COMP360/560 Lab 1 : Turtle Graphics

Due Date: Feb 7, 2021, 11:59pm

1 Overview

In this project you will implement turtle graphics as discussed in class. First, you will create a virtual turtle that can carry out the basic turtle commands. Next, you will use your turtle to draw simple shapes. You will then implement operators, such as shift, spin, and scale, which you will apply to simple shapes to generate more complicated geometry. Finally, you will create recursive turtle functions that generate fractals.

This project is due on Feb 7, 2021 at 11:59pm. It is worth 100 points.

You should work in pairs on this lab. Please let the TAs know of your partnership ASAP.

2 Specification

The given code can be compiled directly without change. The window in **Figure 1** should appear when compilation finishes. The program is divided into three separate sub-regions: the **Display** (black box to the left), the **Input** (white vertical box in the middle), and the **History** (white vertical box to the right). For most of your assignment, you will be typing into the **Input** and the turtle actions will be drawn in the **Display**. When your program starts, the **Display** should be blank.

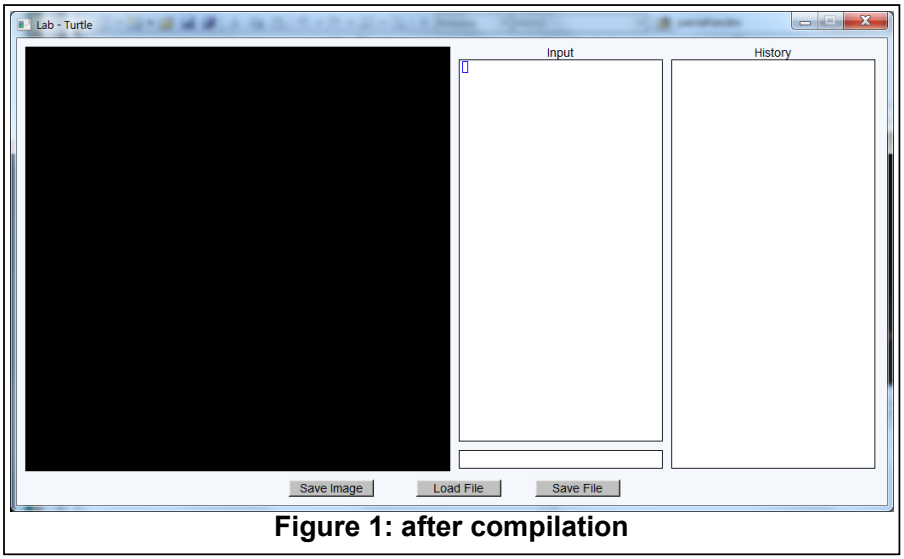

Figure 1: after compilation

This project is divided two parts. **Part One** requires you to fill in the appropriate code for changing the state of the turtle. This includes changing the turtle's position, direction, and scaling factor. It also includes changing the width and the color of the trail of lines drawn by the turtle. This part needs to be completed by modifying the given framework (C++ code). The first part must be completed before you can proceed to the second part of the project.

Part Two requires you to implement turtle drawing commands using a given language. This language has built-in functions that correspond to the C++ code of **Part One**. The language we will be using is based on Logo. The syntax of this language is very close to the syntax of Mathematica. For the sake of brevity, we will refer to the language as **L1** (for Lab 1).

Any portion that requires your implementation will be annotated (in parenthesis) with the point value of that particular section. For example, the work required in Section 2.1 is worth 17 points.

2.1 Part One - Completing the turtle commands (17 Points)

First, open the file **`Rendering/Turtle.h`**. In this file, there are several classes, and their purposes can be easily intuited from their names (such as **`Pt`** for describing a 2D point and **`Line`** for describing a line segment). The class **`TurtleMachine`** is the class you will need to modify.

`TurtleMachine` keeps track of the following information:

- One turtle (a point and a direction)
- Current line color (three real numbers)
- Current line width (one real number)
- Current scaling factor (one real number)
- A list of line segments drawn by the turtle (list of pairs of points)

Your job is to fill in the following empty functions:

- **`void forward(double step)`** - Moves the turtle forward by the amount **`step`** (with respect to the current direction and scaling factor). This function should record a line segment between the starting and the ending points of the turtle's movement.
- **`void move(double step)`** - Same as forward, except does not draw a line segment.
- **`void turn(double ang)`** - Changes the direction that the turtle is facing by **`ang`** radians.
- **`void resize(double scale)`** - Changes the scale of the turtle's movement by a factor of **`scale`**
- **`void color(double r, double g, double b)`** - Changes the color of the lines that are drawn after calling this command.
- **`void lineSize(double width)`** - Changes the width of the lines that are drawn after calling this command.
- **`void reset()`** - Resets the state of all the variables to their default values (i.e. turtle is back at the origin, facing right; color is white; line size is 1; scale is 1).

The lectures in class and the textbook have described some of these functions in more detail. Please first refer to them if you have trouble with this portion. Once you have implemented these functions, you are ready to start writing turtle programs.

2.2 Part Two - Turtle programming with L1

We will first learn the syntax of **L1**. **L1** is a simple, bare-bone language that allows you to program the turtle and create interesting pictures.

2.2.1 Basic language specs

An **L1** program consists of statements. Statements can be entered into the **Input** for evaluation (one statement at a time). **After you have typed a statement in Input, you can evaluate the statement by holding the `shift` key and pressing `enter`.** In the following sections, we describe valid statements in **L1**.

2.2.1.1 Numbers and symbols

Numbers are defined in standard notation. Only integers and decimal numbers are allowed. The following are proper numbers: "**`1`**", "**`.1`**", "**`1.`**", and "**`-1.2`**". The following are not proper numbers: "**`1,000`**", "**`+2`**", and "**`1e9`**".

Symbols are defined in the same way as in C-style languages. The first character must be a letter ('a'-'z' or 'A'-'Z').

The following characters can either be letters or digits ('0'-'9'). The following are valid symbols: "**abc**", "**abc123**", and "**A123b**". The following are not valid symbols: "**1abc**", "**_abc**", and "**345**". Symbols are used in variable and function definitions.

L1 provides one special constant, **Pi**, which contains the decimal value of Pi. You will want to use this symbol in your implementation of the turtle.

2.2.1.2 Arithmetic

You can perform simple arithmetic with **L1**, using a syntax that is similar to C-style languages. As an exercise, type the following statements into the **Input**:

- **1+1**
- **1+1*2**
- **4/2+2**
- **4/(2+2)**

The small horizontal box (**Output**) under the **Input** should have displayed: **2**, **3**, **3**, and **1**, respectively. Integer arithmetic and real value arithmetic are not the same. For example, **1/2** evaluates to **0**, but **1/2.0** evaluates to **0.5**. Mixing integers with real values will produce real values (in general, think C-style behavior).

Like C-style languages, the order of operation between operators of the same priority is determined from left to right. However, to avoid confusion, we encourage you to use parenthesis liberally to enforce order of operation.

2.2.1.3 Variable definition

Variables can be defined in a syntax similar to C-style languages. Type the following two statements into the **Input**. Remember you can only evaluate one statement at a time in **Input**.

1. **x=0**
2. **x+1**
3. **y**
4. **x=x+1**
5. **x**

The **Output** should display **1** for the second statement, and fail on the third statement. The third statement fails because **y** is undefined. The fourth statement adds **1** to the previous value of **x** and saves the new value into **x**. Hence, the fifth statement should evaluate to **1**.

2.2.1.4 Function definition

The definition of a function has two parts, separated by the **=** sign. The first part is defined by first choosing a function name (a symbol), followed by a comma-separated list of parameters (list of symbols). The parameters are enclosed by square brackets(**[a,b,c,...]**). The second part is a statement that can reference the parameters as variables. The following are examples of valid function definitions,

- **f[a,b]=a+b**

- `g[a]=a+1`
- `h[a]=Move[a+2]`

Recursive function definition is also possible but needs to be done with care. A later section will cover recursion.

2.2.2 Special functions

These special functions will be necessary in fully implementing turtle graphics.

2.2.2.1 `Block[A,B,C,...]` - execute multiple statements

`Block` groups sequential statements into a single composite statement. Assume we are given statements **`A`**, **`B`**, and **`C`**. We can build the sequence of executing **`A`**, **`B`**, and **`C`** by writing **`Block[A,B,C]`**. The value of a **`Block`** statement is the value of the last statement executed in the sequence, **`C`**. Consider the example,

- `Block[x=1, x=x+1, x]`

This will execute **`x=1`**, **`x=x+1`**, and **`x`** in that order. The **Output** should display the value of **`x`**, which is **`2`**.

2.2.2.2 `For[i,a,b,S]` - looping structure

The **`For[ ]`** function allows for repeated execution of a statement, **`S`**; **`i`** represents the counting variable; **`a`** and **`b`** are integers with the relationship that **`a<=b`**. The statement **`S`** can refer to **`i`** as a variable; **`i`** will count from **`a`** to **`b`** with respect to the loop iteration. Try the following examples,

1. `Block[x=0, For[i,1,5,x=x+i], x]`
2. `Factorial[n]=Block[x=1, For[i,1,n,x=x*i], x]`

For statement 1, the **Output** will display the value of **`x`** which is **`1+2+3+4+5=15`**. Statement 2 defines a factorial function that computes **`n!`**.

2.2.2.3 `Eq[a,b,T,F]` - conditional

The **`Eq[ ]`** function allows for branching. You can consider **`Eq`** as equivalent to the more familiar notation **`if(a==b) {T} else {F}`**. Statement **`T`** will only execute if **`a`** is equal to **`b`**. Otherwise, statement **`F`** will execute. Try the following,

- `test[a,b]=Eq[a,b*2,1,0]`

This function will return **`1`** if **`a`** is **`2`** times **`b`**. So **`test[2,1]`** evaluates to **`1`**, but **`test[1,1]`** evaluates to 0.

2.2.2.4 `Cos[x]` and `Sin[x]` - trignometric functions

Basic trig functions that you may use to implement turtle programs. The parameter **`x`** should be an angle in **radians**.

2.2.2.5 `Print[x,...]` - Standard output print function

This function prints the value of its parameters. The functions can take any number of parameters greater than one. For two or more parameters, the output will be a comma-separated list of the values of the parameters.

2.2.2.6 `Int[x]` - Converts a numeric value to an integer

This function converts a numeric value, which can be either an integer or a floating point value, to an integer. A floating point to integer conversion will truncate any value after the decimal point. For example, both **`Int[1.1]`** and **`Int[1.6]`** will return the integer **`1`**.

2.2.2.7 `ResetAll[ ]` - clear all states

This function resets all variable and function definitions. It also resets the state of the turtle.

2.2.2.8 `Hold[ ]` - passing procedures as arguments to functions

To allow for proper passage of procedures as arguments, we need an identifier that indicate "lazy evaluation" of the parameter arguments. In general, if we write a statement **`f[x]`**, where **`f[x]=Block[x,...]`**, **`x`** is evaluated once at the function instantiation, and once more when the function uses the variable **`x`**. For example, if we write

- **`f[x]=x`**
- **`Block[a=0,f[a=a+1],a]`**

What value would you expect **`a`** to hold? Most people would answer **`a=1`**. By substituting **`a=a+1`** into **`f[x]`**, we get **`f[a=a+1]=(a=a+1)=(a=0+1)=(a=1)`**. But **`L1`** greedily evaluates its function parameters, which means **`a=a+1`** will be evaluated twice, so **`a=2`** (try it). To prevent greedy evaluation, we use the following syntax,

- **`f[x]=x`**
- **`Block[a=0,f[Hold[a=a+1]],a]`**

Then, the output value is **`1`**, which is what we expect. This feature will be essential for functions such as **`Shift`**,**`Spin`**, and **`Scale`**.

2.2.2.9 General note about recursion

In general, your recursive functions should take the form

- **`f[n]=Eq[n,0,baseCase,f[n-1]]`**

where **`baseCase`** is a statement you want to evaluate as the base case of the recursion. Most (if not all) of the fractals that you are required to implement should have this form. It is also possible to have the base case as a variable, such as,

- **`f[n,baseCase]=Eq[n,0,baseCase,f[n-1,baseCase]]`**

In this case, as mentioned in the previous section, you need to call the function **f** with a lazily-evaluated base case statement. So calling **f** might look like,

- **f[3,Hold[Block[Move[10],Forward[30],Turn[Pi/2]]]]**

Note the use of the **Hold** function.

2.2.3 Turtle functions

The turtle functions are built-in commands that correspond to the functions you have implemented in **Part One**. Following are their syntactic definitions,

- **Move[x]** - **x** should evaluate to a real number.
- **Forward[x]** - **x** should evaluate to a real number.
- **Turn[x]** - **x** should evaluate to a real number.
- **Resize[x]** - **x** should evaluate to a real number.
- **Color[r,g,b]** - **r,g,b** should evaluate to real numbers.
- **ResetTurtle[ ]** - resets the position, orientation, color, and linewidth of the turtle.

Once you have finished **Part One**. You can test out these commands. As an example, you can try the following statement,

- **For[ii,1,3,Block[Forward[50],Turn[(Pi/6.0)*4]]]**

This code should draw a triangle, with sides of length **50**.

2.2.4 Simple turtle programs (17 Points)

You will need to complete the following functions (or programs) in **L1**. You can save your progress by using the "Save file" button. ***Save often so that if the program unexpectedly crashes, you won't feel compelled to destroy your monitor.*** The saved files are simple text files that you can edit freely. In a saved file, statements are separated by "#", and they are executed in the order from top to bottom when loaded.

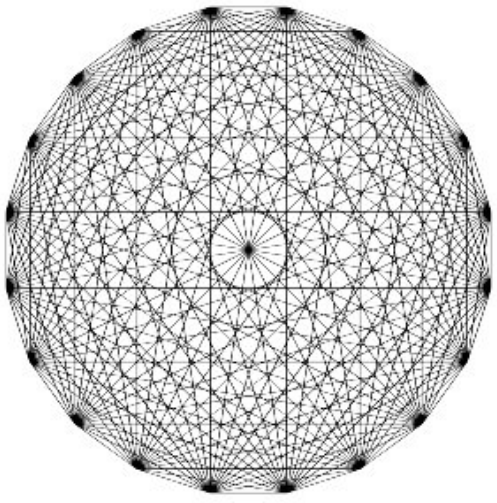

Figure 2: Rosette

- **Polygon[length,sides]** - Draws a polygon with **sides** sides, with each side of length **length**.
- **Star[length,vertices]** - Draws a star with **vertices** vertices, where each edge is of length **length**.
- **Circle[radius]** - Draws a circle with radius **radius**.
- **InscribedStar[radius,vertices]** - Draws a circle of radius **radius** inscribed with a star with **vertices** vertices. The points of the star should lie along the circle.
- **Wheel[length,vertices]** - Draws a wheel with **vertices** vertices, where each outer edge is of length **length**. Remember that a wheel is a polygon with every vertex connected to the center.
- **Rosette[length,vertices]** - Draws a rosette with **vertices** vertices, where each outer edge is of length **length**. Remember that a rosette is a polygon with every vertex connected by a straight line to every other vertex (**Figure 2**).

- **`Spiral[length,angle,scale]`** - Draw a spiral whose initial arm is of length **`length`**, whose arms meet at **`angle`** radians, and whose arms decrease in length by a factor of **`scale`**.

2.2.5 Turtle program operators (17 points)

- **`Shift[distance,repeats,procedure]`** - This operator should draw the geometry described by **`procedure`**, then shift the turtle (using the **`Move`** command) over **`distance`** steps, repeating the sequence **`repeats`** times. The **`procedure`** can be any statement, but **`Shift`** must be called with the **`Hold`** function (see Section 2.2.2.6).
- **`Spin[angle,repeats,procedure]`** - Like **`Shift`**, this operator should draw the **`procedure`**, then rotate the turtle by **`angle`** radians, repeating this process **`repeats`** times.
- **`Scale[scale,repeats,procedure]`** - This operator should draw the **`procedure`**, then scale the turtle's step size by a factor of **`scale`**, repeating this process **`repeats`** times.

Your program should also respond to the following fractal commands.

2.2.6 Turtle fractals (17 points)

Figure 3a: `KochSnowflakeSide` at recursion level 1

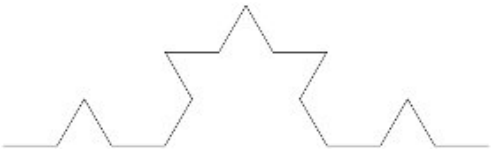

Figure 3b: `KochSnowflakeSide` at recursion level 2

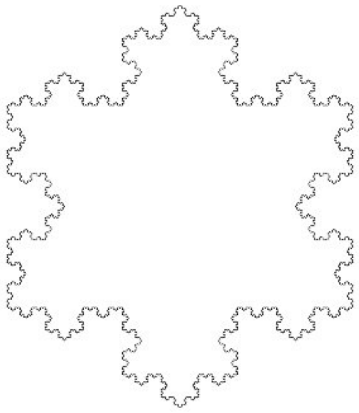

Figure 4:The Koch Snowflake, created by the `Snowflake` program with `sides = 3`.

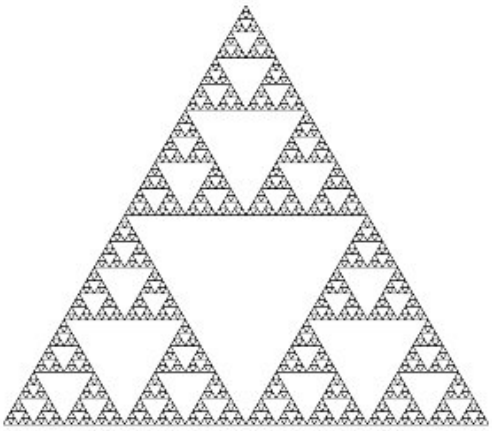

Figure 5: The Sierpinski Gasket, a `PolyGasket` with `3` sides and a scale factor of `0.5`.

- **`KochSnowflakeSide[level,length]`** - Draws one side of a Koch snowflake (that is, a Koch curve). The parameter **`level`** determines how many recursive calls to make for each side of the curve. The overall structure should start, at level 1, as a line of length **`length`**(**Figure 3a**). At level 2 the program should draw a triangular bump in the middle of the line, and so on (**Figure 3b**).
- **`SnowflakeSide[level,sides,length]`** - Draws a bump fractal containing a polygon with **`sides`** sides. This is the generalization of the Koch curve to arbitrary polygons. (A call of **`SnowflakeSide[N,3,L]`** would be equivalent to **`KochSnowflakeSide[N,L]`**.)
- **`Snowflake[level,sides,length]`** - Generates a polygon of **`sides`** sides, where each side is generated using the **`SnowflakeSide`** program with the same arguments. Thus, if **`side = 3`**, this command should

produce a Koch snowflake as shown above - a triangle where each side is the fractal defined by **`SnowflakeSide[level,3,length]`** (**Figure 4**).

- **`PolyGasket[level,sides,length,scale]`** - This function should produce a gasket from a polygon with **`sides`** sides, each of length **`length`**. Recursive copies of the gasket should be scaled by the factor **`scale`**. For example, if **`sides`** is **`3`** and **`scale`** is **`0.5`**, the result should be the Sierpinski gasket (**Figure 5**).

2.2.7 Additional fractals (16 points)

For the remaining fractals, you should create your own command syntax. Be sure to clearly document how to call these turtle programs in your README file. Implement turtle programs to generate the five fractals shown below (**Figures 6&7**).

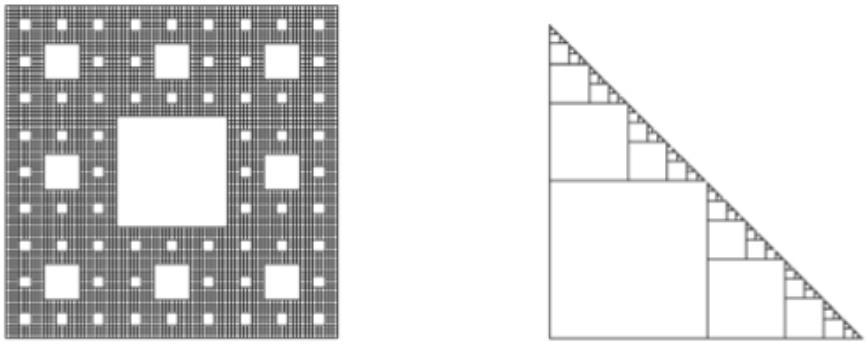

Figure 6: The Square Gasket and the Fractal Staircase.

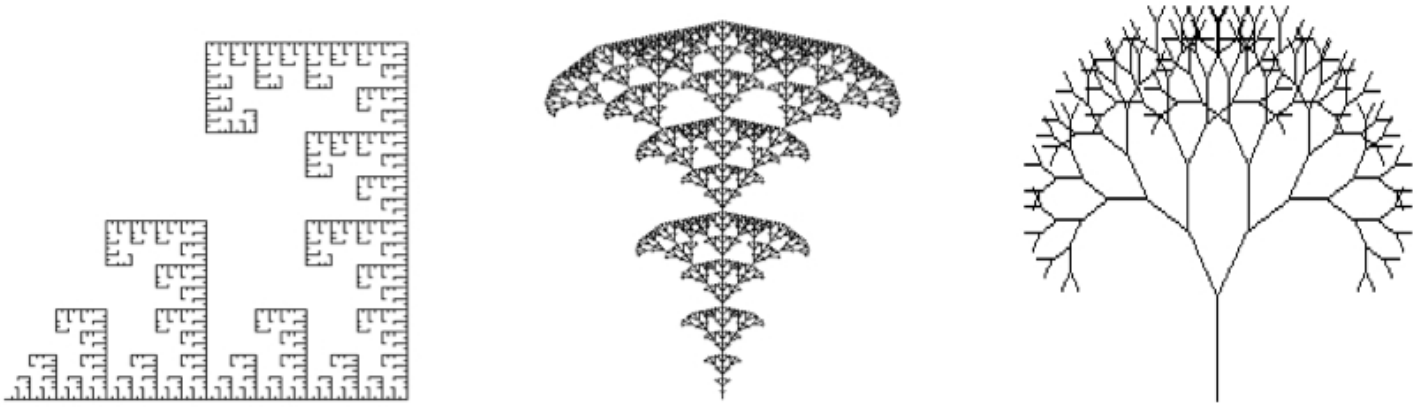

Figure 7: The Hangman Fractal, a Fractal Bush, and a Fractal Tree.

2.2.8 Flag of Freedonia (16 points)

Freedonia has just emerged from several centuries of a brutal absolute monarchy. Rufus T. Firefly, the newly elected democratic prime minister of Freedonia, has declared a contest among his citizens to create an appropriate flag for the new Republic of Freedonia to replace the old flag of the monarchy, which displayed a lion disemboweling a lamb on a bloody red background. You are going to enter this contest, using turtle graphics to generate a new flag for the fledgling republic. The best flag wins the grand prize, sharing a bowl of duck soup with the prime minister. Your grade will reflect the artistic quality of your flag. Once again, you should document in your README the commands necessary to produce your flag.

2.3 Notes

- The turtle does not have to return to its original position or scale after commands **`Shift`**, **`Spin`**, and **`Scale`**.

- The number of times to repeat the lines (spiral arms) in the **`Spiral`** should be sufficiently large that it generates a reasonable spiral.
- For the inscribed star with an even number of vertices, you have the freedom to decide what to draw. Please describe your decision in the README.
- Colors are generally implemented as a set of three numbers, representing the amount of red, green, and blue light to project on the screen. Often we treat these numbers as floating point values normalized to the range [0.0, 1.0]. Thus, for the **`Color`** command, you are given three floating point numbers; the triple (1.0, 1.0, 1.0) represents pure white, and (0.0, 0.0, 0.0) represents pure black.

3 What to turn in

Use the given "lab_turtle" directory. This directory should contain your source code, all files needed to build your program in Visual C++, all text files containing each of your turtle programs **(In each file just include a single turtle program)** and all the images generated by your turtle program. The grader should be able to load up the text file and test all of your turtle programs. The outcome of each turtle program should be the same as the image you submit. Also include a README pdf file containing:

- Your name.
- A brief overview of how your code is organized.
- Any further instructions on running your fractals or other programs.
- Any other implementation details you think your grader should know.

You should turn in a .zip file containing the directory of the VS project in Canvas. Please name your .zip file as Lab1 yourNetID.zip. If your project will be completed late, email the TAs and let them know.