Technical Questionnaire

Willum Bolt

1)

To simplify the process of adding more messages, the text for each popup has been stored in a list. Adding more strings to the list causes the list size to increase. The list size is a value that can be retrieved and used as the maximum value in a random integer generator. This random integer can then be used to reference a specific position in the list, allowing it to be printed. This implementation allows for the quick addition of new popups as it only requires a new line of text to be added to the list.

C# implementation.

```
void ShowRandomMessage(){
        List<string> popUpList = new List<string>{
            "CDC: swine flu doesn't come from eating pork",
            "CDC pioneers new anti-malarial strategies",
            "CDC warns of drug resistant 'nightmare' bacteria",
            "CDC finds 'frequent' fecal contamination in pools",
            "CDC preparedness funds hit by proposed 2014 budget"
        };
        int popUpID = Random.Range(0,popUpList.Count);
        ShowPopup(popUpList[popUpID]);
    }
```

To keep code tidy and easily maintainable, the strings could be separated out into a separate text file. This could then be loaded into the list.

2)

The below approaches would both be able to achieve the goal of producing the outline of a mesh. The shader approach is much more versatile and would ideally be the one to use in this procedural scenario as it wouldn’t require the calculations to determine the outer edges of the mesh.

Shader – Ideal Approach

Creating a unity shader would be a method for creating an outline and would allow for flexibility in altering the appearance of the border. This approach would take the procedurally generated 2D mesh, scale it upwards in both the x and y axis and then re-render it behind the procedural 2D mesh. This process would create an outline that could be increased in thickness by increasing or decreasing the scaling value of the 2D mesh.

Unity has a variety of tools that enables blending with shaders, with a variety of blend modes that can allow for the modification of channels such as RGB and the alpha channel. This could be applied to the 2D mesh outline. Because the shader is essentially copying the 2D mesh that has been procedurally generated, the vertices of the mesh can be utilized as points at which colours differ. This vertex shading approach would allow for gradients to be achieved.

This approach shouldn’t have significant impacts on performance.

Line Renderer -Alternative Approach

Though the shader approach would ideally be used, I have included the line render approach as it is a possible alternative in the case that using shaders is not possible.

Assuming it is possible to find out what vertices make up the outer edges of the procedurally generated mesh it could be possible to create the outline using unity’s built in line renderer. The thickness of the line can be changed, and it also allows for the thickness of the border to vary at different points along the line.

Line Renderer has the functionality to allow for gradients by enabling the renderer to have multiple unity materials. Because line renderer uses unity’s materials, many of the line’s visual properties can be altered.

If the procedurally generated mesh remains stationary during runtime then the vertices of the line renderer would only need to be calculated once, making the renderer have a very small impact on the applications performance.

3)

Given points P1 and P2 are 2D points and velocity V is a 2D vector. P2 = P1 + (V * t) where t is the elapsed time.

P2 = P1 + (V* t)

Example:

P1 = (2,3),V = (2,-1) and T = 5.

5*2 = 10 5 * -1 = -5 2 + 10 = 12 3 + -5 = -2

After elapsed time 5, the entity that started at P1(2,3) is now at P2(12,-2).

4)

Given |A| and *theta* are known, the x and y components of A are given as follows:

x = |A|cos(*theta*)

y = |A|sin(*theta*)

5)

Given the points A and B, the gradient can be calculated by finding the change in y and dividing by the change in x. dy/dx. For AB this will be called g. The perpendicular gradient to this would be -1/g. Using y = mx + c where m is the gradient -1/g and the x and y points of A, the c value can be calculated. C = y – mx. Because c and m are now known it is possible to find any point on this perpendicular line by choosing an x value and calculating y or re-arranging y = mx + c to calculate x from a y value.

6)

By using a queue to store the digits to be used in the moving average calculation there is no need to store every value that has been passed in, only those that would exist in the current window. Keeping track of the current sum of the values in the window allows for the average to be computed by dividing by N (the size of the window).

To add a value and compute the new moving average first a digit is added to the queue, this value is then added to the sum of values. The first item in the queue is then subtracted from the total sum of values and then removed from the queue. As it is known how many values are in the queue the average can be calculated using *currentSum / N,* where N is the number of digits in the window.

Because the number of operations needed to compute the average is not dependent on the number N digits, the run-time complexity of this solution is O(1), constant. In terms of storage, the class only needs to store N digits, with the oldest digit leaving first leaving room for the newest due to the way the queue has been implemented.

```
public class MovingAverage
{
    private int computeRange; //N digits to compute
    private int count = 0; //Current queue size
    private Queue<float> values = new Queue<float>(); //queue of values
    private float currentSum; //Current sum of values
    private float currentAverage; //Current average of values

    //Constructor accepts size of window
    public MovingAverage(int N){
        computeRange = N;
    }
    //Appends value and returns new average
    public float GetNewMovingAverage(float value){
        AddValue(value);
        return GetAverage();
    }
    //Returns the current average of the window
    public float GetAverage(){
        float average;
        if(count == computeRange){
            average = currentSum / computeRange;
        }else{
            average = currentSum / count;
        }
        return average;
    }
    //Adds a value to the values for the average
    public void AddValue(float value){
        if(values.Count < computeRange){
        AddToQueue(value);
        count++;
        }else{
        AddToQueue(value);
        RemoveFromQueue();
        }
    }
    //Adds a value to the queue and updates sum of window
    private void AddToQueue(float value){
        values.Enqueue(value);
        currentSum += value;
    }
    //Removes value from the queue and updates sum of window
    private void RemoveFromQueue(){
        currentSum -= values.Peek();
        values.Dequeue();
    }
}
```

7)

The two approaches below tackle the issue of randomly selecting traits with different mutation chances whilst ensuring each trait cannot be chosen more than once and that each category is halved in probability each time it is chosen from. Approach 2 is faster but separates the traits out into category lists.

Both approaches utilize the cumulative sum of probabilities approach to determine what trait is selected. However, Approach 2 only requires the total sum of a category's traits' mutation_chance to be fully calculated 3 times (once for each category).

Both have a run-time complexity of O(N) however whilst Approach 1 only performs the random probability selection once per day (selecting based on mutation_chance * category_chance), it does require the total sum of chances to be re-calculated each day.

Because Approach 2 separates out the category and the trait selection there is no need to recalculate the sum of probabilities for the traits each time, despite a categories chance halving. There is no longer a need to cycle through the traits and sum them together. Due to the way Approach 2 is structured, assuming the sum of the traits has already been calculated, the algorithm only has to step through the trait list once to find out the selected trait (in the process of subtracting from the random number).

Approach 1 – Selecting category and trait together

Assumptions made:

- ^{1}The category chance/probability is stored, starting at a value of 1 - category_chance
- ^{2}All traits are being stored in a list to allow for the removal of a trait once already selected. It is possible to work out from a trait's properties what category it is in.

Stages carried out in a day:

1. Calculate the cumulative sum of the probabilities of picking each trait using: category_chance * mutation_chance
2. Pick a random_number between 0 and the sum of all probabilities.
3. Step through each element in the list of traits, subtracting their mutation_chance from the random_number until it is less than 0. The current trait is the selected trait.
4. Half the category_chance for the selected trait, resultantly altering the probability calculation for all traits in this category.
5. Remove the trait from the list of traits, removing its ability to be chosen again.
6. Return to step 1 in preparation for next day.

Approach 2 – Selecting category and then trait

Assumptions made:

- The traits are being stored in three different lists.
- Same as the second two points in the approach above $^{1+2}$ but with three distinct lists instead.
- The values for category_chance is being stored in a 1-dimensional array of length 3, starting at a value of 1 for each position in the array.

Stages:

1. Calculate the sum of the category_chance array.
2. Pick a random_number between 0 and the sum of the category_chance array.
3. Step through each element in the list of traits, subtracting their probability from the random_number until the value is less than 0. This is the selected category.
4. If this is the first pass for this category, calculate the sum of the mutation_chance for the traits in this category and store (category_sum), else skip to step 5.
5. Pick a random_number between 0 and the sum of this category's traits' mutation_chance.
6. Step through each element in the list of traits, subtracting their probability from the random_number until it is less than 0. Once 0 is passed the current trait is the selected trait.
7. Subtract the selected trait's probability from the category_sum for this category.
8. Remove the trait from the list.
9. Half the category_chance for the selected category.
10. Return to step 1 in preparation for next day.

8 Bonus)

```
std::string GetDate() {
	std::stringstream ss;
	ss << "January 2018";
	return ss.str();
}
const char* date = GetDate().c_str();
printf("Today it's: %s\n", date);
```

A line in the above code that could potentially cause a crash, depending on what else is running in the application and how quickly memory is being re-used, is ‘const char* date = GetDate().c_str()’. Because GetDate() returns a string ‘ss.str()’ which only exists at this time, because it’s never assigned to a variable. The line of code declaring the constant character pointer ends up pointing to a memory location that is no longer dedicated to that string. If for any reason the data in this location is overwritten the *date* variable would no longer be valid.

This is why, when running just this snippet of code, there appears to be no issue and no crash. Because no other lines of code have attempted to access the supposedly free space in memory that the ss.str() line used.