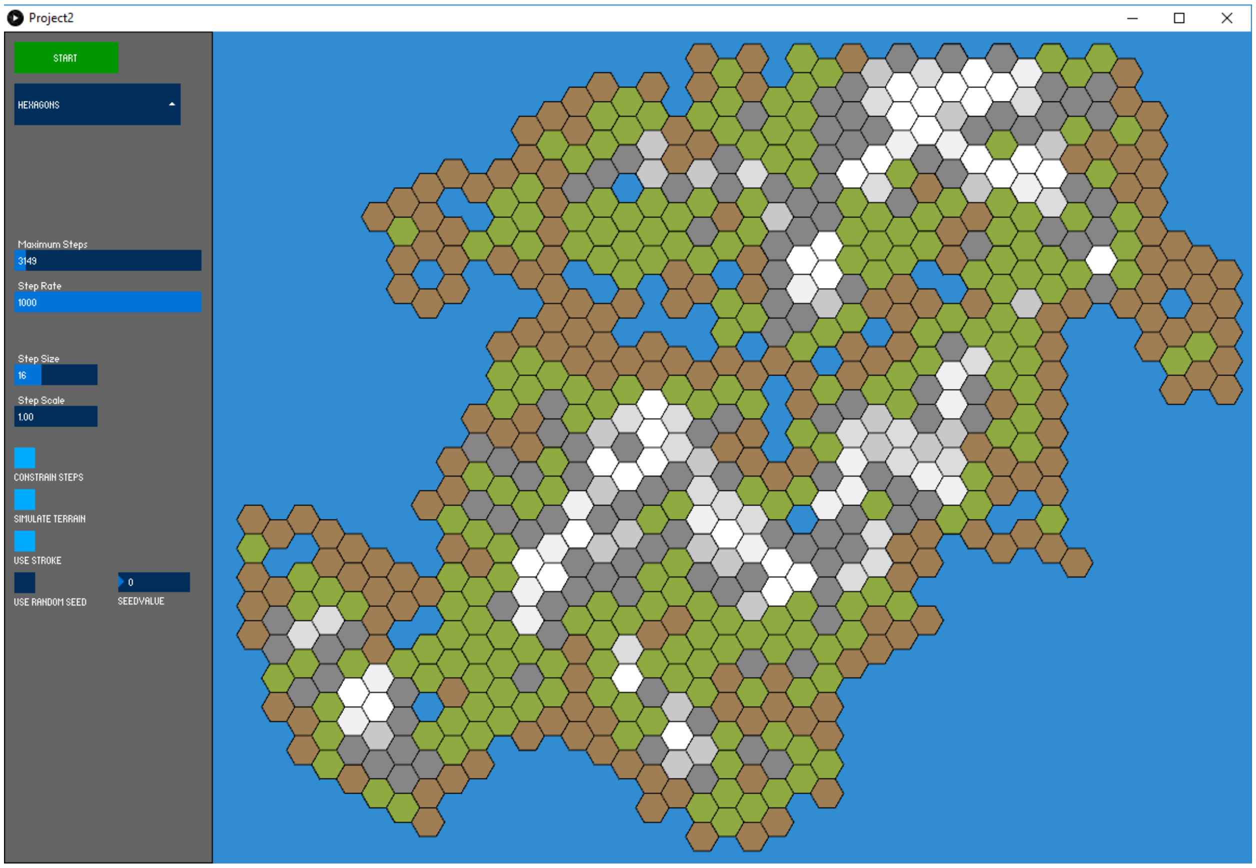
Project 2 – Random Walk Variant

Overview

For this project, you are going to implement 2 variations on the previous assignment. Overall the concept will be the same, but instead of individual points/pixels, you will take steps using 2 different kinds of shapes: squares and hexagons.

Description

For this assignment you are going to create a program that looks something like the following:



HEXAGONS

Maximum Steps 3149

Step Rate 1000

Step Size 16

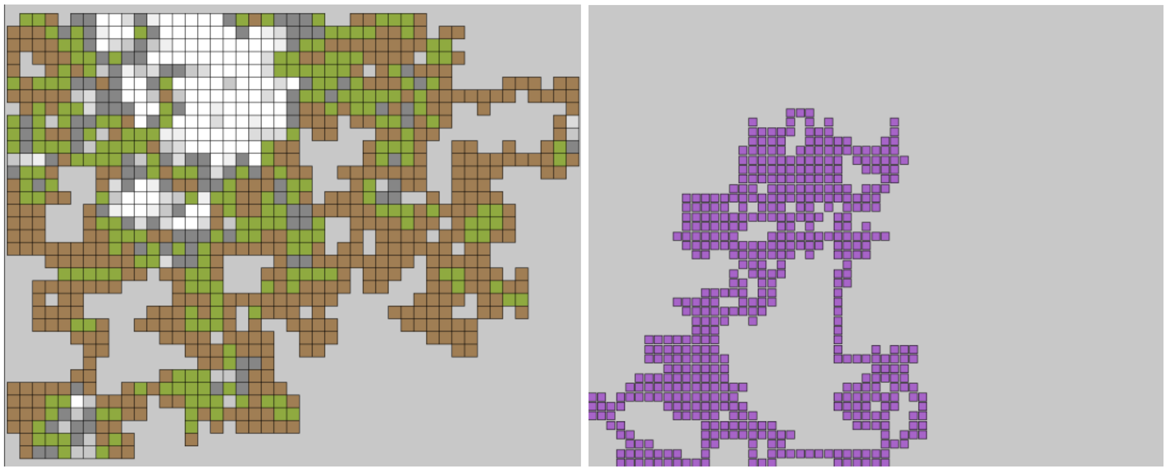
Step Scale 1.00

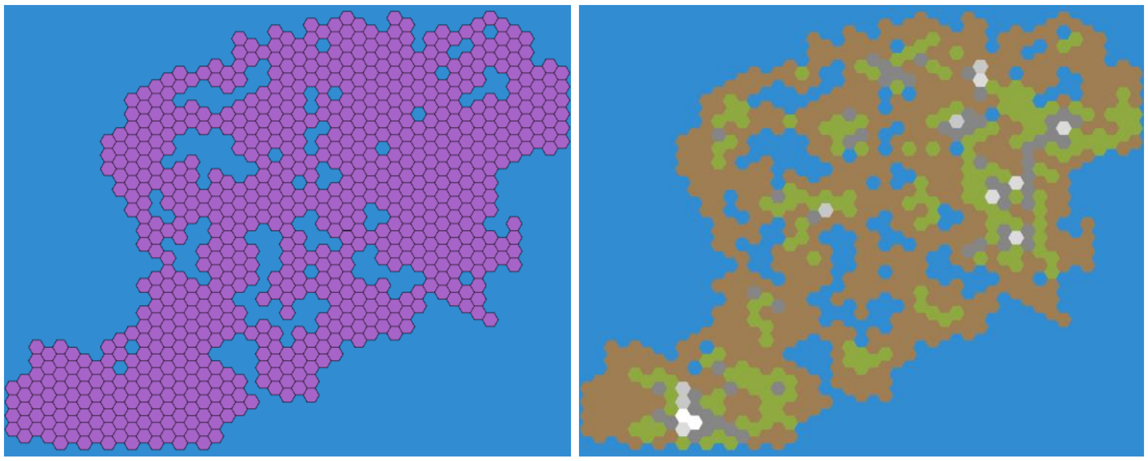
SEEDVALUE 0

The layout of this program will have two main parts:

- The UI, which consists of a variety of controls, detailed in the next section
- The viewable area, which will contain the rendered shapes as the algorithm progresses

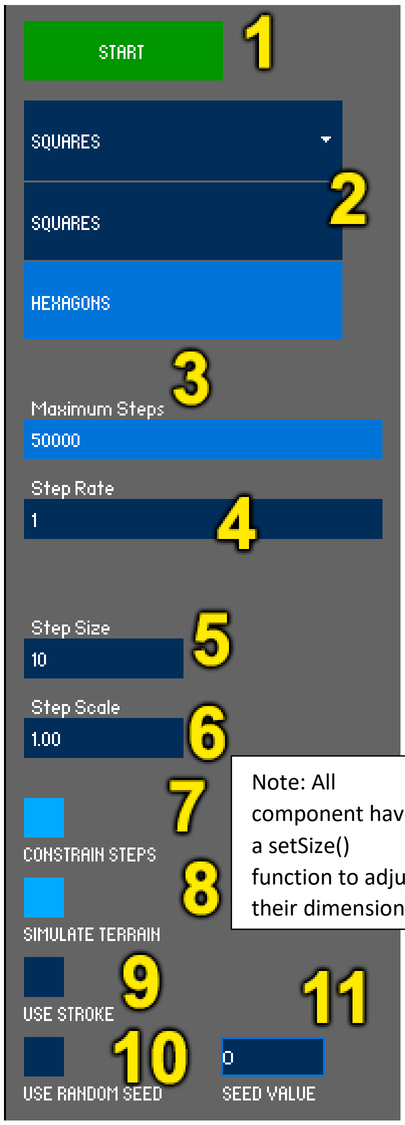
Example Images


~5000 squares with terrain coloring vs 1200 squares with smaller size and spacing of 1.45


~2300 hexagons with and without stroke and terrain coloring, using the same seed value

UI Specifications

The UI for this project is not complicated, but there are a fair number of components that will need implementation. The details for these are described below. The background of the UI is a simple rectangle with a width of 200. Many applications will have multiple panels like this, often with dynamic properties. (WINDOW SIZE: 1200x800)



Note: All component have a setSize() function to adjust their dimensions

1. Where the magic happens, the start button. You can change the background color by using the setColorBackground() function.
2. A DropdownList object. The following functions will be helpful:
    a. `addItem()` – Adds and item to the list
    b. `setItemHeight()` – set the height of entries in the list
    c. `setBarHeight()` – set the height of the "selected" item on the top
    d. `getValue()` – retrieve the 0-based index of whatever is selected
3. A Slider for the maximum number of steps, with a range of 100 to 50,000. The text above it is a Textlabel. To disable the text on a Slider, you can call getCaptionLabel().setVisible(false), or just pass an empty string to the setCaptionLabel() function.
4. A Slider to indicate how many steps to take in a single frame, with a range of 1 to 1,000
5. A Slider indicating the size of a single step. This will be represent the length of a square's side, or the radius of a hexagon. The range of this should be 10 to 30.
6. A Slider for an additional modifier to the length of a step from one position to the next. This is a multiplier with a range of 1.0 to 1.5.
7. A Toggle indicating whether or not the walking process should remain bound to the viewable area or not (the rest of the window NOT covered by this UI panel). To access the on/off value of a Toggle object, simply catch the return of the getState() function.
8. Another Toggle representing if the drawing process should color the steps to give a rough approximation of terrain elevation, or instead use a fixed color for each space.
9. A toggle to indicate whether shapes should be drawn with a stroke outline or not.
10. A toggle to indicate whether a specific value should be used to seed the random number generator or not.
11. A Textfield to let you specify a seed value for the random number generator. You can seed the generator by calling the function `randomSeed()`. Some helpful functions for this control:
    a. `setInputFilter()` – Pass ControlP5.INTEGER to this function to limit input to integers
    b. `getText()` – This retrieves the data stored in this control, as a string. In order to use this as a number, you will have to convert it. (Java has `Integer.parseInt()` to help with this.)

NOTE: If you are using a high resolution display, the function `pixelDensity(2)` can be called after size() to increase the pixel count of your window. This will make the controls a bit larger without having to manually change their sizes (and the text of the labels).

Class Structure

You should create at least 3 classes for this project:

- An abstract base class with the core functions in a random walk algorithm
- A class to control the square walk
- A class to handle the hexagonal walk

Why set up classes like this? So you can write like this:

```
RandomWalkBaseClass someObject = null;

if (optionA)
    someObject = new SquareClass();
else if (optionB)
    someObject = new HexagonClass();

someObject.DoSomeWalkStuff(); // Three cheers for polymorphism!
```

Class Features

On a basic level, your classes should do two things:

Update() – Generate a random number, figure out where to move, and move to that location
Draw() – Draw the specific shape according to the relevant properties: location, size, color, with or without a stroke, etc

Based on the UI controls and program layout, your base class will need to store information regarding the following:

- The maximum number of steps
- The number of steps already taken
- The distance of each step
- The scale for each step, to add borders/padding around each shape
- Whether or not to use color
- Whether or not to use a stroke when drawing a shape
- The boundaries of the viewable area. Normally it would be 0 -> width-1 and 0 -> height-1, but you will need to take the side panel into account, as you don't want to draw over (or under) the UI.

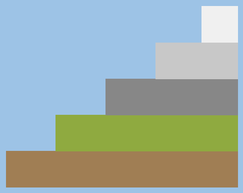
Color Details

The program approximates the idea of building up elevation depending on how many times a particular location is "visited." How you do you store each of these visits? You would need to associate a particular location with a count. A `HashMap<>` is perfectly suited to this scenario (this is just like a `map<key, value>` in C++, or a `Dictionary<TKey, TValue>` in C-Sharp. A **PVector** with the current x and y location serve as a key, with the count stored as the value.

Based on the number of visits to a particular location, the color for that shape will change to represent dirt, grass, rock, and snow at the highest levels. The arbitrarily chosen formula and colors used for this is as follows:

For a visit count less than 4, fill with the color 160, 126, 84 (dirt)
For a visit count less than 7, fill with the color 143, 170, 64 (grass)
For a visit count less than 10, fill with the color 135, 135, 135 (rock)
Otherwise, fill a color equal to the number of visits * 20, ultimately capping at the maximum value of 255—this will allow up to 4 more colors (for values of 200, 220, 240, and 255)

If you want to modify this formula a little bit, by adding more granularity between the steps (what about 2 steps, or 5?), you may do so. Overall there should be a similar progression from bottom to top.



Random Numbers in Processing

The `random()` function generates a floating point number from the minimum value (inclusive) to the maximum (exclusive). So `random(1,4)` will give you a number between 1.0 and 3.999999999, but never four. Instead, using `random(1,5)` will give you between 1-4.99999999 which, when stored in an integer, will be 1-4.

Random Walk - Squares

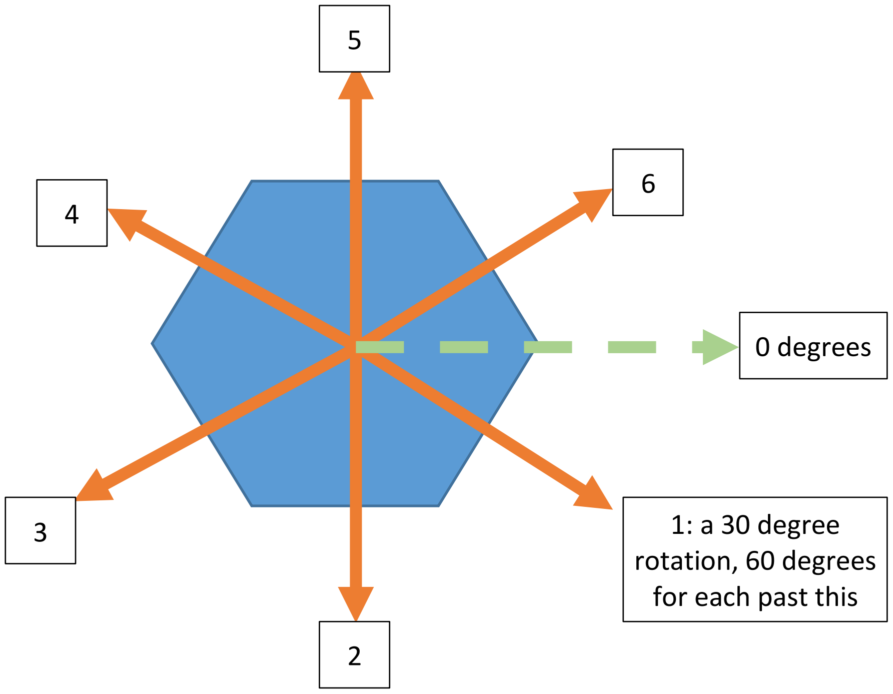
The square version of this is largely the same as walking with pixels. Generate a random number between 1 and 4, and then move in that direction. The amount of the move will depend on the size of the step, which represents the length of a side of the square, as well as the step scale variable. Be sure to take into account whether the shapes should be constrained to the viewable area or not.

Typically with any sort of grid or arrangement of shapes, we think of those shapes as being centered on some point. The rect() function defaults to drawing the shape from its top-left corner. You can call the rectMode() function to specify that a rectangle should be drawn relative to its CENTER, instead of the top-left corner.

Random Walk - Hexagons

Drawing a hexagon is a bit more complex, as is determining where to move it.

To update it, generate a random number from 1 to 6 (since random() in Processing is float based, you can generate a number from 1-7 and truncate it to an int), and based on that number, move in a particular direction. For this assignment you should draw your hexagons such that they have flat tops, which means your movement would look something like this:



Why is the 1 different? Because a 0 degree rotation would be straight along the positive X axis, and from there to the first adjacent side is only a 30-degree rotation. From there, it is a 60-degree rotation to the next side (for a total of 90 degrees from the start), 60 more (for a total of 150) to the next, and so on.

With knowledge of the angle, and the radius, translating that into an X/Y offset is easy using the sin() and cos() functions. There’s a small catch, however! The radius is that of the circle which circumscribes the hexagon. If you were to step by just the radius, you would have overlapping shapes.

Instead… the distance between two regular hexagons set adjacent to one another is: $\sqrt{\mathbf{3}} * \boldsymbol{radius}$. This derives from the relationship between the vertical and horizontal distance between the centers of each hexagon, which form a 30-60-90 triangle. For everything you ever wanted to know about hexagons: https://www.redblobgames.com/grids/hexagons/

Drawing a hexagon can be done using the `beginShape()`, `endShape()` and `vertex()` functions. Take a look at the examples on the Processing website, or in some of the class demos. Like the squares, other details of this shape (the color and stroke) should be stored already and used accordingly.

Converting pixel/Cartesian coordinates to hexagonal coordinates

Floating-point numbers can suffer from a loss of precision, especially over time. The result of a formula may have a small rounding error in it, and feeding the output of that formula into the same formula could result in another small loss of precision, and so on…

With squares you should be fine, but the hexagonal calculations can cause problems, especially when you are trying to store a final (x, y) location in something like a HashMap. Over time, revisiting the location of a previous hexagon will likely have a slightly different (x, y) value and so will be considered a "new" visit to that location.

By converting some (x, y) Cartesian coordinates to hexagonal coordinates, this floating-point error will be eliminated. The math for this can be rather complex, so a helper function is provided for you. This function is called **CartesianToHex()**, and an example of its usage is provided in the sketch file, **hexTest.pde**.

Submissions

Create a .zip file with any code files you created for this project (in Processing they are files with the extension .pde), and name the file ***LastName.FirstName.Project2.zip***. Submit the .zip file on the Canvas page for Project 2.

Where could you take this next?

This could be used as the start of a terrain generator, or a level generator for a video game. The data created could be saved to a file for later use, or fed into another system to further refine the shapes that were created.

Tips

- Use the reference page! Processing has a very thorough reference at https://processing.org/reference/
- Ditto for ControlP5. The reference and example pages are very thorough. Using a new tool or library can initially be a bit overwhelming, but step by step you'll learn more about it.
- Tackle one thing at a time. Even in a small project such as this, breaking it into smaller pieces can help you to make sense of the larger application.
- When rendering things with different properties, it's helpful to have each thing be self-contained—each thing should set its own rendering properties, to ensure it will still work if something else changed those properties.
- Don't be afraid to create throwaway prototypes! Try out some idea in another sketch to see if it works, then integrate that into your larger application. This can be especially helpful if you have some bug or something else in your project is preventing you from testing if a new addition is working or not.

Grading

Item	Description	Maximum Points
UI	Components present and functional	10
Squares	Squares positioned properly according to all size and step scale properties	10
Hexagons	Hexagons positioned properly according to all size and step scale properties	20
Color	Buildup of color implemented based on number of "visits" to a particular location	20
Constraints	Steps are constrained to the drawable area if this option is turned on	10
Code Structure	Main functionality broken into separate classes: abstract base class, and two derived classes for the square and hexagonal walks	20
Random seed	Reproduction of particular patterns possible with a specific random seed value	10
	Total	100