right)`**.

Same instructions as *Task 4.2*, but do this instead for step 2b:

2b. If either c1 or c2 are said to be true, push the char value `V_TRUE`.

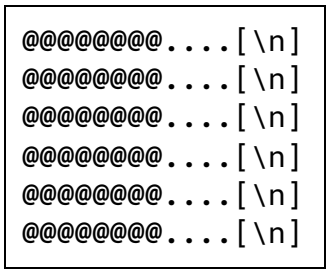
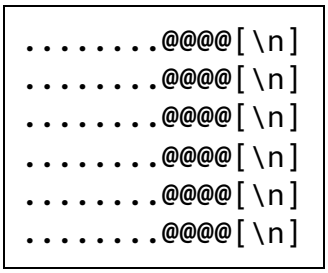
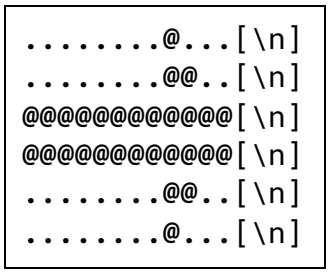
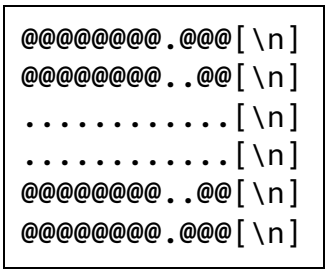
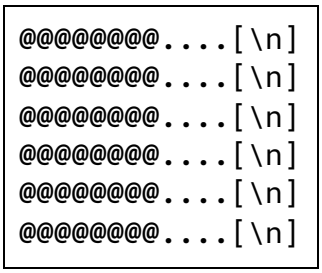
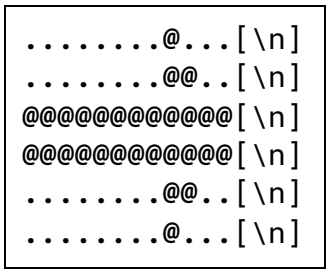
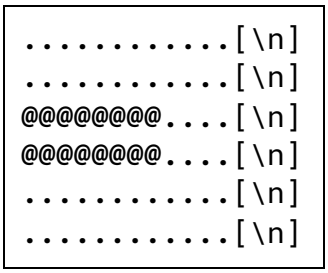
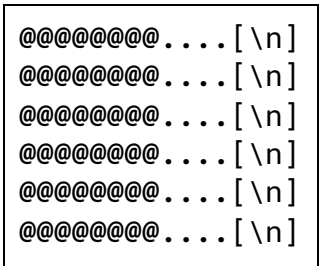
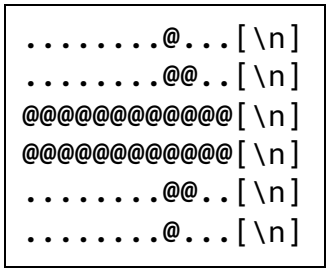
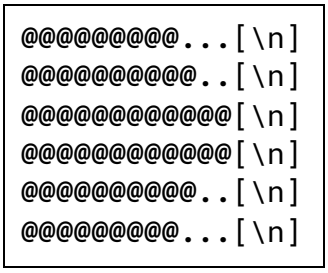
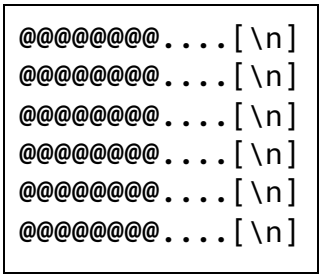
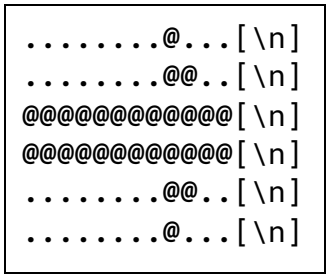
Task 4.4: Implement **`operator ^ (const Image left, const Image right)`**.

Same instructions as *Task 4.2*, but do this instead for step 2b:

2b. If exactly one of c1 or c2 is said to be true, push the char value `V_TRUE`.

---

[2] CSG stands for Constructive Solid Geometry.
[3] 1 image for the unary NOT operator, and 2 images for the binary AND, OR, and XOR operators.
[4] Concept: Parallel Arrays.
[5] The value of character constant `V_FALSE` (`'.'` by default) is said to be false for the purposes of this computation and the value of the character constant `V_TRUE` ('@') or any # other character not equal to `V_FALSE` is said to be true.
[6] Note: V_FALSE and V_TRUE are character constants defined near the beginning of the starter code. In my examples I've assigned `V_FALSE = '.'` and `V_TRUE = '@'`.
[7] "Said to be true" means: NOT EQUAL TO `V_FALSE`.

Op Name	LEFT (*image arg*)		RIGHT (*image arg*)		OUT (*Evaluation*)
NOT	(*No left*) (*unary op*)	!		=	
	(*No left*) (*unary op*)	!		=	
AND		&		=	
OR		|		=	
XOR		^		=