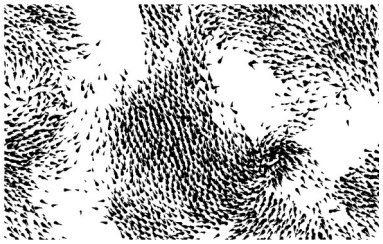
FIZ425E/ Computational Analysis of Physical Systems 17.05.2023

Gökay AKÇAY

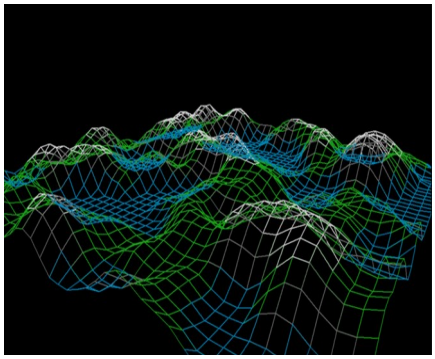
SPRING 2023 / Project Report
Generating Procedural Terrains in 2D/3D using Perlin Noise Mapping

Introduction:

Perlin noise is a type of a gradient noise which is been used in many fields from game and VR technologies to motion picturing in cinema field. This method was developed by Ken Perlin in 1983, working on animated motion picture movie, Tron, at Disney company. It created groundbreaking effect in the field of image and texture like generating coherent, natural-looking patterns and textures. It is often used to create realistic simulations of natural phenomena such as clouds, waves, wind and terrain.



Boids / Flow Mechanics



Procedeural Terrains

Perlin Clouds

This project will be about the application of the Perlin Noise mapping to mesh-grid systems. The project can be specified in two parts:

- How to Generate Perlin Noise Map? Implementation to Code using Python
- Converting 2D Perlin Map into 3D using OpenGL library (PyOpenGL,Pygame) and creating procedural terrains.
- Visual Improvements using Tkinter GUI.

I - How to Generate Perlin Noise Map & Implementation to Code using Python:

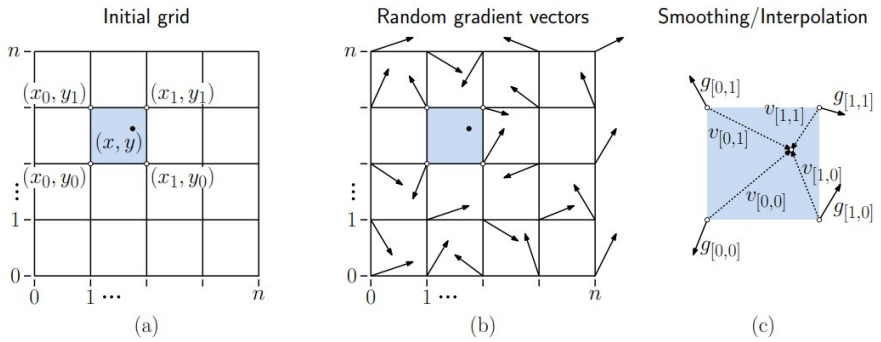


Figure (I)

To most noticing part which differs Perlin noise maps from the simple noise map is the smoothing process. By using different interpolation methods like cubic, linear etc.. the smooth transitions between the points can be created. To do that we have to create a matrix of finite sample random values. Then generate noise function that makes interpolation between each of these values. To make an interpolation between these values in 2D the gradient vectors must be used. The gradient vectors calculates the slope of a surface in 2D or 3D in terms of x,y,z.

As shown in the figure (I) to create a smooth transitions between points. Following instructions can be used.

I) Select a point in the matrix of random values. Imagine that this point is inside a square.

II) Create 4 random gradient vectors and displacement vectors each corners of the square. Displacement vector is defined below:

$$v_{[0,0]} = (x, y) - (x_0, y_0) \quad \text{and} \quad v_{[0,1]} = (x, y) - (x_0, y_1)$$

Here x_0 and y_0 is the selected point in the matrix.

III) To calculate the scalar displacement values, multiply the gradient of each corners with the unit vectors of x and y. Thus, the maximization or the minimization can be found. This is basically refers how randomly created gradient vector and displacement vector placed to each other. Are they Orthanormal or not?

$$\delta_{[0,0]} = (v_{[0,0]} \cdot g_{[0,0]}) \quad \text{and} \quad \delta_{[0,1]} = (v_{[0,1]} \cdot g_{[0,1]})$$

IV) However, on the equation above the distance of the points from the center is not taken into account. We want to apply the gradient effect closer to the each vertex. However, it neglects the proximity of the points to the vertices. To make that Perlin developed fade function below: (This is a improved version of a cubic interpolation)

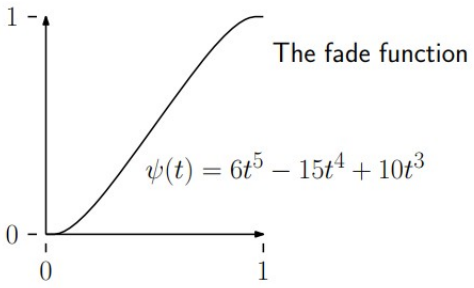


Fig. 2: The fade function.

V) When all values are calculated. The noise can be created with the function below:

$$\text{noise}(x, y) = \Psi(1 - x, 1 - y)\delta_{[0,0]} + \Psi(x, 1 - y)\delta_{[1,0]} + \Psi(1 - x, y)\delta_{[0,1]} + \Psi(x, y)\delta_{[1,1]}.$$

VI) Select again random number in permutation table and apply the same operations. In **I, II, III** Briefly, it is shown in step **V**

Implementation to the Code:

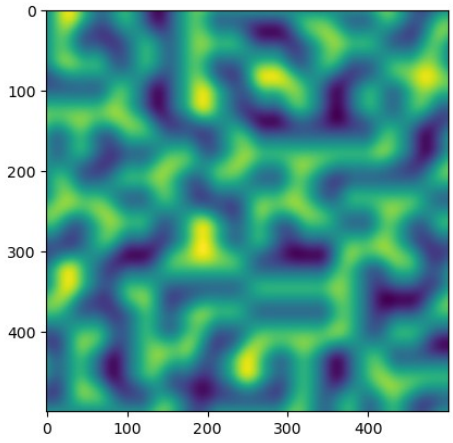
```
import numpy as np

# This class creates perlin noise map.
class TerrainModel:
    @staticmethod
    def lerp(a, b, x):
        # Linearly interpolates between two points. In this model
        return a + x * (b - a)

    @staticmethod
    def fade(f):
        # Apply fading to displacement vector. Decide whether it is close to the vertices or not.
        return 6 * f ** 5 - 15 * f ** 4 + 10 * f ** 3

    @staticmethod
    def gradient(c, x, y):
        # These are the displacement vectors of our corners from the selected point.
        vectors = np.array([[0, 1], [0, -1], [1, 0], [-1, 0]])
        gradient_co = vectors[c % 4]

        return gradient_co[:, :, 0] * x + gradient_co[:, :, 1] * y

    @staticmethod
    def perlin(x, y, seed=0):

        np.random.seed(seed)
        # Create values from 0 to 2^n. Here 2^n defined as 128.
        ptable = np.arange(128, dtype=int)
```

```
        # To make a random permutation table shuffle the array.
        np.random.shuffle(ptable)

        ptable = np.stack([ptable, ptable]).flatten()

        # Grid coordinates.
        xi, yi = x.astype(int), y.astype(int)
        # Displacement vectors.
        xg, yg = x - xi, y - yi
        # Apply fading operation to the displacement vectors.
        xf, yf = TerrainModel.fade(xg), TerrainModel.fade(yg)
        # Create a gradient vectors for upper right, upper left, lower right and lower left. Then multiply it by
displacement vectors.
        n00 = TerrainModel.gradient(ptable[ptable[xi] + yi], xg, yg)
        n01 = TerrainModel.gradient(ptable[ptable[xi] + yi + 1], xg, yg - 1)
        n11 = TerrainModel.gradient(ptable[ptable[xi + 1] + yi + 1], xg - 1, yg - 1)
        n10 = TerrainModel.gradient(ptable[ptable[xi + 1] + yi], xg - 1, yg)

        # Apply interpolation horizontally (bottom left corner, bottom right corner)
        x1 = TerrainModel.lerp(n00, n10, xf)
        x2 = TerrainModel.lerp(n01, n11, xf)

        # Find the height value (it will be the height in 3D mesh)
        return TerrainModel.lerp(x1, x2, yf)
```

I named the class as Terrain Model because while adapting it to 3D ,the terrain will be created by using these TerrainModel methods. I also defined all the methods as statictype because gradient, fade and lerp methods can be used single. However, the main method which triggers to these methods is “Perlin” method. We are going to use only “Perlin” method in later part.

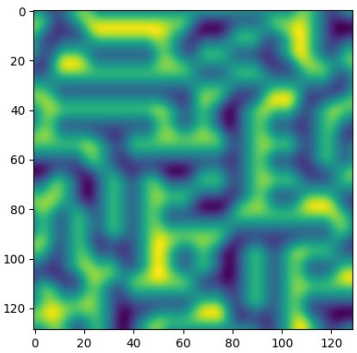
Functionality of the methods is explained with comment lines in the code.

Here is the visual output of the TerrainModel class.

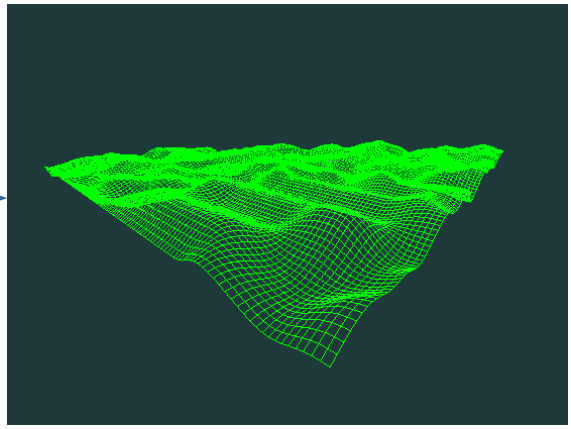


(500x500) Perlin Noise Map

II - Converting 2D Perlin Map into 3D using OpenGL library and creating procedural terrains.



Aim



In this section, we aim to convert our 2D Perlin into 3D openGL surface. This part is much easier than our previous part. To shortly explain the steps I did:

I – I defined all the points in (*n x n)* vertices matrix.

II – Then connected related vertices to each other and put them into edges matrix.

III – I iterated each element in the edge matrix and drew each edges using OpenGL library.

I explained it more detailed in my code block:

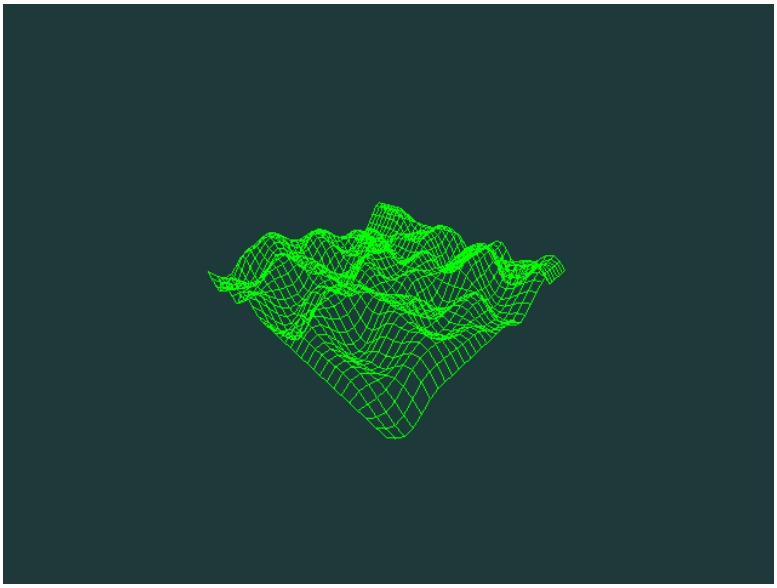
```
#This part creates a openGL frame to draw our mesh. Initialization values like frame size, lighting type, color type, perspective view, frame time etc are defined here.
class OpenGlFrame:

    #Initilization of OpenGL Panel
    def __init__(self,cell_size,seed,resolution):

        pg.init()
        display = (640,480)
        pg.display.set_mode(display,pg.OPENGL|pg.DOUBLEBUF)

        self.seed = seed
        self.cell_size = cell_size
        self.resolution = resolution
        self.clock = pg.time.Clock()
        self.display = display

        self.mainLoop(display)

    # Main operations like drawing the terrain and updating by each time frame are made here.
    def mainLoop(self,display):

        glClearColor(0.1, 0.2, 0.2, 1)

        gluPerspective(90, display[0] / display[1], 0.1, 120.0)
```

```
        glTranslatef(0, -10, -20)

        glRotatef(30, 40, 0, 0)
        glRotatef(135, 0, 20, 0)

        running = True

        terrain = TerrainMesh(self.cell_size,self.seed,self.resolution)

        #In this loop frame is updated each defined waiting time here it is 10 miliseconds.
        while (running):

            for event in pg.event.get():
                if (event.type == pg.QUIT):
                    running = False

            # Rotate our terrain around itself 4 degree in each iteration.
            glRotatef(4, 0, 15, 0)
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

            #Green wireframe color
            glColor3f(0, 1, 0)

             #Render the terrain in each 10 ms.
            terrain.render()

            pg.display.flip()
            pg.time.wait(10)

        self.quit()

    def quit(self):
        pg.quit()

#Creating the Mesh using vertices and edge matrices, and defining them to OpenGL
class TerrainMesh:

    def __init__(self, cell_size=10, seed=32, terrain_resolution=10):
        self.cell_size = cell_size
        self.seed = seed
        self.terrain_resolution = terrain_resolution
        self.update_terrain()

    def update_terrain(self):

        # If the terrain resolution increase, there will be more interpolation between points.
        # Briefly, high resolution causes high noise.
        lin_array = np.linspace(1, self.terrain_resolution, self.cell_size + 1, endpoint=False)

        #Make this linear interpolation x and y
        x, y = np.meshgrid(lin_array, lin_array)

        # We created our perlin noise with this code.
        _terrain_model = TerrainModel.perlin(x, y, seed=self.seed)
```

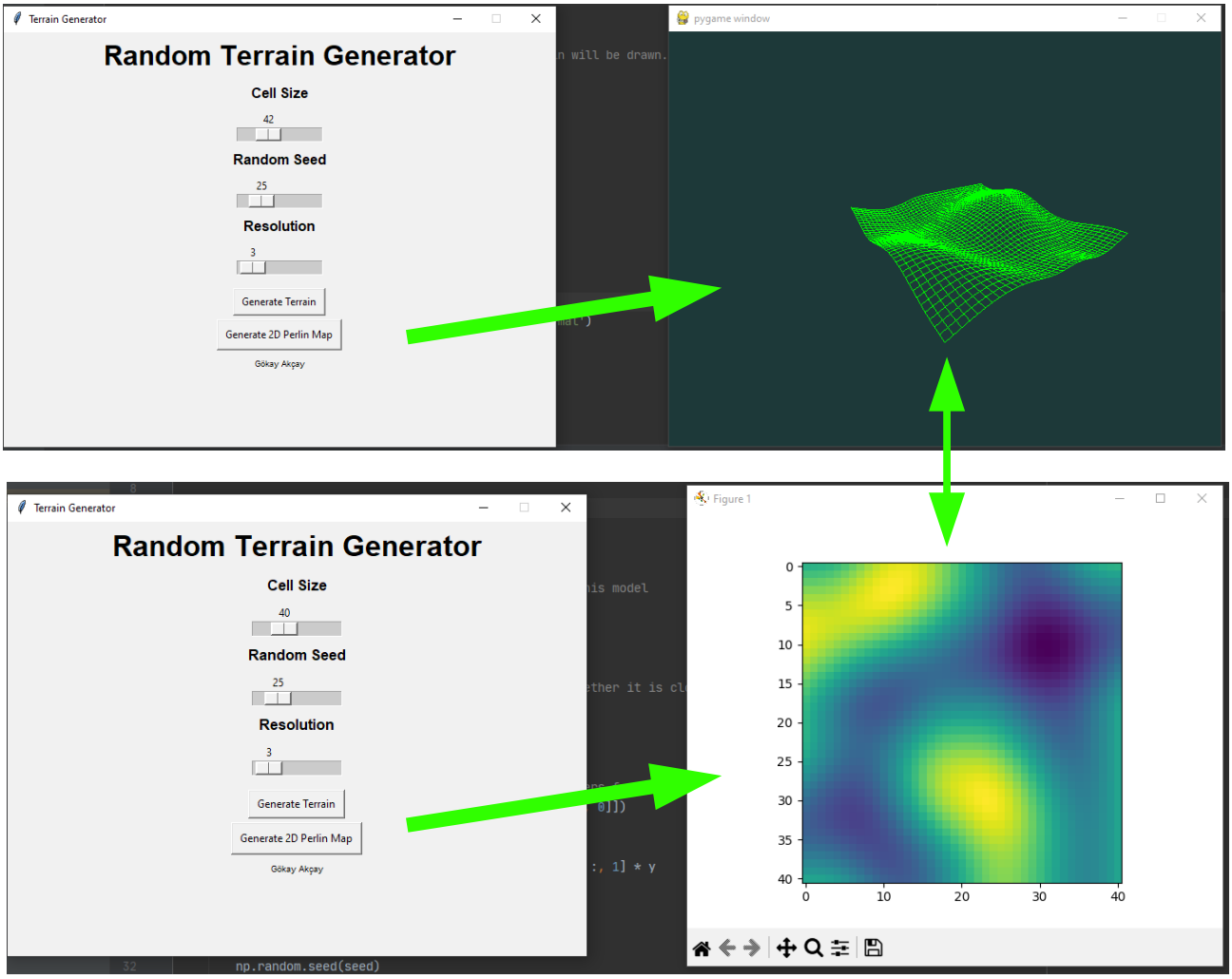
```
        # This is to exaggeerate values and height. Can be changed 8 is set as a default value.
        _terrain_model *= 8

        # First create matrix of empty points. If the cell size 4 x 4 that means there should be 5 x 5 points.
        _vertices = np.zeros(((self.cell_size + 1) ** 2, 3))



        x_idx = 0
        z_idx = 0

        ##First create the vertices
        for i in range(0, _vertices.shape[0]):

            # Creates points. x_idx = x position in space, terrain model adjusts the height, z_idx is z posiiton in space.
            # Do it for each iterations and adds into _vertices matrix
            _vertices[i] = [x_idx, _terrain_model[z_idx, x_idx], z_idx]
            x_idx += 1
            # It creates the vertices in row order. So, if it is end of the row then reset the x_idx to 0 to prevent
overflow.
            if (x_idx > (self.cell_size)):
                x_idx = 0
                z_idx += 1

        # We've created the vertices so far. It is time to connect them each other.
        _edges = np.zeros(((self.cell_size + 1) * (self.cell_size) * 2, 2), dtype="int")



        # This loop binds points to each other horizontally and vertically.
        while (True):

            edge_index = 0

            # In even indexes the edges is bound horizontally.
            for e in range(0, _edges.shape[0], 2):

                _edges[e] = (edge_index, edge_index + 1)

                edge_index += 1

                if (edge_index % (self.cell_size + 1) == self.cell_size):
                    edge_index += 1

            edge_index = 0

            # In odd indexes the edges is bound vertically.
            for k in range(1, _edges.shape[0], 2):

                # Here , edge_index + self.cell_size + 1 refers the the point above the selected point.
                _edges[k] = (edge_index, edge_index + self.cell_size + 1)

                edge_index += 1

                # Since there is no upper point in the last line. Break the loop.
                if (edge_index + self.cell_size >= (self.cell_size + 1) ** 2):
```

```
                break

            break
        print(_edges.shape)

        self.edges = _edges
        self.vertices = _vertices


    # For a better optimization terrain is only rendered when this method is called.
    # Since there will be a lot of calculations in high cell sizes, I preferred to have it calculated when necessary rather than having it done every time.
    def render(self):
        self.update_terrain()
        glBegin(GL_LINES)
        for edge in self.edges:
            for vertex in edge:
                glVertex3fv(self.vertices[vertex])
        glEnd()

    #To destroy drawn vertices and edges. Not used yet.
    def destroy(self):
        glDeleteVertexArrays(1, (self.vao,))
        glDeleteBuffers(1, (self.vbo,))


# When this method is called, the openGL frame will be drawn. This will be used in Ttkinter GUI. Whenever the button is clicked this method will be called.
def showTerrain(cell_size,seed,resolution):
    if __name__ == '__main__':
        terrain_frame = OpenGlFrame(cell_size,seed,resolution)
```

Output:



III – Visual Improvements using Tkinter GUI,



Since the technical part of our code is finished, we need to make it ready for visual presentation. I used Tkinter GUI to make it. We used parameters in terrain model like cell size, seed and linear interpolation values (resolution) in previous parts. To enable the user to set these initializing values I created 3 sliders for each. I added another button called Generate 2D Perlin Map which shows it on Matplotlib heatmap. So that, the user can see the structure of the terrain before moving using it on 3D.

Here is my code:

```
#Creating the Mesh using vertices and edge matrices, and defining them to OpenGL
class TerrainMesh:

    def __init__(self, cell_size=10, seed=32, terrain_resolution=10):
        self.cell_size = cell_size
```

```
        self.seed = seed
        self.terrain_resolution = terrain_resolution
        self.update_terrain()

    def update_terrain(self):

        # If the terrain resolution increase, there will be more interpolation between points.
        # Briefly, high resolution causes high noise.
        lin_array = np.linspace(1, self.terrain_resolution, self.cell_size + 1, endpoint=False)

        #Make this linear interpolation x and y
        x, y = np.meshgrid(lin_array, lin_array)

        # We created our perlin noise with this code.
        _terrain_model = TerrainModel.perlin(x, y, seed=self.seed)

        # This is to exaggeerate values and height. Can be changed 8 is set as a default value.
        _terrain_model *= 8

        # First create matrix of empty points. If the cell size 4 x 4 that means there should be 5 x 5 points.
        _vertices = np.zeros(((self.cell_size + 1) ** 2, 3))



        x_idx = 0
        z_idx = 0

        ##First create the vertices
        for i in range(0, _vertices.shape[0]):

            # Creates points. x_idx = x position in space, terrain model adjusts the height, z_idx is z posiiton in space.
            # Do it for each iterations and adds into _vertices matrix
            _vertices[i] = [x_idx, _terrain_model[z_idx, x_idx], z_idx]
            x_idx += 1
            # It creates the vertices in row order. So, if it is end of the row then reset the x_idx to 0 to prevent
overflow.
            if (x_idx > (self.cell_size)):
                x_idx = 0
                z_idx += 1

        # We've created the vertices so far. It is time to connect them each other.
        _edges = np.zeros(((self.cell_size + 1) * (self.cell_size) * 2, 2), dtype="int")



        # This loop binds points to each other horizontally and vertically.
        while (True):

            edge_index = 0

            # In even indexes the edges is bound horizontally.
            for e in range(0, _edges.shape[0], 2):

                _edges[e] = (edge_index, edge_index + 1)

                edge_index += 1
```

```
                if (edge_index % (self.cell_size + 1) == self.cell_size):
                    edge_index += 1

            edge_index = 0

            # In odd indexes the edges is bound vertically.
            for k in range(1, _edges.shape[0], 2):

                # Here , edge_index + self.cell_size + 1 refers the the point above the selected point.
                _edges[k] = (edge_index, edge_index + self.cell_size + 1)

                edge_index += 1

                # Since there is no upper point in the last line. Break the loop.
                if (edge_index + self.cell_size >= (self.cell_size + 1) ** 2):
                    break

            break
        print(_edges.shape)

        self.edges = _edges
        self.vertices = _vertices


    # For a better optimization terrain is only rendered when this method is called.
    # Since there will be a lot of calculations in high cell sizes, I preferred to have it calculated when necessary
rather than having it done every time.
    def render(self):
        self.update_terrain()
        glBegin(GL_LINES)
        for edge in self.edges:
            for vertex in edge:
                glVertex3fv(self.vertices[vertex])
        glEnd()

    #To destroy drawn vertices and edges. Not used yet.
    def destroy(self):
        glDeleteVertexArrays(1, (self.vao,))
        glDeleteBuffers(1, (self.vbo,))


# When this method is called, the openGL frame will be drawn. This will be used in Ttkinter GUI. Whenever the
button is clicked this method will be called.
def showTerrain(cell_size,seed,resolution):
    if __name__ == '__main__':
        terrain_frame = OpenGlFrame(cell_size,seed,resolution)

## This part will be about Tkinter GUI.

# Initialization values of the Tkinter GUI.
root = tk.Tk()
root.geometry('640x480')
root.resizable(False, False)
root.title('Terrain Generator')
```

```
# Explanations of adjustments like cell size and seed etc..

#Title text
project_title_text = Label(root,text ="Random Terrain Generator", font='Helvetica 24 bold')
project_title_text.pack(ipady=5)

cell_text = Label(root,text ="Cell Size", font='Helvetica 12 bold')
cell_text.pack(ipady=5)

# For setting the cell size I used a scaler, It is yet 4 to 128 due to lack of optimization.
cell_size_scale = Scale(root, from_=4, to=128,orient = "horizontal")
cell_size_scale.pack()

seed_text = Label(root,text ="Random Seed", font='Helvetica 12 bold')
seed_text.pack(ipady=5)

# Seed is can be also set up by scaler.
seed_scale = Scale(root, from_=1, to=128,orient = "horizontal")
seed_scale.pack()

terrain_res_text = Label(root,text ="Resolution", font='Helvetica 12 bold')
terrain_res_text.pack(ipady=5)

# Terrain res is can be also set up by scaler.
terrain_res_scale = Scale(root, from_=1, to=40,orient = "horizontal")
terrain_res_scale.pack(ipady=5)

# Get the values from slider values. With these values create the terrain in OpenGL frame.
def generate_button_clicked():
    showTerrain(cell_size_scale.get(),seed_scale.get(),terrain_res_scale.get())

# Optionally, top view of the terrain can be seen with this method. It creates a matplotlib frame and creates a
heatmap of perlin noise in 2D.
def perlin_map2D_button_clicked():
    TerrainModel.getTerrain2DView(cell_size_scale.get(), seed_scale.get(),terrain_res_scale.get())

# When this button is clicked, the terrain will be generated.
generate_terrain_button = tk.Button(
    root,
    text='Generate Terrain',
    command=lambda: generate_button_clicked()
)

generate_terrain_button.pack(
    ipadx=5,
    ipady=3,
    pady = 2,
    expand=False
)

generate_terrain_button.pack(
    ipadx=5,
    ipady=3,
)

# When this button is clicked, perlin map of the generated terrain will be drawn.
```

```
perlin_map2D_button = tk.Button(
    root,
    text='Generate 2D Perlin Map',
    command=lambda: perlin_map2D_button_clicked()
)

perlin_map2D_button.pack(
    ipadx=5,
    ipady=5,
    pady = 2,
    expand=False,
)

seed_text = Label(root,text ="Gökay Akçay", font='Helvetica 7 normal')
seed_text.pack(ipady=5)


# Renders the Tkinter Frame.
root.mainloop()
```

The working code (all parts included):

It is the final version of the project. Only this code works

```
# Gökay Akçay 090200147

import pygame as pg
import numpy as np
from OpenGL.GL import *
from OpenGL.raw.GLU import gluPerspective
import matplotlib.pyplot as plt
import tkinter as tk
from tkinter import Scale, Label

# This class creates perlin noise map.
class TerrainModel:
    @staticmethod
    def lerp(a, b, x):
        # Linearly interpolates between two points. In this model
        return a + x * (b - a)

    @staticmethod
    def fade(f):
        # Apply fading to displacement vector. Decide whether it is close to the vertices or not.
        return 6 * f ** 5 - 15 * f ** 4 + 10 * f ** 3

    @staticmethod
    def gradient(c, x, y):
        # These are the displacement vectors of our corners from the selected point.
        vectors = np.array([[0, 1], [0, -1], [1, 0], [-1, 0]])
        gradient_co = vectors[c % 4]

        return gradient_co[:, :, 0] * x + gradient_co[:, :, 1] * y
```

```
    @staticmethod
    def perlin(x, y, seed=0):

        np.random.seed(seed)
        # Create values from 0 to 2^n. Here 2^n defined as 128.
        ptable = np.arange(128, dtype=int)

        # To make a random permutation table shuffle the array.
        np.random.shuffle(ptable)

        ptable = np.stack([ptable, ptable]).flatten()

        # Grid coordinates.
        xi, yi = x.astype(int), y.astype(int)
        # Displacement vectors.
        xg, yg = x - xi, y - yi
        # Apply fading operation to the displacement vectors.
        xf, yf = TerrainModel.fade(xg), TerrainModel.fade(yg)
        # Create a gradient vectors for upper right, upper left, lower right and lower left. Then multiply it by
displacement vectors.
        n00 = TerrainModel.gradient(ptable[ptable[xi] + yi], xg, yg)
        n01 = TerrainModel.gradient(ptable[ptable[xi] + yi + 1], xg, yg - 1)
        n11 = TerrainModel.gradient(ptable[ptable[xi + 1] + yi + 1], xg - 1, yg - 1)
        n10 = TerrainModel.gradient(ptable[ptable[xi + 1] + yi], xg - 1, yg)

        # Apply interpolation horizontally (bottom left corner, bottom right corner)
        x1 = TerrainModel.lerp(n00, n10, xf)
        x2 = TerrainModel.lerp(n01, n11, xf)

        # Find the height value (it will be the height in 3D mesh)
        return TerrainModel.lerp(x1, x2, yf)

    @staticmethod
    def getTerrain2DView(cell_size, seed=0,terrain_resolution = 10):
        lin_array = np.linspace(1, terrain_resolution, cell_size + 1, endpoint=False)
        x, y = np.meshgrid(lin_array, lin_array)

        _terrain_model = TerrainModel.perlin(x, y, seed=seed)
        _terrain_model *= 8
        plt.imshow(_terrain_model, origin='upper')
        plt.show()


#This part creates a openGL frame to draw our mesh. Initialization values like frame size, lighting type, color
type, perspective view, frame time etc are defined here.
class OpenGlFrame:

    #Initilization of OpenGL Panel
    def __init__(self,cell_size,seed,resolution):

        pg.init()
        display = (640,480)
        pg.display.set_mode(display,pg.OPENGL|pg.DOUBLEBUF)

        self.seed = seed
        self.cell_size = cell_size
```

```
        self.resolution = resolution
        self.clock = pg.time.Clock()
        self.display = display

        self.mainLoop(display)

    # Main operations like drawing the terrain and updating by each time frame are made here.
    def mainLoop(self,display):

        glClearColor(0.1, 0.2, 0.2, 1)

        gluPerspective(90, display[0] / display[1], 0.1, 120.0)

        glTranslatef(0, -10, -20)

        glRotatef(30, 40, 0, 0)
        glRotatef(135, 0, 20, 0)

        running = True

        terrain = TerrainMesh(self.cell_size,self.seed,self.resolution)

        #In this loop frame is updated each defined waiting time here it is 10 miliseconds.
        while (running):

            for event in pg.event.get():
                if (event.type == pg.QUIT):
                    running = False

            # Rotate our terrain around itself 4 degree in each iteration.
            glRotatef(4, 0, 15, 0)
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

            #Green wireframe color
            glColor3f(0, 1, 0)

            terrain.render()
            pg.display.flip()
            pg.time.wait(10)

        self.quit()

    def quit(self):
        pg.quit()

#Creating the Mesh using vertices and edge matrices, and defining them to OpenGL
class TerrainMesh:

    def __init__(self, cell_size=10, seed=32, terrain_resolution=10):
        self.cell_size = cell_size
        self.seed = seed
        self.terrain_resolution = terrain_resolution
        self.update_terrain()

    def update_terrain(self):
```

```
        # If the terrain resolution increase, there will be more interpolation between points.
        # Briefly, high resolution causes high noise.
        lin_array = np.linspace(1, self.terrain_resolution, self.cell_size + 1, endpoint=False)

        #Make this linear interpolation x and y
        x, y = np.meshgrid(lin_array, lin_array)

        # We created our perlin noise with this code.
        _terrain_model = TerrainModel.perlin(x, y, seed=self.seed)

        # This is to exaggeerate values and height. Can be changed 8 is set as a default value.
        _terrain_model *= 8

        # First create matrix of empty points. If the cell size 4 x 4 that means there should be 5 x 5 points.
        _vertices = np.zeros(((self.cell_size + 1) ** 2, 3))



        x_idx = 0
        z_idx = 0

        ##First create the vertices
        for i in range(0, _vertices.shape[0]):

            # Creates points. x_idx = x position in space, terrain model adjusts the height, z_idx is z posiiton in space.
            # Do it for each iterations and adds into _vertices matrix
            _vertices[i] = [x_idx, _terrain_model[z_idx, x_idx], z_idx]
            x_idx += 1
            # It creates the vertices in row order. So, if it is end of the row then reset the x_idx to 0 to prevent
overflow.
            if (x_idx > (self.cell_size)):
                x_idx = 0
                z_idx += 1

        # We've created the vertices so far. It is time to connect them each other.
        _edges = np.zeros(((self.cell_size + 1) * (self.cell_size) * 2, 2), dtype="int")



        # This loop binds points to each other horizontally and vertically.
        while (True):

            edge_index = 0

            # In even indexes the edges is bound horizontally.
            for e in range(0, _edges.shape[0], 2):

                _edges[e] = (edge_index, edge_index + 1)

                edge_index += 1

                if (edge_index % (self.cell_size + 1) == self.cell_size):
                    edge_index += 1

            edge_index = 0
```

```
            # In odd indexes the edges is bound vertically.
            for k in range(1, _edges.shape[0], 2):

                # Here , edge_index + self.cell_size + 1 refers the the point above the selected point.
                _edges[k] = (edge_index, edge_index + self.cell_size + 1)

                edge_index += 1

                # Since there is no upper point in the last line. Break the loop.
                if (edge_index + self.cell_size >= (self.cell_size + 1) ** 2):
                    break

            break
        print(_edges.shape)

        self.edges = _edges
        self.vertices = _vertices


    # For a better optimization terrain is only rendered when this method is called.
    # Since there will be a lot of calculations in high cell sizes, I preferred to have it calculated when necessary rather than having it done every time.
    def render(self):
        self.update_terrain()
        glBegin(GL_LINES)
        for edge in self.edges:
            for vertex in edge:
                glVertex3fv(self.vertices[vertex])
        glEnd()

    #To destroy drawn vertices and edges. Not used yet.
    def destroy(self):
        glDeleteVertexArrays(1, (self.vao,))
        glDeleteBuffers(1, (self.vbo,))


# When this method is called, the openGL frame will be drawn. This will be used in Ttkinter GUI. Whenever the button is clicked this method will be called.
def showTerrain(cell_size,seed,resolution):
    if __name__ == '__main__':
        terrain_frame = OpenGlFrame(cell_size,seed,resolution)

## This part will be about Tkinter GUI.

# Initialization values of the Tkinter GUI.
root = tk.Tk()
root.geometry('640x480')
root.resizable(False, False)
root.title('Terrain Generator')

# Explanations of adjustments like cell size and seed etc..

#Title text
project_title_text = Label(root,text ="Random Terrain Generator", font='Helvetica 24 bold')
project_title_text.pack(ipady=5)
```

```
cell_text = Label(root,text ="Cell Size", font='Helvetica 12 bold')
cell_text.pack(ipady=5)

# For setting the cell size I used a scaler, It is yet 4 to 128 due to lack of optimization.
cell_size_scale = Scale(root, from_=4, to=128,orient = "horizontal")
cell_size_scale.pack()

seed_text = Label(root,text ="Random Seed", font='Helvetica 12 bold')
seed_text.pack(ipady=5)

# Seed is can be also set up by scaler.
seed_scale = Scale(root, from_=1, to=128,orient = "horizontal")
seed_scale.pack()

terrain_res_text = Label(root,text ="Resolution", font='Helvetica 12 bold')
terrain_res_text.pack(ipady=5)

# Terrain res is can be also set up by scaler.
terrain_res_scale = Scale(root, from_=1, to=40,orient = "horizontal")
terrain_res_scale.pack(ipady=5)

# Get the values from slider values. With these values create the terrain in OpenGL frame.
def generate_button_clicked():
    showTerrain(cell_size_scale.get(),seed_scale.get(),terrain_res_scale.get())

# Optionally, top view of the terrain can be seen with this method. It creates a matplotlib frame and creates a
heatmap of perlin noise in 2D.
def perlin_map2D_button_clicked():
    TerrainModel.getTerrain2DView(cell_size_scale.get(), seed_scale.get(),terrain_res_scale.get())

# When this button is clicked, the terrain will be generated.
generate_terrain_button = tk.Button(
    root,
    text='Generate Terrain',
    command=lambda: generate_button_clicked()
)

generate_terrain_button.pack(
    ipadx=5,
    ipady=3,
    pady = 2,
    expand=False
)

generate_terrain_button.pack(
    ipadx=5,
    ipady=3,
)

# When this button is clicked, perlin map of the generated terrain will be drawn.
perlin_map2D_button = tk.Button(
    root,
    text='Generate 2D Perlin Map',
    command=lambda: perlin_map2D_button_clicked()
)
```

```
perlin_map2D_button.pack(
    ipadx=5,
    ipady=5,
    pady = 2,
    expand=False,
)

seed_text = Label(root,text ="Gökay Akçay", font='Helvetica 7 normal')
seed_text.pack(ipady=5)


# Renders the Tkinter Frame.
root.mainloop()
```

Used Libraries: *Numpy, OpenGL, Tkinter, Matplotlib, Pygame*

References:

- CMSC 425: Lecture 12
  Procedural Generation: 1D Perlin Noise
- CMSC 425: Lecture 12
  Procedural Generation: 2D Perlin Noise
- Youtube / GetIntoGameDev (OpenGL with Python)
- KhanAcademy
  computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise
- Geeks for Geeks